Développement d'algorithmes répartis corrects par construction

Manamiary Bruno Andriamiarina

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Développement d'algorithmes répartis corrects par construction

THÈSE

présentée et soutenue publiquement le 20 Octobre 2015

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Manamiary Bruno Andriamiarina

Composition du jury

Présidente :	Chrisment Isabelle	Professeur à l'Université de Lorraine, Telecom Nancy, LORIA
Rapporteurs :	Attiogbé Jérémie-Christian	Professeur de l'Université de Nantes, LINA
	Quéinnec Philippe	Professeur des Universités à l'INPT/ENSEEIHT, IRIT Toulouse
Examinateurs :	Méry Dominique	Professeur à l'Université de Lorraine, Telecom Nancy, LORIA - *Directeur de thèse*
	Merz Stephan	Directeur de recherche INRIA, LORIA
	Mosbah Mohamed	Professeur des Universités à l'ENSEIRB-MATMECA, LaBRI Bordeaux
	Poppleton Michael R.	Maître de Conférence à l'Université de Southampton, United Kingdom

Mis en page avec la classe thesul.

Sommaire

Chapitre 7
Les algorithmes de snapshot

Chapitre 8
Les systèmes auto-stabilisants

Table des figures

Première partie

Contexte des travaux

1

Introduction

Sommaire

Actuellement, nous assistons à l'émergence de l'utilisation des réseaux et des systèmes répartis, qui peuvent être à grande échelle (échelle d'une ville, d'un pays) ou à échelle locale (dans une habitation). Les systèmes répartis dynamiques mobiles de type ad-hoc (MANET - Mobile ad-hoc Network) deviennent de plus en plus populaires, ce qui est dû notamment à de nouvelles technologies, telles que les smartphones, les tablettes, ordinateurs portables, consoles de jeu, etc, ainsi qu'aux architectures de réseau sans fil, comme le wifi, le bluetooth ou les technologies 3G/4G. Ces systèmes répartis font coopérer et interagir des entités de calculs autonomes, utilisant des technologies hétérogènes (smartphones, tablettes, puces électroniques, etc) et ne partageant aucune connaissance, donnée, etc globales communes. L'importance croissante de ces réseaux est accompagnée par l'apparition, ainsi que la multiplication constante, de services distants, tels que le commerce, l'accès aux comptes bancaires, la recherche d'information, le stockage de données en ligne, etc. Des algorithmes particuliers, appelés algorithmes répartis, et conçus pour fonctionner pour des architectures réparties, permettent aux entités de calculs reliées entre elles par l'intermédiaire d'un réseau, de coopérer et/ou de collaborer à une ou plusieurs tâches communes (par exemple, la fourniture de services, tels que le routage, la communication, etc).

Nous nous intéressons à la conception de solutions algorithmiques réparties correctes et à la vérification d'algorithmes répartis existants. Nous caractérisons ces algorithmes notamment par les services qu'ils fournissent, à titre d'exemple, l'élection d'un leader, l'auto-stabilisation dans un système réparti, etc. Ces algorithmes répartis s'exécutent et fournissent des services pour des systèmes répartis présentant des particularités, telles que la mobilité, l'hétérogénéité, etc, et en outre, doivent satisfaire des propriétés de sûreté, de vivacité, d'équité. Différentes techniques pour s'assurer que ces algorithmes sont corrects, satisfont les propriétés requises et fournissent les services qui leur sont demandés, existent, et parmi elles, notamment les méthodes formelles.

Trois étapes principales peuvent être dégagées lors de l'utilisation des méthodes formelles pour la conception et le développement d'un algorithme réparti :

1. La mise au point d'un cahier des charges, spécifiant en langage naturel, les exigences requises, les services et les propriétés attendus pour l'algorithme réparti.
2. La formalisation de la spécification décrite dans le cahier des charges dans un modèle de l'algorithme, c'est-à-dire une représentation symbolique de ce dernier, généralement abstraite. Cette étape permet ici de s'assurer que le modèle respecte ce qui a été écrit dans le cahier des charges.
3. La dernière étape consiste à démontrer que l'implémentation de l'algorithme réparti satisfait aux exigences de la spécification. Il s'agit de l'étape de *vérification formelle*.

Une méthode pour vérifier que l'implémentation respecte les exigences de la spécification est le *raffinement de modèles* : il s'agit d'établir une spécification abstraite du système et de l'enrichir au fur et à mesure, jusqu'à l'obtention d'un modèle d'implémentation. Le code source sera dérivé à partir du modèle d'implémentation. Il s'agit de l'application de la méthodologie de *correction-par-construction* [14]. La notion de *raffinement* et le paradigme de *correction-par-construction* constituent le cœur de nos travaux. Dans notre cas, vérifier une implémentation revient donc à démontrer que le modèle de la spécification d'un algorithme cible est *raffiné* par le modèle de l'implémentation de ce même algorithme, c'est-à-dire de montrer que le modèle de l'implémentation *simule* tous les comportements abstraits exprimés dans le modèle de la spécification.

Nous avons utilisé, durant cette thèse, la méthode formelle EVENT-B [1], intégrant le raffinement et la démonstration de théorèmes, pour montrer la correction d'un système étudié. EVENT-B repose sur la logique du premier ordre ainsi que sur la théorie des ensembles. Cette méthode permet de formaliser des systèmes dynamiques (matériels ou logiciels). L'état d'un système est défini dans un modèle EVENT-B par une liste (*finie*) de variables d'états dont les valeurs évoluent avec le déclenchement des événements du modèle ; chaque événement du modèle est caractérisé par une *garde* qui exprime les conditions nécessaires au déclenchement de l'événement et des *actions* modifient les valeurs des variables d'état. Un *invariant* contenu dans le modèle spécifie des contraintes sur les états du système étudié, ainsi que les propriétés requises. La preuve de théorèmes, par l'intermédiaire de démonstrations automatiques, permet de vérifier la consistance de l'invariant, par rapport aux événements du modèle. Ces mécanismes sont enrichis par le *raffinement de modèles*. La relation de raffinement entre un modèle concret et un modèle abstrait est établie en démontrant la consistance d'un *invariant de collage*, qui relie les variables d'état concrètes et abstraites.

Le raffinement de modèles fourni par Event-B nous a permis de développer pas à pas, de manière incrémentale des modèles d'algorithmes répartis, à partir d'abstractions décrivant des spécifications, jusqu'à un niveau proche d'une forme algorithmique locale focalisée sur les entités de calculs participant à l'algorithme. Nous nous sommes principalement intéressés à caractériser les algorithmes répartis que nous avons étudiés par des propriétés de vivacité [13], pour exprimer et guider aussi bien le développement des modèles Event-B que les étapes du développement par raffinement. Nous nous sommes aussi intéressés à la réutilisation possible des composants d'un système complexe : soit en les recomposant pour obtenir un modèle du système étudié, soit en les réutilisant dans d'autres systèmes. Un de nos objectifs a aussi été d'intégrer à notre méthodologie de développement par raffinement, l'analyse de propriétés temporelles, telles que la vivacité et l'équité, pour guider la modélisation formelle d'un algorithme réparti : nous avons pour cela utilisé la logique TLA [13], pour exprimer ces propriétés temporelles, vérifier leur validité et établir des relations entre des propriétés caractérisant différents niveaux de raffinement du modèle de l'algorithme réparti étudié.

Parmi les systèmes que nous avons étudiés, figurent notamment : des systèmes de type FPGA et NoC [3], des algorithmes de routage répartis (algorithmes de routage xy [3] et Anycast RP [21, 22]), des systèmes répartis auto-stabilisants [15], des algorithmes de *snapshot* [23], des algorithmes répartis présentant des aspects non-fonctionnels, principalement probabilistes (Election du leader - IEEE 1394 [20, 2], algorithmes de coloriage d'un graphe [4]).

1.1 Systèmes répartis dynamiques

Durant cette thèse, nous avons étudié et analysé des algorithmes répartis pouvant fonctionner dans des *systèmes répartis dynamiques* (e.g. algorithmes de routage xy [3] et Anycast RP [21, 22], protocoles d'auto-stabilisation [15]). Dans cette section, nous détaillons les systèmes répartis dynamiques et nous donnons les différences par rapport aux systèmes répartis fixes (dont la topologie ne change pas), dits *classiques*.

Un système réparti dynamique est couramment appelé Manet (Mobile ad hoc network ou Réseau Mobile ad hoc) [8, 10]. Il s'agit d'un système complexe (voir figure 1.1), composé de nœuds (entités) mobiles. Cette capacité des nœuds permet d'établir des réseaux et systèmes répartis dans des endroits où aucune infrastructure réseau et communication n'existe, par exemple dans les zones frappées par des catastrophes naturelles, etc. Les réseaux ainsi formés sont des associations temporaires de nœuds qui communiquent les uns avec les autres par l'intermédiaire de moyens de communication sans fil (Wifi/IEEE 802.11, Bluetooth, Hyperlan), caractérisée par une *portée* du signal. Par conséquent, les nœuds capables de communiquer directement sont les nœuds dont les portées des signaux se chevauchent. Les communications entre deux nœuds distants (dont les portées des signaux ne se chevauchent pas) se font par des sauts multiples utilisant des nœuds intermédiaires.

Les nœuds ont la liberté de rejoindre ou de quitter le réseau quand ils le veulent, ce qui est la cause principale des changements de topologie du système réparti. La mobilité des nœuds et les déplacements de leurs utilisateurs rendent les changements encore plus rapides et imprévisibles. Il faut aussi noter que les entités (nœuds) reliées entre elles sont hétérogènes : il ne s'agit généralement pas du même matériel, du même système d'exploitation, etc. De plus, les nœuds sont anonymes et ne connaissent généralement rien des capacités de leurs voisins. D'autres problèmes liés à la sécurité physique des entités, à l'énergie (capacité des batteries des appareils mobiles), à l'adaptation à un changement d'ordre de grandeur du réseau, à la fiabilité des canaux de communication sans fil (par rapport aux canaux filaires traditionnels), à la bande passante, etc, doivent aussi être pris en compte. Il faut noter que dans les systèmes répartis dynamiques, les nœuds assurent généralement les rôles de clients et de serveurs, ainsi que les fonctionnalités de routeurs.

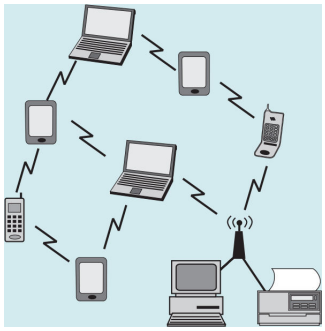


FIGURE 1.1 – Un MANET [8, 10]

Un système réparti dynamique possède les complexités citées précédemment. Cependant, ces systèmes constituent des alternatives prometteuses aux systèmes répartis fixes traditionnels [8, 10], principalement dans les zones où les architectures réparties fixes ne peuvent être déployées. Les principaux avantages sont dûs à l'indépendance des nœuds du réseau d'un système réparti dynamique, ce qui permet d'obtenir un réseau disposant de qualités d'auto-création, auto-organisation et auto-gestion. Un système réparti dynamique peut aussi être utilisé avec un système réparti déjà existant : il peut, par exemple, être utilisé avec le réseau internet et permettre ainsi à ses composantes d'utiliser les services de l'internet et de proposer les leurs.

De nos jours, l'apparition de nouvelles technologies et appareils de communication mobiles tels que les smartphones, les ultraportables, les assistants personnels, etc, font que les usagers mobiles peuvent accéder depuis des appareils portables/mobiles à des services nécessitant des connexions réseaux et des acheminements de données, tels que la consultation des mails, la localisation GPS, les échanges de fichiers, etc. Les besoins des utilisateurs combinés à la prolifération des appareils mobiles et aux avancées technologiques sans fil favorisent l'utilisation des MANETs et nous font entrer dans l'ère de l'informatique ubiquitaire [8, 10], où les usagers accèdent aux informations et données, n'importe où, n'importe quand.

Pour conclure cette section, nous disons que la conception et le développement formel d'un algorithmes réparti, s'exécutant dans un système réparti dynamique, ne doit pas seulement prendre en compte les spécifications de l'algorithme, avec les exigences et propriétés requises, mais doit aussi tenir compte de l'environnement réparti, de la mobilité et des possibles modifications pouvant avoir lieu dans le réseau (modification de la topologie : apparitions et suppressions de liens dans le réseau) [9, 17].

1.2 Objectifs et apports de la thèse

1.2.1 Objectifs de la thèse

Nous avons abordé durant cette thèse la problématique suivante : *la modélisation de systèmes répartis dynamiques*. L'objectif de cette thèse est la mise en place d'une méthodologie de conception et de vérification formelle des algorithmes répartis s'exécutant dans des systèmes répartis dynamiques, mobiles. Nous nous plaçons dans le cadre d'une approche *orientée composant* : notre but est la composition d'algorithmes répartis développés indépendamment les uns des autres, pour obtenir d'autres algorithmes répartis plus complexes.

Nous nous plaçons donc dans un contexte de réutilisation, d'instanciation, d'adaptation et de combinaison de modèles formels d'algorithmes répartis, dans le but d'obtenir des systèmes autrement plus complexes. L'objectif est de simplifier le développement formel d'un algorithme réparti complexe en réutilisant d'anciens développements formels pouvant être intégrés au problème étudié, réduisant ainsi la charge de travail au niveau de la modélisation et de l'effort de preuve.

Pour cela, nous avons pris en compte les aspects suivants :

— Comment composer des algorithmes répartis prouvés corrects, sans que cette composition n'influe sur les propriétés démontrées dans les composants indépendants. Des contraintes sur les compositions d'algorithmes, ainsi que la définition de l'environnement en commun dans lequel ces derniers doivent s'exécuter ont donc été posées.

— Démontrer la correction et le respect des spécifications par un algorithme réparti résultant d'une compositions d'algorithmes corrects plus simples, en réutilisant la correction des composants.
— Les points précédents impliquent donc que nous réutilisons les preuves de chacun des composants durant la composition, sans conséquences négatives ou remise en cause. Les propriétés prouvées dans les composants peuvent être réutilisées pour déduire de nouvelles propriétés relatives à la composition. Les preuves de ces nouvelles propriétés (relatives à la composition) doivent être les seules preuves supplémentaires à effectuer lors de la composition.
— La systématisation de la réutilisation de composants, de modèles, de preuves à l'aide de *plugins* ou *greffons* fournis par l'outil de modélisation [19].

1.2.2 Apports de la thèse

Le premier apport de cette thèse est de proposer l'enrichissement [5, 16] de la sémantique de la méthode Event-B à l'aide de la logique temporelle TLA [13], pour la prise en compte des traces d'exécution d'un système étudié. Cette extension nous permet d'exprimer simplement les algorithmes répartis étudiés : si le service s rendu par un algorithme est caractérisé par une pré-condition P et une post-condition Q, nous exprimons ce service à l'aide d'une propriété de vivacité, utilisant l'opérateur « *leads to* » ($\rightsquigarrow$), $P \rightsquigarrow Q$, qui signifie que lorsque la pré-condition P est satisfaite, alors fatalement, dans le futur, la post-condition Q le sera aussi.

Le second apport de la thèse découle de la première, sous la forme du paradigme *service-as-event* [5, 16], qui permet de définir l'abstraction d'un algorithme réparti à l'aide de ses services identifiés : chaque propriété de vivacité exprimant un service est représenté par un événement abstrait. La condition de déclenchement de l'événement abstrait est la précondition du service et le résultat de l'exécution de l'événement satisfait la post-condition du service. Des représentations graphiques appelées diagrammes de raffinement [5, 16] permettent de représenter simplement les abstractions ainsi obtenues et les propriétés de vivacité les caractérisant.

Le troisième apport est de proposer l'utilisation du raffinement pour définir une méthodologie de décomposition d'un problème complexe en étapes algorithmiques plus petites et plus simples : à partir d'un modèle abstrait et étape par étape, nous identifions les principales phases algorithmiques d'un algorithme réparti. Le modèle abstrait est caractérisé par une propriété de vivacité de type « *leads to* » $P \rightsquigarrow Q$. Pas à pas, par le raffinement et en utilisant les règles d'inférences (transitivité, etc) relatives à l'opérateur « *leads to* » [13], nous ajoutons entre ces pré et post-conditions initiales des pré et post-conditions intermédiaires décrivant les phases algorithmiques d'un algorithme réparti. Le modèle du problème initial est ainsi décomposé en sous-modèles indépendants modélisant chacun une des phases identifiées de l'algorithme réparti étudié.

Le dernier apport est la proposition d'une méthode de composition et de réutilisation de modèles formels d'algorithmes répartis, guidée par les pré et post-conditions, ainsi que les propriétés de vivacité de type *leads to*. Nous composons des phases algorithmiques selon les prédicats pré et post-conditions caractérisant respectivement les entrées et sorties de chacun des composants : nous assemblons deux composants A et B entre eux, si le prédicat post-condition contraignant les résultats de A correspond au prédicat pré-condition caractérisant les entrées de B.

Nous avons ainsi développé et prouvé des algorithmes répartis, notamment l'algorithme de routage Anycast RP, proposé par CISCO Systems [21], les communications dans les Réseaux-sur-puces [7, 3] (communications entre les cœurs d'une puce, algorithme de routage XY), des algorithmes de snapshot [23, 6, 23, 11, 12, 18, 24] permettant de prendre une photographie des états globaux d'un système réparti à un instant t, et des algorithmes auto-stabilisants, proposés par Marquezan et al [15].

1.3 Publications en rapport avec la thèse

Les publications en lien avec cette thèse sont les suivantes :

1. Journaux internationaux avec comité de lecture :
   — Manamiary Bruno Andriamiarina, Dominique Méry, Neeraj Kumar Singh. **Revisiting Snapshot Algorithms by Refinement- Based Techniques – Extended Version.** Dans COM-

SIS : Computer Science and Information Systems (`http://www.comsis.org`), Vol. 11, Issue 1, page 251. Janvier 2014.

2. Conférences internationales avec comité de lecture et actes :
   — Manamiary Bruno Andriamiarina, Dominique Méry, Neeraj Kumar Singh. **Analysis of Self-⋆ and P2P Systems using Refinement.** Dans ABZ 2014. Toulouse, France. Juin 2014.
   — Manamiary Bruno Andriamiarina, Dominique Méry, Neeraj Kumar Singh. **Integrating Proved State-Based Models for Constructing Correct Distributed Algorithms.** Dans iFM 2013 : 10th International Conference on integrated Formal Methods, pages 268-284. Turku, Finlande. Juin 2013.
   — Manamiary Bruno Andriamiarina, Dominique Méry, Neeraj Kumar Singh. **Revisiting Snapshot Algorithms by Refinement-based Techniques.** Dans PDCAT, pages 343-349. IEEE Computer Society, 2012. Beijing, China. 14-16 Décembre 2012.
3. Ateliers internationaux avec comité de lecture et actes :
   — Manamiary Bruno Andriamiarina, Hayat Daoud, Mostefa Belarbi, Dominique Méry, Camel Tanougast. **Formal Verification of Fault Tolerant NoC-based Architecture.** Dans First International Workshop on Mathematics and Computer Science (IWMCS2012), Tiaret, Algérie. Décembre 2012.
   — Manamiary Bruno Andriamiarina. **Stepwise Development of Distributed Algorithms.** Dans FM 2011 : Doctoral Symposium. Limerick, Irlande. 2011.
4. Colloques nationaux avec comité de lecture et actes :
   — Manamiary Bruno Andriamiarina. **Prise en Compte des Aspects Probabilistes dans le Raffinement.** Dans AFADL 2013 : Approches Formelles dans l'Assistance au Développement de Logiciels. LORIA, Nancy, France. Avril 2013.

Bibliographie

[1] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering.* Cambridge University Press, 2010.

[2] J.-R. Abrial, D. Cansell, and D. Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3) :215–227, 2003.

[3] M. B. Andriamiarina, H. Daoud, M. Belarbi, D. Méry, and C. Tanougast. Formal Verification of Fault Tolerant NoC-based Architecture. In *First International Workshop on Mathematics and Computer Science (IWMCS2012)*, Tiaret, Algérie, Dec. 2012.

[4] M. B. Andriamiarina and D. Méry. Stepwise development Of Distributed Vertex Coloring Algorithms (Full Report). Technical report, Lorraine Research Laboratory in Computer Science and its Applications (LORIA), University of Lorraine, Nancy, France, July 2011.

[5] M. B. Andriamiarina, D. Méry, and N. K. Singh. Integrating proved state-based models for constructing correct distributed algorithms. In E. B. Johnsen and L. Petre, editors, *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 2013.

[6] K. M. Chandy and L. Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1) :63–75, 1985.

[7] S. D. Chawade, M. A. Gaikwad, and R. M. Patrikar. Article : Review of xy routing algorithm for network-on-chip architecture. *International Journal of Computer Applications*, 43(21/973-93-80867-69-8) :20–23, April 2012. Published by Foundation of Computer Science, New York, USA.

[8] I. Chlamtac, M. Conti, and J. J.-N. Liu. Mobile ad hoc networking : imperatives and challenges. *Ad Hoc Networks*, 1(1) :13–64, 2003.

[9] T. S. Hoang, H. Kuruma, D. Basin, and J.-R. Abrial. Developing topology discovery in event-b. *Sci. Comput. Program.*, 74(11-12) :879–899, Nov. 2009.

[10] J. Hoebeke, I. Moerman, B. Dhoedt, and P. Demeester. An Overview of Mobile Ad Hoc Networks : Applications and Challenges. *Journal of the Communications Network*, 3 :60–66, July 2004.

[11] A. D. Kshemkalyani, M. Raynal, and M. Singhal. An introduction to snapshot algorithms in distributed computing. *Distributed Systems Engineering*, 2(4) :224, 1995.

[12] T.-H. Lai and T. H. Yang. On distributed snapshots. *Inf. Process. Lett.*, 25(3) :153–158, 1987.

[13] L. Lamport. A temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3) :872–923, May 1994.

[14] G. T. Leavens, J.-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In *Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006)*, pages 221–235. ACM, Oct. 2006.

[15] C. C. Marquezan and L. Z. Granville. *Self-* and P2P for Network Management - Design Principles and Case Studies*. Springer Briefs in Computer Science. Springer, 2012.

[16] D. Méry. Refinement-based guidelines for algorithmic systems. *International Journal of Software and Informatics*, 3(2-3) :197–239, June/September 2009.

[17] D. Méry and N. K. Singh. Analysis of DSR protocol in event-B. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, SSS'11, pages 401–415, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] C. Morgan. Global and logical time in distributed algorithms. *Inf. Process. Lett.*, 20(4) :189–194, 1985.

[19] Project RODIN. Rigorous open development environment for complex systems. http ://www.eventb.org/, 2004-2010.

[20] J. Rehm and D. Cansell. Proved Development of the Real-Time Properties of the IEEE 1394 Root Contention Protocol with the Event B Method. In *ISoLA*, pages 179–190, 2007.

[21] C. Systems. Anycast RP. `http://www.cisco.com/en/US/docs/ios/solutions_docs/ip_multicast/White_papers`.

[22] C. Systems. Anycast RP using PIM. `http://tools.ietf.org/html/draft-ietf-pim-anycast-rp-07`.

[23] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, New York, NY, USA, 1994.

[24] Z. Yang and T. A. Marsland. Global snapshots for distributed debugging : An overview. Technical Report TR-92-03, Computing Science Department, University of Alberta, 1992.

2

Etat de l'Art

Sommaire

2.1 Introduction

Ce chapitre présente un état de l'art, dans lequel nous nous intéressons aux méthodes, techniques, langages et outils utilisés pour spécifier des systèmes, algorithmes répartis et leurs propriétés, pour vérifier formellement que ces systèmes satisfont les propriétés et s'assurer ainsi de la correction de ces systèmes étudiés. Il est structuré de la manière suivante : la section 2.2 présente les méthodes, techniques et outils de modélisation et vérification des algorithmes répartis présents dans la littérature. Finalement, nous présentons dans la section 2.3 nos conclusions : nous y justifions notamment le choix des langages et méthodes formels que nous utilisons dans cette thèse, par rapport à cet état de l'art.

2.2 Modélisation, formalisation et vérification des algorithmes répartis

Cette section présente les méthodes, techniques et outils de la vérification formelle des algorithmes répartis permettant de montrer que ces algorithmes sont corrects et satisfont leurs spécifications. Les algorithmes répartis sont caractérisés par les aspects suivants [35] : la concurrence et la coopération entre les processus composant les systèmes répartis supports des algorithmes étudiés, le non-déterminisme, les hypothèses à poser sur l'environnement du système étudié, les interférences possibles inter-processus ou entre les processus et leurs environnements, etc.

Nous trouvons dans la littérature différentes méthodes de spécification et de vérification prenant en compte ces caractéristiques, notamment la concurrence, la coopération inter-processus, les interférences et le non- déterminisme. Nous présentons ces méthodes dans cette section, en particulier celles proposées par Hoare [27, 29], Owicki et Gries [57], Schlichting et Schneider [66], Apt, Olderog et De Boer [5],Apt, Francez et De Roever [6], Jones [36, 38, 37], Chandy et Misra [15], Milner [53], Manna et Pnuelli [48, 50, 49, 9], Schneider et Treharne [67], Tel [73], Raynal [64, 65].

Nous présentons ensuite des langages de modélisation de systèmes et algorithmes répartis (I/O Automata, LOTOS, UML, SDL, etc), ainsi que les méthodes (model-checking, preuve de théorèmes), outils (SPIN, PVS, etc) et techniques permettant de vérifier formellement ces modèles.

2.2.1 Spécification de processus et programmes concurrents

Hoare introduit et propose dans [28] une méthode déductive pour la vérification de programmes séquentiels : un programme S est caractérisé par deux assertions P et Q, et la notation suivante $\{P\}S\{Q\}$ signifie que si P est vraie avant l'exécution de S, alors si le programme S termine, Q est vraie après l'exécution de S ; des axiomes, des assertions à propos des états du programme S, des règles d'inférences permettent de prouver la correction de ce dernier. Hoare étend ensuite ces notions et concepts aux programmes et processus parallèles [27].

Hoare est aussi à l'origine de CSP (Communicating Sequential Processes) [29]. Il s'agit d'un *algèbre de processus* permettant de modéliser les interactions entre les processus composant un système : CSP permet de décrire le comportement interne d'un processus, ainsi que sa synchronisation (communication) avec son environnements, à l'aide d'événements qui définissent les possibilités de chaque processus (ce qu'un processus peut ou ne peut pas faire).

Un autre algèbre de processus, appelé CCS (Calculus of Communicating Systems) a été introduit par Milner [53]. CCS permet de représenter un système étudié par un ensemble de processus : un processus est un agent capable de communiquer avec son environnement, via une interface, composée de canaux/ports de communications entrants et sortants. Un processus est défini par son comportement interne et des opérateurs de composition (choix non-déterministe, composition parallèle, etc) permettent de modéliser des comportements plus complexes impliquant un processus et son environnement.

Owicki et Gries [57] étendent la méthode déductive de vérification de programmes séquentiels de Hoare en prenant en compte la possible concurrence et les possibles interférences entre les processus d'un système : ces derniers peuvent en effet partager des ressources. La non-interférence entre les processus, lors de l'utilisation de ressources communes, doit par conséquent être démontrée, lors de la preuve de la correction d'un système.

Jones [36, 38, 37] présente la méthode Rely-Guarantee, pour prendre en compte la concurrence et les interférences entre processus dans un système réparti d'une manière différente de celle proposée par Owicki et Gries, en privilégiant une approche locale axée sur les composants : un système est validé par la correction de chacun de ses composants (processus), qui sont spécifiés à l'aide de quatre propriétés, c'est-à-dire, une propriété *pre* qui est la condition d'exécution d'un programme (calculs, traitements, etc), une propriété *rely* qui caractérise les interférences du composant avec l'environnement, une propriété *guar* qui caractérise les transitions internes du composant et une propriété post qui est la condition satisfaite après l'exécution du programme. La spécification d'un processus du système comprend donc ses interactions possibles avec son environnement.

Apt, Francez et De Roever introduisent dans [6] une méthode de preuve pour des programmes concurrents spécifiés en CSP [26]. Comparées aux approches de Jones, Owicki et Gries décrites précédemment, cette dernière ne se focalise pas sur la notion d'interférences entre processus, mais plutôt sur une notion de coopération. Apt et al ajoutent donc des preuves supplémentaires de coopération entre processus, lors de la preuve de la correction d'un système composé de processus concurrents.

Schlichting et Schneider [66] introduisent une méthode pour prouver la correction de systèmes composés de processus concurrents communicant par messages. Des règles d'inférence permettent de démontrer des propriétés de systèmes dont les composants, reliés par des canaux (pouvant être non-fiables), se synchronisent par transmission de messages, rendez-vous, appel distant de procédures, etc, et aussi de contrôler des facteurs comme les possibles interférences entre processus.

Apt, Olderog et De Boer [5] proposent pour les programmes répartis, une méthode basée sur des *transformations* de programmes, pour la simplification de preuves de propriétés telles que la correction totale et/ou partielle du programme ou encore l'équité : à titre d'exemple, des programmes répartis sont transformés en programmes séquentiels non-déterministes et des propriétés de correction, etc, sont prouvées en utilisant des assertions, axiomes, règles d'inférence proches de celles présentées par Hoare.

Chandy et Misra mettent au point, en 1988, UNITY [15] qui est une méthodologie de spécification, de conception et de vérification de programmes concurrents/parallèles. Elle est constituée d'un langage de programmation permettant de décrire les systèmes étudiés, à l'aide de variables et d'actions modifiant les valeurs de ces variables, d'une logique, fragment de la logique temporelle, permettant d'exprimer des propriétés (sûreté, progrès) qu'un programme doit satisfaire, ainsi que d'un système de déduction (axiomes et règles d'inférence) permettant de prouver qu'un programme satisfait les propriétés désirées.

Manna et Pueli [48, 50] proposent une méthodologie de vérification des systèmes réactifs et concurrents basée sur l'utilisation de la logique temporelle pour la spécification des propriétés des systèmes étudiés, représentés sous la forme de systèmes de transitions équitables. Cette méthodologie permet notamment de vérifier que de propriétés telles que la vivacité, la sûreté [50], sont satisfaites par les systèmes étudiés. L'outil de vérification formelle associé est la plateforme de vérification formelle SteP (Stanford Temporal Prover) [9] : un model-checker et un prouveur, ainsi qu'un langage visuel appelé « *verification diagrams* » (pour guider organiser, afficher les preuves) [49] permettent de vérifier qu'un système satisfait sa spécification.

Schneider et Treharne [67] proposent une combinaison des méthodes B et CSP pour la vérification de systèmes concurrents : chaque processus d'un système étudié est représenté par un modèle B, et un processus appelé « contrôleur » décrit en CSP, permet à chaque modèle B représentant un composant du système de communiquer et de se synchroniser avec d'autres modèles en utilisant les canaux/port de communications du « contrôleur ».

Gerard Tel présente dans [73], une collection d'algorithmes répartis (élection du leader, algorithmes de snapshot, etc) et se focalise surtout sur l'analyse de ces algorithmes, dont les processus (acteurs des algorithmes) membres des systèmes répartis supports communiquent de manière point-à-point et par messages. Les systèmes analysés sont représentés dans un cadre formel basé sur les systèmes de transitions, permettant ainsi l'étude de propriétés telles que la sûreté et la vivacité.

Michel Raynal introduit aux algorithmes concurrents dans [64] et aux algorithmes répartis dans [65]. Il y présente les concepts, principes et mécanismes liés à la concurrence, principalement à la synchronisation entre processus [64]. Similairement à Gérard Tel dans [73], Raynal se focalise aussi sur les algorithmes répartis, fonctionnant dans des systèmes répartis dont les processus communiquent par messages, dans son ouvrage [65].

Généralement, ces méthodes (e.g. Hoare [27], Owicki et Gries [57]) nécessitent la spécification complète

concrète du système à étudier, contrairement à notre démarche qui est de détailler la spécification du système pas à pas, par raffinement, à partir d'une abstraction à laquelle des détails sont ajoutés au fur et à mesure.

Nous caractérisons les algorithmes répartis, que nous modélisons, par les services qu'ils fournissent : nous exprimons chaque service s par des propriétés de vivacité de type $P \rightsquigarrow Q$, signifiant que P est la condition attendue par le service (ou un sous-service) pour s'exécuter (*pré-condition*) et Q est la condition vérifiée après l'exécution du service (ou sous-service) (*post-condition*), et qu'une exécution du service(ou sous-service) à partir d'un état satisfaisant P conduit *fatalement* à un état satisfaisant Q. Le raffinement et les règles d'inférence liées à ces propriétés de vivacité nous permettent ensuite de décomposer graduellement ces services, ainsi que les propriétés de vivacité les caractérisant, jusqu'à l'obtention des programmes locaux les implémentant, lesquels sont aussi caractérisés par des propriétés de vivacités, ainsi que des pré et post-conditions. Par rapport aux méthodes proposées par Hoare [27] ou Owicki et Gries [57], l'effort de preuve est divisé, grâce au raffinement : la correction des programmes s'exécutant dans un système réparti et implémentant les services fournis par le système est garantie par les obligations de preuves à décharger lors des étapes du processus de raffinement de modèles.

Nous tenons compte des interférences à partir du niveau abstrait : comme nous modélisons l'algorithme et son environnement dans un même modèle, nous veillons à contrôler les possibles interactions des services avec l'environnement, au niveau abstrait, en prenant en compte les possibles perturbations (e.g. pertes de messages, défaillance d'un composant, etc) et en définissant les réactions du service par rapport à ces dernières. Le raffinement nous permet de préserver cette gestion des interférences, jusqu'au niveau local des processus. Nous nous rapprochons ainsi de la méthode de Jones [36, 38, 37] pour la spécification d'un service et des composants (processus) qui le fournissent : nous intégrons, dans les spécifications des services ou des composants, les interactions avec l'environnement.

Pour nos cas d'études, nous proposons l'utilisation d'« actions ou commandes gardées » pour la modélisation et la représentation des systèmes que nous vérifions. Cette méthodologie de description et de représentation est un concept bien établi dans le domaine de la formalisation des systèmes concurrents (actions ou commandes gardées de Dijkstra [21], Unity [15], TLA/TLA$^+$ [42, 43], etc). En résumé, par rapport aux méthodes de spécification vues dans cette section, ce que nous introduisons est l'utilisation du raffinement, ainsi que des propriétés de vivacité pour la spécification d'un algorithme réparti : au niveau abstrait, nous spécifions l'algorithme par ses services, et l'utilisation du raffinement de modèle, ainsi que des propriétés de vivacité nous permet d'arriver à un niveau local proche d'un algorithme, où les processus sont visibles. Dans la sous-section suivante, nous nous intéressons aux méthodes et techniques existant dans la littérature et qui ont été utilisées pour la modélisation et la vérification d'une catégorie de systèmes concurrents, qui est celle des systèmes répartis.

2.2.2 Méthodes, outils, langages pour la modélisation et la vérification mécanique d'algorithmes répartis

Nous présentons dans cette sous-section des méthodes, outils, langages et techniques mises en œuvre pour la modélisation et la mécanisation de la vérification des algorithmes et systèmes répartis. Nous abordons dans un premier temps des techniques utilisées pour vérifier qu'un modèle d'un algorithme et de son système support satisfait une spécification : il s'agit du model-checking et de la démonstration de propriétés ; puis nous aborderons divers langages permettant de construire des modèles de systèmes, analysés généralement à l'aide des techniques citées précédemment et nous comparons ensuite les éléments présentés dans cette sous-section à la méthodologie que nous appliquons pour modéliser les systèmes répartis.

2.2.2.1 Techniques de vérification de propriétés

Le model-checking [17, 16] permet de vérifier si un algorithme réparti étudié, plus précisément un modèle de ce dernier, satisfait une spécification généralement écrite en logique temporelle [42, 61], par exploration exhaustive et automatique des états possibles du modèles. Nous citons ici deux model-checkers bien connus et utilisés pour la vérification des algorithmes répartis : SPIN [31] et PVS [58]. Le model-checking est une option intéressante pour la vérification d'algorithmes répartis, parce que dans le cas

où une propriété ne serait pas vérifiée par un modèle, le model-checker utilisé fournit généralement un contre-exemple, instructif et guidant l'utilisateur dans la correction du modèle ou de la propriété à vérifier. Cependant, le model-checking est limité par l'explosion combinatoire des états [16], due aux caractéristiques inhérentes aux algorithmes et systèmes répartis [35] pouvant rendre un modèle complexe, telles que le nombre de processus, les hypothèses à poser sur l'environnement du système étudié, les interférences possibles entre les processus et l'environnement, le non-déterminisme, les types et structures de données, etc. Lors de la vérification par model-checking (e.g. Cublicle [18, 19]), des caractéristiques des systèmes répartis étudiés (e.g. nombre de processus, les délais de communication ou la consommation d'énergie), peuvent ne pas être constantes, mais plutôt être des paramètres [40, 35, 40, 18, 19]. Cependant, vérifier les propriétés (sûreté, vivacité, équité) de tels systèmes paramétrés par model-checking est indécidable [35, 40, 18, 19].

La démonstration de propriétés constitue une autre technique de vérification des algorithmes répartis : un modèle mathématique de l'algorithme réparti à étudier est analysé, dans le but de montrer que le modèle satisfait la spécification de l'algorithme. Des obligations de preuves, écrites dans un langage mathématico-logique et générées à partir du modèle, doivent être démontrées dans un système de déduction logique (i.e. axiomes et règles d'inférence), à l'aide d'assistants de preuves [59, 75]. Certaines caractéristiques des systèmes répartis, que nous avons déjà vu précédemment, tels que la concurrence entre les processus, le non-déterminisme, les interférences, etc peuvent complexifier la modélisation d'un algorithme réparti. Par contre, par rapport au model-checking, la démonstration de propriétés n'étant pas basée sur l'exploration des états et permettant de valider toutes les instances d'un modèle, nous ne sommes plus contraints par les aspects tels que le nombre de processus, etc. PVS est un exemple de plateforme de vérification proposant en plus du model- checking, la preuve de théorèmes et ayant permis de vérifier des protocoles de communication [24]. Nous avons choisi durant nos travaux d'utiliser la méthode Event-B [1], notamment à cause du raffinement de modèles et de la démonstration de propriétés ; nous utilisons ainsi les prouveurs et assistants de preuves conçus pour Event-B, lors de la vérification de nos modèles et de la validation du raffinement de modèles.

2.2.2.2 Langages de spécification et de conception d'algorithmes et de systèmes répartis

Les I/O automata (automates à entrées/sorties) ont été introduits par Lynch et al [47] et permettent de représenter un système réparti par une composition d'automates (machines à états), représentant chacun un composant du système réparti caractérisé par des actions internes et des actions de communications avec son environnement (entrées/sorties). Ces automates ont permis à Lynch et al [46] d'étudier des algorithmes répartis de consensus, communications, allocations de ressources, synchronisation.

Le langage LOTOS [11, 33] a été introduit principalement pour la description de protocoles de communication [30, 56, 20], mais permet aussi, plus généralement, de décrire les systèmes répartis, par un ensemble de processus, modélisés par des boîtes noires accomplissant des actions internes et des actions externes d'interaction et de synchronisation avec l'environnement, par l'intermédiaire d'interface appelées « portes ».

UML [39, 62, 12] est un langage graphique permettant de décrire un système réparti à l'aide de diagrammes de comportement et d'interactions, représentant les fonctionnalités, activités et actions du système étudié, ainsi que de diagrammes de structure décrivant l'architecture du système. UML s'avère très pratique pour comprendre et visualiser un système réparti [74, 70], mais est dépourvu d'une sémantique formelle [74, 8, 7, 70], de la base mathématique et des aspects de raisonnement qui se retrouvent dans les méthodes formelles.

SDL [34] est un langage de description formelle, adopté comme un standard par les organisations ITU et ISO pour l'analyse de systèmes communicants concurrentiels, dont les systèmes répartis. Ces systèmes répartis sont modélisés par des blocs composés de processus exécutant des actions et/ou produisant des signaux permettant les communications inter-blocs et/ou avec l'environnement.

ADA [72] est un langage de programmation, dont la première version a été mise au point dans les années 1980, et permettant de développer divers systèmes dont notamment les systèmes concurrents [32], répartis [60]. Le langage, notamment depuis la version Ada 2012, permet la programmation par contrat [51, 52] : des assertions exprimant des pré/post-conditions, des invariants peuvent être introduites dans le code d'un programme et sont vérifiées à l'exécution. Les outils permettant de vérifier que les

programmes ADA satisfont les assertions les caractérisant (pré/post-conditions, invariants), sont fournis par l'environnement GNATprove [25] : la compilation à l'aide de GNAT d'un code source ADA annoté, permet la génération d'une spécification en WhyML, langage supporté par la plateforme de vérification Why3. Des obligations de preuves sont engendrées et sont déchargées automatiquement par des prouveurs tels que Alt-Ergo.

Certains des langages et méthodes de modélisation que nous avons présentés ici possèdent nativement des systèmes de déduction (UNITY), des techniques de preuves sur les traces d'exécutions (I/O Automata) ou encore intègrent des paradigmes telles que la programmation par contrat (ADA) pour la vérification de propriétés, telles que les propriétés d'invariance/sûreté, de vivacité, d'inclusions de traces/simulation, etc. Cependant, nous remarquons aussi que généralement (i.e. cas de LOTOS, I/O Automata, UML, SDL, ADA) une autre technique est utilisée pour pour la vérification : il s'agit de coupler les méthodes et langages à des outils externes [74, 71, 13, 23, 69, 68, 55], c'est-à-dire que les modèles des systèmes étudiés et les propriétés à vérifier sont traduits dans des notations exploitables par des model-checkers (SPIN [31], PVS [59, 58]), des prouveurs et assistants de preuves (PVS [59, 58], Isabelle [75]), des plateformes tels que IF [14, 13], CADP [22], Why [10], permettant ainsi de vérifier mécaniquement si les systèmes étudiés satisfont les propriétés requises.

2.2.2.3 Discussions

Nous avons passé en revue, dans les sous-sections de cet état de l'art, des langages et méthodes permettant de modéliser des systèmes et algorithmes répartis, tels que les I/O automata, LOTOS, SDL, UML, ADA et UNITY. Ces langages et méthodes ne permettent généralement que d'établir des modèles des systèmes étudiés sous forme de composants réalisant des tâches simples et coopérant à la réalisation de fonctionnalités plus compliquées. Ils (hormis UNITY et dans une certaine mesure ADA et les I/O automata) ne permettent généralement pas de démontrer les propriétés de vivacité, sûreté et équité relatives à ces systèmes. La vérification formelle des systèmes étudiés se fait ainsi communément par un interfaçage avec des outils formels externes (assistants de preuves, prouveurs, model-checkers, etc). Nous avons aussi remarqué que certaines de ces méthodes présentent des notions intéressantes, telles que le raffinement (notamment les I/O automata, SDL, UNITY), permettant d'établir des liens de simulation entre un modèle de spécification et un modèle d'implémentation. Cependant, le raffinement nous semble peu ou pas utilisé, lors de la modélisation des algorithmes et systèmes répartis : généralement, les modèles ne sont pas élaborés pas-à-pas, à partir d'abstractions, mais plutôt construits en une seule étape.

Notre méthodologie de développement formel des algorithmes et systèmes répartis est basée sur l'utilisation de la méthode formelle Event-B [1]. Comparée aux méthodes vues plus haut dans cet état de l'art, notre méthodologie repose donc principalement sur l'application du paradigme de « correction-par-construction » [44], et l'utilisation intensive du raffinement : nous partons d'un modèle très simple, très abstrait du système étudié, que nous enrichissons pas-à-pas, jusqu' à obtenir une forme algorithmique locale proche d'un code informatique. Event-B dispose en outre d'un véritable petit laboratoire qui lui est dédiée : la plateforme Rodin [63], qui lui fournit des prouveurs, pour démontrer des propriétés d'invariance/sûreté, la consistance du raffinement, ainsi qu'un model-checker ProB [45] et de nombreux autres outils, qui sont des extensions/greffons (plugins) de la plateforme. UNITY et Event-B partagent des aspects, notamment la notion de commandes (UNITY) gardées (événements en Event-B) : une commande est exécutée si sa garde est vraie ; ainsi que la possibilité de raffiner des spécifications abstraites. Nous notons qu'Event-B permet la vérification de propriétés de sûreté et d'invariance, alors qu'UNITY permet de raisonner sur d'autres propriétés, notamment celles de vivacité. Durant cette thèse, un de nos objectifs a été notamment d'enrichir la sémantique d'Event-Bpour la prise en compte de propriétés de vivacité et d'équité, ce qui aurait pu être fait à l'aide d'UNITY. Cependant, nous avons choisi pour cela la logique TLA [42], pour les raisons que nous expliquons ci-après.

Nous utilisons aussi la logique temporelle TLA [42] pour caractériser les algorithmes et systèmes étudiés, notamment à l'aide d'hypothèses d'équité (forte et/ou faible) et de propriétés de vivacité exprimées à l'aide de l'opérateur leadsto ($\rightsquigarrow$). Nous caractérisons donc un système par des propriétés de « progrès », similairement à ce qui est fait avec UNITY. Nous avons choisi d'utiliser la logique TLA, car nous la considérons en un sens plus générale et plus complète par rapport à UNITY : **(1)** TLA permet de prendre en compte facilement différentes propriétés d'équité sur les actions du système, telles que l'équité faible,

l'équité forte, ou encore l'absence d'équité, alors qu'UNITY repose seulement sur l'équité inconditionnelle [54] (chacune des actions d'un système est observée infiniment souvent) ; les autres types d'équités ou l'absence d'équité devant être pris en compte en les exprimant en tant qu'axiomes avec UNITY ou en redéfinissant certains opérateurs d'UNITY, tels que *ensures* [54]. **(2)** UNITY est un fragment restreint de la logique temporelle et ne permet pas, par exemple, l'utilisation de formules temporelles imbriquées [41], et manque ainsi d'expressivité par rapport à TLA. Par conséquent, nous estimons que si nous devons faire évoluer notre méthodologie pour prendre en compte des propriétés de vivacité autres que celles de progrès, TLA est plus adaptée.

2.3 Conclusion

Nous avons cité dans ce chapitre des travaux relatifs à notre sujet qui est la modélisation et la vérification des algorithmes et systèmes répartis : nous avons passé en revue dans un premier temps les méthodes de spécification de processus et programmes concurrents vues dans la littérature, puis nous nous sommes intéressés aux langages, techniques et outils permettant de mettre en œuvre la conception, la modélisation et la vérification de systèmes répartis.

Nous proposons [3, 2, 4], par rapport aux travaux vus dans cet état de l'art, une méthodologie de conception et de vérification des algorithmes répartis basée sur le paradigme de « correction-par-construction » et caractérisée par l'utilisation intensive du raffinement de modèles, divisant ainsi les efforts de preuves et de modélisation : les modèles sont détaillés progressivement, au fur et à mesure du raffinement. Pour la mise en œuvre de cette méthodologie, nous utilisons la méthode Event-B dont le cœur est le raffinement de modèles et qui dispose d'une plateforme très riche qui lui est dédiée (Rodin) [63]. Notre procédé de développement formel est aussi caractérisé par le fait qu'il est guidé, à chaque pas/étape de raffinement, par des propriétés de vivacité et d'équité caractérisant les fonctionnalités et services offerts par les algorithmes répartis étudiés. Pour prendre en compte ces propriétés, nous nous servons de la logique temporelle TLA [42] pour étendre la sémantique d'Event-B.

Le chapitre suivant présente les deux méthodes formelles utilisées dans nos travaux : Event-B [1] et TLA [42].

Bibliographie

[1] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, 2010.

[2] M. B. Andriamiarina, D. Méry, and N. K. Singh. Revisiting snapshot algorithms by refinement-based techniques. In H. Shen, Y. Sang, Y. Li, D. Qian, and A. Y. Zomaya, editors, *13th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2012, Beijing, China, December 14-16, 2012*, pages 343–349. IEEE, 2012.

[3] M. B. Andriamiarina, D. Méry, and N. K. Singh. Integrating proved state-based models for constructing correct distributed algorithms. In E. B. Johnsen and L. Petre, editors, *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 2013.

[4] M. B. Andriamiarina, D. Méry, and N. K. Singh. Analysis of self-* and p2p systems using refinement. In Y. Aït-Ameur and K.-D. Schewe, editors, *ABZ*, volume 8477 of *Lecture Notes in Computer Science*, pages 117–123. Springer, 2014.

[5] K. R. Apt, F. S. de Boer, and E. Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 2009.

[6] K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 2(3) :359–385, July 1980.

[7] L. Apvrille, J.-P. Courtiat, C. Lohr, and P. de Saqui-Sannes. Turtle : A real-time uml profile supported by a formal validation toolkit. *IEEE Transactions on Software Engineering*, 30(7) :473–487, 2004.

[8] D. B. Aredo. *Formal Development of Open Distributed Systems : Integration of UML and PVS*. PhD thesis, University of Oslo, 2004.

[9] N. Bjørner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H. Sipma, and T. E. Uribe. Verifying temporal properties of reactive systems : A step tutorial. *Formal Methods in System Design*, 16(3) :227–270, 2000.

[10] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3 : Shepherd your herd of provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[11] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14(1) :25–59, Mar. 1987.

[12] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[13] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, L. Mounier, and J. Sifakis. If : An intermediate representation for sdl and its applications. In *SDL Forum*, pages 423–440, 1999.

[14] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. Tools and Applications II : The IF Toolset. In F. Bernardo, Marco ; Corradini, editor, *Formal Methods for the Design of Real-Time Systems International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer SCience*, pages 237–267, Bertinoro, Italie, Sept. 2004. Springer.

[15] K. M. Chandy. *Parallel program design : a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[16] E. M. Clarke. The birth of model checking. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2008.

[17] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, december 1999. ISBN 0-262-03270-8.

[18] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Cubicle : A parallel smt-based model checker for parameterized systems - tool paper. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 718–724. Springer, 2012.

[19] S. Conchon, A. Mebsout, and F. Zaïdi. Vérification de systèmes paramétrés avec Cubicle. In *Vingt-quatrièmes Journées Francophones des Langages Applicatifs*, Aussois, France, Feb. 2013.

[20] J. D. Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12) :1334–1340, Dec. 1983.

[21] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8) :453–457, Aug. 1975.

[22] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010 : A Toolbox for the Construction and Analysis of Distributed Processes. In *Tools and Algorithms for the Construction and Analysis of Systems - TACAS 2011*, Saabrucken, Allemagne, Apr. 2011.

[23] J. Guitton, J. Kanig, and Y. Moy. Why hi-lite ada ? In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[24] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, FME '96, pages 662–681, London, UK, UK, 1996. Springer-Verlag.

[25] D. Hoang, Y. Moy, A. Wallenburg, and R. Chapman. Spark 2014 and gnatprove. *International Journal on Software Tools for Technology Transfer*, pages 1–13, 2014.

[26] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1984.

[27] C. Hoare. *Towards a theory of parallel programming*, pages 231–244. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12 :576–580, 1969.

[29] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8) :666–677, Aug. 1978.

[30] D. Hogrefe. OSI formal specification case study : The Inres protocol and service. Technical Report 91-012, University of Bern, Switzerland, 1991.

[31] G. Holzmann. *Spin Model Checker, the : Primer and Reference Manual.* Addison-Wesley Professional, first edition, 2003.

[32] D. J. Howe and S. Michell. An approach to formal verication of real time concurrent ada programs. *Ada Lett.*, XXIII(4) :87–92, Sept. 2003.

[33] ISO/IEC. *Information Processing Systems – Open Systems Interconnection : LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, 1989.

[34] ITU-T. *Recommendation Z.100 : CCITT Specification and Description Language (SDL)*, 1988.

[35] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Towards modeling and model checking fault-tolerant distributed algorithms. In *Proceedings 20th International Symposium on Model Checking Software (SPIN'13), Springer LNCS 7976*, pages 209–226. Springer, 2013.

[36] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference.* PhD thesis, Oxford University, jun 1981. Printed as : Programming Research Group, Technical Monograph 25.

[37] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[38] C. B. Jones. Granularity and the development of concurrent programs. *Electr. Notes Theor. Comput. Sci.*, 1 :302–306, 1995.

[39] P. S. Kaliappan. Designing dependable distributed systems using model driven architecture techniques – an approach. In *Proc. of the Computer Science Report 03/07*, pages 49–56, 2007.

[40] I. Konnov, H. Veith, and J. Widder. Who is afraid of model checking distributed algorithms ? CAV Workshop $(EC)^2$, 2012.

[41] L. Lamport. Verification and specification of concurrent programs. In *A Decade of Concurrency*, volume 803, pages 347–374. Springer-Verlag, 1993.

[42] L. Lamport. A temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3) :872–923, May 1994.

[43] L. Lamport. *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[44] G. T. Leavens, J.-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In *Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006)*, pages 221–235. ACM, Oct. 2006.

[45] M. Leuschel and M. J. Butler. Prob : an automated analysis toolset for the b method. *STTT*, 10(2) :185–203, 2008.

[46] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[47] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2 :219–246, 1989.

[48] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[49] Z. Manna and A. Pnueli. Temporal verification diagrams. In M. Hagiya and J. C. Mitchell, editors, *TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 726–765. Springer, 1994.

[50] Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995.

[51] B. Meyer. Applying "design by contract". *Computer*, 25(10) :40–51, Oct. 1992.

[52] B. Meyer. *Eiffel : The Language*. Prentice Hall International Ltd., 1992.

[53] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[54] J. Misra. How to reason with strong-fairness and no-fairness. *Notes on UNITY*, 31, 1992.

[55] D. Méry and A. Mokkedem. Crocos : An integrated environment for interactive verification of sdl specifications. In G. Bochmann and D. Probst, editors, *Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 343–356. Springer Berlin Heidelberg, 1993.

[56] Nokia Research Center (Finland). INRES Protocol. http ://cadp.inria.fr/case-studies/98-g-inres.html, 1998.

[57] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6 :319–340, 1976.

[58] S. Owre, J. M. Rushby, , and N. Shankar. PVS : A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[59] S. Owre and N. Shankar. A brief overview of pvs. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 22–27. Springer, 2008.

[60] L. Pautet, T. Quinot, and S. Tardieu. Building modern distributed systems. In D. Craeynest and A. Strohmeier, editors, *Reliable Software Technologies : Ada Europe 2001, 6th Ade-Europe International Conference Leuven, Belgium, May 14-18, 2001, Proceedings*, volume 2043 of *Lecture Notes in Computer Science*, pages 123–135. Springer, 2001.

[61] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[62] POPKIN SOFTWARE. Modeling Systems With UML - A Popkin Software White Paper. `http://www.akira.ruc.dk/~keld/teaching/OOP_e09/uml_modeling.pdf`.

[63] Project RODIN. Rigorous open development environment for complex systems. http ://www.eventb.org/, 2004-2010.

[64] M. Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013.

[65] M. Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer, 2013.

[66] R. D. Schlichting and F. B. Schneider. Using message passing for distributed programming : Proof rules, disciplines. *ACM Trans. Program. Lang. Syst.*, 6(3) :402–431, 1984.

[67] S. Schneider and H. Treharne. Csp theorems for communicating b machines. *Formal Asp. Comput.*, 17(4) :390–422, 2005.

[68] G. Scollo. Formal Description of the OSI Session Layer : Transport Service. In P. V. Eijk and M. Diaz, editors, *Formal Description Technique Lotos : Results of the Esprit Sedos Project*, 1989.

[69] N. Sidorova and M. Steffen. Verifying large sdl-specifications using model checking. In R. Reed and J. Reed, editors, *SDL 2001 : Meeting UML*, volume 2078 of *Lecture Notes in Computer Science*, pages 403–420. Springer Berlin Heidelberg, 2001.

[70] C. Snook and M. Butler. Uml-b : Formal modeling and design aided by uml. *ACM Trans. Softw. Eng. Methodol.*, 15(1) :92–122, Jan. 2006.

[71] C. Snook and M. Butler. Uml-b and event-b : An integration of languages and tools. In *Proceedings of the IASTED International Conference on Software Engineering*, SE '08, pages 336–341, Anaheim, CA, USA, 2008. ACTA Press.

[72] S. T. Taft, R. A. Duff, R. Brukardt, E. Plödereder, P. Leroy, and E. Schonberg. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E)*, volume 8339 of *Lecture Notes in Computer Science*. Springer, 2013.

[73] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, New York, NY, USA, 1994.

[74] I. Traor'e. An outline of pvs semantics for uml statecharts. *J. UCS*, 6(11) :1088–1108, 2000.

[75] M. Wenzel, L. C. Paulson, and T. Nipkow. The isabelle framework. In A. Mohamed, Munoz, and Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 33–38. Springer, 2008.

Deuxième partie

Méthodologie de développement d'algorithmes répartis

3

Formalismes à états : Event-B et TLA

Sommaire

3.1 Introduction

Notre principal objectif durant cette thèse est la modélisation et la vérification d'algorithmes répartis [7, 6, 8] ; pour ce faire, nous proposons une méthodologie basée sur le paradigme de « *correction-par-construction* » [17] : le raffinement de modèles, que nous utilisons pour le développement graduel des systèmes étudiés, est le cœur de notre méthodologie, qui est aussi guidée par des propriétés temporelles (de vivacité) caractérisant les systèmes étudiés. Par conséquent, nous avons choisi durant cette thèse d'utiliser les méthodes formelles suivantes : la méthode Event-B [3], permettant le développement par raffinement de systèmes, ainsi que la logique temporelle TLA [16]. Ce chapitre nous introduit à ces méthodes formelles que nous avons employées pour nos travaux.

Ce chapitre est structuré de la manière suivante. La section 2 nous présente Event-B : les modèles, le raffinement de modèles, les techniques et outils que nous avons utilisés durant cette thèse. La section 3 nous introduit de manière générale à TLA et se focalise particulièrement sur les propriétés de vivacité auxquelles nous nous sommes intéressé lors de la modélisation des algorithmes répartis, c'est-à-dire les propriétés de fatalité exprimées par des opérateurs « *leadsto* » ($\rightsquigarrow$). La dernière section présente nos conclusions sur l'intégration et l'utilisation conjointe d'Event-B et TLA.

3.2 Event-B

Event-B permet le développement et l'analyse formels d'une grande variété de systèmes appelés systèmes dynamiques discrets (matériels ou logiciels). Ces systèmes peuvent être : une machine (ordinateur, smartphone,etc) exécutant des instructions, effectuant des calculs, etc, des composants électroniques, un réseau d'appareils électroniques, un système de contrôle d'accès avec les personnes, les bâtiments et les voies d'accès mis en jeu, etc. Event-B utilise la logique du premier ordre et la théorie des ensembles pour la spécification des systèmes. Ces fondements mathématiques permettent de démontrer que chaque étape de la construction d'un système est correcte : cette correction est attestée par la preuve de conditions de vérification.

Un modèle Event-B est caractérisé par des états décrits par des variables d'état, ainsi qu'un ensemble d'actions appelées événements, modifiant la valeur des variables et décrivant ainsi les changements d'états du modèle. La notation formelle d'Event-B permet d'exprimer des propriétés de sûreté, sous la forme de clauses dans le modèle du système étudié. Une propriété de sûreté spécifie les propriétés du système qui sont toujours vraies : elles sont valides à l'état initial et sont préservées durant les changements d'état. La démonstration de telles propriétés se fait en déchargeant des obligations de preuves engendrées au préalable pour le modèle Event-B.

Une notion essentielle de la modélisation en Event-B est le raffinement, permettant de représenter le système étudié à différents niveaux d'abstraction : un utilisateur modélise le système étudié d'abord par une spécification mathématique abstraite, et ensuite, par ajout pas à pas de contraintes, de détails à cette spécification abstraite, une forme algorithmique proche d'un code informatique est atteinte. Le but est de tendre, à l'aide d'une chaîne de raffinements partant du modèle abstrait, à des spécifications de plus en plus fines, de moins en moins abstraites. Chaque étape du raffinement est validée par des preuves mathématiques permettant de démontrer la correction et la consistance existant entre les différents niveaux de raffinement successifs. Ce processus est appelé « *correction-par-construction* » [17].

Nous détaillons maintenant la modélisation d'un système en Event-B. Un modèle Event-B est composé de deux entités :

— Un *contexte* qui décrit la spécification non-dynamique d'un système.
— Une *machine* qui est la spécification dynamique du système

Les sous-sections suivantes présentent ces deux composantes d'un modèle Event-B plus en détails.

3.2.1 Contextes

Un contexte exprime les propriétés statiques d'un modèle Event-B. Il contient des ensembles porteurs, des constantes, des axiomes et des théorèmes. Les axiomes caractérisent les propriétés des ensembles porteurs et des constantes. Les théorèmes sont des propriétés qui sont prouvées à partir des axiomes. Les

obligations de preuves associées à un contexte sont liées à la démonstration des théorèmes placés dans le contexte : les théorèmes doivent être démontrés à partir des axiomes et autres théorèmes préalablement définis. Un contexte peut être étendu (clause *extends*) par un autre contexte. Dans ce cas, le contexte qui étend peut introduire de nouveaux ensembles, constantes, propriétés, etc. Un contexte peut être « *vu* » par une machine à l'aide de la clause *sees*. Un contexte C est vu indirectement par une machine M, si une extension de C est vue par la machine M.

```
CONTEXT
  /* Nom du Contexte */
EXTENDS
  /* Contextes étendus par le Contexte */
SETS
  /* Ensembles porteurs */
CONSTANTS
  /* Constantes */
AXIOMS
  /* Propriétés et Axiomes */
THEOREMS
  /* Théorèmes */
```

FIGURE 3.1 – Un contexte

La figure 3.2 suivante résume les liens existant entre les machines et les contextes.

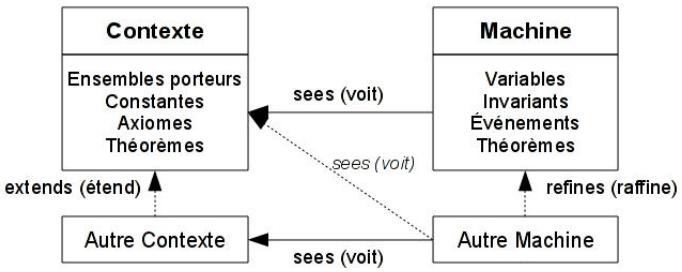


FIGURE 3.2 – Relations entre machines et contextes

3.2.2 Machines

Une machine exprime le comportement dynamique d'un modèle EVENT-B. Une machine peut contenir des variables, des invariants, des théorèmes, des événements et des variants. Les variables notées x, définissent les états de la machines et sont des objets mathématiques simples : ensembles, relations binaires, fonctions, nombres, etc. Un invariant noté $I(x)$ exprime des contraintes sur ces variables ; $I(x)$ doit être valide à l'état initial et doit être préservé lors des changements d'états (changement de valeurs des variables x).

Les événements modifiant les valeurs des variables d'états sont définis dans une machine. Une machine utilise les informations statiques (à l'aide de la clause *sees*) définies dans un ou plusieurs contextes. Ces mécanismes sont enrichis par le raffinement qui fait le lien entre une machine abstraite et une autre plus concrète (clause $refines$), par ajout de nouvelles variables et/ou de nouveaux événements.

```
MACHINE
  /* Nom de la machine */
REFINES
  /* Nom de la machine abstraite raffinée par cette machine */
SEES
  /* Contextes vus par la machine */
VARIABLES
  /* Variables d'état du modèle */
INVARIANT
  /* Propriétés d'invariance du système */
VARIANT
  /* Variant du système */
THEOREMS
  /* Théorèmes de la machine */
EVENTS
  /* Événements de la machine */
```

FIGURE 3.3 – Une machine

Le raffinement nous permet de développer graduellement des machines EVENT-B et de valider chaque étape du développement par la preuve. Le raffinement peut s'exprimer simplement de la manière suivante : Une machine P raffine une machine M, si P simule M.

3.2.3 Les événements

Une machine est caractérisée par une liste (finie) x de *variables d'état* pouvant être modifiées par une liste (finie) d'événements ; un invariant $I(x)$ exprime des propriétés qui doivent être *toujours satisfaites* par les variables x et *maintenues* par l'activation des événements. Le tableau 3.1 décrit les notations ensemblistes [2, 3] utilisées pour formaliser un système.

Nom	**Syntaxe**	**Définition**
Relation binaire	$s \leftrightarrow t$	$\mathcal{P}(s \times t)$
Composition	$r_1; r_2$	$\{x, y \mid x \in a \quad \wedge \quad y \in b \quad \wedge$ $\exists z.(z \in c \quad \wedge \quad x, z \in r_1 \quad \wedge \quad z, y \in r_2)\}$
Relation inverse	r^{-1}	$\{x, y \mid x \in \mathcal{P}(a) \quad \wedge \quad y \in \mathcal{P}(b) \quad \wedge \quad y, x \in r\}$
Domaine	$\mathsf{dom}(r)$	$\{a \mid a \in s \quad \wedge \quad \exists b.(b \in t \quad \wedge \quad a \mapsto b \in r)\}$
Co-domaine	$\mathsf{ran}(r)$	$\mathsf{dom}(r^{-1})$
Identité	$\mathsf{id}(s)$	$\{x, y \mid x \in s \quad \wedge \quad y \in s \quad \wedge \quad x = y\}$
Restriction	$s \lhd r$	$\mathsf{id}(s); r$
Co-restriction	$r \rhd s$	$r;\ \mathsf{id}(s)$
Anti-restriction	$s \lhd\!\!\!- r$	$(\mathsf{dom}(r) - s) \lhd r$
Anti-co-restriction	$r \rhd\!\!\!- s$	$r \rhd (\mathsf{ran}(r) - s)$
Image	$r[w]$	$\mathsf{ran}(w \lhd r)$
Surcharge	$q \lhd\!\!+ r$	$(\mathsf{dom}(r) \lhd\!\!\!- q) \cup r$
Fonction partielle	$s \nrightarrow t$	$\{r \mid r \in s \leftrightarrow t \quad \wedge \quad (r^{-1}; r) \subseteq \mathsf{id}(t)\}$
Fonction totale	$s \rightarrow t$	$\{f \mid f \in s \nrightarrow t \quad \wedge \quad dom(f) = s\}$
Injection totale	$s \rightarrowtail t$	$\{f \mid f \in s \rightarrow t \quad \wedge \quad f^{-1} \in t \nrightarrow s\}$

TABLE 3.1 – Notations ensemblistes de la méthode B

Chaque événement d'une machine peut être décomposé une *garde* $G(t, x)$ et une *action* $R(t, x, x')$, t étant les variables (paramètres) locales de l'événement et x' les nouvelles valeurs des variables x. La garde exprime la *condition nécessaire* pour que l'événement ait lieu, l'action décrit *comment les variables d'état évoluent* au moment où l'action se produit. Un événement peut prendre trois formes (voir table 3.2) :

— La première forme est la forme générale d'un événement, dite forme *non-déterministe*. Une garde, représentée par $\exists t \cdot G(t, x)$, exprime la condition nécessaire pour que l'événement puisse se produire. $x : |R(t, x, x')$ est l'action de l'événement. Les variables d'état de la machine sont représentées par x et t est un vecteur de variables locales à l'événement distinctes. La présence des variables locales t fait que cet événement est non-déterministe.

— La seconde forme ne possède aucune variable locale, elle ne dépend que des variables d'état x de la machine. Cette forme est dite *gardée*.
— La troisième forme est la plus simple, car la garde est toujours vraie.

Ces trois formes d'événements sont équivalentes sémantiquement à des prédicats *avant-après* que nous notons BA (Before-After, en anglais). Ces prédicats notés $BA(e)(x, x')$ décrivent pour chaque événement e d'une machine, la relation entre les valeurs x des variables d'état avant et après le déclenchement des événements. Les valeurs des variables d'état après le déclenchement de l'événement e sont notées x'. Le tableau ci-dessous résume la garde ($grd(e)(x)$) et le prédicat avant-après ($BA(e)(x, x')$) pour chaque forme d'événement e.

Événement e	**Garde :** $grd(e)(x)$	**Prédicat avant-après :** $BA(e)(x, x')$
BEGIN $x :\mid R(x, x')$ END	$TRUE$	$R(x, x')$
WHEN $G(x)$ THEN $x :\mid R(x, x')$ END	$G(x)$	$G(x) \ \wedge \ R(x, x')$
ANY t WHERE $G(t, x)$ THEN $x :\mid R(t, x, x')$ END	$\exists t \cdot G(t, x)$	$\exists t \cdot G(t, x) \ \wedge \ R(t, x, x')$

TABLE 3.2 – Gardes et prédicats « avant-après »

Les changements d'états sont exprimés à l'aide des *susbtitutions généralisées*. La partie action d'un événement est constituée d'une substitution qui fait évoluer la valeur des variables x d'une machine. Les substitutions ont trois formes possibles :

Type	**Substitution généralisée**
Vide	*skip*
Déterministe	$x := E(x, t)$
Non-Déterministe	$x :\in E(x, t)$ $x :\mid P(x, x')$

TABLE 3.3 – Substitutions généralisées

— **Vide :** *skip* signifie que l'action de l'événement est vide. Les nouvelles valeurs x' des variables sont donc identiques aux anciennes x.
— **Déterministe :** $x := E(x, t)$ est une substitution simple ressemblant à une affectation où la variable x prend la valeur de l'expression $E(x, t)$.
— **Non-déterministe :**
  — $x :\mid P(x, x')$ est la forme la plus générale des substitutions, elle exprime la relation entre la valeur des variables avant (x) et après la substitution (x'). *Les variables x sont modifiées de telle manière que les nouvelles valeurs de x, notée x', satisfont le prédicat $P(x, x')$.*
  — $x :\in E(x, t)$ signifie que la valeur d'un élément de $E(x, t)$ sera affectée à x.

Des obligations de preuve (PO) sont engendrées pour les modèles EVENT-B. Les obligations de preuves sont de plusieurs sortes, et nous en citons quelques unes : WD (Bonne définition), INV (Préservation de l'invariant), GRD (renforcement de la garde), SIM (simulation d'une action), FIS (Faisabilité d'un événement), etc. Les obligations de preuve doivent être déchargées pour garantir la correction des modèles B Événementiel. Elles concernent différents critères des modèles : certaines (WD) peuvent servir à montrer la bonne définition des expressions utilisées, démontrer (INV) les propriétés d'invariance d'une machine, prouver la consistance d'un raffinement (GRD et SIM), etc.

3.2.4 Le raffinement de modèle

Le raffinement d'un modèle formel nous permet d'enrichir ce modèle *pas-à-pas*, à l'aide d'une approche *incrémentale*. Le raffinement est le fondement du paradigme de *correction-par-construction* [17], que nous utilisons pour modéliser des systèmes. Le mécanisme de raffinement permet de renforcer l'invariant d'un modèle et de le rendre plus fin en y ajoutant des détails. La transformation d'un modèle abstrait en un modèle plus concret, plus fin, se fait en ajoutant de nouvelles variables au modèle abstrait (et aussi en en supprimant), en raffinant chaque événement abstrait en un ensemble d'événements plus concrets et en introduisant de nouveaux événements. Un invariant $J(x, y)$, que nous appelons *invariant de collage* relie les variables abstraites, que nous notons x, aux variables abstraites, notées y. Des obligations de preuve garantissent que :

— Chaque événement abstrait est correctement raffiné par sa version concrète.
— Chaque nouvel événement introduit raffine *skip*.
— Il n'arrive jamais que les nouveaux événements introduits se déclenchent indéfiniment et prennent la main sans la rendre aux événements abstraits.
— Il n'y a pas plus de blocage dans la machine concrète que dans la machine abstraite qu'elle raffine.

La première obligation de preuve générée et à décharger concerne l'initialisation concrète. Soit l'initialisation concrète, l'événement suivant :

Initialisation $\triangleq$
BEGIN
$y : |INITC(y')$
END

$INITC$ est le prédicat caractérisant les états initiaux du modèle concret, et $INITA$ le prédicat caractérisant l'initialisation abstraite. $INITC(y')$ attribue une valeur initiale aux nouvelles variables concrètes et à celles conservées du modèle abstrait. Il faut alors démontrer qu'étant donnée une valeur initiale concrète y', il existe une valeur abstraite correspondante x' avec laquelle l'invariant concret (de collage) J est vérifié :

$$INITC(y') \Rightarrow \exists x' \cdot (INITA(x') \wedge J(x', y'))$$

La seconde obligation de preuve à démontrer est celle montrant que chaque événement abstrait est correctement raffiné par sa version concrète. Soit un modèle abstrait AM avec des variables d'état x et un invariant $I(x)$. Le modèle AM est raffiné par un modèle concret CM avec des variables d'état y et un invariant de collage noté $J(x, y)$. Un événement e se trouvant dans le modèle AM est raffiné par un événement f placé dans le modèle CM. Les prédicats avant-après de e et de f sont respectivement $BA(e)(x, x')$ et $BA(f)(y, y')$. L'obligation de preuve qui garantit que l'événement abstrait e est correctement raffiné par l'événement concret f est la suivante :

$$I(x) \wedge J(x, y) \wedge BA(f)(y, y') \Rightarrow \exists x' \cdot (BA(e)(x, x') \wedge J(x', y'))$$

Le raffinement d'un événement e par un événement f signifie que l'événement f simule l'événement e.

Les nouveaux événements introduits dans une étape de raffinement peuvent être vus comme des événements cachés, invisibles à l'environnement abstrait du système et hors de contrôle de ce dernier. Exiger que les nouveaux événements dans un modèle concret raffinent *skip* signifie que les effets de ces derniers ne sont pas observables au niveau abstrait : les exécutions de ces nouveaux événements se produisent généralement entre les exécutions des événements abstraits visibles. L'obligation de preuve suivante exprime que chaque nouvel événement f du modèle concret doit raffiner *skip* ($x = x'$) :

$$I(x) \wedge J(x, y) \wedge BA(f)(y, y') \Rightarrow J(x, y')$$

L'obligation de preuve suivante est à propos d'un *variant* ($V(y)$), qui doit décroître à chaque déclenchement d'un nouvel événement, empêchant ainsi les nouveaux événements de s'exécuter indéfiniment aux dépens des événements abstraits. Il faut démontrer que le variant décroît à chaque exécution d'un nouvel événement f (les nouveaux événements finiront ainsi par redonner la main aux événements abstraits) :

$$I(x) \wedge J(x, y) \wedge BA(f)(y, y') \Rightarrow V(y') < V(y)$$

La dernière obligation de preuve consiste à démontrer qu'il n'y a pas plus de blocage dans la machine concrète que dans la machine abstraite qu'elle raffine. Il s'agit de prouver que le raffinement, qui se fait généralement par renforcement des gardes des événements abstraits, n'induit pas plus de blocage dans le système. Nous notons $grd(f_1)(x)$, ..., $grd(f_n)(x)$ les gardes des événements concrets f_i (où $1 \leq i \leq n$) et $grd(e_1)(x)$, ..., $grd(e_m)(x)$ les gardes des événements abstraits e_j (où $1 \leq j \leq m$). Nous avons alors l'obligation de preuve suivante à décharger :

$$I(x) \wedge J(x,y) \wedge (grd(e_1)(x) \vee \ldots \vee grd(e_n)(x)) \Rightarrow (grd(f_1)(x) \vee \ldots \vee grd(f_m)(x))$$

En résumé, quand nous raffinons un modèle, un événement existant peut être raffiné en renforçant sa garde et/ou son prédicat avant-après, réduisant ainsi au fur et à mesure le non-déterminisme. De nouveaux événements raffinant *skip* peuvent aussi être ajoutés. Il convient de noter ici l'importance de la condition de *faisabilité* : chaque état possible doit avoir un successeur. Ce raffinement que nous appliquons nous permet de garantir que l'ensemble des traces du modèle concret est contenu (en tenant compte des étapes de bégaiement/modulo le bégaiement) dans les traces du modèle abstrait.

3.2.5 Les patrons de conception

Abrial et al introduisent dans [5, 14], les patrons de conception pour EVENT-B : un patron de conception est un développement formel (avec les constantes, variables, invariants, événements correspondants) dédié à la formalisation d'un problème commun, récurrent [14] (e.g. les actions/réactions dans les systèmes réactifs, la transmission de messages, etc). Un patron de conception inclue une spécification abstraite et des raffinements de cette spécification.

Abrial et al distinguent deux étapes lors de l'utilisation d'un patron de conception [5, 14] :

— L'adaptation : il s'agit ici de faire correspondre (*matching*) les constantes, les variables, les événements de la spécification abstraite d'un patron de conception avec des éléments d'un modèle d'un développement formel plus large, contenant le problème formalisé par le patron de conception. Cette correspondance est une forme d'instanciation, qui consiste à renommer, si nécessaire, les éléments du patron de conception, en accord avec ceux qui leur correspondent dans le développement plus large.
— L'incorporation : il s'agit de la *composition* des éléments de la spécification abstraite du patron de conception et de ceux du modèle du développement plus large, possédant des éléments correspondant à ceux du patron de conception [5, 14]. Les raffinements présentés par le patron de conception peuvent être alors incorporés pour obtenir des raffinements du développement plus large.

La figure 3.4 ci-après, tirée de [14], illustre l'adaptation et l'incorporation d'un patron de conception de conception composé de deux modèles $\mathsf{p_0}$ et $\mathsf{p_1}$, à un développement formel plus large comportant $n+1$ modèles, permettant de produire un raffinement $\mathsf{m_{n+1}}$ du modèle $\mathsf{m_n}$ du développement plus large.

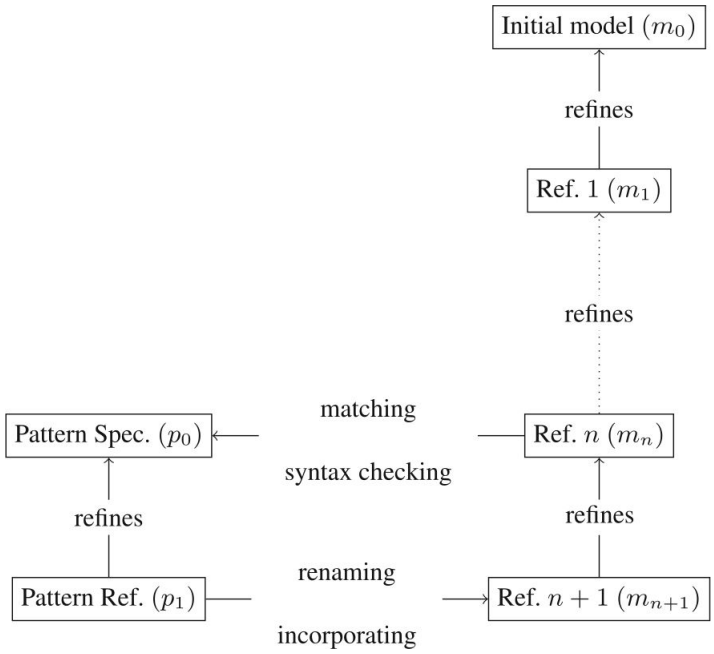


FIGURE 3.4 – Utilisation d'un patron de conception en EVENT-B

En résumé, les patrons de conception introduits par Abrial et al [5, 14] illustrent une application de la réutilisation de modèles existants et des preuves de correction relatives à ces modèles, permettant ainsi d'économiser du temps et des efforts lors du développement formel de systèmes complexes.

3.2.6 Outils pour Event-B

Différents outils existent pour l'application de la méthode EVENT-B. L'outil historique est l'Atelier B [10], développé par la société Clearsy. Un autre outil, appelé plateforme RODIN [21], a été développé récemment [4] dans le cadre d'un projet européen. Il s'agit d'une extension (dédiée à EVENT-B) de l'environnement de développement Eclipse. Ces deux environnements permettent de créer des contextes, des machines, de mettre en œuvre le raffinement de modèles, de générer les obligations de preuves correspondant à des propriétés, de décharger ces obligations de preuves de manière automatique ou interactive, etc. Ils peuvent aussi être couplés à des outils permettant l'animation de modèles, etc. Dans les deux environnements, des prouveurs [11, 22] issus de l'atelier B (prouveurs PP, ML) ainsi que d'autres prouveurs (prouveur NEWPP) sont utilisés pour démontrer les obligations de preuves générées pour les modèles EVENT-B développés.

Durant cette thèse, nous avons essentiellement utilisé la plateforme RODIN comme environnement de développement formel pour nos études de cas. Les fonctionnalités proposées par cette plateforme peuvent être étendues, notamment à l'aide de greffons (plug-ins). Nous citons, dans cette sous-section, les différents plug-ins que nous avons utilisés lors de nos travaux, et que nous pouvons classer en trois catégories :

1. Les plug-ins relatifs aux preuves et démonstrations.
   — « Atelier B Provers » [11, 22] : ce plug-in intègre à la plateforme RODIN les prouveurs de l'Atelier B (PP, ML).
   — « SMT Plug-in » [13] est une interface permettant d'utiliser des solveurs SMT (Satisfiability Modulo Theories) pour décharger les obligations de preuves engendrées pour un modèle. Ce plug-in complète celui vu précédemment.
2. Les animateurs et model-checkers.
   — « ProB » [18] est un animateur et un model-checker dédié aux méthodes B et EVENT-B. Nous l'utilisons pour animer nos modèles et ainsi vérifier que les comportements de ces derniers sont bien ceux attendus. Il nous permet aussi de nous assurer que les propriétés que nous

voulons vérifier ne contiennent pas d'erreurs et aussi que les modèles ne sont pas sujets à des phénomènes tels que les interblocages (deadlocks), etc.

3. Les plug-ins axés sur la réutilisation de modèles et de preuves.
   — « Refactoring Framework » [23] : le renommage des éléments d'un développement Event-B (constantes, variables, événements, modèles, etc) peut entraîner une perte de preuves et ainsi obliger l'utilisateur à décharger à nouveau des obligations de preuve. Le plug-in « Refactoring Framework » permet d'éviter ce cas et procure aussi d'autres fonctionnalités utiles : il permet de renommer en une seule opération un élément, ainsi que toutes ses occurrences dans un développement formel.
   — « Pattern » [14] permet la réutilisation et l'instanciation d'anciens développements formels (considérés comme des patrons de conception) dans de nouveaux développements présentant des aspects/problèmes auxquels les anciens développements répondent.

Nous avons introduit dans cette section les notions de base relatives à la méthode Event-B. Nous y avons présenté les composantes d'un modèle, le raffinement de modèles, les différentes obligations de preuve liées à la correction des modèles et à la consistance du raffinement, ainsi que des techniques (utilisation des patrons de conception) et outils de modélisation. Le concept de *raffinement* est un des points clés d'Event-B et constitue la pierre angulaire des développements formels présentés durant cette thèse. La section suivante nous introduit à la logique temporelle TLA [16].

3.3 La logique TLA

Nous présentons la logique TLA, en nous inspirant librement des travaux de son créateur, Leslie Lamport [16]. TLA (Logique Temporelle des Actions) combine une logique temporelle avec une logique des actions. Elle permet de spécifier des systèmes réactifs, par rapport à des actions et des comportements. Elle permet notamment de vérifier si des propriétés temporelles, telles que la vivacité (fatalité, convergence), l'équité, sont satisfaites par les systèmes étudiés. Notre objectif est d'intégrer la logique TLA dans notre méthodologie [7, 6, 8] de développement formel avec Event-B, pour fournir ainsi un cadre temporel pour l'étude et l'analyse des algorithmes répartis.

3.3.1 Formules TLA

Un système étudié est modélisé en TLA par l'ensemble de toute ses exécutions possibles et ce sont ces comportements qui sont analysés pour vérifier la correction du système par rapport aux propriétés (vivacité, sûreté, équité) requises. Dans cette section, nous introduisons brièvement quelques formules TLA qui nous permettront de mieux comprendre les concepts de cette logique, ainsi que la modélisation et l'analyse de systèmes.

Nous commençons par définir les comportement et états d'un système étudié. Soient Var un ensemble infini dénombrable de variables et Val, un ensemble de valeurs. Un état s est défini comme une valuation des variables, il s'agit d'une fonction de Var vers Val : $s \in \mathsf{Var} \rightarrow \mathsf{Val}$. La notation $s(x)$, qui est aussi notée $s[\![x]\!]$ représente la valeur de la variable x à l'état s. Nous notons $\mathbf{St}$ l'ensemble de tous les états possibles. Un comportement est défini come une *suite* infinie d'états s_0, s_1, ..., s_i, ..., notée $\langle s_0, s_1, \ldots, s_i, \ldots \rangle$ en TLA.

Un fonction d'état est une expression non-booléenne construite à partir de variables et de symboles de constantes. La signification $[\![f]\!]$ d'une fonction d'état f est une fonction de l'ensemble des états possibles ($\mathbf{St}$) vers l'ensemble des valeurs (Val). Nous notons $s[\![f]\!]$ la valeur associée à f, à l'état s. Sémantiquement, $s[\![f]\!]$ est définie comme suit :

$$s[\![f]\!] \triangleq f(\forall\ `x' : s[\![x]\!]/x)$$

$f(\forall\ `x' : s[\![x]\!]/x)$ est la valeur de f à l'état s, où chaque variable x contenue dans f est remplacée par sa valeur (à l'état s) $s[\![x]\!]$.

Un prédicat est une expression booléenne construite à partir de variables et de symboles de constantes. Soient P un prédicat, s un état, x la valeur des variables à l'état s. $s[\![P]\!]$ est la valeur obtenue en remplaçant

les variables x dans P par leurs valeurs $s[\![x]\!]$ à l'état s. Un état s satisfait un prédicat P, si, et seulement si, $s[\![P]\!]$ est vraie.

Une action A est une expression booléenne contenant des variables non-primées, primées et des symboles de constantes, elle représente une relation entre une paire d'états consécutifs, un ancien (exprimé à l'aide des variables non-primées et des symboles de constantes) et un nouveau (exprimé à l'aide des variables primées et des symboles de constantes). Soient $\langle s, t\rangle$ une paire d'états consécutifs, x des variables ; $s[\![x]\!]$ est la valeur des variables à l'état s et $t[\![x]\!]$ la valeur des variables à l'état t. Le couple $\langle s, t\rangle$ satisfait l'action A si et seulement si $s[\![A]\!]t$ est vraie. $s[\![A]\!]t$ est la valeur obtenue en remplaçant les variables non-primées x par leurs valeurs $s[\![x]\!]$ à l'état s et celle primées (x') par les valeurs de x, $t[\![x]\!]$ à l'état t :

$$s[\![A]\!]t \triangleq A(\forall\ `x' : s[\![x]\!]/x, t[\![x]\!]/x')$$

Nous explicitons aussi ici quelques notations relatives aux actions, ainsi que leurs significations :

- ENABLED A est un prédicat qui est vrai dans un état s, si et seulement s'il est possible à partir de cet état s d'effectuer un pas A, c'est-à-dire, s'il existe un état t accessible depuis s par l'action A. ENABLED A est défini de la manière suivante : $s[\![\text{ENABLED } A]\!] \triangleq \exists t \in \mathbf{St} : s[\![A]\!]t$.
- $\langle A\rangle_x$ définit un pas A qui change les valeurs des variables x en de nouvelles valeurs x'. $\langle A\rangle_x$ est défini comme suit : $\langle A\rangle_x \triangleq A \wedge (x' \neq x)$.
- $[A]_x$ signifie que soit l'action A est satisfaite, soit nous avons un bégaiement : les valeurs des variables x ne changent pas. Sémantiquement, $[A]_x$ est défini comme suit : $[A]_x \triangleq A \vee (x' = x)$. $(x' = x)$ est définie par la notation *Unchanged* x.

Une formule temporelle est construite à partir de formules élémentaires en utilisant les opérateurs booléens et l'opérateur unaire $\Box$ (*« toujours »*). Une formule temporelle est considérée comme une assertion sur les comportements possibles du système étudié. Soient σ un comportement telle que $\sigma \ \widehat{=}\ \langle s_0, s_1, \ldots\rangle$ et F une formule temporelle. Nous notons $[\![F]\!]$ la signification de F : il s'agit d'une fonction de l'ensemble des comportements vers les booléens. Le comportement σ satisfait F, si, et seulement si, la valuation booléenne de $\sigma[\![F]\!]$ est vraie.

La satisfaction d'une formule par un comportement $\sigma \triangleq \langle s_0, s_1, \ldots\rangle$ est exprimée à partir de la validité de ses composantes, par rapport à σ. Nous donnons ici quelques exemples :

1. $\sigma[\![F \wedge G]\!] \triangleq \sigma[\![F]\!] \wedge \sigma[\![G]\!]$
2. $\sigma[\![\neg F]\!] \triangleq \neg\sigma[\![F]\!]$
3. $\sigma[\![F \Rightarrow G]\!] \triangleq \sigma[\![F]\!] \Rightarrow \sigma[\![G]\!]$
4. $\Box F$ (« Toujours F ») :

   $$\langle s_0, s_1, \ldots\rangle[\![\Box F]\!] \triangleq \forall n \in \mathbb{N} : \langle s_n, s_{n+1}, \ldots\rangle[\![F]\!]$$

   $\langle s_0, s_1, \ldots\rangle[\![\Box F]\!]$ exprime que F est vraie pour tous les suffixes du comportement $\langle s_0, s_1, \ldots\rangle$.
5. $\Diamond F$ (« Fatalement F », définie par $\Diamond F \triangleq \neg\Box\neg F$) :

   $$\langle s_0, s_1, \ldots\rangle[\![\Diamond F]\!] \triangleq \exists n \in \mathbb{N} : \langle s_n, s_{n+1}, \ldots\rangle[\![F]\!]$$

   Le comportement $\langle s_0, s_1, \ldots\rangle$ satisfait $\Diamond F$, si, et seulement si, F est vérifiée par un suffixe de ce comportement.
6. $\Box\Diamond F$ (« Infiniment souvent F ») :

   $$\langle s_0, s_1, \ldots\rangle[\![\Box\Diamond F]\!] \triangleq \forall n \in \mathbb{N} : \exists m \in \mathbb{N} : \langle s_{n+m}, s_{n+m+1}, \ldots\rangle[\![F]\!]$$

   Le comportement $\langle s_0, s_1, \ldots\rangle$ satisfait $\Box\Diamond F$ si et seulement si F est par vérifiée une infinité de préfixes du comportement $\langle s_0, s_1, \ldots\rangle$.
7. Action A :
   (a) Cas de $[\![A]\!]$. Le comportement σ satisfait $[\![A]\!]$, si et seulement si la première paire d'états de σ est un *pas* A : $\langle s_0, s_1, \ldots\rangle[\![A]\!] \triangleq s_0[\![A]\!]s_1$.
   (b) Cas de $[\![\Box A]\!]$ *(cas particulier de 4 pour les actions)* :

   $$\langle s_0, s_1, \ldots\rangle[\![A]\!] \triangleq \forall n \in \mathbb{N} : s_n[\![A]\!]s_{n+1}$$

Le comportement σ satisfait $[\![\Box A]\!]$, si et seulement si chaque paire d'états $\langle s_n, s_{n+1}\rangle$ de σ est un *pas* A.

8. Prédicat P :
   (a) Cas de $[\![P]\!]$. Le comportement σ satisfait $[\![A]\!]$, si et seulement si le premier état de σ satisfait P :
   $$\langle s_0, s_1, \ldots\rangle[\![P]\!] \triangleq s_0[\![P]\!]$$
   (b) Cas de $[\![\Box P]\!]$ *(cas particulier de 4 pour les prédicats)* :
   $$\langle s_0, s_1, \ldots\rangle[\![\Box P]\!] \triangleq \forall n \in \mathbb{N} : s_n[\![P]\!]$$
   Le comportement σ satisfait $[\![\Box P]\!]$, si et seulement si le prédicat P est vrai pour tous les états $s_n \in \sigma$.

Une formule temporelle F est *valide* (notée $\models F$), si, et seulement si, elle est satisfaite par tous les comportements σ possibles du système étudié :

$$\models F \triangleq \forall \sigma \in \mathsf{St}^{\infty} : \sigma[\![F]\!]$$

où St^{∞} représente l'ensemble de tous les comportements du système étudié.

3.3.2 Propriétés exprimables en TLA

TLA nous permet aussi d'exprimer et de vérifier les propriétés d'un système, telles que les propriétés de sûreté, de vivacité et aussi de définir des hypothèses d'équité sur les comportements du système. Les formules TLA permettent de caractériser des ensembles de comportements satisfaisant les dites formules. Nous présentons dans cette section les propriétés exprimables et vérifiables en TLA :

— Des propriétés de sûreté de la forme
$\Box P$, qui exprime que le prédicat P est toujours vrai. Une propriété de sûreté exprime le fait que *« quelque chose de mauvais n'arrivera jamais »*.

— Des propriétés de vivacité exprimant le fait que *« quelque chose de bien / un événement heureux arrivera un jour »*. Un exemple de propriété de vivacité pour un programme p est $\Diamond termine$, signifiant que le programme p terminera fatalement. De telles propriétés seront donc vérifiées à partir d'un certain état, lors de l'exécution du système. La vivacité est exprimée dans TLA en fonction de l'*équité* (fairness).

— L'équité définit des contraintes sur les comportements infinis d'un système. Nous distinguons deux types d'équité :
   — *L'équité faible*, définie en TLA comme suit :
   $WF_x(A) \triangleq \Diamond\Box \textsc{Enabled}\langle A\rangle_x \Rightarrow \Box\Diamond\langle A\rangle_x$ et signifie : si une action A est toujours activable à partir d'un état, alors elle va être activée infiniment souvent.
   — *L'équité forte*, définie en TLA comme suit :
   $SF_x(A) \triangleq \Box\Diamond \textsc{Enabled}\langle A\rangle_x \Rightarrow \Box\Diamond\langle A\rangle_x$ et signifie : si une action A est activable infiniment souvent, alors elle va être activée infiniment souvent.

3.3.3 Spécification TLA d'un système

Une spécification TLA d'un système modélise l'ensemble des exécutions possibles de ce dernier. Il s'agit d'une formule définie par la forme suivante :

$$Init \wedge \Box[\mathsf{NEXT}]_x \wedge L$$

— $Init$ est un prédicat contraignant les états initiaux du système étudié.
— NEXT est une relation sur les paires d'états successifs. $\Box[\mathsf{NEXT}]_x$ signifie que tout couple d'états successifs satisfait NEXT ou ne change pas les variables x du modèle (bégaiement).
— L exprime des conditions d'équité sur les comportements infinis du système étudié (voir la sous-section « Équité » de la section précédente) : des actions doivent être exécutées si elles sont activables à partir d'un certain moment. Il s'agit d'une conjonction d'hypothèses d'équité fortes et d'équité faibles, de type $WF_x(F_1)$ et $SF_x(F_2)$, où F_1 et F_2 sont des combinaisons d'actions.

Nous remarquons qu'en TLA, aucune distinction n'est faite entre une spécification et une propriété : tout est formule.

3.3.4 L'opérateur « *leads to* »

Nous nous intéressons particulièrement à des formules temporelles de vivacité définies à l'aide de l'opérateur *« leads to »/« conduit à »* ($\rightsquigarrow$). En effet, notre but est de décrire les comportements d'un système à l'aide de telles formules, de type $P \rightsquigarrow Q$ et d'utiliser les règles (transitivité, disjonction, règles de combinaison, etc.) qui y sont reliées.

Définition de $P \rightsquigarrow Q$. La formule $P \rightsquigarrow Q$ est définie de la manière suivante : $P \rightsquigarrow Q \triangleq \Box(P \Rightarrow \Diamond Q)$. Cette formule signifie qu'à chaque fois que la propriété P est vérifiée, la propriété Q est vérifiée ou elle le sera *fatalement* dans le futur.

Règles de preuves. Nous considérons maintenant quelques règles de preuves [16] permettant de démontrer ces propriétés définies à l'aide de l'opérateur *« leads to »* ($\rightsquigarrow$) :

— **LATTICE.**

$$\frac{\begin{array}{l}\succ \text{ est un ordre partiel bien fondé sur un ensemble } S \ (S, \succ) \\ F \wedge c \in S \Rightarrow (H_c \rightsquigarrow (G \vee \exists d \in S \cdot (c \succ d) \wedge H_d))\end{array}}{F \Rightarrow ((\exists c \in S \cdot H_c) \rightsquigarrow G)}$$

LATTICE utilise les structures bien fondées : La première hypothèse exprime que $(S, \succ)$ est une relation binaire telle qu'il ne va pas exister une chaîne descendante infinie $x_1 \succ x_2, \ldots$ d'éléments $x_i \in S$. La seconde hypothèse requiert qu'une occurrence de H, pour une valeur $c \in S$, conduit soit à G, soit à une nouvelle occurrence de H, pour une valeur $d \in S$, strictement inférieure à c. Puisque la première hypothèse nous assure qu'aucune chaîne descendante infinie n'existe dans S, alors G sera fatalement vérifiée.

— **WF1.**

$$\frac{\begin{array}{l}P \wedge [N]_x \Rightarrow (P' \vee Q') \\ P \wedge \langle N \wedge A \rangle_x \Rightarrow Q' \\ P \Rightarrow \text{ENABLED} \langle A \rangle_x\end{array}}{\Box[N]_x \wedge WF_x(A) \Rightarrow (P \rightsquigarrow Q)}$$

Dans cette règle WF1, P et Q sont des prédicats d'état, $[N]_x$ définit la relation entre deux états consécutifs du système étudié, chaque pas est considéré comme un pas $[N]_x$ et A est une action sur laquelle une hypothèse d'équité faible est posée ($WF_x(A)$). La première hypothèse exprime que le successeur d'un état satisfaisant P, satisfait soit P, soit Q. La seconde hypothèse exprime que le successeur d'un état satisfaisant P, après l'activation de l'action $\langle A \rangle_x$, doit satisfaire Q et la troisième hypothèse exprime que lorsque P est vrai, alors l'action $\langle A \rangle_x$ est activable. La conclusion exprime que, puisque chaque pas (dont A) est considéré comme un pas $[N]_x$ et qu'une hypothèse d'équité faible est posée sur A ($WF_x(A)$), nous pouvons déduire la propriété $P \rightsquigarrow Q$.

— **SF1.**

$$\frac{\begin{array}{l}P \wedge [N]_x \Rightarrow (P' \vee Q') \\ P \wedge \langle N \wedge A \rangle_x \Rightarrow Q' \\ \Box P \wedge \Box[N]_x \wedge \Box F \Rightarrow \Diamond \text{ENABLED} \langle A \rangle_x\end{array}}{\Box[N]_x \wedge SF_x(A) \wedge \Box F \Rightarrow (P \rightsquigarrow Q)}$$

Similairement à la règle WF1, P et Q sont des prédicats d'état, $[N]_x$ définit la relation entre deux états consécutifs du système étudié, chaque pas (dont A) est considéré comme un pas $[N]_x$. Dans la règle SF1, une hypothèse d'équité forte est posée sur l'action A ($SF_x(A)$) et F est une formule temporelle qui peut être requise et exprimer, par exemple, soit des conditions d'équités en plus,

soit des propriétés *« leads to »*, soit des invariants, etc. Les deux premières hypothèses de cette règle sont les mêmes que celles de WF1, déjà expliquées précédemment. La troisième hypothèse assure que si P est toujours vrai et que des pas $[N]_x$ sont activables, alors l'action $\langle A \rangle_x$ est *fatalement* activable. La conclusion exprime que, puisque chaque pas est considéré comme un pas $[N]_x$ et qu'une hypothèse d'équité forte est posée sur A ($SF_x(A)$), nous pouvons déduire la propriété $P \rightsquigarrow Q$.

— **Autres règles de vérification.** Ces règles de vérification sont utilisées pour combiner, décomposer, etc, des propriétés définies à l'aide de l'opérateur *leads to* ($\rightsquigarrow$).

— Transitivité (trans) :

$$\frac{P \rightsquigarrow R \qquad R \rightsquigarrow Q}{P \rightsquigarrow Q} \ (trans)$$

L'opérateur *leads to* est transitif : si P conduit à R et R conduit à Q, alors nous pouvons déduire que P conduit à Q.

— Disjonction (disj) :

$$\frac{P \rightsquigarrow Q \qquad R \rightsquigarrow Q}{(P \vee R) \rightsquigarrow Q} \ (disj)$$

Les deux prémisses de la règle de disjonction sont les suivantes : P conduit à Q et R conduit à Q. Nous pouvons par conséquent déduire qu'une occurrence soit de P, soit de R conduit à Q.

— Déduction (dedu) :

$$\frac{P \Rightarrow Q}{P \rightsquigarrow Q} \ (dedu)$$

La règle de déduction nous permet de considérer l'implication comme une forme de *« leads to »* : si P implique Q, alors nous pouvons déduire que P conduit à Q.

— Simplification (simpl) :

$$\frac{(P \wedge I) \rightsquigarrow Q \qquad Inv(I)}{P \rightsquigarrow Q} \ (simpl)$$

La règle de simplification nous permet de considérer que si $(P \wedge I)$ conduit à Q, et que I est invariant, alors nous pouvons déduire que P conduit à Q.

— Quantification existentielle (exists) :

$$\frac{P \rightsquigarrow Q}{(\exists x : P(x)) \rightsquigarrow (\exists x : Q(x))} \ (exists)$$

Cette règle exprime que si P conduit à Q, alors $P(x)$, où x est une valeur des variables d'états satisfaisant P, conduit à $Q(x)$, où x est une valeur des variables d'états satisfaisant Q.

3.3.5 Outils pour TLA

Des outils permettant l'édition et la vérification de spécifications TLA existent, notamment :

— La plateforme « TLA Toolbox » [15] est un environnement de développement intégré dédié à TLA, basé sur Eclipse, et permettant d'éditer et de vérifier des spécifications TLA. La plateforme inclue, pour la vérification, le système de preuves « TLAPS » [9] et le model-checker « TLC » [24].

— « TLC » [24] est un model-checker prenant en compte un sous-ensemble du langage TLA et permettant de détecter les erreurs dans les spécifications TLA : il permet de vérifier des propriétés de sûreté et de vivacité d'une spécification, par exploration de l'espace d'états de cette dernière. En cas d'erreur, TLC donne un contre-exemple sous la forme d'une trace montrant l'erreur. TLC est un outil intéressant : il est capable d'exploiter les ressources d'un environnement réparti (multi-processeurs/machines), cependant, il reste sujet à des problèmes d'explosions combinatoires des états, de mémoire [19].

— « TLAPS » [9] est un système (de preuves) permettant la preuve interactive d'une spécification TLA. TLAPS parcourt les spécifications TLA, génère ensuite des obligations de preuves à décharger et fait appel à différents prouveurs et/ou assistants de preuves (Zenon, Isabelle, Solveurs SMT, etc) [9, 20, 12] pour vérifier ces obligations de preuves. TLAPS permet, comme la plateforme Rodin et ses prouveurs pour Event-B, de vérifier des propriétés de sûreté/invariance de spécifications TLA. Cependant, il ne permet pas encore de raisonner sur des propriétés temporelles (vivacité) [1, 12]. Ayant choisi Event-B comme méthode principale de développement formel, que nous guidons à l'aide de propriétés temporelles exprimées en TLA, nous avons préféré utiliser pour les travaux effectués durant cette thèse la plateforme Rodin, mais suivant l'évolution du système TLAPS, concernant notamment les propriétés temporelles, nous n'excluons pas de l'utiliser conjointement avec Rodin pour nos travaux futurs.

Dans cette section, nous avons introduit brièvement TLA. Nous avons surtout expliqué les concepts relatifs à TLA que nous utiliserons dans notre méthodologie d'analyse des algorithmes répartis, pour prendre en compte les propriétés temporelles telles que la vivacité, la fatalité, l'équité. La section suivante présente nos conclusions pour ce chapitre.

3.4 Conclusion

Nous proposons une méthodologie basée sur l'application du raffinement et les propriétés de vivacité pour la modélisation d'un algorithme réparti [7, 6, 8]. Notre objectif est de guider le développement formel par raffinement à l'aide de propriétés de vivacité : chaque modèle produit à chaque étape d'un développement par raffinement est caractérisé par un ensemble de propriétés de vivacité, et des règles de preuve et d'inférence nous permettent d'établir des liens entre des ensembles de propriétés de vivacité caractérisant des niveaux de raffinement différents.

Pour appliquer notre méthodologie, nous avons choisi la méthode Event-B et la logique temporelle TLA. Event-B nous fournit le raffinement de modèles et des outils nous permettant de démontrer mécaniquement la consistance et la correction du raffinement, ainsi que la satisfaction par un algorithme étudié de ses propriétés de sûreté/d'invariance. Pour étendre la sémantique d'Event-B, prendre en compte les traces d'exécution d'un modèle et pouvoir ainsi raisonner sur les propriétés temporelles telles que la vivacité et d'équité, nous proposons l'utilisation de la logique TLA, qui fournit un cadre solide et puissant de spécification et de vérification des propriétés temporelles.

Bibliographie

[1] M. Abadi, F. McSherry, D. G. Murray, and T. L. Rodeheffer. Formal analysis of a distributed algorithm for tracking progress. In D. Beyer and M. Boreale, editors, *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, volume 7892 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2013.

[2] J.-R. Abrial. *The B-book : Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[3] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, 2010.

[4] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for event-b. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering*, ICFEM'06, pages 588–605, Berlin, Heidelberg, 2006. Springer-Verlag.

[5] J.-R. Abrial and T. S. Hoang. Using design patterns in formal methods : An event-b approach. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigün, editors, *ICTAC*, volume 5160 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2008.

[6] M. B. Andriamiarina, D. Méry, and N. K. Singh. Revisiting snapshot algorithms by refinement-based techniques. In H. Shen, Y. Sang, Y. Li, D. Qian, and A. Y. Zomaya, editors, *13th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2012, Beijing, China, December 14-16, 2012*, pages 343–349. IEEE, 2012.

[7] M. B. Andriamiarina, D. Méry, and N. K. Singh. Integrating proved state-based models for constructing correct distributed algorithms. In E. B. Johnsen and L. Petre, editors, *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 2013.

[8] M. B. Andriamiarina, D. Méry, and N. K. Singh. Analysis of self-* and p2p systems using refinement. In Y. Aït-Ameur and K.-D. Schewe, editors, *ABZ*, volume 8477 of *Lecture Notes in Computer Science*, pages 117–123. Springer, 2014.

[9] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the tla+ proof system. In J. Giesl and R. Hähnle, editors, *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 142–148. Springer Berlin Heidelberg, 2010.

[10] ClearSy. Atelier B. `http://www.atelierb.eu/`.

[11] ClearSy, Aix-en-Provence (F). *B4FREE*, 2004. http ://www.b4free.com.

[12] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA+ Proofs. In Dimitra Giannakopoulou and Dominique Méry, editor, *18th International Symposium On Formal Methods - FM 2012*, volume 7436, pages 147–154, Paris, France, 2012. Springer. The original publication is available at www.springerlink.com.

[13] D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. Smt solvers for rodin. In J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 194–207. Springer Berlin Heidelberg, 2012.

[14] T. S. Hoang, A. Furst, and J.-R. Abrial. Event-b patterns and their tool support. *Software Engineering and Formal Methods, International Conference on*, 0 :210–219, 2009.

[15] L. Lamport. The TLA Toolbox. `http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html`.

[16] L. Lamport. A temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3) :872–923, May 1994.

[17] G. T. Leavens, J.-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In *Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006)*, pages 221–235. ACM, Oct. 2006.

[18] M. Leuschel and M. J. Butler. Prob : an automated analysis toolset for the b method. *STTT*, 10(2) :185–203, 2008.

[19] T. Lu, S. Merz, and C. Weidenbach. Model Checking the Pastry Routing Protocol. In Jens Bendisposto and Michael Leuschel and Markus Roggenbach, editor, *10th International Workshop Automated Verification of Critical Systems*, 10th International Workshop Automated Verification of Critical Systems, pages 19–21, Düsseldorf, Germany, Sept. 2010. Universität Düsseldorf. short communication.

[20] S. Merz and H. Vanzetto. Harnessing SMT Solvers for TLA+ Proofs. In Gerald Lüttgen and Stephan Merz, editor, *12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)*, volume 53, Bamberg, Germany, Dec. 2012. EASST.

[21] Project RODIN. Rigorous open development environment for complex systems. http ://www.eventb.org/, 2004-2010.

[22] Rodin Platform. Rodin User's Handbook v2.7. `http://handbook.event-b.org/current/html/atelier_b_provers.html`, 2012.

[23] R. Silva. Refactoring Framework. `http://rodin-b-sharp.sourceforge.net/`, 2009.

[24] Y. Yu, P. Manolios, and L. Lamport. Model checking tla+ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer Berlin Heidelberg, 1999.

4

Modélisation et vérification par raffinement des algorithmes répartis

Sommaire

4.1 Introduction

Ce chapitre présente les techniques utilisées pour la modélisation et la vérification des algorithmes répartis. L'analyse de ces algorithmes et de leurs systèmes répartis supports présentent des difficultés, tels que l'absence de connaissance globale des processus composant les systèmes, la mobilité de ces derniers, leur hétérogénéité, la synchronisation des processus durant les algorithmes, etc.

Nous proposons une méthodologie [10, 9] de développement formel des algorithmes répartis, basée sur le raffinement de modèles, fourni par la méthode Event-B [12, 3, 2] et l'utilisation de propriétés temporelles, exprimées à l'aide de la logique TLA [23] : il s'agit de guider chaque étape du raffinement par des propriétés *leads to*. Dans le cas des algorithmes répartis, les services et fonctionnalités offerts par un algorithme sont caractérisés par des propriétés *leads to*, puis exprimées à l'aide d'événements (sous hypothèses d'équité), et enrichies au fur et à mesure du raffinement, grâce à des règles d'inférence.

4.2 Modélisation des algorithmes répartis

Nous présentons dans cette section un cadre sémantique, inspiré de la logique TLA et capable de prendre en compte les propriétés temporelles des systèmes étudiés (vivacité, fatalité, équité) : il s'agit des modèles relationnels, proposés par Méry et al [15, 27]. Ce cadre sémantique va nous permettre de considérer un modèle Event-B en tant que *système de transitions*, mettant en jeu les états possibles du modèle et définissant les transitions entre ces états. Nous pourrons ainsi raisonner sur les comportements du modèle (les *traces*) et interpréter les aspects temporels du modèle.

4.2.1 Algorithmes répartis

Nous définissons dans cette section les concepts et notions exploités durant ce chapitre, principalement la notion d'*algorithme réparti*.

Définition 4.2.1. Un **algorithme réparti** [31] pour un ensemble $\mathbb{P} = \{p_1, ..., p_N\}$ de processus est une collection d'algorithmes locaux, c'est-à-dire d'algorithmes dont les variables sont locales, un pour chaque processus de $\mathbb{P}$. Les communications entre les processus sont opérées par messages.

Nous caractérisons un algorithme réparti par les *services* qu'il doit fournir.

Définition 4.2.2. Un **service** [30] est une fonctionnalité (e.g. exclusion mutuelle, routage) fournie par un algorithme et définissant quelles actions l'algorithme peut réaliser pour le compte des utilisateurs. L'accès à un service par un utilisateur se fait par l'intermédiaire d'une interface et est régi par des contraintes, règles et politiques spécifiées dans la description de l'algorithme réparti fournissant le service.

Nous nous intéressons notamment à identifier les phases [10] algorithmiques permettant de fournir un service.

Définition 4.2.3. Une **phase** est une étape algorithmique issue de la décomposition d'un service en étapes plus simples à analyser et à résoudre. Les phases composant un service sont synchronisées et coordonnées les unes par rapport aux autres.

Nous illustrons maintenant ces définitions par quelques exemples :

— Nous avons les algorithmes qui nous fournissent comme service l'exclusion mutuelle [21] : un seul processus se trouve dans une section critique. Ce service peut être décomposé en trois phases : **(1)** une phase de requête d'entrée en section critique par un processus, **(2)** l'entrée en section critique du processus et **(3)** sa sortie de la section critique.
— Le protocole DSR [26], dont le service fournit est le routage. Nous identifions deux phases pour le service de routage : **(1)** une première phase de découverte de la route entre une source et un destinataire, **(2)** une phase de maintenance de la route (en cas de changement de topologie du réseau, etc).

4.2.2 Modélisation relationnelle d'un algorithme réparti

Nous caractérisons un modèle abstrait relationnel $\mathcal{AM}$ d'un système (ici un algorithme réparti $\mathcal{A}$) par un espace d'états Σ, un ensemble d'états initiaux $Init_p$, un ensemble d'états terminaux $Term_p$ (pouvant être vide si le système ne se termine pas) et une relation binaire NEXT sur Σ. Σ est défini comme suit : $\Sigma \mathrel{\widehat{=}} \mathsf{Var} \to \mathsf{Val}$, où Var est l'ensemble des variables et Val l'ensemble des valeurs possibles des variables. Soient s et s' deux états de l'ensemble Σ ; une transition de l'état s à s' est notée : $s \xrightarrow[\mathsf{NEXT}]{} s'$. Soit x les variables du système ; la valeur de x est notée $s[\![x]\!]$ à l'état s, et $s'[\![x]\!]$ à l'état s'.

Nous proposons, à partir de ces éléments, une simplification de la référence aux états et aux valeurs des variables lors de ces états : si nous considérons la transition $s \xrightarrow[\mathsf{NEXT}]{} s'$, la valeur $s[\![x]\!]$ de x, à l'état s, sera notée x et la valeur $s'[\![x]\!]$, à l'état suivant s', sera notée x'. Nous introduisons ainsi la notion de variable primée de TLA [23] : x est la valeur avant la transition considérée et x' la valeur après.

Nous pouvons maintenant simplement exprimer la définition des état initiaux à l'aide d'un prédicat INIT caractérisant les valeurs des variables x à l'état initial et nous pouvons décrire les transitions entre états en utilisant les relations entre les variables primées et non-primées : $s \xrightarrow[\mathsf{NEXT}]{} s'$ s'écrira ainsi comme une relation $\mathsf{NEXT}(x, x')$. L'ensemble des états Σ est une fonction allant de l'ensemble Var des variables à l'ensemble Val des valeurs. Nous identifions maintenant cet ensemble d'états à l'ensemble des valeurs possibles des variables x.

A partir de ces éléments, nous pouvons maintenant proposer une définition plus générale d'un modèle relationnel $\mathcal{M}$ d'un algorithme réparti $\mathcal{A}$.

Définition 4.2.4. Un modèle relationnel $\mathcal{M}$ d'un algorithme réparti $\mathcal{A}$ est une structure [23] :

$$(Th(s,c), x, \mathsf{Val}, \mathsf{INIT}(x), \{r_0, \ldots, r_n\})$$

— $Th(s,c)$ est une théorie définissant les ensembles s, les constantes c et les propriétés $P(s,c)$ statiques de ces éléments.
— x est la liste des variables de $\mathcal{M}$ modélisant l'état global du système réparti support de $\mathcal{A}$ et/ou les états locaux des processus $P_1, \ldots, P_n$ composant ce système.
— Val est l'ensemble des valeurs possibles de x.
— $\mathsf{INIT}(x)$ est le prédicat définissant les conditions initiales du modèle $\mathcal{M}$.
— $\{r_0, \ldots, r_n\}$ est un ensemble fini de relations reliant les valeurs des variables x avant (x) et après (x'). Il s'agit de l'ensemble des relations avant-après (before-after) $BA(e)(x, x')$ définies pour les événements e de $\mathcal{M}$ et l'un des événements correspond à *skip* : nous supposons que r_0 est la relation $Id[\mathsf{Val}]$ (identité sur Val) correspondant à $BA(skip)(x, x')$.
— La relation NEXT associée au modèle $\mathcal{M}$ est la relation entre les valeurs des variables avant (x) et après (x') (donc entre deux états consécutifs du modèle), et est définie par : $\mathsf{NEXT} \mathrel{\widehat{=}} r_0 \vee \ldots \vee r_n$.

Pour plus de précision et pour éviter les confusions, nous définissons les valeurs suivantes pour une variable x :

— x est la valeur courante de la variable x.
— x' est la valeur suivante de la variable x.

Le modèle relationnel $\mathcal{M}$ de l'algorithme réparti $\mathcal{A}$ va nous permettre d'introduire les *traces* des exécutions possibles du modèle, et de les analyser pour l'étude de propriétés temporelles, telles que la vivacité et l'équité.

4.2.3 Expression des propriétés de sûreté

Une propriété de sûreté exprime que *rien de mauvais ne peut arriver* [23]. Quelques exemples de propriétés de sûreté pour des algorithmes répartis sont les suivants :

— Pour les algorithmes d'exclusion mutuelle, un exemple de propriété de sûreté est *« un seul processus au plus utilise la ressource partagée/la section critique »*.

— Pour le protocole de l'élection du leader [4] dans un graphe acyclique et connexe, un exemple de propriété de sûreté possible est *« l'arbre de recouvrement du graphe est un sous-ensemble de ce dernier »*.

Nous donnons ici une définition d'une propriété de sûreté [15, 27, 31] pour un algorithme réparti $\mathcal{A}$.

Définition 4.2.5. Soit $(Th(s,c), x, \mathsf{Val}, \mathsf{INIT}(x), \{r_0, \ldots, r_n\})$ un modèle relationnel d'un algorithme réparti $\mathcal{A}$. Une propriété P est une propriété de sûreté pour l'algorithme réparti $\mathcal{A}$, si $\forall x, y \cdot x, y \in \mathsf{Val} \wedge \mathsf{INIT}(x) \wedge \mathsf{NEXT}^*(x,y) \Rightarrow P(y)$

L'expression NEXT^* désigne la fermeture réflexive transitive de la relation NEXT. Nous reprenons les deux exemples de propriété de sûreté citées précédemment en les exprimant selon la définition 4.2.5 donnée ci-dessus :

Exemple 4.2.1. Les algorithmes d'exclusion mutuelle [21]. Nous présentons ici un modèle très abstrait d'un algorithme d'exclusion mutuelle.

— Le modèle Event-B de cet algorithme définit un ensemble de processus P, non vide.

— Le modèle possède deux variables : *own* qui contient les processus utilisant la section critique, *req* qui contient les processus demandant l'accès à la section critique.

— Le modèle contient trois événements :

— request : un processus p demande l'accès à section critique. La variable *req* est mise à jour, on y ajoute le processus p.

```
EVENT request ≙
  ANY
    p
  WHERE
    p ∈ P
    p ∉ req
  THEN
    req := req ∪ {p}
  END
```

— take : un processus p accède à la section critique, s'il a effectué une requête ($p \in req$) et si elle est libre ($own = \varnothing$).

```
EVENT take ≙
  ANY
    p
  WHERE
    p ∈ P
    p ∈ req
    own = ∅
  THEN
    req := req \ {p}
    own := own ∪ {p}
  END
```

— drop : un processus p utilisant la section critique la libère.

```
EVENT drop ≙
  ANY
    p
  WHERE
    p ∈ P
    p ∈ own
  THEN
    own := own \ {p}
  END
```

— Nous posons :

$$\text{NEXT}(own, req, own', req') \;\widehat{=}\; \begin{array}{l} \vee(own' = own \wedge req = req') \\ \vee BA(\mathsf{request})(own, req, own', req') \\ \vee BA(\mathsf{take})(own, req, own', req') \\ \vee BA(\mathsf{drop})(own, req, own', req') \end{array}$$

— $\text{INIT}(own, req) \;\widehat{=}\; (own = \varnothing \wedge req = \varnothing)$

— La propriété de sûreté *« un seul processus au plus utilise la ressource partagée/la section critique »* peut s'exprimer de la manière suivante : $(own = \varnothing) \vee (\exists p \cdot p \in P \wedge own = \{p\})$. En utilisant la définition 4.2.5, nous avons :

$$\forall \begin{pmatrix} own, \\ req, \\ \overline{own}, \\ \overline{req} \end{pmatrix} \cdot \begin{pmatrix} \wedge \text{INIT} \begin{pmatrix} own, \\ req \end{pmatrix} \\ \\ \wedge \text{NEXT}^* \begin{pmatrix} own, \\ req, \\ \overline{own}, \\ \overline{req} \end{pmatrix} \end{pmatrix} \Rightarrow \begin{pmatrix} \vee(\overline{own} = \varnothing) \\ \vee(\exists p \cdot p \in P \wedge \overline{own} = \{p\}) \end{pmatrix}$$

Exemple 4.2.2. Le protocole de l'élection du leader (premier raffinement présenté dans [4]).

— Le modèle Event-B de cet algorithme définit un ensemble de nœuds ND, ainsi qu'un graphe g, acyclique et connexe.

— Le modèle possède trois variables : rt qui représente le nœud leader de l'arbre recouvrant de g, sp qui représente l'arbre de recouvrement de g et tr, variable intermédiaire utilisée lors de la construction progressive de l'arbre de recouvrement de g.

— Le modèle contient deux événements :

— progress : l'arbre de recouvrement de g est construit pas-à-pas, par l'établissement de liens de parenté entre nœuds voisins (un nœud y devient le parent d'un nœud x, s'ils sont voisins, si le nœud y n'a pas encore de parent et que le nœud x est parent de tous ses voisins, sauf y). L'arbre de recouvrement ainsi construit est stocké dans la variable tr. La construction s'arrête dès qu'une racine x de tr est découverte : le nœud x est le parent de tous ses voisins.

— election : l'événement est déclenché lorsqu'une racine x de tr est découverte. La variable rt prend alors la valeur x et sp prend celle de tr.

— Nous posons :

$$\text{NEXT}(rt, sp, tr, rt', sp', tr') \;\widehat{=}\; \begin{array}{l} \vee(rt' = rt \wedge sp' = sp \wedge tr' = tr) \\ \vee BA(\mathsf{progress}) \begin{pmatrix} rt, sp, tr, \\ rt', sp', tr' \end{pmatrix} \\ \vee BA(\mathsf{election}) \begin{pmatrix} rt, sp, tr, \\ rt', sp', tr' \end{pmatrix} \end{array}$$

— $\text{INIT}(rt, sp, tr) \;\widehat{=}\; (rt \in ND \wedge sp = \varnothing \wedge tr = \varnothing)$

— La propriété de sûreté *« l'arbre de recouvrement du graphe est un sous-ensemble de ce dernier »* peut s'exprimer de la manière suivante : $sp \subseteq g$. Nous considérons deux cas pour $\overline{sp}$: **(1)** $\overline{sp} = \varnothing$, si le calcul de l'arbre de recouvrement est encore en cours ou **(2)** $\overline{sp} = \overline{tr}$, c'est-à-dire si un arbre de recouvrement du graphe g a été calculé dans la variable tr.
En utilisant la définition 4.2.5, nous avons :

$$\forall \begin{pmatrix} rt, sp, tr, \\ \overline{rt}, \overline{sp}, \overline{tr} \end{pmatrix} \cdot \begin{pmatrix} \text{INIT} \begin{pmatrix} rt, \\ sp, \\ tr \end{pmatrix} \wedge \text{NEXT}^* \begin{pmatrix} rt, sp, tr, \\ \overline{rt}, \overline{sp}, \overline{tr} \end{pmatrix} \end{pmatrix} \Rightarrow (\overline{sp} \subseteq g)$$

La démonstration d'une propriété de sûreté se fait soit en vérifiant la propriété pour toutes les valeurs possibles dans Val (à condition que Val soit un ensemble fini), soit en utilisant un principe d'induction [15, 27].

Propriété 4.2.1. (Principe d'induction) [15, 27]
Soit $(Th(s, c), x, \mathsf{Val}, \text{INIT}(x), \{r_0, \ldots, r_n\})$ un modèle relationnel $\mathcal{M}$ d'un algorithme réparti $\mathcal{A}$. Une propriété $P(x)$ est une propriété de sûreté pour l'algorithme $\mathcal{A}$, si et seulement si, il existe une propriété d'état $I(x)$, telle que :

$$\forall x, x' \in \mathsf{Val} \ : \ \begin{cases} (1)\ \mathsf{INIT}(x) \Rightarrow I(x) \\ (2)\ I(x) \Rightarrow P(x) \\ (3)\ I(x) \wedge \mathsf{NEXT}(x, x') \Rightarrow I(x') \end{cases}$$

La propriété $I(x)$ est appelée un invariant inductif de l'algorithme $\mathcal{A}$ et est une propriété de sûreté particulière plus forte que les autres propriétés de sûreté [15, 27]. Nous justifions maintenant ce principe d'induction.

PREUVE [15, 27] :

— SUPPOSONS QUE : il existe une propriété $I(x)$ telle que :

$$\forall x, x' \in \mathsf{Val} \ : \ \begin{cases} (1)\ \mathsf{INIT}(x) \Rightarrow I(x) \\ (2)\ I(x) \Rightarrow P(x) \\ (3)\ I(x) \wedge \mathsf{NEXT}(x, x') \Rightarrow I(x') \end{cases}$$

PROUVONS QUE : $P(x)$ est une propriété de sûreté pour pour l'algorithme $\mathcal{A}$.

PREUVE : Soient x et $x' \in \mathsf{Val}$ tels que $\mathsf{INIT}(x) \wedge \mathsf{NEXT}^*(x, x')$. On peut construire une suite telle que : $(x = x_0) \xrightarrow[\mathsf{NEXT}]{} x_1 \xrightarrow[\mathsf{NEXT}]{} x_2 \xrightarrow[\mathsf{NEXT}]{} \ldots \xrightarrow[\mathsf{NEXT}]{} (x_i = x')$. L'hypothèse (1) nous permet de déduire $I(x_0)$. L'hypothèse (3) nous permet de déduire $I(x_1)$, $I(x_2)$, $I(x_3)$, ..., $I(x_i)$. En utilisant l'hypothèse (2) pour x', nous en déduisons que x' satisfait P. □

— SUPPOSONS QUE : $\forall x, y \cdot x, y \in \mathsf{Val} \wedge \mathsf{INIT}(x) \wedge \mathsf{NEXT}^*(x, y) \Rightarrow P(y)$

PROUVONS QUE : il existe une propriété $I(x)$ telle que :

$$\forall x, x' \in \mathsf{Val} \ : \ \begin{cases} (1)\ \mathsf{INIT}(x) \Rightarrow I(x) \\ (2)\ I(x) \Rightarrow P(x) \\ (3)\ I(x) \wedge \mathsf{NEXT}(x, x') \Rightarrow I(x') \end{cases}$$

PREUVE : Nous considérons la propriété suivante : $I(x) \ \widehat{=} \ \exists y \in \mathsf{Val} \cdot \mathsf{INIT}(y) \wedge \mathsf{NEXT}^*(y, x)$. $I(x)$ exprime que la valeur x est accessible à partir d'une valeur initiale y. Les trois propriétés sont simples à vérifier pour $I(x)$. $I(x)$ est appelé le plus fort invariant de l'algorithme $\mathcal{A}$. □

— Les deux étapes précédentes permettent de déduire l'équivalence. □

La règle d'induction nous permet de distinguer deux types de propriétés :

— Les propriétés d'invariance, qui nécessitent un pas d'induction.

— Les propriétés de sûreté, qui sont plus générales et qui nécessitent des propriétés d'invariance pour pouvoir être démontrées.

La méthode EVENT-B utilise ces deux types de propriétés : les invariants sont définis dans la clause INVARIANTS et les propriétés de sûreté dans la clause THEOREMS. Nous rappelons ici que toute propriété d'invariance est une propriété de sûreté, mais l'inverse n'est pas vrai.

4.2.3.1 Les propriétés d'invariance et de sûreté en Event-B

Nous rappelons ici qu'un modèle EVENT-B est constitué de deux structures : un contexte et une machine. Le contexte contient les ensembles s et constantes c du modèle, ainsi que les propriétés (axiomes, théorèmes) caractérisant ces ensembles et constantes. Nous notons $C(s, c)$ le contexte de preuve, donné par les propriétés des ensembles s et des constances c. La machine contient les variables x, les événements e et des propriétés sur les variables (invariants, théorèmes).

L'invariant $I(x)$ d'un modèle se présente sous la forme d'une liste de prédicats étiquettés inv_1, inv_2, ..., inv_n et est interprété comme une conjonction. Les conditions initiales doivent satisfaire l'invariant $I(x)$:

$$C(s, c) \wedge Init(x, s, c) \Rightarrow I(x) \ (INV1)$$

L'invariant $I(x)$ doit être préservé par chaque événement e du modèle. La condition de préservation d'un invariant par chaque événement e est la suivante :

$$C(s, c) \wedge I(x) \wedge BA(e)(c, s, x, x') \Rightarrow I(x') \ (INV2)$$

La faisabilité d'un événement e signifie que sous l'hypothèse définie par l'invariant $I(x)$ et par la garde $grd(e)(x)$ de l'événement, il existe toujours x', tel que $BA(e)(c, s, x, x')$. La condition de faisabilité pour chaque événement e du modèle est la suivante :

$$C(s,c) \wedge I(x) \wedge grd(e)(x) \Rightarrow \exists x' \cdot BA(e)(c,s,x,x') \ (FIS)$$

La propriété de sûreté $A(s,c,x)$ d'un modèle se présente aussi sous la forme d'une liste de prédicats étiquettés thm_1, thm_2, ..., thm_n et elle est interprétée comme une conjonction. Elles sont déduites de la preuve que l'invariant $I(x)$ implique la propriété de sûreté $A(s,c,x)$. La condition de vérification de la propriété de sûreté est la suivante :

$$C(s,c) \wedge I(x) \Rightarrow A(s,c,x) \ (THM)$$

Le raffinement est un procédé incrémental permettant d'enrichir et de détailler un modèle formel. Quand un modèle abstrait est raffiné par un modèle concret, ce dernier *simule* le modèle abstrait. Le raffinement préserve les propriétés d'un modèle abstrait : cela signifie que les propriétés de sûreté du modèle abstrait sont aussi des propriétés de sûreté du modèle concret. Les propriétés d'invariance sont préservées de la même manière : le modèle concret préserve les invariants du modèle abstrait.

Pour conclure, les propriétés que la méthode Event-B permet de prouver sont des propriétés de sûreté, exprimées généralement sous la forme d'invariants et/ou de théorèmes dans les modèles. La méthode permet aussi de raisonner sur certaines propriétés de vivacité, par exemple la *terminaison*. Cependant, Event-B ne permet pas de prendre en compte des propriétés de vivacité plus générales, telles que le progrès ou la persistance [17]. Nous proposons donc dans les sections suivantes une extension de la méthode Event-B, pour pouvoir prendre en compte les propriétés de traces et de vivacité.

4.3 Modélisation temporelle d'un algorithme réparti

Le modèle relationnel vu dans les sections précédentes va nous permettre de définir les traces d'exécution pour les modèles Event-B. Nous pourrons ainsi prendre en compte les propriétés de vivacité des modèles Event-B.

4.3.1 Traces d'exécution d'un modèle Event-B

Une trace d'exécution engendrée par un modèle Event-B $\mathcal{M}$ est une suite σ infinie de valeurs que peuvent prendre les variables x du modèle. Chaque élément σ_n de la suite décrit les valeurs des variables d'état x (à un indice n).

Définition 4.3.1. Une trace engendrée par $\mathcal{M}$ est une suite σ infinie de valeurs :

$$\sigma_0\sigma_1\sigma_2\sigma_3 \ldots \sigma_i\sigma_{i+1} \ldots$$

où $\sigma_0 \in \mathsf{INIT}$ et $\forall i \in \mathbb{N} : (\sigma_i, \sigma_{i+1}) \in \mathsf{NEXT}$.

L'ensemble de traces simulant le modèle Event-B $\mathcal{M}$ est par conséquent un sous-ensemble de $\mathbb{N} \rightarrow \mathsf{Val}$, qui est l'ensemble des suites infinies formées à partir de l'ensemble Val des valeurs possibles des variables x de $\mathcal{M}$. L'ensemble de toutes les traces simulant le modèle Event-B $\mathcal{M}$ est noté $tr(\mathcal{M})$.

Nous rappelons qu'un événement du modèle Event-B $\mathcal{M}$ représente l'observation d'une transition du modèle et est une substitution généralisée : un événement est composée d'une garde et d'une action/substitution qui est observée lorsque la garde est vraie. L'*équité* définit des contraintes sur les comportements infinis du modèle $\mathcal{M}$. Elle permet de garantir que certains événements peuvent être observés s'ils sont observables (autorisés) à partir d'un certain moment. Nous allons utiliser des éléments de la logique TLA [23] pour expliquer l'équité. Pour un événement e de $\mathcal{M}$, nous définissons :

- Son observabilité $ENABLED\langle e \rangle_x$ comme suit : $ENABLED\langle e \rangle_x \ \widehat{=} \ \exists y \cdot BA(e)(x,y)$, c'est-à-dire que la garde de e, notée $G(x)$, est vraie et qu'il existe une valeur après y des variables de $\mathcal{M}$, telle que l'action de e, notée $R(x,y)$, est observable.
- son observation (modifiant les valeurs des variables x du modèle $\mathcal{M}$) $\langle e \rangle_x$ de la manière suivante : $\langle e \rangle_x \ \widehat{=} \ BA(e)(x,x') \wedge (x \neq x')$.

Deux types d'équité sont distingués pour un événement e :

— Une équité faible, définie de la manière suivante :

$$WF_x(e) \ \widehat{=} \ \Diamond\Box ENABLED\langle e\rangle_x \Rightarrow \Box\Diamond\langle e\rangle_x$$

Si l'événement e est observable indéfiniment à partir d'un état, alors e est observé infiniment souvent.

— Une équité forte, définie comme suit :

$$SF_x(e) \ \widehat{=} \ \Box\Diamond ENABLED\langle e\rangle_x \Rightarrow \Box\Diamond\langle e\rangle_x$$

Si l'événement e est observable infiniment souvent, alors e est observé infiniment souvent.

Une condition d'équité L contraignant les exécutions du modèle Event-B $\mathcal{M}$ est une conjonction de contraintes d'équité faibles et fortes sur les événements de $\mathcal{M}$, de la forme : $WF_x(e_i) \wedge \ldots \wedge SF_x(e_j)$, où e_i et e_j sont des événements de $\mathcal{M}$.

Des conditions d'équité sur les exécutions d'un modèle Event-B sont nécessaires lors de la vérification de propriétés de vivacité, car elles permettent d'exclure les traces *non-équitables* (ne respectant pas les conditions d'équité), et de ne raisonner que sur celles dites *équitables* (satisfaisant les conditions d'équité).

Pour illustrer les notions de traces équitables et non-équitables, nous utilisons un modèle abstrait d'un algorithme de routage, contenant un ensemble de paquets PCK à envoyer, des variables $x = (sent, lost, got)$, un invariant $I(x) = (sent \subseteq PCK) \wedge (got \cup lost \subseteq sent) \wedge (got \cap lost = \varnothing)$ (l'ensemble des paquets envoyés est un sous-ensemble de PCK, les paquets reçus ou perdus ont été envoyés par leurs sources, et les paquets sont soit reçus ou perdus). Ce modèle contient aussi les événements suivants :

— SENDING modélise l'envoi d'un paquet p.

— RESENDING modélise le renvoi d'un paquet p perdu.

— LOSING modélise la perte d'un paquet p, qui a été envoyé au préalable et n'a été ni perdu ni reçu.

— RECEIVING modélise la réception d'un paquet p, qui a été envoyé au préalable et n'a été ni perdu ni reçu.

```
EVENT SENDING ≙
  ANY
    p
  WHERE
    p ∈ PCK ∧ p ∉ sent
  THEN
    sent := sent ∪ {p}
  END
```

```
EVENT RESENDING ≙
  ANY
    p
  WHERE
    p ∈ PCK ∧ p ∈ lost
  THEN
    lost := lost \ {p}
  END
```

```
EVENT LOSING ≙
  ANY
    p
  WHERE
    p ∈ PCK ∧ p ∈ sent ∧ p ∉ (got ∪ lost)
  THEN
    lost := lost ∪ {p}
  END
```

```
EVENT RECEIVING ≙
  ANY
    p
  WHERE
    p ∈ PCK ∧ p ∈ sent ∧ p ∉ (got ∪ lost)
  THEN
    got := got ∪ {p}
  END
```

et son état initial est défini par : $INIT(x) \ \widehat{=} \ (sent = \varnothing \wedge got = \varnothing \wedge lost = \varnothing)$.

Exemple 4.3.1. Nous supposons que l'hypothèse d'équité L posée sur le modèle abstrait du routage ci-dessus est la suivante : $L \ \widehat{=} \ WF_x(\mathsf{SENDING}) \wedge WF_x(\mathsf{RESENDING}) \wedge SF_x(\mathsf{RECEIVING})$.

Soit $\sigma \ \widehat{=} \ \sigma_0\sigma_1\sigma_2 \ldots \sigma_k\sigma_{k+1} \ldots$ une trace d'exécution du modèle, où :

— $\sigma_0 \ \widehat{=} \ (sent = \varnothing \wedge got = \varnothing \wedge lost = \varnothing)$ ($\sigma_0 \in \mathsf{INIT}$).

— $\sigma_1 \ \widehat{=} \ (p \in sent \wedge got = \varnothing \wedge lost = \varnothing)$ (événement SENDING : un paquet p a été envoyé par sa source).

— $\sigma_2 \ \widehat{=} \ (p \in sent \wedge got = \varnothing \wedge p \in lost)$ (événement LOSING : p est perdu).

— pour tout i, tel que $i \in \mathbb{N}$ et $i \geq 3$, nous avons $\sigma_i = \sigma_1$ et $\sigma_{i+1} = \sigma_2$. Les seuls événements observés sont les événements RESENDING et LOSING.

Une telle trace σ où l'événement RECEIVING n'est jamais observé, alors qu'il est pourtant autorisé infiniment souvent à partir de σ_1 ($p \in sent \wedge got = \varnothing \wedge lost = \varnothing$) et est sous hypothèse d'équité forte, ne respecte pas l'hypothèse d'équité L. L'hypothèse d'équité permet d'exclure de telles traces.

L'observation d'un modèle Event-B peut se faire à travers l'analyse de ses traces d'exécution. Pour mettre en œuvre les notions liées aux traces, à la vivacité et à l'équité vues précédemment, nous présentons,

dans la section suivante, une extension de la méthode Event-B permettant de les prendre en compte.

4.3.2 Cadre temporel pour la méthode Event-B

Nous souhaitons utiliser la logique temporelle pour guider le développement formel par raffinement de modèles Event-B. Ainsi, pour raisonner sur les traces d'exécution et pouvoir prendre en compte les propriétés de vivacité (propriétés de fatalité exprimées à l'aide de l'opérateur « *leads to* » ($\rightsquigarrow$), etc), nous proposons l'extension de l'expressivité et de la sémantique de la méthode Event-B en utilisant des éléments de la logique temporelle TLA [23].

Définition 4.3.2. Le cadre temporel pour un modèle Event-B $\mathcal{M}$ est défini par la spécification TLA [23] $\mathcal{S}pec(\mathcal{M})$ suivante :

$$\mathsf{INIT}(x) \wedge \Box[\mathsf{NEXT}]_x \wedge L$$

où :
- $\mathsf{INIT}(x)$ est le prédicat définissant les conditions initiales du modèle $\mathcal{M}$.
- Nous notons E l'ensemble des événements du modèle $\mathcal{M}$ et nous définissons NEXT comme suit : $\mathsf{NEXT} \equiv \exists e.e \in E \wedge BA(e)(x, x')$. La formule $\Box[\mathsf{NEXT}]_x$ exprime que chaque paire d'états consécutifs satisfait NEXT et les valeurs des variables d'état x du modèle changent ou demeurent les mêmes.
- L est une condition d'équité et exprime des contraintes sur les comportements infinis : L est une conjonction de contraintes d'équités faibles et fortes sur des combinaisons d'événements $e \in E$ de $\mathcal{M}$.

La spécification TLA $\mathcal{S}pec(\mathcal{M})$ décrit tous les comportements autorisés du modèle Event-B $\mathcal{M}$, c'est-à-dire l'ensemble des traces d'exécution équitables (respectant l'hypothèse d'équité L) simulant le modèle Event-B $\mathcal{M}$. Nous donnons par conséquent ici une définition de l'ensemble des traces équitables $tfair(\mathcal{M}, L)$ simulant le modèle Event-B $\mathcal{M}$, sous l'hypothèse d'équité L. Une trace σ, engendrée par le modèle Event-B $\mathcal{M}$, fait partie des traces simulant ce dernier, si, et seulement si, σ satisfait la spécification TLA $\mathcal{S}pec(\mathcal{M})$ du modèle $\mathcal{M}$, que nous notons $\sigma \models \mathcal{S}pec(\mathcal{M}) : \sigma_0 \in \mathsf{INIT}$, et pour tout i, tel que $i \in \mathbb{N}$, $(\sigma_i, \sigma_{i+1}) \in \mathsf{NEXT}$, et σ respecte la condition d'équité L.

Définition 4.3.3. Soit σ une trace d'exécution simulant le modèle $\mathcal{M}$, sous hypothèse d'équité L.
$\sigma \in tfair(\mathcal{M}, L)$ si, et seulement, si $\sigma \models \mathcal{S}pec(\mathcal{M})$

Nous pouvons maintenant définir des propriétés de vivacité (i.e. propriétés utilisant l'opérateur *« leads to »* ($\rightsquigarrow$)) pour le modèle m et raisonner sur ces propriétés. En effet, notre but est de décrire les comportements d'un système à l'aide de telles formules, de type $P \rightsquigarrow Q$. Nous rappelons que la formule $P \rightsquigarrow Q$ est définie par $\Box(P \Rightarrow \Diamond Q)$ et signifie qu'à chaque fois que la propriété P est vérifiée, la propriété Q est vérifiée ou le sera *fatalement* dans le futur.

Nous définissons maintenant la formule temporelle $P \rightsquigarrow Q$ par rapport à l'ensemble $tfair(\mathcal{M}, L)$ des traces équitables simulant le modèle Event-B $\mathcal{M}$.

Définition 4.3.4. Le modèle $\mathcal{M}$ sous hypothèse d'équité L satisfait $P \rightsquigarrow Q$, si pour toutes les traces $\sigma \in tfair(\mathcal{M}, L)$, la propriété suivante est vérifiée : $\forall i \cdot (i \geq 0 \wedge P(\sigma_i) \Rightarrow \exists j \cdot (j \geq i \wedge Q(\sigma_j)))$

Nous présentons dans la section suivante le paradigme « service-as-event » [10], basé sur propriétés de vivacité et la méthode Event-B et que nous avons appliqué pour modéliser les algorithmes répartis.

4.3.3 Le paradigme « service-as-event »

Nous introduisons dans cette section le paradigme « service-as-event » [10], inspiré du paradigme « call-as-event » [24], et basé sur les propriétés de vivacité utilisant l'opérateur *« leads to »* ($\rightsquigarrow$) et la méthode Event-B. Nous nous servons des propriétés pour caractériser les modèles Event-B des algorithmes répartis (ces modèles modélisent les possibles changements d'états du système réparti support d'un l'algorithme étudié et/ou des processus composant le système), et pour guider le développement formel par raffinement de ces derniers.

Les objectifs sont ici d'identifier, dans un premier temps, les services fournis par un algorithme réparti, qui sont exprimés à l'aide de propriétés de vivacité, puis abstraitement dans un modèle Event-B ; puis ensuite d'utiliser les règles d'inférence relatives à l'opérateur *« leads to »* (transitivité, etc.) pour guider le raffinement, en décomposant les services en *phases/étapes* locales (e.g. phases d'*initialisation*, de *demande de section critique*, de *stabilisation*, etc) aux processus composant le système réparti support de l'algorithme étudié. Les propriétés de vivacité introduisent la notion de *contrôle* dans les modèles Event-B, car ce sont elles qui permettent la coordination et la synchronisation des différentes étapes/phases composant l'algorithme.

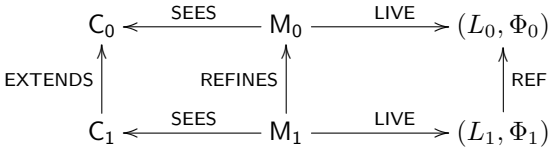


Figure 4.1 – Le paradigme « service-as-event » : méthodologie

Le diagramme 4.1 présente notre démarche : nous enrichissons la sémantique d'Event-B de deux relations LIVE et REF. Nous expliquons la relation LIVE, en prenant pour illustration le niveau abstrait présenté par le diagramme faisant intervenir un contexte $\mathsf{C_0}$, une machine $\mathsf{M_0}$ avec des variables x et un invariant $I(x)$, une collection de propriétés de vivacité Φ_0 (caractérisant $\mathsf{M_0}$) et une hypothèse d'équité L_0 posée sur $\mathsf{M_0}$; LIVE signifie que la spécification de la machine $\mathsf{M_0}$ est : $\mathcal{S}pec(\mathsf{M_0}) \ \widehat{=}\ \mathsf{INIT}(x) \wedge \square[\mathsf{NEXT}]_x \wedge L_0$, et que chaque propriété de vivacité $P \rightsquigarrow Q$ de Φ_0 est dérivable à partir de $\mathcal{S}pec(\mathsf{M_0})$, soit $\mathcal{S}pec(\mathsf{M_0}) \vdash (P \rightsquigarrow Q)$. La relation REF est définie comme suit : chaque propriété de vivacité $P_1 \rightsquigarrow Q_1$ de Φ_1 est dérivable à partir de $\mathcal{S}pec(\mathsf{M_1})$, c'est-à-dire, $\mathcal{S}pec(\mathsf{M_1}) \vdash (P_1 \rightsquigarrow Q_1)$, et $\mathsf{M_1}$ raffine (au sens Event-B) $\mathsf{M_0}$, et la spécification $\mathcal{S}pec(\mathsf{M_1})$ de la machine $\mathsf{M_1}$, ainsi que l'ensemble des propriétés de vivacité Φ_1 associées permettent de dériver toutes les propriétés de vivacités $P \rightsquigarrow Q$ de Φ_0, c'est-à-dire que nous avons $(\mathcal{S}pec(\mathsf{M_1}), \Phi_1) \vdash (P \rightsquigarrow Q)$.

4.3.3.1 Abstraction

Avant d'introduire l'abstraction d'un algorithme réparti à l'aide du paradigme *service-as-event*, nous définissons des notations que nous emploierons dans cette section et durant le reste de ce chapitre : $P \xrightarrow{\mathsf{e}} Q$ (un événement e, sur lequel une hypothèse d'équité faible est posée, est observable quand P est vrai et conduit à Q) et $P \overset{\mathsf{e}}{\Rightarrow} Q$ (un événement e, sur lequel une condition d'équité forte est posée, est observable quand P est vrai et conduit à Q).

Définition 4.3.5. ($P \xrightarrow{\mathsf{e}} Q$). Soit M un modèle Event-B, avec des variables x, un invariant $I(x)$, un événement e et d'autres événements $\mathsf{c_0}, \mathsf{c_1}, \ldots, \mathsf{c_n}$. L'événement e modélise une action du système support de l'algorithme réparti étudié et les événements $\mathsf{c_i}$ $(0 \leq i \leq n)$, modélisent soit d'autres actions possibles du système, soit des actions de l'environnement de ce dernier. Soit P et Q deux propriétés sur les valeurs prises par les variables x du modèle M. P est la condition d'observation de l'événement e et Q est la propriété établie après l'observation de e. $P \xrightarrow{\mathsf{e}} Q$ est l'expression de la propriété de vivacité $P \rightsquigarrow Q$ par l'événement e du modèle M. Les événements e et $\mathsf{c_i}$ $(0 \leq i \leq n)$ doivent satisfaire les propriétés suivantes :

- e satisfait :
  - la propriété $P(x) \wedge BA(\mathsf{e})(x, x') \Rightarrow Q(x')$,
  - la condition de faisabilité : $P(x) \Rightarrow (\exists y \cdot BA(\mathsf{e})(x, y))$.

— Les événements $\mathsf{c_i}$ satisfont la propriété suivante : $P(x) \wedge BA(\mathsf{c_i})(x, x') \Rightarrow (P(x') \vee Q(x'))$, c'est-à-dire que l'observation d'un événement $\mathsf{c_i}$ dans un état satisfaisant la propriété P, ne rend pas fausse la propriété P, condition d'observation attendue par l'événement e ou conduit à Q.
— Une condition d'équité faible est posée sur l'événement e ($WF_x(\mathsf{e})$).

Justification. Nous montrons que si $P \xrightarrow{\mathsf{e}} Q$, alors le modèle M satisfait la propriété $P \rightsquigarrow Q$. Le fait que tous les événements $\mathsf{c_i}$ satisfont $P(x) \wedge BA(\mathsf{c_i})(x, x') \Rightarrow (P(x') \vee Q(x'))$, que e satisfait $P(x) \wedge BA(\mathsf{e})(x, x') \Rightarrow Q(x')$ et que $P(x) \wedge (x = x') \Rightarrow P(x')$ nous permettent de déduire :

$$P \wedge [\mathsf{NEXT}]_x \Rightarrow (P' \vee Q')$$

où $\mathsf{NEXT} \triangleq BA(\mathsf{e})(x, x') \vee BA(\mathsf{c_0})(x, x') \vee \ldots \vee BA(\mathsf{c_n})(x, x')$.

L'événement e satisfait la condition de faisabilité, permettant de déduire $P \Rightarrow ENABLED\langle e \rangle_x$. La propriété $P(x) \wedge BA(\mathsf{e})(x, x') \Rightarrow Q(x')$ nous permet de déduire $P \wedge \langle \mathsf{NEXT} \wedge \mathsf{e} \rangle_x \Rightarrow Q'$. L'hypothèse d'équité posée sur l'événement e est une hypothèse d'équité faible ($WF_x(\mathsf{e})$) ; et l'application de la règle d'inférence WF1 (voir page 36) de TLA nous donne $\mathcal{S}pec(\mathsf{M}) \vdash (P \rightsquigarrow Q)$. Nous pouvons déduire que le modèle M satisfait la propriété $P \rightsquigarrow Q$. □

Définition 4.3.6. ($P \overset{\mathsf{e}}{\Rightarrow} Q$). Soit M un modèle Event-B, avec des variables x, un invariant $I(x)$, un événement e et d'autres événements $\mathsf{c_0}$, $\mathsf{c_1}$, …, $\mathsf{c_n}$. Soit P et Q deux propriétés sur les valeurs prises par les variables x du modèle M. L'événement e modélise une action du système support de l'algorithme réparti étudié et les événements $\mathsf{c_i}$ ($0 \leq i \leq n$), modélisent soit d'autres actions possibles du système, soit des actions de l'environnement de ce dernier. Un ou plusieurs de ces événements peuvent nous conduire à un état satisfaisant une propriété R ($R \neq P$ et $R \neq Q$) quand P est vrai. P est la condition d'observation de l'événement e et Q est la propriété établie après l'observation de e. $P \overset{\mathsf{e}}{\Rightarrow} Q$ est l'expression de la propriété de vivacité $P \rightsquigarrow Q$ par l'événement e du modèle M. Les événements e et $\mathsf{c_i}$ ($0 \leq i \leq n$) doivent satisfaire les propriétés suivantes :

— e satisfait :
  — la propriété $P(x) \wedge BA(\mathsf{e})(x, x') \Rightarrow Q(x')$,
  — la condition de faisabilité : $P(x) \Rightarrow (\exists y \cdot BA(\mathsf{e})(x, y))$.
— Les événements $\mathsf{c_i}$ satisfont les propriétés suivantes :
  — il existe un événement $\mathsf{c_j}$, tel que $P(x) \wedge BA(\mathsf{c_j})(x, x') \Rightarrow R(x')$, c'est-à-dire que l'observation d'un événement $\mathsf{c_j}$ dans un état satisfaisant la propriété P, conduit à un état satisfaisant R, telle que $(R \neq P) \wedge (R \neq Q)$, et $R \rightsquigarrow P$ est satisfaite par M (des événements $\mathsf{c_k}$ de M, sous condition d'équité faible, conduisent de R à P),
  — $\mathsf{c_j}$ satisfait aussi la condition de faisabilité : $P(x) \Rightarrow (\exists y \cdot BA(\mathsf{c_j})(x, y))$,
  — tous les événements c_i, y compris c_j, satisfont la propriété $(P(x) \vee R(x)) \wedge BA(\mathsf{c_i})(x, x') \Rightarrow (P(x') \vee R(x') \vee Q(x'))$.
— Une condition d'équité forte est posée sur l'événement e ($SF_x(\mathsf{e})$).

Justification. Nous montrons que si $P \overset{\mathsf{e}}{\Rightarrow} Q$, le modèle M satisfait la propriété $P \rightsquigarrow Q$. Le fait que tous les événements $\mathsf{c_i}$ satisfont $(P(x) \vee R(x)) \wedge BA(\mathsf{c_i})(x, x') \Rightarrow (P(x') \vee R(x') \vee Q(x'))$, que e satisfait $P(x) \wedge BA(\mathsf{e})(x, x') \Rightarrow Q(x')$ et que $(P(x) \vee R(x)) \wedge (x = x') \Rightarrow (P(x') \vee R(x'))$ nous permettent de déduire :

$$(P \vee R) \wedge [\mathsf{NEXT}]_x \Rightarrow (P' \vee R' \vee Q')$$

où $\mathsf{NEXT} \triangleq BA(\mathsf{e})(x, x') \vee BA(\mathsf{c_0})(x, x') \vee \ldots \vee BA(\mathsf{c_n})(x, x')$.

La propriété $P(x) \wedge BA(\mathsf{e})(x, x') \Rightarrow Q(x')$ nous permet de déduire $(P \vee R) \wedge \langle \mathsf{NEXT} \wedge \mathsf{e} \rangle_x \Rightarrow Q'$. Cette propriété et $P \Rightarrow ENABLED\langle e \rangle_x$, ainsi que l'hypothèse $R \rightsquigarrow P$ nous permettent de déduire que tant que Q, n'est pas satisfait, alors l'événement e sera fatalement activable, soit $(\neg Q) \Rightarrow \Diamond ENABLED\langle e \rangle_x$. Nous pouvons en déduire, en considérant l'hypothèse $R \rightsquigarrow P$, que $\Box(P \vee R) \wedge \Box[\mathsf{NEXT}] \Rightarrow \Diamond ENABLED\langle e \rangle_x$. L'application de la règle d'inférence SF1 (voir page 36) nous donne $(\mathcal{S}pec(\mathsf{M}), (R \rightsquigarrow P)) \vdash ((P \vee R) \rightsquigarrow Q)$. Nous pouvons déduire que le modèle M satisfait la propriété $(P \vee R) \rightsquigarrow Q$, par conséquent $P \rightsquigarrow Q$. □

Nous continuons maintenant par la description de l'utilisation du paradigme *service-as-event*, pour l'abstraction d'un algorithme réparti, dans un modèle Event-B $\mathsf{M_0}$ avec des variables x et un invariant

$I(x)$. Nous commençons par identifier les services fournis par l'algorithme réparti : nous caractérisons l'algorithme par une collection de services $S \mathrel{\widehat{=}} \{s_0, s_1, \ldots, s_m\}$. Chaque service identifié $s_i \in S$ (avec $i \in 0..m$) modifie des variables d'états x et est exprimé par une spécification de type $P_i \rightsquigarrow Q_i$.

— La propriété P_i caractérise les valeurs initiales des variables x attendues avant l'invocation du service s_i. $P_i(x)$ doit être vraie pour que le service s_i puisse être invoqué.

— La propriété Q_i doit être satisfaite après l'invocation du service s_i.

En résumé, nous exprimons abstraitement un algorithme réparti par un ensemble de propriétés de vivacité noté $\Phi_0 \mathrel{\widehat{=}} \{P_0 \rightsquigarrow Q_0, P_1 \rightsquigarrow Q_1, \ldots, P_m \rightsquigarrow Q_m\}$, caractérisant les services offerts par l'algorithme.

Chaque propriété de vivacité de $P_i \rightsquigarrow Q_i$ de Φ_0, donc chaque service s_i offert par l'algorithme, est modélisé par un événement $\mathsf{e_i}$ du modèle abstrait $\mathsf{M_0}$, de la forme :

EVENT $\mathsf{e}_i \mathrel{\widehat{=}}$
WHEN
$P_i(x)$
THEN
$x :| Q_i(x')$
END

et sur lequel une hypothèse d'équité faible ($WF_x(\mathsf{e_i})$) ou forte ($SF_x(\mathsf{e_i})$) a été posée. L'événement e est tel que :

— $P_i \xrightarrow{\mathsf{e_i}} Q_i$, si l'hypothèse d'équité posée sur e est faible ($WF_x(\mathsf{e_i})$),

— $P_i \overset{\mathsf{e_i}}{\Rightarrow} Q_i$, si l'hypothèse d'équité posée sur e est forte ($SF_x(\mathsf{e_i})$).

En résumé, le modèle abstrait $\mathsf{M_0}$ contient aussi bien des événements modélisant des services que des événements modélisant les interactions de l'algorithme réparti étudié avec son environnement. Nous illustrons cette abstraction d'un algorithme réparti, obtenue à l'aide du paradigme « *service-as-event* », en utilisant l'algorithme de l'élection du leader dans un réseau [4].

Exemple 4.3.2. *Abstraction du protocole de l'élection du leader [4].*

Nous caractérisons le service de l'élection du leader comme suit :

— **Ensembles porteurs :** ND (ensemble de nœuds).

— **Constantes :** g satisfaisant la propriété suivante $acyclic(g) \wedge connected(g)$ (g est un graphe acyclique et connexe représentant le réseau).

— **Variables :** $x = (sp, rt)$ (sp est un arbre de recouvrement de g, rt est la racine de l'arbre).

— **Condition d'activation du service (précondition) :**
$P(x) \mathrel{\widehat{=}} sp = \varnothing \wedge rt \in ND$ (l'arbre sp n'est pas encore calculé, rt est un nœud quelconque du graphe g).

— **Condition satisfaite après l'activation du service (postcondition) :** $Q(x) \mathrel{\widehat{=}} spanning(sp, rt, g)$ (un arbre sp recouvrant de g est calculé en un coup, rt est la racine de l'arbre sp).

Ces données nous permettent d'exprimer l'élection du leader à l'aide d'une propriété de vivacité utilisant l'opérateur *leads to* ($\rightsquigarrow$) :

$$(sp = \varnothing \wedge rt \in ND) \rightsquigarrow spanning(sp, rt, g)$$

Nous définissons ensuite une machine Event-B $\mathsf{Leader_0}$ abstraite satisfaisant cette propriété de vivacité $P \rightsquigarrow Q$ et contenant un seul événement abstrait $\mathsf{election_0}$ tel que :

EVENT $\mathsf{election_0} \mathrel{\widehat{=}}$
BEGIN
$sp, rt :| spanning(sp', rt', g)$
END

La condition d'équité associée à la machine $\mathsf{Leader_0}$ est : $WF_x(\mathsf{election_0})$. Nous montrons que $\mathsf{Leader_0}$ satisfait $P \rightsquigarrow Q$.

Démonstration. Le contexte associé à la machine $\mathsf{Leader_0}$ nous permet de démontrer la propriété $P(x) \wedge BA(\mathsf{election_0})(x, x') \Rightarrow Q(x')$, ainsi que la faisabilité (FIS) de l'événement $\mathsf{election_0} : \exists x'.BA(\mathsf{election_0})(x, x')$. La propriété $P(x) \wedge BA(\mathsf{election_0})(x, x') \Rightarrow Q(x')$ nous permet de déduire $P \wedge [\mathsf{NEXT}]_x \Rightarrow (P' \vee Q')$, $P \wedge \langle \mathsf{NEXT} \wedge \mathsf{election_0} \rangle_x \Rightarrow Q'$, où $\mathsf{NEXT} \mathrel{\widehat{=}} BA(\mathsf{election_0})(x, x') \vee (x = x')$. La condition de faisabilité de l'événement $\mathsf{election_0}$ nous permet de déduire $P \Rightarrow ENABLED\langle \mathsf{election_0} \rangle_x$, où $ENABLED\langle \mathsf{election_0} \rangle_x$

est définie par $\exists y \cdot BA(\mathsf{election_0})(x, y)$. L'application de la règle d'inférence **WF1** nous donne $\Box[\mathsf{NEXT}]_x \wedge WF_x(\mathsf{election_0}) \Rightarrow (P \rightsquigarrow Q)$. Finalement, nous obtenons $\mathsf{INIT}(x) \wedge \Box[\mathsf{NEXT}]_x \wedge WF_x(\mathsf{election_0}) \Rightarrow (P \rightsquigarrow Q)$. □

Nous pouvons résumer l'abstraction du protocole de l'élection du leader en instanciant le diagramme présenté par la figure 4.1 :

$$\mathsf{C_0} \xleftarrow{\text{SEES}} \mathsf{Leader_0} \xrightarrow{\text{LIVE}} (WF_x(\mathsf{election_0}), \{P \rightsquigarrow Q\})$$

FIGURE 4.2 – Abstraction du protocole de l'élection du leader

4.3.3.2 Raffinement

Nous expliquons maintenant en détails comment le raffinement de modèles EVENT-B est guidé par les propriétés de vivacité et nous explicitons les liens entre ces dernières et les modèles, en reprenant en détail le diagramme présenté par la figure 4.1.

Nous rappelons que la spécification TLA d'une machine EVENT-B $\mathsf{M_0}$, avec des variables x, un invariant $I(x)$ et un ensemble d'événements E_0 modifiant ces variables est la suivante : $\mathcal{S}pec(\mathsf{M_0}) \triangleq \mathsf{INIT_0}(x) \wedge \Box[\mathsf{NEXT_0}]_x \wedge L_0$, où $\mathsf{INIT_0}(x)$ définit les conditions initiales des variables x de $\mathsf{M_0}$, $\mathsf{NEXT_0} \triangleq \exists e \cdot e \in E_0 \wedge BA(e)(x, x')$ et L_0 une conjonction d'hypothèses d'équités forte et/ou faible sur les événements de $\mathsf{M_0}$. Nous pouvons ainsi raisonner sur les comportements de la machine $\mathsf{M_0}$, grâce à des propriétés de vivacité utilisant l'opérateur « *leads to* » ($\rightsquigarrow$).

Nous nous servons de ces propriétés pour caractériser les modèles B Événementiels des systèmes que nous étudions et pour guider le développement formel par raffinement de ces derniers. La figure 4.3 suivante nous explique les liens entre les modèles B Événementiel et les propriétés de vivacité ($\rightsquigarrow$).

$$\mathsf{C_0} \xleftarrow{\text{SEES}} \mathsf{M_0} \xrightarrow{\text{LIVE}} (L_0, \Phi_0)$$

FIGURE 4.3 – Relations entre modèles et propriétés de vivacité

Le diagramme fait intervenir un contexte $\mathsf{C_0}$, une machine $\mathsf{M_0}$, une collection de propriétés de vivacité Φ_0 et une condition d'équité L_0. La relation LIVE que nous avons introduite signifie pour la machine $\mathsf{M_0}$ que :

- $\mathcal{S}pec(\mathsf{M_0}) \triangleq \mathsf{INIT_0}(x) \wedge \Box[\mathsf{NEXT_0}]_x \wedge L_0$.
- Pour toutes les propriétés de vivacité $P \rightsquigarrow Q$ de Φ_0, $\mathsf{M_0}$, sous hypothèse d'équité L_0, satisfait $P \rightsquigarrow Q : \mathcal{S}pec(\mathsf{M_0}) \vdash (P \rightsquigarrow Q)$.

Le diagramme présenté par la figure 4.4 suivante explique le raffinement du modèle $\mathsf{M_0}$ par un modèle $\mathsf{M_1}$:

$$\begin{array}{ccccc} \mathsf{C_0} & \xleftarrow{\text{SEES}} & \mathsf{M_0} & \xrightarrow{\text{LIVE}} & (L_0, \Phi_0) \\ \uparrow{\scriptstyle \text{EXTENDS}} & & \uparrow{\scriptstyle \text{REFINES}} & & \uparrow{\scriptstyle \text{REF}} \\ \mathsf{C_1} & \xleftarrow{\text{SEES}} & \mathsf{M_1} & \xrightarrow{\text{LIVE}} & (L_1, \Phi_1) \end{array}$$

FIGURE 4.4 – Raffinement de modèle

Nous pouvons expliquer la relation REF par les conditions suivantes :

- Pour tout P, Q, tels que $(P \rightsquigarrow Q) \in \Phi_1$, nous avons $\mathcal{S}pec(\mathsf{M_1}) \vdash (P \rightsquigarrow Q)$.
- $\mathsf{M_1}$ raffine $\mathsf{M_0}$.
- Pour tout P, Q, tels que $(P \rightsquigarrow Q) \in \Phi_0$, nous avons $(\mathcal{S}pec(\mathsf{M_1}), \Phi_1) \vdash (P \rightsquigarrow Q)$.
- Si on se trouve dans le cas suivant, tel que la variable abstraite x du modèle $\mathsf{M_0}$ n'est pas modifiée par les nouveaux événements $Q_1, \ldots, Q_l$ introduits lors du raffinement de $\mathsf{M_0}$ en $\mathsf{M_1}$; que les propriétés d'équité restent identiques, c'est-à-dire que les nouveaux événements N'_j raffinant ceux

abstraits N_j, sont définis à partir de ces derniers en ajoutant la transformation $z = z'$ d'une nouvelle variable z ; que de plus, si un événement N du modèle abstrait M_0 était défini avec une équité donneé $SF_x(N)$ (respectivement $WF_x(N)$), et que le nouvel événement étendu N' par z est défini avec une hypothèse $SF_y(N')$ (respectivement $WF_y(N')$), pour $y = x, z$, alors nous avons $\mathcal{S}pec(\mathsf{M}_1) \Rightarrow \mathcal{S}pec(\mathsf{M}_0)$.

La relation REF exprime un processus similaire au raffinement, dans lequel des propriétés de vivacité de Φ_0 sont détaillées à l'aide des règles d'inférence relatives à l'opérateur *leadsto* ($\rightsquigarrow$). Il s'agit d'une relation basée sur la déduction et la preuve des propriétés de vivacité de Φ_0 à partir de celles de Φ_1 : toute propriété de vivacité $P \rightsquigarrow Q$ de Φ_0 peut être dérivée de Φ_1 et de la machine M_1 (avec des variables y, un prédicat d'initialisation $\mathsf{INIT}_1(y)$, un invariant de collage $J(x, y)$). L'ensemble E_1 des événements de M_1 est construit par rapport aux propriétés de vivacité de Φ_1. Il s'agit ensuite de déterminer une hypothèse d'équité L_1 pour la machine M_1, de telle manière que les propriétés de Φ_1, permettant de dériver les propriétés abstraites de Φ_0, soient satisfaites. La relation REF exprime aussi ainsi que le modèle M_1, sous hypothèse d'équité L_1, satisfait toutes les propriétés de vivacité $P \rightsquigarrow Q$ de Φ_0.

Nous présentons maintenant deux techniques permettant de justifier la préservation de propriétés de vivacité et d'équité lors d'étapes de raffinement répondant à des certaines conditions décrites ci-dessous.

Première technique de raffinement automatique. Cette technique de raffinement, appelée superposition est une stratégie de développement qui a été notammant utilisée dans le cadre de la méthode UNITY [14] pour simplifier le développement des systèmes répartis selon les couches progressivement ajoutées. En particulier, UNITY considère deux règles de superposition :

- La règle dite *d'augmentation* : Une instruction s du modèle abstrait est transformée en une instruction $s \parallel r$ où r ne modifie pas les varaibles du modèle abstrait.
- La règle *d'union restreinte* : Une instruction r est ajoutée au modèle abstrait pourvu que r ne modifie pas les variables du modèle abstrait.

Dans le cas du langage Event-B, cette technique peut être appliquée et on énonce et démontre le résultat suivant. La variable abstraite f d'un modèle M_0 n'est pas modifiée par les nouveaux événements $Q_1, \ldots, Q_l$ introduits lors du raffinement de M_0 en M_1. Les propriétés d'équité restent identiques, c'est-à-dire que les nouveaux événements N'_j raffinant ceux abstraits N_j, sont définis à partir de ces derniers en ajoutant la transformation $z = z'$. De plus, si un événement N du modèle abstrait M_0 était défini avec une équité donneé $SF_x(N)$ (respectivement $WF_x(N)$), alors nous demandons que le nouvel événement étendu N' par h est défini avec une hypothèse $SF_y(N')$ (respectivement $WF_y(N')$), avec $y = x, z$.

Théorème 4.3.1. *Raffinement par superposition*

Supposons que :

- $\mathcal{S}pec(\mathsf{M}_0) \mathrel{\widehat{=}} \mathsf{INIT}_0 \wedge \Box[\mathsf{NEXT}_0]_x \wedge \mathrm{WF}_x(N_1) \wedge \ldots \wedge \mathrm{SF}_x(N_k)$
- $\mathcal{S}pec(\mathsf{M}_1) \mathrel{\widehat{=}} \mathsf{INIT}_1 \wedge \Box[\mathsf{NEXT}_1]_y \wedge \mathrm{WF}_y(N'_1) \wedge \ldots \wedge \mathrm{SF}_y(N'_k) \wedge L_1$
- $y = x, z$
- $\forall i \in 1..k : N'_i \mathrel{\widehat{=}} N_i \wedge z = z'$
- $\mathsf{NEXT}_0 \mathrel{\widehat{=}} N_1 \vee \ldots \vee N_k$
- $\mathsf{NEXT}_1 \mathrel{\widehat{=}} \mathsf{NEXT}'_0 \vee Q_1 \vee \ldots \vee Q_l$
- *L_1 est une conjonction de contraintes d'équité faible et forte sur des combinaisons des événements $Q_1, \ldots, Q_l$ de M_1.*

Alors

- $\mathrm{WF}_y(N'_i) \Rightarrow \mathrm{WF}_x(N_i)$
- $\mathrm{SF}_y(N'_j) \Rightarrow \mathrm{SF}_x(N_j)$
- $\mathcal{S}pec(\mathsf{M}_1) \Rightarrow \mathcal{S}pec(\mathsf{M}_0)$

Démonstration. La preuve est faite comme suit :

⟨1⟩1. Nous montrons que $\mathrm{WF}_y(N'_i) \Rightarrow \mathrm{WF}_x(N_i)$. Nous avons :

$\mathrm{WF}_y(N'_i) \mathrel{\widehat{=}} \Box\Diamond\neg ENABLED\langle N'_i\rangle_y \vee \Box\Diamond\langle N'_i\rangle_y$

$\mathrm{WF}_y(N'_i) \mathrel{\widehat{=}} \Box\Diamond\neg ENABLED\langle N_i \wedge z = z'\rangle_{x,z} \vee \Box\Diamond\langle N_i \wedge z = z'\rangle_{x,z}$

$\mathrm{WF}_y(N'_i) \mathrel{\widehat{=}} \Box\Diamond\neg ENABLED\langle N_i \wedge z = z'\rangle_x \vee \Box\Diamond\langle N_i \wedge z = z'\rangle_x$

et

$\mathrm{WF}_x(N_i) \mathrel{\widehat{=}} \Box\Diamond\neg ENABLED\langle N_i\rangle_x \vee \Box\Diamond\langle N_i\rangle_x$

Nous pouvons en déduire que $\mathrm{WF}_y(N'_i) \Rightarrow \mathrm{WF}_x(N_i)$. □

⟨1⟩2. Nous montrons que $\mathrm{SF}_y(N'_j) \Rightarrow \mathrm{SF}_x(N_j)$. Nous avons :

$\mathrm{SF}_y(N'_j) \mathrel{\widehat{=}} \Diamond\Box\neg ENABLED\langle N'_j\rangle_y \vee \Box\Diamond\langle N'_j\rangle_y$

$\mathrm{SF}_y(N'_j) \mathrel{\widehat{=}} \Diamond\Box\neg ENABLED\langle N_j \wedge z = z'\rangle_{x,z} \vee \Box\Diamond\langle N_j \wedge z = z'\rangle_{x,z}$

$\mathrm{SF}_y(N'_j) \mathrel{\widehat{=}} \Diamond\Box\neg ENABLED\langle N_j \wedge z = z'\rangle_x \vee \Box\Diamond\langle N_j \wedge z = z'\rangle_x$

et

$\mathrm{SF}_x(N_i) \mathrel{\widehat{=}} \Diamond\Box\neg ENABLED\langle N_j\rangle_x \vee \Box\Diamond\langle N_j\rangle_x$

Nous pouvons en déduire que $\mathrm{SF}_y(N'_j) \Rightarrow \mathrm{SF}_x(N_j)$. □

⟨1⟩3. Nous montrons que $\mathcal{S}pec(\mathsf{M_1}) \Rightarrow \mathcal{S}pec(\mathsf{M_0})$. Le raffinement Event-B nous permet de montrer $(\mathsf{INIT}_1 \wedge \Box[\mathsf{NEXT}_1]_y) \Rightarrow (\mathsf{INIT}_0 \wedge \Box[\mathsf{NEXT}_0]_x)$. Les étapes ⟨1⟩1 et ⟨1⟩2 nous permettent de montrer :

$$\mathrm{WF}_y(N'_1) \wedge \ldots \wedge \mathrm{SF}_y(N'_k) \wedge L_1 \Rightarrow \mathrm{WF}_x(N_1) \wedge \ldots \wedge \mathrm{SF}_x(N_k)$$

Nous obtenons par conséquent $\mathcal{S}pec(\mathsf{M_1}) \Rightarrow \mathcal{S}pec(\mathsf{M_0})$. □

⟨1⟩4. QED

□

Une conséquence est que les traces de $\mathsf{M_1}$ sont des traces $\mathsf{M_0}$ et donc que les propriétés de $\mathsf{M_0}$ sont des propriétés de $\mathsf{M_1}$. Il faut noter l'importance des hypothèses d'équité qui exigent que les événements restent les mêmes à une extension près des variables. Une autre conséquence du théorème est la règle de raffinement suivante que nous allons énoncer :

Première règle de raffinement automatique. Supposons que :

- $\mathcal{S}pec(\mathsf{M_0}) \mathrel{\widehat{=}} \mathsf{INIT}_0 \wedge \Box[\mathsf{NEXT}_0]_x \wedge \mathrm{WF}_x(N_1) \wedge \ldots \wedge \mathrm{SF}_x(N_k)$
- $\mathcal{S}pec(\mathsf{M_1}) \mathrel{\widehat{=}} \mathsf{INIT}_1 \wedge \Box[\mathsf{NEXT}_1]_y \wedge \mathrm{WF}_y(N'_1) \wedge \ldots \wedge \mathrm{SF}_y(N'_k) \wedge L_1$
- $y = x, z$
- $\forall i \in 1..k : N'_i \mathrel{\widehat{=}} N_i \wedge z = z'$
- $\mathsf{NEXT}_0 \mathrel{\widehat{=}} N_1 \vee \ldots \vee N_k$
- $\mathsf{NEXT}_1 \mathrel{\widehat{=}} \mathsf{NEXT}'_0 \vee Q_1 \vee \ldots \vee Q_l$
- L_1 est une conjonction de contraintes d'équité faible et forte sur des combinaisons des événements $Q_1, \ldots, Q_l$ de $\mathsf{M_1}$.

Alors, toutes les propriétés d'invariance et de fatalité de $\mathsf{M_0}$ sont préservées ; de plus $\mathsf{M_1}$ raffine $\mathsf{M_0}$ au sens REF.

Seconde technique de raffinement automatique. Cette technique est basée sur le raffinement de données. Dans le cas du raffinement de données, nous changeons d'espace de données et la relation entre les variables x (d'une machine abstraite $\mathsf{M_0}$) et y (d'une machine abstraite $\mathsf{M_1}$) est définie par une fonction r, telle que $r(x) = y$. On demande à ce que cette fonction soit bijective et qu'elle permette de remplacer un nom par un autre, avec comme contrainte que les nouveaux événements concrets c soient équivalents aux anciens événements a à r près. Nous notons a un événement abstrait, de la machine abstraite $\mathsf{M_0}$, modifiant la variable x et c un événement concret, de la machine concrète $\mathsf{M_1}$, raffinant a et modifiant g. Une relation d'équivalence entre a et c à f près, notée $\mathsf{a} =_f \mathsf{c}$, est définie comme suit :

Nom	**Événement** a	**Événement** c
Cas 1	BEGIN $x : \|R(x, x')$ END	BEGIN $\ominus x : \|R(x, x')$ $\oplus y : \|R(y, y')$ END
Cas 2	WHEN $G(x)$ THEN $x : \|R(x, x')$ END	WHEN $\ominus G(x)$ $\oplus G(y)$ THEN $\ominus x : \|R(x, x')$ $\oplus y : \|R(y, y')$ END
Cas 3	ANY t_a WHERE $G(t_a, x)$ THEN $x : \|R(t_a, x, x')$ END	ANY $\ominus t_a$ $\oplus t_c$ WHERE $\ominus G(t_a, x)$ $\oplus G(t_c, y)$ WITH $\oplus t_c = h(t_a)$ THEN $\ominus x : \|R(t_a, x, x')$ $\oplus x : \|R(t_c, y, y')$ END

— Cas 1 : Il s'agit de remplacer l'action abstraite de l'événement a par une action concrète équivalente.
— Cas 2 : Il s'agit de remplacer gardes et actions abstraites par des gardes et actions équivalentes.
— Cas 3 : Les paramètres t_a d'un événement abstrait a peuvent être remplacés par des paramètres concrets t_c. Nous demandons dans ce cas que ces deux paramètres soient reliés par une fonction h bijective, telle que $t_c = h(t_a)$.

En supposant que la relation f soit ainsi définie, on peut donc assurer que :
— $BA(\mathsf{a})(x, x') \equiv BA(\mathsf{c})(y, y')$ où c est obtenu en remplaçant une occurrence de x par y.
— $r(x') = y'$
— $r(x) = y$

Avec ces conditions, les conditions d'équité sont conservées. Il reste à montrer que les deux traces engendrées respectivement par M_0 et M_1 sont conservées à renommage près par r. Pour cela nous allons utiliser les fonctions de raffinement (« refinement mappings ») que nous définissons comme suit :
— Soit deux machines Event-B M_0 et M_1, et $L0$ et $L1$ exprimant respectivement des hypothèses d'équité $FAIR(\mathsf{M_0})$ et $FAIR(\mathsf{M_1})$ qui restreignent les traces engendrées par M_0 et M_1. M_1 raffine M_0, si et seulement si, l'ensemble des propriétés visibles induites par M_1 est un sous-ensemble des propriétés visibles induites par M_0.

Définition 4.3.7. fonction de raffinement. Une fonction de raffinement $f : \Sigma_1 \to \Sigma_0$ est basé sur les conditions suivantes :

1. La fonction de raffinement préserve l'état visible d'un composant : une conséquence du raffinement de chaque événement e de M_0 par un événement de M_1.
2. Les états initiaux de M_1 sont associés (par f) aux états initiaux de M_0
3. Chaque événement e_1 de M_1 raffine un événement e_0 de M_0 ou est un bégaiement de M_0. Alors avec f, nous avons $\mathsf{e}_1 \Rightarrow \mathsf{e}_0$
4. $f(L1) \subseteq L0$

Définition 4.3.8. Raffinement de spécification.

L'existance d'une fonction de raffinement exprime que la spécification M_1 implemente M_0, ce qui signifie que les traces de M_1 sont(au bégaiement près) des traces de M_0.

Démonstration. En utilisant ces deux définitions, nous pouvons montrer que les deux traces engendrées respectivement par M_0 et M_1 sont conservées à renommage près par r :

- La condition 1 de la première définition est vérifiée par le fait que chaque événement de M_0 est raffiné par un événement de M_1.
- Si nous notons $x0$ l'état initial des variables de la machine M_0 et $y0$ l'état initial des variables de la machine M_1, nous avons la relation : $y0 = r(x0)$, et nous vérifions ainsi la condition 2 de la première définition.
- Dans la machine M_1, nous ne rajoutons pas de nouveaux événements, seulement des raffinements des événements de M_0. Nous vérifions de ce fait la condition 3 de la première définition.
- Et nous avons vu précédemment que les conditions d'équité sont conservées, et nous obtenons de ce fait la condition 4 de la première définition.

La première définition nous permet alors de déduire l'existence d'une fonction de raffinement f, et en utilisant la seconde définition, cela signifie que les traces de M_1 sont (au bégaiement près) des traces de M_0. □

Seconde règle de raffinement automatique. Supposons que :

- La relation entre les variables x (d'une machine abstraite M_0) et y (d'une machine abstraite M_1) est définie par une fonction r, telle que $r(x) = y$.
- r est une bijection et permet de remplacer un nom par un autre, avec comme contrainte que les nouveaux événements concrets c soient équivalents aux anciens événements a à r près.
- Une relation d'équivalence entre a et c à f près, notée $\mathsf{a} =_f \mathsf{c}$, existe et est définie comme suit :

Nom	**Événement** a	**Événement** c
Cas 1	BEGIN $x :\mid R(x, x')$ END	BEGIN $\ominus x :\mid R(x, x')$ $\oplus y :\mid R(y, y')$ END
Cas 2	WHEN $G(x)$ THEN $x :\mid R(x, x')$ END	WHEN $\ominus G(x)$ $\oplus G(y)$ THEN $\ominus x :\mid R(x, x')$ $\oplus y :\mid R(y, y')$ END
Cas 3	ANY t_a WHERE $G(t_a, x)$ THEN $x :\mid R(t_a, x, x')$ END	ANY $\ominus t_a$ $\oplus t_c$ WHERE $\ominus G(t_a, x)$ $\oplus G(t_c, y)$ WITH $\oplus t_c = h(t_a)$ THEN $\ominus x :\mid R(t_a, x, x')$ $\oplus x :\mid R(t_c, y, y')$ END

Alors, toutes les propriétés d'invariance et de fatalité de M_0 sont préservées ; de plus M_1 raffine M_0 au sens REF.

4.3.3.3 Synthèse

Nous rappelons la définition du *raffinement* [23] entre les spécifications TLA $\mathcal{S}pec(\mathsf{M}_0)$ et $\mathcal{S}pec(\mathsf{M}_1)$ des machines M_0 et M_1 : $\mathcal{S}pec(\mathsf{M}_1)$ *raffine* $\mathcal{S}pec(\mathsf{M}_0)$, si, $\mathcal{S}pec(\mathsf{M}_1) \Rightarrow \mathcal{S}pec(\mathsf{M}_0)$, soit $(\mathsf{INIT}_1(y) \wedge \Box[\mathsf{NEXT}_1]_y \wedge L_1) \Rightarrow (\mathsf{INIT}_0(x) \wedge \Box[\mathsf{NEXT}_0]_x \wedge L_0)$, ce qui est généralement démontré par l'existence d'une fonction appelée « *refinement mapping* » [1], établissant des correspondances entre les espaces d'états de $\mathcal{S}pec(\mathsf{M}_0)$ et $\mathcal{S}pec(\mathsf{M}_1)$, ainsi qu'entre les événements (actions) permis par $\mathcal{S}pec(\mathsf{M}_0)$ et $\mathcal{S}pec(\mathsf{M}_1)$. La preuve de $\mathcal{S}pec(\mathsf{M}_1) \Rightarrow \mathcal{S}pec(\mathsf{M}_0)$, permet de démontrer, en plus de la préservation des propriétés de vivacité et de

sûreté caractérisant $\mathsf{M_0}$ et $\mathsf{M_1}$, l'inclusion de l'ensemble des traces d'exécution équitables $tfair(\mathsf{M_1}, L_1)$ engendrées par $\mathsf{M_1}$, dans celui $tfair(\mathsf{M_0}, L_0)$ du modèle $\mathsf{M_0}$.

Dans notre cas, nous avons choisi une démarche différente de l'utilisation de fonctions « *refinement mappings* » pour définir le raffinement entre deux spécifications $\mathcal{S}pec(\mathsf{M_1})$ et $\mathcal{S}pec(\mathsf{M_0})$. Nous utilisons pour cela la définition du raffinement Event-B (voir page 30), ainsi que les règles d'inférence liées aux propriétés *leads to* (voir à partir de la page 36).

Préservation des propriétés de sûreté. Nous la démontrons en prouvant $(\mathsf{INIT_1}(y) \wedge \Box[\mathsf{NEXT_1}]_y) \Rightarrow (\mathsf{INIT_0}(x) \wedge \Box[\mathsf{NEXT_0}]_x)$, ce qui nous est donné par la relation de raffinement Event-B existant entre les machines $\mathsf{M_0}$ et $\mathsf{M_1}$.

Démonstration. La démonstration de l'obligation de preuve de raffinement reliant les conditions initiales de $\mathsf{M_1}$ et $\mathsf{M_0}$: $\mathsf{INIT_1}(y') \Rightarrow (\exists x' \cdot (\mathsf{INIT_0}(x) \wedge J(x', y')))$, exprimant que pour chaque valeur initiale concrète y', une valeur initiale x' correspondante existe, nous permet de déduire $\mathsf{INIT_1}(y) \Rightarrow \mathsf{INIT_0}(x)$. Chacun des **(1)** événements e de $\mathsf{M_0}$ est raffiné par une ou des versions concrètes f dans $\mathsf{M_1}$, et **(2)** les nouveaux événements de $\mathsf{M_1}$ raffinent l'événement *skip*, ce qui est exprimé par les obligations de preuves démontrées suivantes : **(1)** $I(x) \wedge J(x, y) \wedge BA(f)(y, y') \Rightarrow \exists x' \cdot (BA(e)(x, x') \wedge J(x', y'))$ et **(2)** $I(x) \wedge J(x, y) \wedge BA(f)(y, y') \Rightarrow J(x, y')$. Ces obligations de preuves, avec celle qui exprime que $\mathsf{M_1}$ n'introduit pas plus de blocage que $\mathsf{M_0}$ (voir page 31), ainsi que les définitions de $[\mathsf{NEXT_1}]_y$ et $[\mathsf{NEXT_0}]_x$, nous permettent de déduire $\Box[\mathsf{NEXT_1}]_y \Rightarrow \Box[\mathsf{NEXT_0}]_x$. Nous obtenons ainsi $(\mathsf{INIT_1}(y) \wedge \Box[\mathsf{NEXT_1}]_y) \Rightarrow (\mathsf{INIT_0}(x) \wedge \Box[\mathsf{NEXT_0}]_x)$. □

Préservation des propriétés de vivacité. Nous rappelons que la condition d'équité L_1 est choisie de manière à ce que chaque propriété $P_1 \rightsquigarrow Q_1$ de Φ_1 soit satisfaite par $\mathsf{M_1}$ et que chaque propriété $P_0 \rightsquigarrow Q_0$ de Φ_0 soit dérivable à partir de $\mathsf{M_1}$ et Φ_1 : les propriétés de Φ_0 sont donc préservées lors du raffinement défini à l'aide de la relation REF et de celle REFINES de Event-B. Nous rappelons aussi que le raffinement Event-B de $\mathsf{M_0}$ par $\mathsf{M_1}$ nous permet la préservation de l'invariant $I(x)$. Si $L_1 \Rightarrow L_0$ peut être démontrée, alors en plus de la préservation des propriétés de sûreté/invariance $I(x)$ et de vivacité Φ_0, nous avons l'inclusion des traces équitables engendrées par $\mathsf{M_1}$ dans celles engendrées par $\mathsf{M_0}$: soit $tfair(\mathsf{M_1}, L_1) \subseteq tfair(\mathsf{M_0}, L_0)$.

Nous illustrons la notion de raffinement que nous avons présentée précédemment en reprenant l'algorithme de l'élection du leader vu dans la section précédente : nous nous intéressons ici au premier raffinement de la machine abstraite $\mathsf{Leader_0}$.

Exemple 4.3.3. *Premier raffinement du protocole de l'élection du leader.*

Nous illustrons l'application du paradigme « service-as-event », au raffinement de modèles, en raffinant la machine $\mathsf{Leader_0}$ vu dans la section précédente en une machine $\mathsf{Leader_1}$. L'idée directrice est ici de considérer que la construction de l'arbre de recouvrement *sp* du graphe acyclique g ne se fait pas en un coup, mais en plusieurs. Une variable tr représentant un sous-graphe de g est introduite, elle contient l'arbre de recouvrement qui est construit pas-à-pas, à l'aide d'un nouvel événement convergent progress : un nœud x n'ayant pas encore de « père », mais étant déjà père de tous ses voisins, sauf un nœud y (lui aussi n'ayant pas de encore père), devient un nœud fils de ce dernier. L'événement progress fait diminuer à chaque itération la taille de l'ensemble $dom(g) \setminus dom(tr)$, jusqu'à ce que celle-ci soit égale à 1, ce qui correspond à la fin de la construction l'arbre de recouvrement tr et à la découverte d'un nœud racine x qui n'appartient pas à $dom(tr)$. Nous considérons donc $dom(g) \setminus dom(tr)$ comme étant le variant que l'événement progress fait décroître. Nous posons $F(n+1) = (card(dom(g) \setminus dom(tr)) = n+1)$, avec $n \in \mathbb{N}$, indiquant que le nombre de racines (et par conséquent d'arbres) à un moment donné est n+1.

Nous caractérisons ce premier raffinement de la manière suivante :

— **Ensembles porteurs :** ND.
— **Constantes :** g.
— **Variables :** $v = (sp, rt, tr)$.
— **Condition d'activation du service :** $F(n+1)$.
— **Condition satisfaite après l'activation du service :** $F(1)$.

Nous exprimons l'élection du leader à l'aide des propriétés de vivacité suivantes :

$$\{F(n+1) \rightsquigarrow F(n), (\exists n \cdot F(n)) \rightsquigarrow F(1)\}$$

La première propriété de vivacité exprime que le nombre de racines diminue fatalement de 1 (à chaque itération) et la seconde propriété exprime qu'il existe un nombre de racines n permettant de converger fatalement à une seule racine.

La machine $\mathsf{Leader_1}$ contient les événements progress et $\mathsf{election_1}$ (raffinement de $\mathsf{election_0}$) suivants :

EVENT progress $\widehat{=}$
ANY
x, y
WHERE
$x \mapsto y \in g \wedge x \notin dom(tr)$
$y \notin dom(tr) \wedge g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$
THEN
$tr := tr \cup \{x \mapsto y\}$
END

EVENT $\mathsf{election_1}$ REFINES $\mathsf{election_0}$ $\widehat{=}$
ANY
x
WHERE
$g[\{x\}] = tr^{-1}[\{x\}]$
THEN
$sp, rt := tr, x$
END

Nous définissons une hypothèse d'équité L_1 sur $\mathsf{Leader_1}$, par des conditions d'équité faible sur les événements progress et $\mathsf{election_1}$, raffinement de $\mathsf{election_0}$: $L_1 = WF_v(\mathsf{progress}) \wedge WF_v(\mathsf{election_1})$. Nous montrons que $\mathsf{Leader_1}$ satisfait $P \rightsquigarrow Q$.

Démonstration. L'hypothèse d'équité faible sur l'événement progress permet de montrer $F(n+1) \rightsquigarrow F(n)$ (pour $n \geq 1$), en utilisant la règle WF1 (page 36) :

- Nous posons : $\mathsf{NEXT} \ \widehat{=} \ BA(\mathsf{progress})(v, v') \vee BA(\mathsf{election_1})(v, v') \vee (v = v')$
- Nous posons $R \ \widehat{=} \ F(n+1)$ et $S \ \widehat{=} \ F(n)$. Nous pouvons voir que l'événement progress fait décroître ce nombre et établit $S' \ \widehat{=} \ F'(n)$.
- Un raisonnement similaire à celui vu dans l'abstraction nous permet d'obtenir les prédicats suivants :
  - $R \wedge [\mathsf{NEXT}]_v \Rightarrow (R' \vee S')$
  - $R \wedge \langle \mathsf{NEXT} \wedge progress \rangle_v \Rightarrow S'$
  - $R \Rightarrow ENABLED \langle progress \rangle_v$
- La règle WF1 nous permet ensuite de prouver : $\Box[\mathsf{NEXT}]_v \wedge WF_v(\mathsf{progress}) \Rightarrow (R \rightsquigarrow S)$.

Nous avons montré $F(n+1) \rightsquigarrow F(n)$. En prenant l'ensemble des naturels positifs $\mathbb{N}_1$ comme ensemble bien fondé, nous pouvons utiliser la règle d'inférence treillis (LATTICE, page 36) pour démontrer la convergence vers $F(1)$: $(\exists n \cdot F(n)) \rightsquigarrow F(1)$. Nous avons aussi la propriété suivante : $P \Rightarrow (\exists n \cdot F(n))$, où $P \ \widehat{=} \ (acyclic(g) \wedge connected(g) \wedge sp = \varnothing \wedge rt \in ND$. En utilisant la règle de déduction (page 37), nous montrons que $P \rightsquigarrow (\exists n \cdot F(n))$. La règle de transitivité (page 37) nous permet de prouver que $P \rightsquigarrow F(1)$. Or, $F(1)$ est la condition d'observation de l'événement progress, permettant la satisfaction de $spanning(rt, sp, g)$, où sp est l'arbre tr et rt est un nœud x du graphe qui est la racine de tr : $F(1) \Rightarrow Q$. La machine $\mathsf{Leader_1}$ satisfait donc $P \rightsquigarrow Q$. □

Le diagramme décrit par la figure 4.5 résume ce premier raffinement du protocole de l'élection du leader :

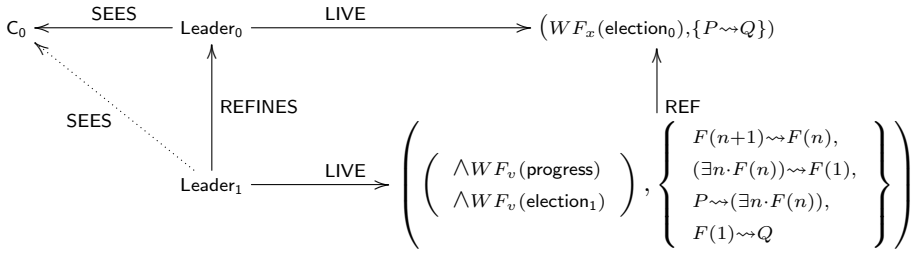


FIGURE 4.5 – Premier raffinement du protocole de l'élection du leader

En résumé, nous avons présenté ici un raffinement bénéficiant à la fois des avantages d'EVENT-B et de TLA : le raffinement EVENT-B nous garantit la préservation des propriétés de sûreté/invariance abstraites lors du raffinement, alors que les règles d'inférences relatives aux propriétés de vivacité, ainsi qu'aux conditions d'équité satisfaites par le modèle concret garantissent la préservation des propriétés de

vivacité caractérisant le modèle abstrait. La principale difficulté repose sur le choix des conditions d'équité à poser sur le modèle concret : si celle-ci est choisie de telle manière que celle posée sur le modèle abstrait raffiné puisse en être déduite, nous pouvons garantir aussi l'inclusion des traces équitables engendrées par le modèle concret dans celles engendrées par le modèle abstrait. Nous illustrons la méthodologie basée sur le paradigme « service-as-event » dans la section suivante, en utilisant le protocole d'élection du leader [4].

4.4 Méthodologie illustrée

L'objectif de cette section est de rendre la méthodologie « service-as-event » plus claire et compréhensible pour le lecteur, à l'aide d'un exemple, qui est le protocole de l'élection du leader dans un réseau [4].

Cependant, nous n'introduisons pas tout de suite cet exemple. Nous commençons d'abord par définir les diagrammes d'assertions [24, 10], permettant de représenter des propriétés de vivacité, sous des conditions d'équité. Nous nous servirons en effet de ces diagrammes lors des explications concernant l'application du paradigme « service-as-event » au protocole d'élection du leader dans un réseau [4].

4.4.1 Diagrammes d'assertions supports du raffinement

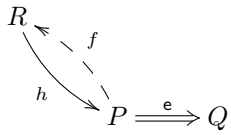
Les diagrammes d'assertions sont des méthodes graphiques, proches des diagrammes de prédicats [13] et des treillis de preuves introduits par Owicki et Lamport [28], et dont l'objectif principal est la représentation de (preuves de) propriétés de vivacité, en tenant compte d'hypothèses d'équité. Nous utilisons ces diagrammes pour principalement guider un utilisateur lors du raffinement de modèles. Ces *diagrammes d'assertions* sont construits en appliquant les règles d'inférence liées à l'opérateur *« leads to »* ($\rightsquigarrow$).

Définition 4.4.1. Soit $\mathcal{M}$ une machine Event-B, x les variables de $\mathcal{M}$, $I(x)$ l'invariant de $\mathcal{M}$, E l'ensemble des événements de $\mathcal{M}$, A un ensemble d'assertions, G un ensemble fini d'assertions appelées *conditions/gardes*, de la forme $g(x)$, V un ensemble de k étiquettes $\{(v_1, \prec_1), (v_2, \prec_2), \ldots, (v_k, \prec_k)\}$. Une étiquette de V associe une expression v_i à une relation binaire $\prec_i$, interprétée comme une relation d'ordre bien fondée.

Un diagramme d'assertions $D = (A, G, E, V, F, \mathcal{M})$ associé à $\mathcal{M}$ est un graphe dirigé dont les étiquettes des nœuds appartiennent à A et dont les arcs (pleins ou en pointillés) ont pour étiquettes des éléments soit de E, soit de E et V, soit de G. F est une fonction associant à chaque événement de E étiquetant un arc plein, une condition d'équité faible ($WF_x(e)$) ou forte ($SF_x(e)$). Nous notons qu'un événement $e \in E$ est observable à un nœud étiqueté par $P \in A$, si, et seulement si, un arc étiqueté par e, reliant directement le nœud P à un nœud quelconque existe.

Le diagramme d'assertions D satisfait les règles suivantes :

- Si P est relié à Q par un arc étiqueté par un événement $e \in E$, c'est à dire, soit $P \xrightarrow{e} Q$, soit $P \overset{e}{\Rightarrow} Q$, soit $P \overset{e}{\dashrightarrow} Q$, alors :
  - $\forall x, x' \cdot P(x) \wedge I(x) \wedge BA(e)(x, x') \Rightarrow Q(x')$
  - $\forall x \cdot P(x) \wedge I(x) \Rightarrow (\exists x' \cdot BA(e)(x, x'))$
- Si P est relié à Q par un arc qui n'est pas en pointillés, étiquetté par un événement $e \in E$, alors nous avons des règles supplémentaires (équité et vivacité) :
  - Si P est relié à Q par un arc simple plein, c'est-à-dire $P \xrightarrow{e} Q$, cela veut dire que :
    - Si des événements f ($f \neq e$) sont observables lorsque P est vrai, leurs observations ne falsifient pas le prédicat P, condition d'activation de l'événement e (ces événements f étiquettent soit des transitions réflexives sur le nœud étiqueté par P, ou peuvent ne pas étiqueter un arc sortant du nœud étiqueté par P) ou peuvent conduire à Q.
    - La condition d'équité posée sur e est $WF_x(e)$.
  - Si P est relié à Q par un arc double, et est relié à un autre nœud R ($R \neq Q$), par un arc en pointillés étiqueté par un événement f ($f \neq e$), et que tous les arcs partant de R, sont pleins, étiquetés par des événements h ($h \neq e$ et $h \neq f$), et ramènent à P, comme illustré ci-dessous :

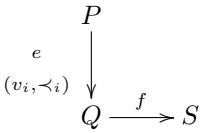


alors :
- Si d'autres événements l ($l \neq e$, $l \neq h$ et $l \neq f$) sont observables dans des états satisfaisant $(P \vee R)$ et que leurs observations ne falsifient pas $(P \vee R)$, ou conduisent à Q.
- La condition d'équité posée sur e est $SF_x(e)$.

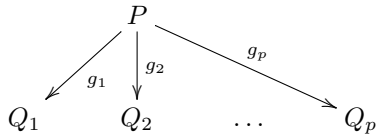
— Dans les deux cas, la propriété $P \rightsquigarrow Q$ est satisfaite.

— Si P est relié à Q par un arc simple plein étiqueté par un événement $e \in E$ et une étiquette $(v_i, \prec_i) \in V$, c'est-à-dire $P \xrightarrow[(v_i,\prec_i)]{e} Q$, alors nous avons des règles supplémentaires (liée à la décroissance de v_i) :
- $\forall x, x' \cdot P(x) \wedge I(x) \wedge BA(e)(x, x') \Rightarrow (v_i(x') \prec_i v_i(x))$
- Si $\bar{x}$ est une valeur des variables de $\mathcal{M}$ telle que $v_i(\bar{x})$ est la valeur minimale de l'expression v_i, et $Q(\bar{x})$, alors une transition, étiquetée par un événement f, et partant que Q et allant vers un nœud S, telle que :
  - $Q(\bar{x}) \wedge I(\bar{x}) \Rightarrow (\exists y \cdot BA(f)(\bar{x}, y))$ est satisfaite,
  - $\forall y \cdot Q(\bar{x}) \wedge I(\bar{x}) \wedge BA(f)(\bar{x}, y) \Rightarrow S(y)$ est satisfaite,
  - $WF_x(f)$ est la condition d'équité posée sur f

  peut être observée, comme illustré ci-dessous :



— Si P est relié à $Q_1, \ldots, Q_p$ par des arcs simples pleins, comme illustré ci-dessous :



alors :
- Chaque arc de P vers Q_i a pour label une garde $g_i \in G$.
- Pour tout i appartenant à $1..p$, les conditions suivantes sont vérifiées
  $\begin{cases} \forall x \cdot P(x) \wedge I(x) \wedge g_i(x) \Rightarrow Q_i(x) \\ \forall x, j.j \in 1..p \wedge j \neq i \wedge P(x) \wedge I(x) \wedge g_i(x) \Rightarrow \neg g_j(x) \end{cases}$
- $\forall x \cdot P(x) \wedge I(x) \Rightarrow (\exists i \cdot i \in 1..p \wedge g_i(x))$

Les diagrammes d'assertions sont utilisés pour inférer la correction totale d'un algorithme construit pas-à-pas par raffinement et à l'aide des règles d'inférence relatives aux propriétés de vivacité utilisant l'opérateur *« leads to »* ($\rightsquigarrow$). Par conséquent, un diagramme d'assertions construit pour un problème donné est correct par rapport aux exigences requises pour le problème. Les diagrammes d'assertions possèdent des propriétés [24, 7] que nous énumérons ici :

Propriété 4.4.1. Soit $\mathcal{M}$ une machine Event-B et $D = (A, G, E, V, F, \mathcal{M})$ un diagramme d'assertions associé.

1. Si $\mathcal{M}$ satisfait $P \rightsquigarrow Q$ et $Q \rightsquigarrow R$, alors $\mathcal{M}$ satisfait $P \rightsquigarrow R$.
2. Si $\mathcal{M}$ satisfait $P \rightsquigarrow Q$ et $R \rightsquigarrow Q$, alors $\mathcal{M}$ satisfait $(P \vee R) \rightsquigarrow Q$.
3. Si I est l'invariant de $\mathcal{M}$ et si $\mathcal{M}$ satisfait $P \wedge I \rightsquigarrow Q$, alors $\mathcal{M}$ satisfait $P \rightsquigarrow Q$.
4. Si I est l'invariant de $\mathcal{M}$ et si $\mathcal{M}$ satisfait $P \wedge I \Rightarrow Q$, alors $\mathcal{M}$ satisfait $P \rightsquigarrow Q$.
5. Si $P \xrightarrow{e} Q$ ou $P \overset{e}{\Rightarrow} Q$ est un lien de D pour la machine $\mathcal{M}$, alors $\mathcal{M}$ satisfait $P \rightsquigarrow Q$.

6. Si $\mathcal{M}$ satisfait $P(x) \rightsquigarrow P(x')$ et que n'importe quel chemin dans D entre $P(x)$ et $P(x')$ fait décroître la valeur d'une expression v, tel que $v(x') \prec v(x)$, alors $\mathcal{M}$ satisfait $(\exists y \cdot P(y)) \rightsquigarrow P(\bar{x})$, où $\bar{x}$ est une valeur des variables de $\mathcal{M}$ telle que $v(\bar{x})$ est la valeur minimale de l'expression v.
7. Si P et Q sont deux nœuds de D, tels qu'il existe un chemin dans D de P à Q et que n'importe quel chemin à partir de P peut être étendu en un chemin contenant Q, alors $\mathcal{M}$ satisfait $P \rightsquigarrow Q$.
8. Si I, U, V, P et Q sont des assertions telles que I est l'invariant de $\mathcal{M}$; $P \wedge I \Rightarrow U$; $V \Rightarrow Q$ et qu'il existe un chemin de U à V et chaque chemin partant de U conduit à V ; alors $\mathcal{M}$ satisfait $P \rightsquigarrow Q$.

Nous prouvons [24] maintenant ces propriétés.

— Preuve de la propriété 1 :
Le modèle $\mathcal{M}$ satisfait $P \rightsquigarrow Q$ et $Q \rightsquigarrow R$. Par transitivité, nous déduisons que $\mathcal{M}$ satisfait $P \rightsquigarrow R$. □

— Preuve de la propriété 2 :
Le modèle $\mathcal{M}$ satisfait $P \rightsquigarrow Q$ et $R \rightsquigarrow Q$. Par application de la règle de disjonction, nous déduisons que $\mathcal{M}$ satisfait $(P \vee R) \rightsquigarrow Q$. □

— Preuve de la propriété 3 :
I est l'invariant de $\mathcal{M}$ et $\mathcal{M}$ satisfait $P \wedge I \rightsquigarrow Q$. Par application de la règle de simplification, nous déduisons que $\mathcal{M}$ satisfait $P \rightsquigarrow Q$. □

— Preuve de la propriété 4 :
I est l'invariant de $\mathcal{M}$ et $\mathcal{M}$ satisfait $P \wedge I \Rightarrow Q$. Par application de la règle de déduction, nous déduisons que $\mathcal{M}$ satisfait $P \wedge I \rightsquigarrow Q$. En appliquant ensuite exactement le même raisonnement que celui de la preuve de la propriété 4, nous prouvons que $\mathcal{M}$ satisfait $P \rightsquigarrow Q$. □

— Preuve de la propriété 5 :

⟨1⟩1. Cas de $WF_x(e)$ et $P \xrightarrow{e} Q$. Une hypothèse d'équité faible est posée sur un événement e, étiquette d'un arc reliant un nœud étiquetté par P à un autre étiqueté par Q, dans le cas suivant **(1)** :
— Si des événements f $(f \neq e)$ sont observables lorsque P est vrai, leurs observations ne falsifient pas le prédicat P, condition d'activation de l'événement e (ces événements f étiquettent soit des transitions réflexives sur le nœud étiqueté par P, ou peuvent ne pas étiqueter un arc sortant du nœud étiqueté par P) ou conduisent à Q.

Nous posons $\mathsf{NEXT} \triangleq BA(e)(x,x') \wedge BA(f)(x,x') \wedge (x = x')$. Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$P \wedge [\mathsf{NEXT}]_x \Rightarrow (P' \vee Q')$$

Soit :
— $P \wedge BA(e)(x,x') \Rightarrow (P' \vee Q')$: l'observation de l'événement e à un état satisfaisant P, conduit soit à un nouvel état satisfaisant P ou à un autre satisfaisant Q.
— $P \wedge BA(f)(x,x') \Rightarrow (P' \vee Q')$: l'observation d'un 'événement f à un état satisfaisant P, conduit soit à un nouvel état satisfaisant P ou à un autre satisfaisant Q.
— $P \wedge (x = x') \Rightarrow (P' \vee Q')$: un bégaiement, à un état satisfaisant P, peut nous conduire à un état satisfaisant P ou à un état satisfaisant Q.

La propriété suivante **(3)** : $P(x) \wedge I(x) \wedge BA(e)(x,x') \Rightarrow Q'$, ainsi que la condition de faisabilité **(4)** : $P(x) \wedge I(x) \Rightarrow (\exists x' \cdot BA(e)(x,x'))$ sont satisfaites par l'événement e. La propriété **(3)** nous permet de déduire $P \wedge \langle \mathsf{NEXT} \wedge e \rangle_x \Rightarrow Q'$ et **4** nous permet de déduire $P \Rightarrow ENABLED\langle e \rangle_x$, où $ENABLED\langle e \rangle_x \triangleq (\exists y \cdot BA(e)(x,y))$. L'application de la règle d'inférence WF1 nous donne $\Box[\mathsf{NEXT}]_x \wedge WF_x(\mathsf{e}) \Rightarrow (P \rightsquigarrow Q)$. Finalement, nous obtenons $\mathsf{INIT}(x) \wedge \Box[\mathsf{NEXT}]_x \wedge WF_x(\mathsf{e}) \Rightarrow (P \rightsquigarrow Q)$. □

⟨1⟩2. Cas de $SF_x(e)$ et $P \overset{e}{\Rightarrow} Q$. Une hypothèse d'équité forte est posée sur un événement e, étiquette d'un arc reliant un nœud étiquetté par P à un autre étiqueté par Q, dans le cas suivant **(1)** :
— Si P est relié à un autre nœud R $(R \neq Q)$, par un arc en pointillés étiqueté par un événement f $(f \neq e)$, que tous les arcs partant de R, sont pleins, étiquetés par des événements h

($h \neq e$ et $h \neq f$), et ramènent à P ; si d'autres événements l ($l \neq e$, $l \neq h$ et $l \neq f$) sont observables dans des états satisfaisant $(P \vee R)$, et leurs observations ne falsifient pas $(P \vee R)$ ou conduisent à Q.

Nous posons $\mathsf{NEXT} \ \widehat{=} \ BA(e)(x,x') \wedge BA(f)(x,x') \wedge BA(h)(x,x') \wedge BA(l)(x,x') \wedge (x = x')$.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(P \vee R) \wedge [\mathsf{NEXT}]_x \Rightarrow ((P' \vee R') \vee Q')$$

Soit :

— $(P \vee R) \wedge BA(e)(x,x') \Rightarrow ((P' \vee R') \vee Q')$: l'observation de l'événement e à un état satisfaisant $(P \vee R)$, conduit à un nouvel état satisfaisant soit P, soit R, soit Q.
— $(P \vee R) \wedge BA(f)(x,x') \Rightarrow ((P' \vee R') \vee Q')$: l'observation d'un événement f à un état satisfaisant $(P \vee R)$, conduit à un nouvel état satisfaisant soit P, soit R, soit Q.
— $(P \vee R) \wedge BA(h)(x,x') \Rightarrow ((P' \vee R') \vee Q')$: l'observation d'un événement h à un état satisfaisant $(P \vee R)$, conduit à un nouvel état satisfaisant soit P, soit R, soit Q.
— $(P \vee R) \wedge BA(l)(x,x') \Rightarrow ((P' \vee R') \vee Q')$: l'observation d'un événement l à un état satisfaisant $(P \vee R)$, conduit à un nouvel état satisfaisant soit P, soit R, soit Q.
— $(P \vee R) \wedge (x = x') \Rightarrow ((P' \vee R') \vee Q')$: un bégaiement, à un état satisfaisant $(P \vee R)$, peut nous conduire à un état satisfaisant soit P, soit R ou à un état satisfaisant Q.

La propriété suivante **(3)** : $P(x) \wedge I(x) \wedge BA(e)(x,x') \Rightarrow Q'$, ainsi que la condition de faisabilité **(4)** : $P(x) \wedge I(x) \Rightarrow (\exists x' \cdot BA(e)(x,x'))$ sont satisfaites par l'événement e. La propriété **(3)** nous permet de déduire $(P \vee R) \wedge \langle \mathsf{NEXT} \wedge e \rangle_x \Rightarrow Q'$. Les propriétés **(3)** et **(4)** nous permettent de déduire que tant que Q n'est pas satisfaite, alors l'événement e sera fatalement activable, soit : $(\neg Q) \Rightarrow \Diamond ENABLED\langle e \rangle_x$, où $ENABLED\langle e \rangle_x \ \widehat{=} \ (\exists y \cdot BA(e)(x,y))$. Nous pouvons en déduire : $\Box(P \vee R) \wedge \Box[\mathsf{NEXT}]_x \Rightarrow \Diamond ENABLED\langle e \rangle_x$. L'application de la règle d'inférence SF1 nous donne $\Box[\mathsf{NEXT}]_x \wedge SF_x(\mathsf{e}) \Rightarrow ((P \vee R) \rightsquigarrow Q)$. Finalement, nous obtenons $\mathsf{INIT}(x) \wedge \Box[\mathsf{NEXT}]_x \wedge SF_x(\mathsf{e}) \Rightarrow ((P \vee R) \rightsquigarrow Q)$. $\mathcal{M}$ satisfait donc $P \rightsquigarrow Q$. □

⟨1⟩3. Q.E.D.

PREUVE : Les étapes ⟨1⟩1 et ⟨1⟩2 nous permettent de montrer que $\mathcal{M}$ satisfait $P \rightsquigarrow Q$. □

— PREUVE DE LA PROPRIÉTÉ 6 :

$\mathcal{M}$ satisfait $P(x) \rightsquigarrow P(x')$ et n'importe quel chemin dans D entre $P(x)$ et $P(x')$ fait décroître la valeur d'une expression v, tel que $v(x') \prec v(x)$. Nous rappelons que v est associée à une relation binaire $\prec$, interprétée comme une relation d'ordre bien fondée. L'application de la règle d'inférence treillis (LATTICE) nous permet de déduire que $\mathcal{M}$ satisfait $(\exists y \cdot P(y)) \rightsquigarrow P(\bar{x})$, où $\bar{x}$ est une valeur des variables de $\mathcal{M}$ telle que $v(\bar{x})$ est la valeur minimale de l'expression v, permettant ainsi de prouver la convergence vers $P(\bar{x})$. □

— PREUVE DE LA PROPRIÉTÉ 7 :

P et Q sont deux nœuds de D, tels qu'il existe un chemin dans D de P à Q et que n'importe quel chemin à partir de P peut être étendu en un chemin contenant Q.

Nous procédons par induction sur le chemin de P à Q, pour démontrer que $\mathcal{M}$ satisfait $P \rightsquigarrow Q$.

⟨1⟩1. La longueur du chemin de P à Q est 1.

PREUVE : La deuxième contrainte sur le chemin à partir de P nous permet d'affirmer que P a un seul unique successeur possible qui est Q. Le label de l'arc reliant P et Q est une garde ou un événement. □

⟨2⟩1. Le label est une garde g.

PREUVE : La définition du diagramme de raffinement D nous permet de dériver la propriété $P \wedge I(x) \wedge g(x) \Rightarrow Q$, ainsi que $P \wedge I(x) \Rightarrow g(x)$. Par conséquent, nous avons $P \wedge I(x) \Rightarrow Q$. La propriété 3 du diagramme de raffinement D nous permet de déduire que $\mathcal{M}$ satisfait $P \rightsquigarrow Q$. □

⟨2⟩2. Le label est un événement e.

PREUVE : La construction de D pour $\mathcal{M}$ nous permet de déduire que $\mathcal{M}$ satisfait $P \rightsquigarrow Q$. □

⟨2⟩3. Q.E.D.

PREUVE : Les cas ⟨2⟩1 et ⟨2⟩2 nous permettent de prouver que $\mathcal{M}$ satisfait $P \rightsquigarrow Q$. □

⟨1⟩2. La longueur du chemin de P à Q est n et nous supposons que la propriété à prouver est vérifiée pour les chemins de longueur $n - 1$.

PREUVE : La longueur du chemin de P à Q est $n > 0$. Alors, il existe soit une assertion R et un lien de P à R dont l'étiquette est un événement e, soit des assertions $R_1, \ldots, R_k$ et des gardes $g_1, \ldots, g_k$, telles que $P \xrightarrow{g_i} R_i$, pour $i \in 1..k$. □

⟨2⟩1. $\mathcal{M}$ satisfait $P \rightsquigarrow S$, où S est soit $\bigvee_{i \in 1..k} R_i$ ou R.

PREUVE :

⟨3⟩1. Il existe une assertion R et un lien de P à R étiquetté par un événement e.

PREUVE : La construction de D pour $\mathcal{M}$ nous permet de déduire que $\mathcal{M}$ satisfait $P \rightsquigarrow R$. □

⟨3⟩2. il existe des assertions $R_1, \ldots, R_k$ et des gardes $g_1, \ldots, g_k$, telles que $P \xrightarrow{g_i} R_i$, pour $i \in 1..k$.

PREUVE : En utilisant le même raisonnement qu'en ⟨2⟩1, nous pouvons déduire que $\mathcal{M}$ satisfait $P \rightsquigarrow \bigvee_{i \in 1..k} R_i$. □

⟨3⟩3. Q.E.D.

PREUVE : Les étapes ⟨3⟩1 et ⟨3⟩2 nous permettent de montrer que $\mathcal{M}$ satisfait $P \rightsquigarrow S$, où S est soit $\bigvee_{i \in 1..k} R_i$ ou R. □

⟨2⟩2. $\mathcal{M}$ satisfait $S \rightsquigarrow Q$, où S est soit $\bigvee_{i \in 1..k} R_i$ ou R.

PREUVE : En considérant les deux cas, nous remarquons que chaque R_i est relié à Q par un chemin de longueur plus petite que n et que chaque chemin conduit à Q. Nous pouvons appliquer l'hypothèse d'induction pour déduire que $\mathcal{M}$ satisfait $R_i \rightsquigarrow Q$ pour tout $i \in 1..k$. En appliquant la règle de confluence, nous déduisons que $\mathcal{M}$ satisfait $\bigvee_{i \in 1..k} R_i \rightsquigarrow Q$. En appliquant la transitivité, nous prouvons que $\mathcal{M}$ satisfait $P \rightsquigarrow Q$. □

⟨2⟩3. Q.E.D.

PREUVE : L'induction et les étapes inductives ⟨1⟩1 et ⟨1⟩2 nous permettent de montrer que $\mathcal{M}$ satisfait $P \rightsquigarrow Q$. □

— PREUVE DE LA PROPRIÉTÉ 8 :
I, U, V, P et Q sont des assertions telles que I est l'invariant de $\mathcal{M}$; $P \wedge I \Rightarrow U$; $V \Rightarrow Q$ et qu'il existe un chemin de U à V et chaque chemin partant de U conduit à V. Par application des règles de déduction et de simplification, nous pouvons déduire que $\mathcal{M}$ satisfait $P \rightsquigarrow U$ et $V \rightsquigarrow Q$. En réutilisant le même raisonnement que lors de la preuve de la propriété 7, nous déduisons que $\mathcal{M}$ satisfait $U \rightsquigarrow V$. Par transitivité, nous déduisons que $\mathcal{M}$ satisfait $P \rightsquigarrow Q$. □

Pour conclure cette section sur les diagrammes d'assertions, nous disons qu'associés à des machines EVENT-B, ils peuvent être utilisés pour montrer la satisfaction de propriétés de vivacités. La justification de ces diagrammes est liée aux propriétés de vivacité qui y sont exprimées et aux règles d'inférences relatives à ces propriétés qui permettent de construire les diagrammes.

4.4.2 Révision de l'élection du leader (suite)

Nous continuons l'illustration de l'utilisation du paradigme « service-as-event », à l'aide de l'algorithme de l'élection du leader dans un réseau [4], en le reprenant à partir du second raffinement.

Exemple 4.4.1. *Deuxième raffinement du protocole de l'élection du leader.*

Par rapport au modèle précédent, ce raffinement Leader_2 détaille le processus qui permet à un nœud y de devenir le parent d'un nœud x. Cette machine contient des **variables** qui sont : $w = (sp, rt, tr, req, ack, cnt)$. Nous nous servons du diagramme d'assertions 4.6 [10, 24] pour présenter les propriétés de vivacité exprimant les étapes intermédiaires conduisant de $F(n+1)$ à $F(n)$:

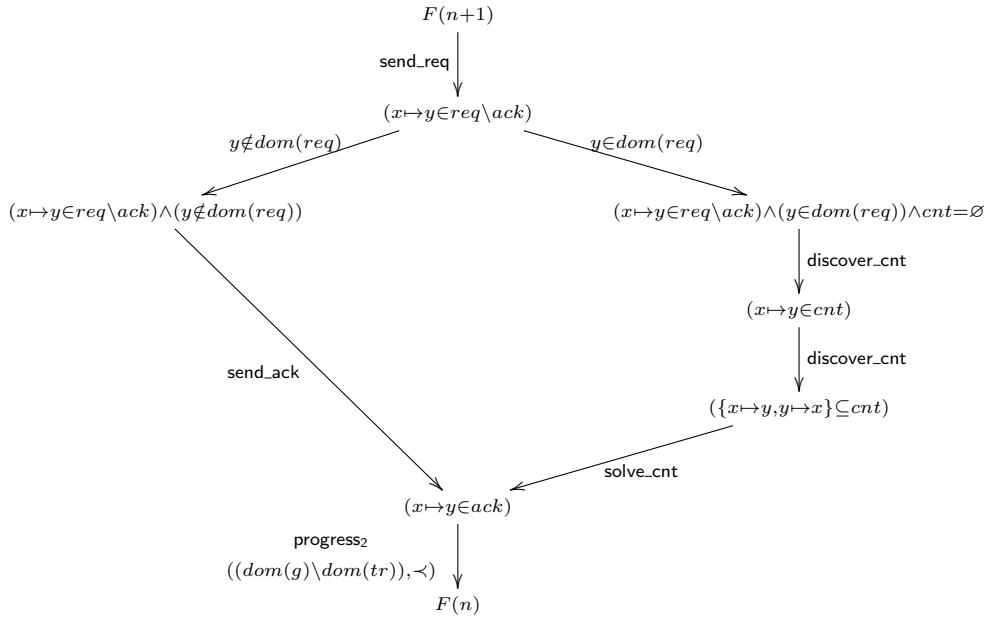


FIGURE 4.6 – Diagramme exprimant $F(n+1) \rightsquigarrow F(n)$

— Quand un nœud x a détecté qu'il est parent de tous ses voisins, sauf d'un seul qui est le nœud y, il envoie à y une requête lui demandant de devenir son parent (événement send_req).
— Le nœud y ayant reçu la requête renvoie ensuite au nœud x une confirmation (événement send_ack).
— Le nœud x termine en établissant que le nœud y est son parent (événement progress), réduisant ainsi le nombre d'arbres de 1.
— Ce raffinement introduit aussi le problème de la contention : les requêtes de demande de parent que s'envoient les deux nœuds adjacents x et y se croisent, auquel cas, l'élection ne peut se faire.
  — Le nœud x découvre la contention : il a envoyé une requête à y, il n'a pas encore reçu de confirmation. Il reçoit à la place une requête de y (événement discover_cnt).
  — Le nœud y découvre la contention de la même manière.
  — La contention étant ainsi découverte, la résolution de la contention (événement solve_cnt) a lieu : nous présentons ici une abstraction qui permet de choisir le nœud x en tant que parent du nœud y.
  — De la même manière que précédemment, le nœud x termine en établissant que le nœud y est son parent (événement progress$_2$), réduisant ainsi le nombre d'arbres de 1.

Nous définissons une hypothèse d'équité L_2 sur Leader$_2$, par des conditions d'équité faible sur les événements send_req, send_ack, discover_cnt, solve_cnt, progress$_2$ et election$_2$, raffinements de progress et election$_1$:

$$\begin{aligned} L_2 = \quad & \wedge WF_w(\text{send\_req}) \\ & \wedge WF_w(\text{send\_ack}) \\ & \wedge WF_w(\text{discover\_cnt}) \\ & \wedge WF_w(\text{solve\_cnt}) \\ & \wedge WF_w(\text{progress}_2) \\ & \wedge WF_w(\text{election}_2) \end{aligned}$$

Nous montrons que Leader$_2$ satisfait $P \rightsquigarrow Q$.

Démonstration. Le diagramme d'assertions 4.6 nous montre que $F(n+1) \rightsquigarrow F(n)$ est satisfaite par la machine Leader$_2$: $F(n+1)$ et $F(n)$ sont deux nœuds du diagramme et n'importe quel chemin à partir de $F(n+1)$ peut être étendu en un chemin contenant $F(n)$. En utilisant la règle d'inférence treillis (LATTICE, page 36) nous pouvons démontrer la convergence vers $F(1)$: $(\exists n \cdot F(n)) \rightsquigarrow F(1)$. Or, nous savons que $P \rightsquigarrow (\exists n \cdot F(n))$ et $F(1) \rightsquigarrow Q$. Nous montrons ainsi que le modèle Leader$_2$ satisfait $P \rightsquigarrow Q$. □

Le diagramme décrit par la figure 4.7 résume le second raffinement du protocole de l'élection du leader. Nous avons décomposé la propriété $F(n+1) \rightsquigarrow F(n)$ à l'aide des règles d'inférence relatives au propriétés de vivacité utilisant l'opérateur ($\rightsquigarrow$). Nous posons $R1 \ \widehat{=}\ (x \mapsto y \in req \setminus ack) \wedge (y \notin dom(req))$, $R2 \ \widehat{=}\ (x \mapsto y \in req \setminus ack) \wedge (y \in dom(req))$, $S \ \widehat{=}\ (x \mapsto y \in ack)$ et $C \ \widehat{=}\ (\{x \mapsto y, y \mapsto x\} \subseteq cnt)$.

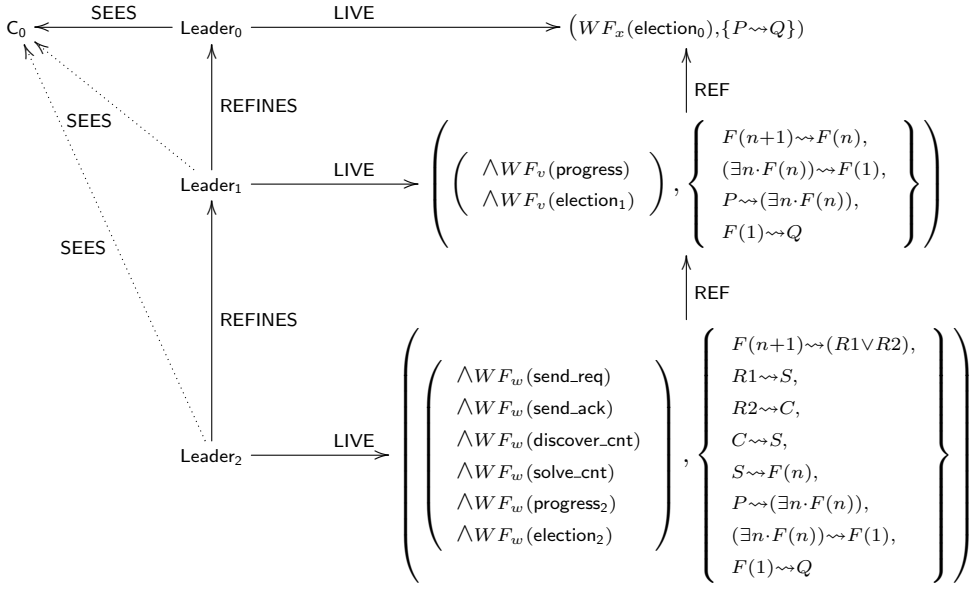


FIGURE 4.7 – Second raffinement du protocole de l'élection du leader

Les propriétés de vivacités caractérisant Leader_2 et présentées par les diagrammes 4.6 et 4.7 nous ont permis de définir les événements de Leader_2, principalement les nouveaux et ceux modifiés lors du raffinement :

EVENT send_req $\widehat{=}$
ANY
x, y
WHERE
$x \mapsto y \in g \wedge x \notin dom(tr)$
$y \mapsto x \notin tr$
$g[\{x\}] = tr^{-1}[\{x\}] \cup \{y\}$
$y \mapsto x \notin ack$
$x \notin dom(req)$
THEN
$req := req \cup \{x \mapsto y\}$
END

EVENT send_ack $\widehat{=}$
ANY
x, y
WHERE
$x \mapsto y \in req \setminus ack$
$y \notin dom(req)$
THEN
$ack := ack \cup \{x \mapsto y\}$
END

EVENT discover_cnt $\widehat{=}$
ANY
x, y
WHERE
$x \mapsto y \in req \setminus ack$
$y \in dom(req)$
THEN
$cnt := cnt \cup \{x \mapsto y\}$
END

EVENT solve_cnt $\widehat{=}$
ANY
x, y
WHERE
$x \mapsto y \in cnt \wedge y \mapsto x \in cnt$
THEN
$ack := ack \cup \{x \mapsto y\}$
$req := \{y\} \lhd req$
$cnt := \varnothing$
END

EVENT progress2 REFINES progress $\widehat{=}$
ANY
x, y
WHERE
$x \mapsto y \in ack \wedge x \notin dom(tr)$
THEN
$tr := tr \cup \{x \mapsto y\}$
END

Nous nous sommes servis des sections précédentes pour introduire le paradigme *« service-as-event »* et illustrer son application, à travers le protocole de l'élection du leader [4]. Ces sections nous ont aussi permis d'établir comment nous nous servons des propriétés de vivacité pour le développement de modèles EVENT-B et pour guider le processus de raffinement.

4.5 Synthèse du paradigme « *service-as-event* »

Nous décrivons de manière synthétique l'application du paradigme *service-as-event*. Soit un algorithme $\mathcal{A}$ réparti étudié, une machine abstraite M0 avec des variables x, un invariant $I(x)$, et son raffinement M1 avec des variables y et un invariant de collage $J(x, y)$.

— La première étape est la construction de la machine M0 modélisant de manière abstraite l'algorithme $\mathcal{A}$.

1. Identifier l'ensemble $S \mathrel{\widehat{=}} \{s_0, s_1, \ldots, s_m\}$ des services s_i fournis par $\mathcal{A}$.
2. Exprimer chaque service s_i par une propriété de vivacité $P_i \rightsquigarrow Q_i$, où P_i est la condition d'invocation de s_i et Q_i la condition vérifiée quand s_i termine. Cette étape revient à représenter l'algorithme $\mathcal{A}$ par un ensemble de propriétés de vivacité $\Phi_0 \mathrel{\widehat{=}} \{P_0 \rightsquigarrow Q_0, P_1 \rightsquigarrow Q_1, \ldots, P_m \rightsquigarrow Q_m\}$, caractérisant les services offerts par l'algorithme.
3. Associer chaque propriété $P \rightsquigarrow Q$ de Φ_0 à un événement de M0 qui sera détaillé plus tard.
4. Déterminer l'hypothèse d'équité L_0 à poser sur M0.
5. Montrer que chaque propriété $P \rightsquigarrow Q$ de Φ_0 est satisfaite par la machine M0 sous hypothèse d'équité L_0 (Relation LIVE, voir figure 4.8), à l'aide des règles d'inférences relatives aux propriétés de vivacité de type « leads to ». Guider les preuves à l'aide des diagrammes d'assertions.
6. Guider le développement de la machine M0 à l'aide de Φ_0 et des diagrammes d'assertions : il s'agit ici d'écrire les événements de la machine M0.

— La seconde étape est le raffinement de la machine M0 par M1.

1. Décomposer les propriétés $P \rightsquigarrow Q$ de Φ_0 à l'aide des règles d'inférences relatives aux propriétés de vivacité de type « leads to », pour obtenir un nouvel ensemble $\Phi_1 \mathrel{\widehat{=}} \{P_0 \rightsquigarrow Q_0, P_1 \rightsquigarrow Q_1, \ldots, P_n \rightsquigarrow Q_n\}$.
2. Reprendre les étapes 3, 4, 5 du point précédent (en remplaçant M0 par M1, Φ_0 par Φ_1, L_0 par L_1).
3. Montrer que pour tout P, Q, tels que $(P \rightsquigarrow Q) \in \Phi_0$, nous avons $(\mathcal{S}pec(\mathsf{M}_1), \Phi_1) \vdash (P \rightsquigarrow Q)$ (Relation REF, voir figure 4.8).
4. Reprendre l'étape 6 du point précédent (en remplaçant M0 par M1, Φ_0 par Φ_1, L_0 par L_1).

— Pour construire d'autres niveaux de raffinements, il suffit d'appliquer la partie concernant M1.

Le diagramme suivant, présenté par la figure 4.8 et vu dans les sections précédentes, explique notre méthode de raffinement :

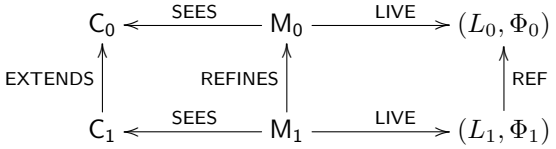


FIGURE 4.8 – Raffinement : synthèse

4.6 Conclusion

Le paradigme *« service-as-event »*, pour la modélisation et la vérification des algorithmes répartis, consiste dans un premier temps à identifier les services offerts par un algorithme réparti. Il s'agit ensuite d'exprimer chaque service à l'aide d'une ou de plusieurs propriété de vivacité de type $P \rightsquigarrow Q$ et de construire un modèle abstrait correspondant à ces services, qui sous certaines hypothèses d'équité, satisfait les propriétés de vivacité exprimant les services offerts par l'algorithme. Le raffinement du modèle abstrait en un modèle plus concret est guidé par les règles d'inférence : nous nous servons de ces dernières pour « détailler » les propriétés de vivacité « abstraites » et obtenir ainsi des propriétés de vivacité plus « concrètes » (les propriétés « abstraites » doivent être dérivables à partir de ces dernières, en utilisant les règles d'inférences). Les propriétés de vivacité « concrètes » ainsi obtenues guident la construction

du modèle concret. Cette étape est assez difficile, notamment la détermination des hypothèses d'équité « concrètes » qui vont contraindre les traces d'exécution du modèle concret. En effet, les nouvelles hypothèses d'équité doivent permettre non seulement de satisfaire les propriétés de vivacité « concrètes », mais aussi préserver celles « abstraites » lors du raffinement.

Notre méthodologie repose sur utilisation *superposée* de la méthode Event-B et de la logique TLA. Nous modélisons les algorithmes répartis à l'aide de la méthode Event-B et nous l'utilisons pour prouver des propriétés de sûreté. Nous utilisons TLA en considérant les modèles Event-B en tant que systèmes d'actions contraintes par des conditions d'équité : nous sommes ainsi en mesure de prouver des propriétés de vivacité. Notre méthodologie se rapproche ainsi de celle proposée par Méry et Poppleton dans [25]. Il faut aussi remarquer que nous ne faisons aucune extension modifiant le langage Event-B. Notre méthodologie diffère ainsi de celles proposées par Hoang et al, qui étendent la méthode Event-B en en modifiant le langage. Hoang et Abrial [17] proposent une définition du *leads to*, basée sur une transition d'événements : si une machine M satisfait $P \rightsquigarrow Q$, des obligations de preuve, montrant que les événements activables en P établissent Q, sont générées et doivent être déchargées ; des obligations de preuve basées sur l'utilisation de variants permettent de prouver la fatalité ($\Box\Diamond P$) ou encore la persistance ($\Diamond\Box P$). Par rapport à la méthodologie que nous décrivons dans ce chapitre, le raisonnement sur les propriétés de vivacité en Event-B proposée par Hoang et Abrial dans [17] n'inclue pas de prise en compte des conditions d'équité, et la préservation des propriétés de vivacité durant le raffinement n'est pas garantie. Hoang et Hudon [19] proposent une extension de la méthode Event-B, inspirée par UNITY [14], lui donnant ainsi une sémantique de traces et permettant de définir des hypothèses d'équité faibles et fortes au niveau des événements. Contrairement à ce que nous proposons, Hoang et Hudon ne distinguent pas la séparation entre les spécifications temporelles, exprimées à l'aide de propriétés de vivacité et de conditions d'équité, et les spécifications Event-B. Le raisonnement sur les propriétés temporelles est inclus dans les modèles Event-B. De manière assez similaire à ce que nous proposons, leur méthodologie nécessite la redéfinition à chaque étape raffinement des hypothèses d'équité, pour préserver les propriétés de vivacité. Une des différences principales se situe aussi sur le fait que nous avons choisi de baser notre méthodologie sur la logique TLA, plus générale qu'UNITY, qui représente seulement un fragment de la logique temporelle [22]. Nous proposons ici une approche explicite basée sur la logique TLA, pour la prise en compte des propriétés de vivacité et d'équité. L'approche habituelle en Event-B est de supposer qu'un ordonnanceur choisit équitablement parmi les événements activables, à tout moment. Ce genre d'ordonnanceur peut être mis en place dans les modèles à l'aide de compteurs, d'indicateurs permettant de définir des priorités pour les événements [6, 11].

Nous nous intéressons aussi à la possibilité de la *réutilisation* des services (ou de certaines parties de ces services) identifiés lors d'anciens développements : ces services ou certaines phases/étapes de ces services peuvent être réutilisés, notamment pour construire de nouveaux systèmes [8], par instanciation puis spécialisation (ajout de détails par raffinement) par rapport au problème étudié. Il s'agit ici d'une notion proche de celle des patrons de conception proposée par Abrial et al [5, 18], car notre objectif est non seulement de réutiliser les éléments de modèles Event-B modélisant ces services, mais également des propriétés qu'ils présentent et des preuves de correction de ces dernières. Nous implémentons cette réutilisation à travers en considérant des modèles en tant que patrons de conception [5, 18] (voir page 31) et en utilisant les outils fournis par la plateforme Rodin pour l'instanciation de ces patrons (greffons Refactory Framework [29], Pattern [18]).

Les chapitres suivants illustrent l'application du paradigme *« service-as-event »* au développement formel par raffinement de systèmes et d'algorithmes répartis, notamment, le protocole de routage Anycast RP, les réseaux-sur-puces, les algorithmes de « snapshot » et d'auto-stabilisation.

Bibliographie

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2) :253–284, May 1991.

[2] J.-R. Abrial. Extending b without changing it (for developing distributed systems). In H. Habrias, editor, *1st Conference on the B method [16]*, pages 169–190, Nov. 1996.

[3] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, 2010.

[4] J.-R. Abrial, D. Cansell, and D. Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3) :215–227, 2003.

[5] J.-R. Abrial and T. S. Hoang. Using design patterns in formal methods : An event-b approach. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigün, editors, *ICTAC*, volume 5160 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2008.

[6] J.-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In *B98*, pages 83–128, 1998.

[7] M. B. Andriamiarina. Stepwise development of distributed algorithms. In *FM 2011 : Doctoral Symposium*, 2011.

[8] M. B. Andriamiarina, H. Daoud, M. Belarbi, D. Méry, and C. Tanougast. Formal Verification of Fault Tolerant NoC-based Architecture. In *First International Workshop on Mathematics and Computer Science (IWMCS2012)*, Tiaret, Algérie, Dec. 2012.

[9] M. B. Andriamiarina, D. Méry, and N. K. Singh. Revisiting snapshot algorithms by refinement-based techniques. In H. Shen, Y. Sang, Y. Li, D. Qian, and A. Y. Zomaya, editors, *13th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2012, Beijing, China, December 14-16, 2012*, pages 343–349. IEEE, 2012.

[10] M. B. Andriamiarina, D. Méry, and N. K. Singh. Integrating proved state-based models for constructing correct distributed algorithms. In Johnsen and Petre [20], pages 268–284.

[11] D. Bert. Preuve de propriétés d'équité en B : étude du protocole du bus SCSI-3. In J. Souquières, editor, *Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL 2001*, pages 221–241, LORIA, Nancy, June 2001.

[12] D. Cansell and D. Méry. The Event-B Modelling Method - Concepts and Case Studies. In D. Bjoerner and M. Henson, editors, *Logics of Specification Languages*, Monographs in Theoretical Computer Science, pages 33–140. Springer, Feb. 2008.

[13] D. Cansell, D. Méry, and S. Merz. Diagram refinements for the design of reactive systems. *J. UCS*, 7(2) :159–174, 2001.

[14] K. M. Chandy. *Parallel program design : a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[15] Dominique, Méry. Modèles et Algorithmes - Modélisation et Vérification de Systèmes. `http://www.loria.fr/~mery/malg/partie-move.pdf`, 2014.

[16] H. Habrias, editor. *First Conference on the B Method*, Nantes, France, April 22-24 1996. IRIN-IUT de Nantes, IRIN-IUT de Nantes. ISBN 2-906082-25-2.

[17] T. Hoang and J.-R. Abrial. Reasoning about liveness properties in event-b. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 456–471. Springer Berlin Heidelberg, 2011.

[18] T. S. Hoang, A. Furst, and J.-R. Abrial. Event-b patterns and their tool support. *Software Engineering and Formal Methods, International Conference on*, 0 :210–219, 2009.

[19] S. Hudon and T. S. Hoang. Systems design guided by progress concerns. In Johnsen and Petre [20], pages 16–30.

[20] E. B. Johnsen and L. Petre, editors. *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, volume 7940 of *Lecture Notes in Computer Science*. Springer, 2013.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, July 1978.

[22] L. Lamport. Verification and specification of concurrent programs. In *A Decade of Concurrency*, volume 803, pages 347–374. Springer-Verlag, 1993.

[23] L. Lamport. A temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3) :872–923, May 1994.

[24] D. Méry. Refinement-based guidelines for algorithmic systems. *International Journal of Software and Informatics*, 3(2-3) :197–239, June/September 2009.

[25] D. Méry and M. Poppleton. Formal modelling and verification of population protocols. In Johnsen and Petre [20], pages 208–222.

[26] D. Méry and N. K. Singh. Analysis of DSR protocol in event-B. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, SSS'11, pages 401–415, Berlin, Heidelberg, 2011. Springer-Verlag.

[27] D. Méry and N. K. Singh. Event B. In J.-L. Boulanger, editor, *Mise en oeuvre de la méthode B*, Informatique et Systèmes d'Informations. HERMES, Apr. 2013.

[28] S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3) :455–495, 1982.

[29] R. Silva. Refactoring Framework. `http://rodin-b-sharp.sourceforge.net/`, 2009.

[30] A. S. Tanenbaum. *Computer networks (4. ed.)*. Prentice Hall, 2002.

[31] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, New York, NY, USA, 1994.

Troisième partie

Etudes de cas et expérimentations

5

Le protocole de routage Anycast RP

Sommaire

5.1 Introduction

Dans ce chapitre, nous allons développer formellement un algorithme de routage ANYCAST RP (RP : Point de Rendez-vous) [12, 17, 18]. L'objectif est ici d'appliquer le paradigme de *« service-as-event »* à un algorithme pouvant s'exécuter dans un système réparti dynamique, c'est-à-dire, dont la topologie peut varier (des processus peuvent rejoindre ou quitter le système à tout moment). Cette section nous introduit brièvement les idées générales concernant le protocole ANYCAST RP.

Le protocole ANYCAST RP a été mis au point par CISCO Systems [17]. Il est fondé sur la notion de diffusion « multicast » [15], que nous expliquons ici informellement : il s'agit pour une source (S) de diffuser un message à destination d'un groupe (G) de destinataires. Les destinataires (aussi appelés récepteurs), intéressés par les messages adressés à ce groupe s'y inscrivent. Ce mécanisme permet d'établir une *route* de la source vers les membres du groupe et est similaire à la construction d'un arbre, où la racine est la source et les feuilles, les destinations membres du groupe acceptant de recevoir des messages de la source. Un protocole appelé Protocol Independent Multicast (PIM) [7, 8, 2, 6], spécialisé dans le routage multicast, permet la mise en place de telles routes arborescentes, partant des sources aux destinataires appartenant aux groupes recevant des messages de ces sources.

Le protocole ANYCAST RP est aussi fondé sur le partage des charges [17] : la source partage avec des routeurs spéciaux appelés Points de Rendez-vous (RP) l'acheminement des messages vers les destinataires ; les routeurs RP se partagent entre eux les tâches d'inscription de destinataires aux groupes recevant des messages des sources, d'acheminement de ces messages et peuvent servir de *back-ups* en cas de panne d'un routeur RP. Il convient de noter ici des caractéristiques très importantes de ces routeurs RP : ils sont connectés aux sources, sont interconnectés et partagent une même adresse IP (nous expliquerons ce point dans les paragraphes suivants).

Pour que les lecteurs comprennent les relations entre les sources, les routeurs RP et les destinataires, nous les clarifions dans ce paragraphe. Chaque hôte destinataire rejoint (se connecte à) un routeur RP : plusieurs sous-arbres ayant un routeur RP pour racine et des destinataires pour feuilles sont donc construits. Les routeurs RP gèrent les inscriptions des destinataires aux flux envoyés par les sources et connaissent ainsi les membres des groupes associés aux sources. Un mécanisme appelé « adressage anycast » [18] permet à plusieurs routeurs de partager une même adresse IP, ce qui est le cas du groupe des routeurs RP. La communication entre la source et les routeurs RP se fait en utilisant la méthode de diffusion « anycast » : la source diffuse son message à l'attention du groupe des routeurs RP, mais choisit parmi les membres du groupe un seul routeur RP, qui recevra le message. Les critères de choix du routeur RP récepteur sont des critères d'optimalité : le plus proche géographiquement, celui ayant le moins de charge, etc. Le routeur RP choisi diffuse le message reçu de la source aux destinataires (directement ou par l'intermédiaire des autres routeurs RP).

Selon son auteur (CISCO Systems) [17], ANYCAST RP est un protocole robuste, fiable, pouvant passer à l'échelle et être utilisé par des applications réseaux critiques. Nous allons étudier dans ce chapitre son fonctionnement dans un réseau mobile. La section suivante détaille l'algorithme ANYCAST RP.

5.2 Présentation du protocole

Cette section présente de manière informelle le protocole ANYCAST RP, tel que décrit dans les documents suivant [12, 17, 18], principalement [18].

Avant de décrire le fonctionnement du protocole ANYCAST RP, nous commençons par définir quelques notions relatives à ce protocole :

- Les messages diffusés sont des *paquets* (il s'agit en fait de parties d'un message plus grand, qui a été découpé en paquets au préalable).
- Le protocole implique trois types d'appareils informatiques :
  - Des Points de Rendez-vous (RPs).
  - Des Routeurs Désignés (DRs) [9] : un Routeur Désigné (DR) est un routeur avec lequel chaque routeur RP du réseau est adjacent (chaque routeur RP possède une connexion directe ou distante avec le routeur DR). Le routeur DR centralise les informations de routage et généralement, il dispose de plusieurs « backups » capables de le remplacer s'il devient indisponible.

— Des hôtes : cet ensemble regroupe les sources de paquets, les destinataires, ainsi que les routeurs qui ne sont ni DR, ni RP.

Le protocole ANYCAST RP est présenté par l'algorithme 1 ci-après. Nous supposons qu'une source s

Algorithm 1 Anycast RP

Paquet : p
Source : s
DRs : $\{dr_1, \ldots, dr_n\}$
RPs : $\{rp_1, \ldots, rp_m\}$
Destinataires (groupe g) : $\{t_1, t_2, \ldots, t_k\}$
Encapsulation d'un paquet p : $enc(p)$
Désencapsulation : $desenc(enc(p))$ $(desenc(enc(p)) = p)$

$\{dr_1$ est le **DR** actif, les autres sont des backups$\}$
$\{rp_1$ est le **RP** plus apte à recevoir du traffic de $dr_1\}$

Require:
(chaque destinataire t_i a rejoint un RP rp_i)
(s est connecté directement à chaque DR dr_i)
(chaque rp_i est connecté aux dr_i)
(les rp_i sont interconnectés)

1: s envoie p à dr_1
2: dr_1 reçoit p
3: dr_1 effectue $register = enc(p)$
4: **while** dr_1 n'a pas reçu de « stop-register » de rp_1 **do**
5: dr_1 envoie $register$ à rp_1
6: **end while**
7: rp_1 reçoit $register$
8: rp_1 effectue $p = desenc(register)$
9: rp_1 envoie p aux t_i qui l'ont rejoint
10: **while** rp_1 n'a pas reçu de « register-stop » de tous les rp_i $(i \neq 1)$ **do**
11: rp_1 envoie $register$ à tous les rp_i $(i \neq 1)$
12: **end while**
13: rp_1 crée un état (s, g)
14: rp_1 envoie un « register-stop » à dr_1
15: **for** rp_i $(i \neq 1)$ **do**
16: recevoir $register$
17: effectuer $p = desenc(register)$
18: envoyer p aux t_i qui l'ont rejoint
19: envoyer un « register-stop » à rp_1
20: créer un état (s, g)
21: **end for**

souhaite diffuser un paquet p à un groupe g de k destinataires : $g = \{t_1, t_2, \ldots, t_k\}$. Les routeurs DRs mis en jeu sont au nombre de n : $DRs = \{dr_1, dr_2, \ldots, dr_n\}$, et les routeurs RPs sont au nombre de m : $RPs = \{rp_1, rp_2, \ldots, rp_m\}$. Nous supposons aussi que les sources sont connectés directement aux routeurs DR, qu'un chemin existe toujours entre chaque routeur RP et chaque routeur DR, que les routeurs RPs sont interconnectés et que les membres du groupe g ont rejoint chacun un routeur RP.

Le protocole ANYCAST RP se déroule de la manière suivante :

— La source s envoie le paquet p au routeur DR actif. Nous supposons ici que ce routeur DR est dr_1 et que les autres routeurs DR sont des backups de dr_1.

— Le routeur dr_1 reçoit le paquet p et l'encapsule dans un message appelé $register$: ce message

permettra de découvrir une route entre la source et les destinataires, s'il n'en existait pas encore et il donnera aussi la possibilité à d'autres destinataires de s'inscrire au groupe de destinataire g.
- Le routeur dr_1 diffuse le message *register* à un routeur RP en utilisant la méthode de diffusion « anycast » : il choisit le routeur RP le plus optimal pour lui (le plus proche de lui géographiquement ou le moins chargé, etc.). Nous supposons que ce routeur RP optimal est rp_1.
- Il faut noter que tant que le routeur dr_1 n'a pas reçu un message appelé « register-stop », pour le message *register*, du routeur rp_1, il continue de renvoyer *register*.
- Lorsque le routeur rp_1 reçoit le message *register*, il le désencapsule et transmet le paquet p ainsi obtenu aux destinataires membres du groupe g qui l'ont rejoint.
- Il retransmet ensuite le message *register* aux autres RPs. Il le retransfère tant qu'il n'a pas reçu de message « register-stop » de ceux-ci.
- Le routeur rp_1 crée un état (s, g) qui signifie que si de nouveaux destinataires souhaitent faire partie du groupe g, ils peuvent rejoindre le routeur rp_1.
- Le routeur rp_1 envoie ensuite à dr_1 un message « register-stop ».
- Quand un routeur RP différent de rp_1 reçoit le message *register*, il le désencapsule, le transmet aux destinataires qui l'ont rejoint, envoie ensuite à rp_1 un message « register-stop » et crée un état (s, g).

La section suivante présente le plan de développement suivi pour la modélisation et la vérification par raffinement de ce protocole de routage.

5.3 Développement formel du protocole

5.3.1 Plan du développement

Cette section présente de manière concise le développement formel du protocole ANYCAST RP, qui se compose d'une abstraction et de 12 raffinements. Nous présentons ici ces niveaux de raffinement (voir figure 6.17) :
- Une machine M0 modélise abstraitement le service fourni par le protocole ANYCAST RP : il s'agit de l'acheminement d'un flux de paquets de données à partir des sources des paquets à des groupes de destinataires (communication *multicast*). Nous modélisons dans cette machine les envois et renvois de paquets, qui peuvent être perdus, par leurs sources et leurs réceptions par leurs destinataires.
- Le premier raffinement M1 détaille le flux entre les sources et les destinataires : avant d'atteindre les destinataires, les paquets passent d'abord par un routeur intermédiaire appelé « Routeur Désigné » (DR). Une hypothèse supplémentaire est posée : un paquet ne peut pas être perdu lors de la communication entre la source et le DR, mais peut être perdu après (entre le DR et les destinataires). Les renvois de paquets sont donc assurés par les Routeurs Désignés.
- Le deuxième raffinement M2 ajoute entre les Routeurs Désignés et les destinataires, des routeurs intermédiaires appelés Points de Rendez-vous (RP) : le routeur DR par lequel transite un paquet, choisi un routeur RP (selon des critères d'optimalité : localisation géographique, performances, etc), auquel il va le transmettre. Le routeur RP se charge ensuite de la diffusion du paquet vers ses destinataires.
- Le troisième raffinement M3 introduit les identités des différents processus acteurs du protocoles : les sources des paquets, leurs destinataires, les Routeurs Désignés, les Points de Rendez-vous sont maintenant clairement identifiés. Les trois précédents niveaux de raffinement ne détaillaient que les flux de paquets de données existants entre les sources, DRs, RPs et destinataires ; nous détaillons maintenant quels sont les processus qui y participent.
- Le quatrième raffinement M4 détaille l'étape de diffusion d'un paquet entre le RP, choisi par un DR pour l'acheminement du paquet, et ses destinataires. Nous y introduisons le fait que chaque destinataire de paquets choisit en premier lieu de « rejoindre » un des RPs du système réparti, c'est-à-dire, de s'y connecter et de recevoir par son intermédiaire les paquets qui lui sont destinés. L'étape de diffusion d'un paquet par le RP choisi aux destinataires est ainsi décomposée en deux sous-étapes :

  1. Le RP choisi transmet le paquet reçu du DR aux destinataires qui l'ont « rejoint », ainsi qu'aux

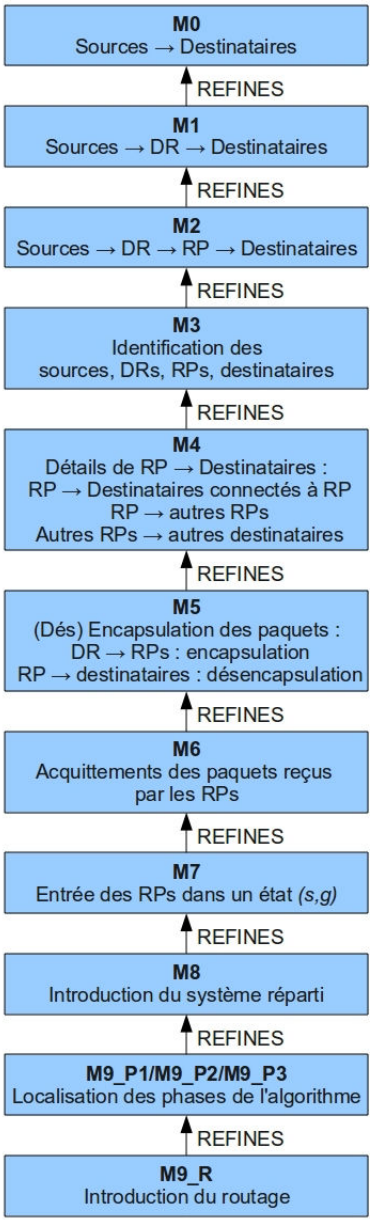


FIGURE 5.1 – ANYCAST RP : plan du développement

autres RPs.

2. Les routeurs RPs non-choisis transmettent le paquet reçu du RP choisi aux destinataires qui les ont « rejoint ».

— Le cinquième raffinement M5 introduit un mécanisme d'encapsulation des paquets dans des messages appelés « Registers », et de désencapsulation de ces messages. Une source transmet un paquet à un routeur DR, qui l'encapsule dans un message « Register », avant de transmettre le message aux routeurs RPs. Ces derniers désencapsulent les messages « Registers » reçus et transmettent les paquets ainsi obtenus vers leurs destinataires qui les ont « rejoint ».
— Le sixième raffinement M6 introduit l'acquittement des messages « Registers » : lors de la réception d'un message « Register », un RP, selon qu'il ait été choisi par le DR émetteur du message ou non, envoie un message d'acquittement appelé « Register-stop », soit au DR émetteur, soit au RP choisi par ce dernier.
— Dans le septième raffinement M7, nous modélisons le fait qu'après avoir reçu un message « Register » encapsulant un paquet, les routeurs RPs entraient, pour ce paquet, dans un état « (s, g) », indiquant que si d'autres destinataires souhaitent rejoindre le groupe g de destinataires du paquet, ils peuvent « rejoindre » les RPs.
— Le raffinement M8 introduit le système réparti : les sources des paquets et les DRs sont connectés entre eux par des liens directs, les DRs et RPs peuvent être distants, de même que les RPs entre eux. Le système réparti ainsi défini peut être modifié (ajout/suppression de lien entre les nœuds), mais conservent quelques propriétés : un chemin existe toujours entre une source et un DR, entre un DR et les RPs, et entre les RPs.
— Le raffinement M9_P1 localise la première phase de l'algorithme de routage, entre les sources et les DRs : les variables et événements correspondants sont localisés sur les sources et les DRs.
— Le raffinement M9_P2 localise la deuxième phase de l'algorithme de routage, entre les DRs et les RPs : les variables et événements correspondants sont localisés sur les DRs et les RPs.
— Le raffinement M9_P3 localise la deuxième phase de l'algorithme de routage, entre les RPs et les destinataires : les variables et événements correspondants sont localisés sur les RPs et les destinataires des paquets.
— Nous introduisons dans la dernière machine M9_R, les tables de routage, une abstraction d'un mécanisme permettant la mise à jour de ces dernières en cas de modification du système réparti, ainsi que le fait que durant un intervalle de temps (assez long), le système réparti peut être considéré comme stable, si aucune modification n'a eu lieu lors de cet intervalle.

La section suivante détaille ces différents niveaux de raffinement, en y présentant les modèles correspondants.

5.3.2 Développement et modélisation du protocole ANYCAST RP

Cette section décrit la modélisation et la vérification par raffinement du protocole ANYCAST RP. Pour ce développement formel, nous appliquons les paradigmes de *correction-par-construction* et *service-as-event* : nous exprimons les services, les fonctionnalités offertes par des propriétés de vivacité, que nous utilisons ensuite pour construire les modèles EVENT-B.

5.3.2.1 Modèle abstrait du protocole ANYCAST RP

Nous commençons par abstraire le protocole ANYCAST RP, à l'aide d'une machine EVENT-B M0. Pour cela, nous modélisons une diffusion de type multicast, impliquant directement les sources de paquets et les destinataires, sans passer par les routeurs DRs ou RPs. Les paquets circulant entre les sources et les destinataires peuvent être perdus, mais un mécanisme permet de s'assurer que s'ils sont perdus, les paquets sont renvoyés et seront un jour reçus par leurs destinataires.

Nous définissons dans un contexte C0, utilisé par la machine EVENT-B M0, un ensemble de paquets PCK non-vide : il s'agit des paquets que les sources devront envoyer aux groupes de destinataires.

CONTEXT C0
SETS
PCK
AXIOMS
$axm1$: $PCK \neq \varnothing$
END

Nous introduisons ensuite, dans M0, des variables d'état $x = (sent, got, lost)$, sous-ensembles de PCK :
— *sent* modélise les paquets envoyés par leurs sources aux groupes de destinataires.
— *got* modélise les paquets reçus par leurs destinataires.
— *lost* modélise les paques perdus après les envois, avant leur réception par les destinataires.
L'invariant contraignant ces variables est noté $I(x)$ et est défini comme suit :

$inv1$: $sent \subseteq PCK$
$inv2$: $got \subseteq PCK$
$inv3$: $lost \subseteq PCK$
$inv4$: $got \subseteq sent$
$inv5$: $lost \subseteq sent$
$inv6$: $got \cap lost = \varnothing$

— $inv1$, $inv2$ et $inv3$ typent les variables *sent*, *got* et *lost*.
— $inv4$ exprime que les paquets reçus par les destinataires ont été envoyés au préalable par des sources.
— $inv5$ exprime que les paquets perdus ont été envoyés au préalable par des sources.
— $inv6$ exprime que les paquets sont soit reçus ou perdus (après envoi), mais ne peuvent pas se trouver dans les deux états « reçus » ou « perdus » en même temps.
Nous avons aussi défini, pour M0, quatre événements : SENDING0, RESENDING0, RECEIVING0, LOSING0, modélisant respectivement l'envoi d'un paquet p par une source, le renvoi d'un paquet p (à cause d'une perte, etc.), la réception d'un paquet p par son groupe de destinataires, la perte d'un paquet p lors de l'acheminement entre sa source et ses destinataires. L'hypothèse d'équité L_0 que nous posons sur la machine M0 est la suivante :

$$L_0 \ \widehat{=} \ WF_x(\text{SENDING0}) \wedge WF_x(\text{RESENDING0}) \wedge SF_x(\text{RECEIVING0})$$

Nous posons des hypothèses d'équité faible sur les événements SENDING0, RESENDING0 et une hypothèse d'équité forte sur l'événement RECEIVING0. Par contre, nous ne posons aucune hypothèse d'équité sur l'événement LOSING0, qui représente une action de l'environnement (perte de paquets). Nous détaillerons ces événements plus loin dans cette section.

Nous posons aussi :

$$\begin{aligned} \text{NEXT} \ \widehat{=} \quad & \vee BA(\text{SENDING0})(x, x') \\ & \vee BA(\text{LOSING0})(x, x') \\ & \vee BA(\text{RESENDING0})(x, x') \\ & \vee BA(\text{RECEIVING0})(x, x') \\ & \vee (x = x') \end{aligned}$$

et définissons les propriétés suivantes :
— $P \ \widehat{=} \ (p \in PCK \wedge p \notin sent)$, qui exprime qu'un paquet p n'a pas encore été envoyé par une source.
— $R \ \widehat{=} \ (p \in sent \wedge p \in lost)$, exprimant qu'un paquet p a été envoyé par une source et est perdu.
— $S \ \widehat{=} \ (p \in sent \wedge p \notin (got \cup lost)$, qui exprime qu'un paquet p a été envoyé par une source et n'est ni perdu, ni reçu par ses destinataires : il transite entre sa source et ses destinataires.
— $Q \ \widehat{=} \ (p \in got)$, exprimant qu'un paquet p est reçu par ses destinataires.
Le diagramme 5.2 nous permet d'exprimer simplement le protocole Anycast RP (diffusion multicast d'un paquet p par une source) à l'aide des propriétés de vivacité suivantes :

1. $P \rightsquigarrow S$ qui exprime qu'un paquet p non-encore envoyé par sa source, transitera fatalement entre cette dernière et les destinataires.
2. $R \rightsquigarrow S$ qui exprime qu'un paquet p encore envoyé par sa source et perdu, sera fatalement renvoyé à ses destinataires.

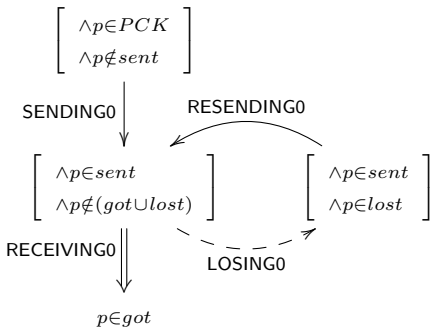


FIGURE 5.2 – Diagramme d'assertions pour l'abstraction du protocole ANYCAST RP

3. $S \rightsquigarrow Q$ qui exprime qu'un paquet p transitant vers ses destinataires sera fatalement reçu par ces dernier.

Ces propriétés expriment qu'un paquet p sera reçu fatalement par ses destinataires. L'ensemble Φ_0 des propriétés caractérisant M0 est donc : $\Phi_0 \triangleq \{P \rightsquigarrow S, R \rightsquigarrow S, S \rightsquigarrow Q\}$.

Démonstration. Nous montrons maintenant que la machine M0 satisfait les propriétés de vivacité de Φ_0.

⟨1⟩1. Nous montrons que M0 satisfait $P \rightsquigarrow S$. Une hypothèse d'équité faible est posée sur l'événement SENDING0 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de SENDING0 sont observables lorsque P est vrai, leurs observations ne falsifient pas le prédicat P, condition d'observation de SENDING0, ou conduisent à S.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$P \wedge [\mathsf{NEXT}]_x \Rightarrow (P' \vee S')$$

Soit :
— $P \wedge BA(\mathsf{SENDING0})(x, x') \Rightarrow (P' \vee S')$
— $P \wedge BA(\mathsf{RESENDING0})(x, x') \Rightarrow (P' \vee S')$
— $P \wedge BA(\mathsf{RECEIVING0})(x, x') \Rightarrow (P' \vee S')$
— $P \wedge BA(\mathsf{LOSING0})(x, x') \Rightarrow (P' \vee S')$
— $P \wedge (x = x') \Rightarrow (P' \vee S')$

La propriété suivante **(3)** : $P(x) \wedge BA(\mathsf{SENDING0})(x, x') \Rightarrow S(x')$, et la condition de faisabilité **(4)** : $P(x) \Rightarrow (\exists x' \cdot BA(\mathsf{SENDING0})(x, x'))$ sont satisfaites par l'événement SENDING0. La propriété **(3)** nous permet de déduire $P \wedge \langle \mathsf{NEXT} \wedge \mathsf{SENDING0} \rangle_x \Rightarrow S'$ et **(4)** nous permet de déduire $P \Rightarrow ENABLED\langle \mathsf{SENDING0} \rangle_x$, où $ENABLED\langle \mathsf{SENDING0} \rangle_x \triangleq (\exists y \cdot BA(\mathsf{SENDING0})(x, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M0}) \vdash P \rightsquigarrow S$, donc que M0 satisfait $P \rightsquigarrow S$. □

⟨1⟩2. Nous montrons que M0 satisfait $R \rightsquigarrow S$. L'événement LOSING0 peut nous conduire d'un état satisfaisant S à un état satisfaisant R. Nous montrons qu'une opération de renvoi modélisée par l'événement RESENDING0 permet de retourner à un état satisfaisant S. Une hypothèse d'équité faible est posée sur l'événement RESENDING0 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de RESENDING0 sont observables lorsque R est vrai, leurs observations ne falsifient pas le prédicat R, condition d'observation de RESENDING0, ou conduisent à S.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$R \wedge [\mathsf{NEXT}]_x \Rightarrow (R' \vee S')$$

Soit :
— $R \wedge BA(\mathsf{RESENDING0})(x, x') \Rightarrow (R' \vee S')$
— $R \wedge BA(\mathsf{SENDING0})(x, x') \Rightarrow (R' \vee S')$
— $R \wedge BA(\mathsf{RECEIVING0})(x, x') \Rightarrow (R' \vee S')$
— $R \wedge BA(\mathsf{LOSING0})(x, x') \Rightarrow (R' \vee S')$
— $R \wedge (x = x') \Rightarrow (R' \vee S')$

La propriété **(3)** : $R(x) \wedge BA(\mathsf{RESENDING0})(x, x') \Rightarrow S(x')$, et la condition de faisabilité **(4)** : $R(x) \Rightarrow (\exists x' \cdot BA(\mathsf{RESENDING0})(x, x'))$ sont satisfaites par l'événement RESENDING0. La propriété **(3)** permet de déduire $R \wedge \langle \mathsf{NEXT} \wedge \mathsf{RESENDING0} \rangle_x \Rightarrow S'$ et **(4)** permet de déduire $R \Rightarrow ENABLED\langle \mathsf{RESENDING0} \rangle_x$, où $ENABLED\langle \mathsf{RESENDING0} \rangle_x \ \widehat{=} \ (\exists y \cdot BA(\mathsf{RESENDING0})(x, y))$. La règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M0}) \vdash R \rightsquigarrow S$, donc que M0 satisfait $R \rightsquigarrow S$. □

⟨1⟩3. Nous montrons que M0 satisfait $S \rightsquigarrow Q$. Pour cela, nous allons prouver que M0 satisfait $(R \vee S) \rightsquigarrow Q$. Une hypothèse d'équité forte est posée sur l'événement RECEIVING0 et nous nous trouvons dans le cas suivant **(1)** : un événement LOSING0 peut nous ramener, d'un état satisfaisant S, à un état satisfaisant R (tel que $R \rightsquigarrow S$), sinon les observations des autres événements, différents de LOSING0 et RECEIVING0, ne falsifient pas $(R \vee S)$ ou peuvent nous conduire à Q.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(R \vee S) \wedge [\mathsf{NEXT}]_x \Rightarrow ((R' \vee S') \vee Q')$$

Soit :
— $(R \vee S) \wedge BA(\mathsf{SENDING0})(x, x') \Rightarrow ((R' \vee S') \vee Q')$
— $(R \vee S) \wedge BA(\mathsf{RESENDING0})(x, x') \Rightarrow ((R' \vee S') \vee Q')$
— $(R \vee S) \wedge BA(\mathsf{LOSING0})(x, x') \Rightarrow ((R' \vee S') \vee Q')$
— $(R \vee S) \wedge BA(\mathsf{RECEIVING0})(x, x') \Rightarrow ((R' \vee S') \vee Q')$
— $(R \vee S) \wedge (x = x') \Rightarrow ((R' \vee S') \vee Q')$
La propriété **(3)** : $S(x) \wedge BA(\mathsf{RECEIVING0})(x, x') \Rightarrow Q(x')$, ainsi que la condition de faisabilité **(4)** : $S(x) \Rightarrow (\exists x' \cdot BA(\mathsf{RECEIVING0})(x, x'))$ sont satisfaites par RECEIVING0. **(3)** nous permet de déduire $(R \vee S) \wedge \langle \mathsf{NEXT} \wedge \mathsf{RECEIVING0} \rangle_x \Rightarrow Q'$. Les propriétés **(3)**, **(4)** et $R \rightsquigarrow S$ nous permettent de déduire que tant que Q n'est pas satisfait, alors RECEIVING0 sera fatalement activable : $(\neg Q) \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING0} \rangle_x$, où $ENABLED\langle \mathsf{RECEIVING0} \rangle_x \ \widehat{=} \ (\exists y \cdot BA(\mathsf{RECEIVING0})(x, y))$. Nous pouvons en déduire, en tenant compte de $R \rightsquigarrow S : \Box(R \vee S) \wedge \Box[\mathsf{NEXT}]_x \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING0} \rangle_x$. L'application de la règle SF1 donne $(\mathcal{S}pec(\mathsf{M0}), R \rightsquigarrow S) \vdash ((R \vee S) \rightsquigarrow Q)$. La machine M0 satisfait $(R \vee S) \rightsquigarrow Q$, donc $S \rightsquigarrow Q$. □

⟨1⟩4. Les étapes ⟨1⟩1, ⟨1⟩2 et ⟨1⟩3 nous permettent de déduire que M0 satisfait les propriétés de vivacité de Φ_0. QED.

□

Nous nous aidons de ces propriétés de vivacité de Φ_0 et par conséquent du diagramme 5.2 pour détailler le modèle Event-B abstrait M0 du protocole Anycast RP. Nous commençons par les initialisations des variables *sent*, *got* et *lost*.

```
INITIALISATION ≙
  BEGIN
    act1 : sent := ∅
    act2 : got := ∅
    act3 : lost := ∅
  END
```

Ces variables sont initialisées à l'aide de l'ensemble vide (∅) parce qu'aucun paquet n'a encore été envoyé, ni reçu, ni perdu à l'état initial.

Nous détaillons maintenant les événements SENDING0, RESENDING0, RECEIVING0 et LOSING0 :

```
EVENT SENDING0 ≙
  ANY
    p
  WHERE
    grd1 : p ∈ PCK
    grd2 : p ∉ sent
  THEN
    act1 : sent := sent ∪ {p}
  END
```

```
EVENT RESENDING0 ≙
  ANY
    p
  WHERE
    grd1 : p ∈ PCK
    grd2 : p ∈ sent
  THEN
    act1 : lost := lost \ {p}
  END
```

```
EVENT RECEIVING0 ≙
  ANY
    p
  WHERE
    grd1 : p ∈ PCK
    grd2 : p ∈ sent
    grd2 : p ∉ (got ∪ lost)
  THEN
    act1 : got := got ∪ {p}
  END
```

EVENT LOSING0 $\widehat{=}$
ANY
p
WHERE
$grd1 \ : \ p \in PCK$
$grd2 \ : \ p \in sent$
$grd3 \ : \ p \notin (got \cup lost)$
THEN
$act1 \ : \ lost := lost \cup \{p\}$
END

— L'événement SENDING0 exprime que si un paquet p n'a pas encore été envoyé, alors il est envoyé sa source.
— L'événement RESENDING0 exprime que si un paquet p a déjà été envoyé, alors il peut être renvoyé par sa source. Si le paquet p a été perdu, son renvoi permet de l'enlever de l'ensemble des paquets perdus ($lost$).
— L'événement RECEIVING0 exprime que si un paquet p a été envoyé par sa source et n'a pas encore été reçu ou perdu, alors il est reçu par son groupe de destinataires.
— L'événement LOSING0 exprime que si un paquet p a été envoyé par sa source et n'a pas encore été reçu ou perdu, alors il peut être perdu.

Nous pouvons résumer cette abstraction du protocole ANYCAST RP par le diagramme 5.3 suivant :

$$\mathsf{C0} \xleftarrow{\text{SEES}} \mathsf{M0} \xrightarrow{\text{LIVE}} \left(\left(\begin{array}{l} \wedge WF_x(\text{SENDING0}) \\ \wedge WF_x(\text{RESENDING0}) \\ \wedge SF_x(\text{RECEIVING0}) \end{array} \right), \left\{ \begin{array}{l} P \rightsquigarrow S, \\ R \rightsquigarrow S, \\ S \rightsquigarrow Q \end{array} \right\} \right)$$

FIGURE 5.3 – Abstraction du protocole ANYCAST RP

Nous introduisons dans la section suivante les routeurs intermédiaires DR entre les sources et les groupes de destinataires.

5.3.2.2 Premier raffinement : Routeur Désigné (DR)

Ce premier raffinement M1 introduit un routage intermédiaire entre les sources et les groupes de destinataires : il s'agit des Routeurs Désignés (DRs). L'ensemble des variables x_1 de cette machine est constitué des variables x de la machine précédente auxquelles sont rajoutées deux nouvelles variables intermédiaires : dr_rcvd et dr_sent, sous-ensembles de PCK et représentant respectivement les paquets reçus par les DRs et ceux envoyés par ces derniers à leurs destinataires. L'invariant contraignant ces variables est noté $I_1(x_1)$ et est défini de la manière suivante :

$inv1 \ : \ dr\_rcvd \subseteq PCK$
$inv2 \ : \ dr\_sent \subseteq PCK$
$inv3 \ : \ dr\_rcvd \subseteq sent$
$inv4 \ : \ dr\_sent \subseteq dr\_rcvd$
$inv5 \ : \ (got \cup lost) \subseteq dr\_sent$

— $inv1$ et $inv2$ servent à typer dr_rcvd et dr_sent.
— $inv3$ exprime que chaque paquet reçu par un routeur DR a été envoyé par une source.
— $inv4$ exprime que chaque paquet envoyé par un routeur DR a été au préalable reçu par ce dernier.
— $inv5$ exprime que les paquets reçus par les destinataires ou perdus ont été envoyés par des routeurs DRs.

Nous posons une hypothèse dans ce raffinement : un paquet p ne peut pas être perdu lors de son transfert entre sa source et les DRs ; par contre, il peut se perdre lors de son acheminement des DRs aux destinataires. Le routeur DR ayant pris en charge le paquet se charge de le rediffuser en cas de perte.

Nous définissons des événements SENDING1, RECEIVING1, LOSING1, DR_RESENDING1, raffinements respectifs de SENDING0, RECEIVING0, LOSING0, DR_RESENDING0. Nous y définissons aussi deux nouveaux événements : DR_RECEIVING1, DR_SENDING1. Par rapport au premier modèle, les événements

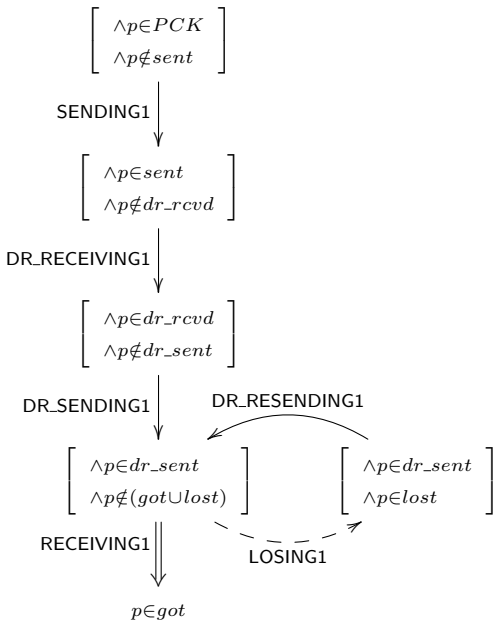


FIGURE 5.4 – Diagramme d'assertions pour le premier raffinement

introduisant de nouveaux éléments sont DR_RECEIVING1, DR_SENDING1 et DR_RESENDING1 qui modélisent respectivement la réception, l'envoi et le renvoi d'un paquet p par un routeur DR. Nous expliquerons en détails les événements du modèle après. L'hypothèse d'équité L_1 posée sur le modèle M1 est la suivante :

$$\begin{aligned} L_1 \quad \widehat{=} \quad & \wedge WF_{x_1}(\mathsf{SENDING1}) \\ & \wedge WF_{x_1}(\mathsf{DR\_RECEIVING1}) \\ & \wedge WF_{x_1}(\mathsf{DR\_SENDING1}) \\ & \wedge WF_{x_1}(\mathsf{DR\_RESENDING1}) \\ & \wedge SF_{x_1}(\mathsf{RECEIVING1}) \end{aligned}$$

Nous posons une hypothèse forte sur l'événement RECEIVING1, et des hypothèses d'équité faible sur les autres événements du modèles différents de LOSING1, qui modélise une action de l'environnement et non de l'algorithme (perte de paquets). Nous définissons aussi la relation NEXT comme suit :

$$\begin{aligned} \mathsf{NEXT} \quad \widehat{=} \quad & \vee BA(\mathsf{SENDING1})(x_1, {x_1}') \\ & \vee BA(\mathsf{DR\_RECEIVING1})(x_1, {x_1}') \\ & \vee BA(\mathsf{DR\_SENDING1})(x_1, {x_1}') \\ & \vee BA(\mathsf{DR\_RESENDING1})(x_1, {x_1}') \\ & \vee BA(\mathsf{RECEIVING1})(x_1, {x_1}') \\ & \vee BA(\mathsf{LOSING1})(x_1, {x_1}') \\ & \vee (x_1 = {x_1}') \end{aligned}$$

Le diagramme de preuve 5.4 suivant prend en compte l'hypothèse d'équité L_1 et présente la liste Φ_1 de propriétés de vivacité exprimant le premier raffinement du protocole ANYCAST RP. Par rapport au diagramme 5.2, nous avons étendu le précédent diagramme en prenant en compte les Routeurs Désignés et en y ajoutant les propriétés de vivacités relatives aux étapes intermédiaires introduites par ces derniers. Nous posons :

— $T \ \widehat{=} \ (p \in sent \wedge p \notin dr\_rcvd)$, exprimant qu'un paquet p a été envoyé par une source mais n'est pas encore reçu par un DR.
— $U \ \widehat{=} \ (p \in dr\_rcvd \wedge p \notin dr\_sent)$, exprimant qu'un paquet p a été reçu par un DR mais pas encore transmis par ce dernier.

— $V \triangleq (p \in dr\_sent \wedge p \notin (got \cup lost))$, exprimant qu'un paquet p a été envoyé par un DR et n'est pas encore reçu par ses destinataires et n'est pas perdu.
— $W \triangleq (p \in dr\_sent \wedge p \in lost)$, exprimant qu'un paquet p a été envoyé par un DR et est perdu.

La liste Φ_1 des propriétés de vivacité, satisfaites par M1, est définie comme suit :

$$\Phi_1 \triangleq \{P \rightsquigarrow T, T \rightsquigarrow U, U \rightsquigarrow V, W \rightsquigarrow V, V \rightsquigarrow Q\}$$

Démonstration. Nous prouvons maintenant que la machine M1 satisfait les propriétés de vivacité de Φ_1.

⟨1⟩1. Nous montrons que M1 satisfait $P \rightsquigarrow T$. Une hypothèse d'équité faible est posée sur l'événement SENDING1 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de SENDING1 sont observables lorsque P est vrai, leurs observations ne falsifient pas le prédicat P, condition d'observation de SENDING1, ou conduisent à T.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$P \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow (P' \vee T')$$

Soit :
— $P \wedge BA(\mathsf{SENDING1})(x_1, {x_1}') \Rightarrow (P' \vee T')$
— $P \wedge BA(\mathsf{DR\_RECEIVING1})(x_1, {x_1}') \Rightarrow (P' \vee T')$
— $P \wedge BA(\mathsf{DR\_SENDING1})(x_1, {x_1}') \Rightarrow (P' \vee T')$
— $P \wedge BA(\mathsf{DR\_RESENDING1})(x_1, {x_1}') \Rightarrow (P' \vee T')$
— $P \wedge BA(\mathsf{RECEIVING1})(x_1, {x_1}') \Rightarrow (P' \vee T')$
— $P \wedge BA(\mathsf{LOSING1})(x_1, {x_1}') \Rightarrow (P' \vee T')$
— $P \wedge (x_1 = {x_1}') \Rightarrow (P' \vee T')$

La propriété suivante **(3)** : $P(x_1) \wedge \wedge BA(\mathsf{SENDING1})(x_1, x_1') \Rightarrow T(x_1')$, et la condition de faisabilité **(4)** : $P(x_1) \wedge \Rightarrow (\exists x_1' \cdot BA(\mathsf{SENDING1})(x_1, x_1'))$ sont satisfaites par l'événement SENDING1. **(3)** permet de déduire $P \wedge \langle \mathsf{NEXT} \wedge \mathsf{SENDING1} \rangle_{x_1} \Rightarrow T'$ et **(4)**, $P \Rightarrow ENABLED \langle \mathsf{SENDING1} \rangle_{x_1}$, où $ENABLED \langle \mathsf{SENDING1} \rangle_{x_1} \triangleq (\exists y \cdot BA(\mathsf{SENDING1})(x_1, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M1}) \vdash P \rightsquigarrow T$, donc que M1 satisfait $P \rightsquigarrow T$. □

⟨1⟩2. Nous montrons que M1 satisfait $T \rightsquigarrow U$. Une hypothèse d'équité faible est posée sur l'événement DR_RECEIVING1 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de DR_RECEIVING1 sont observables lorsque T est vrai, leurs observations ne falsifient pas le prédicat T, condition d'observation de DR_RECEIVING1, ou conduisent à U.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$T \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow (T' \vee U')$$

Soit :
— $T \wedge BA(\mathsf{DR\_RECEIVING1})(x_1, {x_1}') \Rightarrow (T' \vee U')$
— $T \wedge BA(\mathsf{SENDING1})(x_1, {x_1}') \Rightarrow (T' \vee U')$
— $T \wedge BA(\mathsf{DR\_SENDING1})(x_1, {x_1}') \Rightarrow (T' \vee U')$
— $T \wedge BA(\mathsf{DR\_RESENDING1})(x_1, {x_1}') \Rightarrow (T' \vee U')$
— $T \wedge BA(\mathsf{RECEIVING1})(x_1, {x_1}') \Rightarrow (T' \vee U')$
— $T \wedge BA(\mathsf{LOSING1})(x_1, {x_1}') \Rightarrow (T' \vee U')$
— $T \wedge (x_1 = {x_1}') \Rightarrow (T' \vee U')$

La propriété **(3)** : $T(x_1) \wedge BA(\mathsf{DR\_RECEIVING1})(x_1, x_1') \Rightarrow U(x_1')$, et la condition de faisabilité **(4)** : $T(x_1) \wedge \Rightarrow (\exists x_1' \cdot BA(\mathsf{DR\_RECEIVING1})(x_1, x_1'))$ sont satisfaites par DR_RECEIVING1. **(3)** permet de déduire $T \wedge \langle \mathsf{NEXT} \wedge \mathsf{DR\_RECEIVING1} \rangle_{x_1} \Rightarrow U'$ et **(4)**, $T \Rightarrow ENABLED \langle \mathsf{DR\_RECEIVING1} \rangle_{x_1}$, où $ENABLED \langle \mathsf{DR\_RECEIVING1} \rangle_{x_1} \triangleq (\exists y \cdot BA(\mathsf{DR\_RECEIVING1})(x_1, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M1}) \vdash T \rightsquigarrow U$, donc que M1 satisfait $T \rightsquigarrow U$. □

⟨1⟩3. Nous montrons que M1 satisfait $U \rightsquigarrow V$. Une hypothèse d'équité faible est posée sur l'événement DR_SENDING1 et nous nous trouvons dans ce cas **(1)** : si les autres événements différents de DR_SENDING1 sont observables lorsque U est vrai, leurs observations ne falsifient pas le prédicat U, condition d'observation de DR_SENDING1, ou conduisent à V.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$U \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow (U' \vee V')$$

Soit :
— $U \wedge BA(\mathsf{DR\_SENDING1})(x_1, {x_1}') \Rightarrow (U' \vee V')$
— $U \wedge BA(\mathsf{SENDING1})(x_1, {x_1}') \Rightarrow (U' \vee V')$

— $U \wedge BA(\mathsf{DR\_RECEIVING1})(x_1, {x_1}') \Rightarrow (U' \vee V')$
— $U \wedge BA(\mathsf{DR\_RESENDING1})(x_1, {x_1}') \Rightarrow (U' \vee V')$
— $U \wedge BA(\mathsf{RECEIVING1})(x_1, {x_1}') \Rightarrow (U' \vee V')$
— $U \wedge BA(\mathsf{LOSING1})(x_1, {x_1}') \Rightarrow (U' \vee V')$
— $U \wedge (x_1 = {x_1}') \Rightarrow (U' \vee V')$

La propriété **(3)** : $U(x_1) \wedge \wedge BA(\mathsf{DR\_SENDING1})(x_1, x_1') \Rightarrow V(x_1')$, et la condition de faisabilité **(4)** : $U(x_1) \Rightarrow (\exists x_1' \cdot BA(\mathsf{DR\_SENDING1})(x_1, x_1'))$ sont satisfaites par l'événement DR_SENDING1. **(3)** permet de déduire $U \wedge \langle \mathsf{NEXT} \wedge \mathsf{DR\_SENDING1} \rangle_{x_1} \Rightarrow V'$ et **(4)**, $U \Rightarrow ENABLED \langle \mathsf{DR\_SENDING1} \rangle_{x_1}$, où $ENABLED \langle \mathsf{DR\_SENDING1} \rangle_{x_1} \ \widehat{=} \ (\exists y \cdot BA(\mathsf{DR\_SENDING1})(x_1, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M1}) \vdash U \rightsquigarrow V$, donc que M1 satisfait $U \rightsquigarrow V$. □

⟨1⟩4. Nous montrons que M1 satisfait $W \rightsquigarrow V$. Une hypothèse d'équité faible est posée sur l'événement DR_RESENDING1 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de DR_RESENDING1 sont observables lorsque W est vrai, leurs observations ne falsifient pas le prédicat W, condition d'observation de l'événement DR_RESENDING1, ou conduisent à V.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$W \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow (W' \vee V')$$

Soit :
— $W \wedge BA(\mathsf{DR\_RESENDING1})(x_1, {x_1}') \Rightarrow (W' \vee V')$
— $W \wedge BA(\mathsf{SENDING1})(x_1, {x_1}') \Rightarrow (W' \vee V')$
— $W \wedge BA(\mathsf{DR\_RECEIVING1})(x_1, {x_1}') \Rightarrow (W' \vee V')$
— $W \wedge BA(\mathsf{DR\_SENDING1})(x_1, {x_1}') \Rightarrow (W' \vee V')$
— $W \wedge BA(\mathsf{RECEIVING1})(x_1, {x_1}') \Rightarrow (W' \vee V')$
— $W \wedge BA(\mathsf{LOSING1})(x_1, {x_1}') \Rightarrow (W' \vee V')$
— $W \wedge (x_1 = {x_1}') \Rightarrow (W' \vee V')$

La propriété **(3)** : $W(x_1) \wedge BA(\mathsf{DR\_RESENDING1})(x_1, x_1') \Rightarrow V(x_1')$, et la condition de faisabilité **(4)** : $W(x_1) \Rightarrow (\exists x_1' \cdot BA(\mathsf{DR\_RESENDING1})(x_1, x_1'))$ sont satisfaites par DR_RESENDING1. **(3)** permet de déduire $W \wedge \langle \mathsf{NEXT} \wedge \mathsf{DR\_RESENDING1} \rangle_{x_1} \Rightarrow V'$ et **(4)**, $W \Rightarrow ENABLED \langle \mathsf{DR\_RESENDING1} \rangle_{x_1}$, où $ENABLED \langle \mathsf{DR\_RESENDING1} \rangle_{x_1} \ \widehat{=} \ (\exists y \cdot BA(\mathsf{DR\_RESENDING1})(x_1, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M1}) \vdash W \rightsquigarrow V$, donc que M1 satisfait $W \rightsquigarrow V$. □

⟨1⟩5. Nous montrons que M1 satisfait $V \rightsquigarrow Q$. Pour cela, nous allons prouver M1 satisfait $(W \vee V) \rightsquigarrow Q$. Une hypothèse d'équité forte est posée sur l'événement RECEIVING1 et nous nous trouvons dans le cas suivant **(1)** : un événement LOSING1 peut nous ramener, d'un état satisfaisant V, à un état satisfaisant W, sinon les observations des autres événements, différents de LOSING1 et RECEIVING1, ne falsifient pas $(W \vee V)$ ou peuvent mener à Q.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(W \vee V) \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow ((W' \vee V') \vee Q')$$

Soit :
— $(W \vee V) \wedge BA(\mathsf{SENDING1})(x_1, x_1') \Rightarrow ((W' \vee V') \vee Q')$
— $(W \vee V) \wedge BA(\mathsf{DR\_RECEIVING1})(x_1, x_1') \Rightarrow ((W' \vee V') \vee Q')$
— $(W \vee V) \wedge BA(\mathsf{DR\_SENDING1})(x_1, x_1') \Rightarrow ((W' \vee V') \vee Q')$
— $(W \vee V) \wedge BA(\mathsf{DR\_RESENDING1})(x_1, x_1') \Rightarrow ((W' \vee V') \vee Q')$
— $(W \vee V) \wedge BA(\mathsf{LOSING1})(x_1, x_1') \Rightarrow ((W' \vee V') \vee Q')$.
— $(W \vee V) \wedge BA(\mathsf{RECEIVING1})(x_1, x_1') \Rightarrow ((W' \vee V') \vee Q')$
— $(W \vee V) \wedge (x_1 = x_1') \Rightarrow ((W' \vee V') \vee Q')$

La propriété suivante **(3)** : $V(x_1) \wedge \wedge BA(\mathsf{RECEIVING1})(x_1, x_1') \Rightarrow Q(x_1')$, ainsi que la condition de faisabilité **(4)**, définie par : $V(x_1) \wedge \Rightarrow (\exists x_1' \cdot BA(\mathsf{RECEIVING1})(x_1, x_1'))$ sont satisfaites par l'événement RECEIVING1. **(3)** permet de déduire $(W \vee V) \wedge \langle \mathsf{NEXT} \wedge \mathsf{RECEIVING1} \rangle_{x_1} \Rightarrow Q'$. **(3)**, **(4)** et la propriété $W \rightsquigarrow V$ nous permettent de déduire que tant que Q n'est pas satisfait, alors l'événement RECEIVING1 sera fatalement observable, soit : $(\neg Q) \Rightarrow \Diamond ENABLED \langle \mathsf{RECEIVING1} \rangle_{x_1}$, avec $ENABLED \langle \mathsf{RECEIVING1} \rangle_{x_1} \ \widehat{=} \ (\exists y \cdot BA(\mathsf{RECEIVING1})(x_1, y))$. Nous en déduisons, en tenant en compte l'hypothèse $W \rightsquigarrow V$: $\Box(W \vee V) \wedge \Box[\mathsf{NEXT}]_{x_1} \Rightarrow \Diamond ENABLED \langle \mathsf{RECEIVING1} \rangle_{x_1}$. L'application de la règle SF1 nous donne $(\mathcal{S}pec(\mathsf{M1}), W \rightsquigarrow V) \vdash ((W \vee V) \rightsquigarrow Q)$. La machine M1 satisfait $(W \vee V) \rightsquigarrow Q$, donc $V \rightsquigarrow Q$. □

⟨1⟩6. QED.

□

Nous prouvons ensuite que M1 satisfait la liste $\Phi_0 \triangleq \{P \rightsquigarrow S, R \rightsquigarrow S, S \rightsquigarrow Q\}$ de propriétés de vivacité caractérisant M0.

Démonstration. Nous montrons que M1 satisfait les propriétés de vivacité de Φ_0 caractérisant le niveau abstrait.

⟨1⟩1. Nous montrons que M1 satisfait $P \rightsquigarrow S$.
- ⟨2⟩1. Les propriétés $P \rightsquigarrow T$, $T \rightsquigarrow U$, $U \rightsquigarrow V$ et la règle d'inférence transitivité (*trans*) nous permettent de déduire $P \rightsquigarrow V$.
- ⟨2⟩2. Nous avons l'équivalence suivante qui est vérifiée : $S \Leftrightarrow (T \vee U \vee V)$ **(1)**, c'est-à-dire qu'un paquet p envoyé par une source, non reçu par ses destinataires et non perdu, est soit envoyé par un source et non reçu par un DR, soit reçu par un DR et non encore envoyé par ce dernier, soit envoyé par un DR et non reçu par ses destinataires et non perdu ; et vice versa. Cette équivalence, ainsi que la règle de déduction (*dedu*) nous permettent d'obtenir $(T \vee U \vee V) \rightsquigarrow S$, donc $V \rightsquigarrow S$.
- ⟨2⟩3. Les étapes ⟨2⟩1 et ⟨2⟩2, ainsi que l'application de la règle de transitivité (*trans*) nous permettent de déduite $\mathcal{S}pec(\mathsf{M1} \vdash P \rightsquigarrow S$. QED. □

⟨1⟩1. Nous montrons que M1 satisfait $R \rightsquigarrow S$.
- ⟨2⟩1. Nous avons posé l'hypothèse suivante pour ce raffinement : $(p \in sent \wedge p \in lost) \Rightarrow (p \in dr\_sent \wedge p \in lost)$, c'est-à-dire $R \Rightarrow W$, exprimant qu'un paquet envoyé par une source ne peut être perdu que s'il a été retransmis par un DR. Cette propriété $R \Rightarrow W$ nous permet de déduire $R \rightsquigarrow W$ (règle de déduction (*dedu*)). Les propriétés $R \rightsquigarrow W$, $W \rightsquigarrow V$ nous permettent de montrer par transitivité (*trans*) $R \rightsquigarrow V$.
- ⟨2⟩2. L'équivalence **(1)** ainsi que la règle de déduction (*dedu*) nous permettent d'obtenir $(T \vee U \vee V) \rightsquigarrow S$, donc $V \rightsquigarrow S$.
- ⟨2⟩3. Les étapes ⟨2⟩1 et ⟨2⟩2, ainsi que l'application de la règle de transitivité (*trans*) nous permettent de déduite $\mathcal{S}pec(\mathsf{M1}) \vdash R \rightsquigarrow S$. QED. □

⟨1⟩2. Nous montrons que M1 satisfait $S \rightsquigarrow Q$.
- ⟨2⟩1. L'équivalence **(1)**, ainsi que la règle de déduction (*dedu*) nous permettent de déduire $S \rightsquigarrow (T \vee U \vee V)$. Cette propriété de vivacité, ainsi que $T \rightsquigarrow U$ et $U \rightsquigarrow V$ nous permettent de déduire $S \rightsquigarrow V$. Par transitivité (*trans*), nous déduisons M1 satisfait $S \rightsquigarrow Q$ à partir des propriétés $S \rightsquigarrow V$ et $V \rightsquigarrow Q$. □

⟨1⟩3. QED.

□

Nous nous servons des propriétés de vivacité définies dans la liste Φ_1 et du diagramme 5.4 pour construire la machine EVENT-B M1, raffinement [2] de M0. Nous commençons par détailler les initialisations des variables. Celles définies dans le modèle M0 ne changent pas. Nous ne décrivons donc ici que les initialisations des nouvelles variables *dr_rcvd* et *dr_sent*.

```
INITIALISATION ≙
  BEGIN
    ...
    ⊕ act4 : dr_rcvd := ∅
    ⊕ act5 : dr_sent := ∅
  END
```

Ces variables sont initialisées à l'aide de l'ensemble vide (∅) parce qu'aucun paquet n'a encore été envoyé, ni reçu par un routeur DR à l'état initial.

Nous détaillons maintenant les événements du modèle. Le raffinement SENDING1 n'apportant rien de nouveau par rapport à SENDING0, nous ne nous détaillons que les autres événements du modèle :

2. ⊕ : ajouter des éléments à un modèle, ⊖ : retirer des éléments d'un modèle.

EVENT DR_RECEIVING1 $\widehat{=}$
ANY
p
WHERE
$grd1 : p \in PCK$
$grd2 : p \in sent$
$grd3 : p \notin dr\_rcvd$
THEN
$act1 : dr\_rcvd = dr\_rcvd \cup \{p\}$
END

EVENT DR_SENDING1 $\widehat{=}$
ANY
p
WHERE
$grd1 : p \in PCK$
$grd2 : p \in dr\_rcvd$
$grd3 : p \notin dr\_sent$
THEN
$act1 : dr\_sent = dr\_sent \cup \{p\}$
END

EVENT DR_RESENDING1
REFINES RESENDING0 $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd2 : p \in sent$
$\oplus grd2 : p \in dr\_sent$
...
THEN
...
END

EVENT RECEIVING1
REFINES RECEIVING0 $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd2 : p \in sent$
$\oplus grd2 : p \in dr\_sent$
...
THEN
...
END

EVENT LOSING1
REFINES LOSING0 $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd2 : p \in sent$
$\oplus grd2 : p \in dr\_sent$
...
THEN
...
END

— l'événement DR_RECEIVING1 modélise la réception d'un paquet p par un routeur DR. Le paquet p a été au préalable envoyé par une source et n'a pas encore été reçu par un routeur DR.
— l'événement DR_SENDING1 modélise l'envoi d'un paquet p reçu par un routeur DR par ce dernier.
— Pour les évenements DR_RESENDING1, RECEIVING1, LOSING1, nous remplaçons la garde abstraite ($grd2 : p \in sent$) par une garde plus concrète ($grd2 : p \in dr\_sent$) : nous introduisons le fait que ces événements modélisant respectivement le renvoi, la réception et la perte d'un paquet p, sont observables uniquement après que le paquet p ait été envoyé par un routeur DR.

Ce premier raffinement peut se résumer par le diagramme 5.5 suivant :

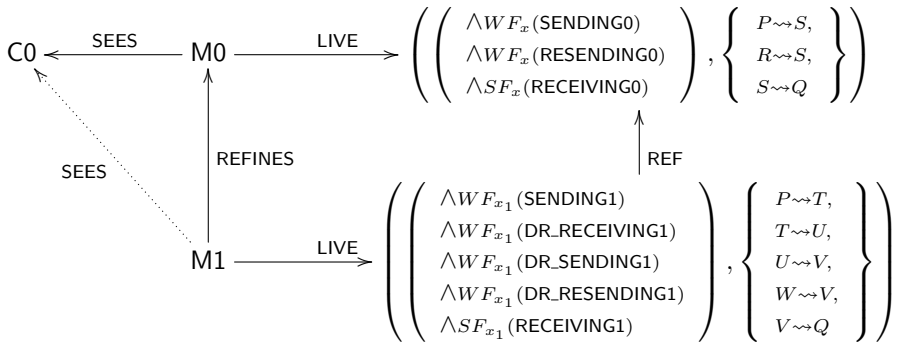


FIGURE 5.5 – Premier raffinement du protocole ANYCAST RP

Le prochain raffinement, décrit dans la section qui suit, introduit les Points de Rendez-vous (RPs) dans les modèles du protocole ANYCAST RP. Pour les niveaux de raffinements qui suivent, nous ne détaillons pas les propriétés de vivacité et de preuves dans le présent document de thèse, mais nous les donnons en annexe [3].

5.3.2.3 Deuxième raffinement : Point de Rendez-vous (RP)

Nous introduisons, dans ce raffinement M2, un routage supplémentaire entre les Routeurs Désignés (DRs) et les groupes de destinataires : il s'agit des routeurs Points de Rendez-vous (RPs). Ce second raffinement modélise de manière abstraite les principales phases du protocole de routage ANYCAST RP :

3. http://andriami.bruno.free.fr/andriami/main/short/annexe.pdf

— Nous avons une première diffusion de type « *unicast* » d'un paquet, d'une source à un Routeur Désigné (DR) : il s'agit d'une communication entre un seul émetteur (la source) et un seul destinataire (le routeur DR).
— Nous avons ensuite une seconde diffusion de type « *anycast* » du paquet, du routeur DR à un des routeurs RPs, qui a été choisi comme destinataire intermédiaire, à l'aide de critères d'optimalité (routeur RP géographiquement proche du routeur DR ou charge du routeur minimale, etc.)
— Nous avons enfin une diffusion « *multicast* » du paquet, du routeur RP au groupe de destinataires du paquet.

Nous notons x_2 l'ensemble des variables de la machine M2, constitué des variables x_1 du raffinement M1 précédent, ainsi que de nouvelles variables : rp_rcvd et rp_sent, sous-ensembles de PCK et représentant respectivement les paquets reçus et envoyés par les routeurs RPs. L'invariant contraignant ces variables est noté $I_2(x_2)$ et est défini de la manière suivante :

$$
\begin{array}{l}
inv1 \ : \ rp\_rcvd \subseteq PCK \\
inv2 \ : \ rp\_sent \subseteq PCK \\
inv3 \ : \ rp\_rcvd \subseteq dr\_sent \\
inv4 \ : \ rp\_sent \subseteq rp\_rcvd \\
inv5 \ : \ got \subseteq rp\_sent \\
inv6 \ : \ lost \cap rp\_rcvd = \varnothing
\end{array}
$$

— $inv1$ et $inv2$ sont des invariants de typage.
— $inv3$ exprime que chaque paquet reçu par un routeur RP a été au préalable envoyé par un routeur DR.
— $inv4$ exprime que chaque paquet envoyé par un routeur RP a été reçu par ce dernier.
— $inv5$ exprime que chaque paquet reçu ses destinataires a été envoyé par un routeur RP.
— $inv6$ exprime qu'un paquet est soit perdu, soit reçu par un routeur RP. Avec cet invariant, nous exprimons le fait que la perte de paquet ne peut se produire que lors de l'acheminement du paquet entre les routeurs DRs et RPs.

Nous posons dans ce raffinement une hypothèse sur la perte et la rediffusion des paquets : la perte d'un paquet ne peut se produire que durant l'acheminement de ce dernier entre les DRs et les RPs, auquel cas, le DR ayant diffusé le paquet se chargera de l'envoyer à nouveau. Nous y définissons aussi des événements SENDING2, DR_RECEIVING2, DR_SENDING2, DR_RESENDING2, RECEIVING2, LOSING2, raffinements respectifs des événements du modèle M1, SENDING1, DR_RECEIVING1, DR_SENDING1, DR_RESENDING1, RECEIVING1, LOSING1. Nous avons aussi introduit des événements RP_RECEIVING2 et RP_SENDING2 qui modélisent respectivement la réception d'un paquet par un routeur RP et la diffusion d'un paquet aux destinataires par ce même routeur.

Comme dans les machines précédentes, nous définissons une liste de propriétés de vivacité Φ_2, prenant en compte l'introduction des routeurs RPs, détaillée en annexe[4], caractérisant la machine M2 et nous l'utilisons pour construire cette machine, second raffinement de l'abstraction M0. Les états initiaux des variables abstraites ne changeant pas, nous ne donnons ici que les initialisations des variables rp_rcvd et rp_sent.

$$
\begin{array}{l}
\text{INITIALISATION} \ \widehat{=} \\
\quad \text{BEGIN} \\
\quad\quad \ldots \\
\quad\quad \oplus \ act6 \ : \ rp\_rcvd := \varnothing \\
\quad\quad \oplus \ act7 \ : \ rp\_sent := \varnothing \\
\quad \text{END}
\end{array}
$$

Ces variables sont initialisées à l'aide de l'ensemble vide ($\varnothing$), parce qu'à l'état initial, aucun paquet n'a été ni reçu, ni envoyé par les routeurs RPs.

Nous présentons ensuite les événements de M2. Certains événements n'étant pas modifiés lors du raffinement, nous nous concentrons sur ceux qui ont changé, ainsi que les nouveaux événements, notamment RP_RECEIVING2, RP_SENDING2, RECEIVING2 et LOSING2.

4. `http://andriami.bruno.free.fr/andriami/main/short/annexe.pdf`

EVENT RP_RECEIVING2 $\widehat{=}$
ANY
p
WHERE
$grd1$: $p \in PCK$
$grd2$: $p \in dr\_sent$
$grd3$: $p \notin lost$
THEN
$act1$: $rp\_rcvd := rp\_rcvd \cup \{p\}$
END$_2$

EVENT RP_SENDING2 $\widehat{=}$
ANY
p
WHERE
$grd1$: $p \in PCK$
$grd2$: $p \in rp\_rcvd$
THEN
$act1$: $rp\_sent := rp\_sent \cup \{p\}$
END

EVENT RECEIVING2
REFINES RECEIVING1 $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd2$: $p \in dr\_sent$
$\oplus grd2$: $p \in rp\_sent$
$\ominus grd3$: $p \notin (got \cup lost)$
$\oplus grd3$: $p \notin got$
...
THEN
...
END

EVENT LOSING2 REFINES LOSING1 $\widehat{=}$
...
WHERE
...
$\ominus grd3$: $p \notin (got \cup lost)$
$\oplus grd3$: $p \notin (rp\_rcvd \cup lost)$
...
THEN
...
END

— L'événement RP_RECEIVING2 modélise la réception d'un paquet p envoyé par un routeur DR et non perdu, par un routeur RP.
— L'évenement RP_SENDING2 modélise l'envoi d'un paquet p par un routeur RP. Il faut au préalable que le paquet p ait été reçu par le routeur RP. Nous insistons ici sur le fait que le routeur RP peut recevoir un même paquet plusieurs fois, suite à un renvoi du paquet par un routeur DR, ce qui est la raison pour laquelle nous ne mettons pas dans la garde de l'événement, une condition qui exprime que le paquet p ne doit pas encore avoir été reçu ($p \notin rp\_rcvd$).
— L'événement RECEIVING2 modélise la réception d'un paquet p diffusé par un routeur RP par les destinataires du paquet. Nous avons supposé que la perte de paquets ne pouvait seulement se produire que pendant l'acheminement des paquets entre les routeurs DR et RP. C'est pourquoi, nous avons enlevé la condition qui exprime que le paquet p n'est pas perdu ($p \in lost$) de la garde de l'évenement.
— L'événement LOSING2 modélise la perte d'un paquet p : la garde (modifiée) de l'événement exprime que le paquet peut être perdu s'il a été envoyé par un routeur DR et s'il n'a pas encore été perdu ou reçu par un routeur RP.

Ce deuxième raffinement est résumé par le diagramme 5.6 suivant :

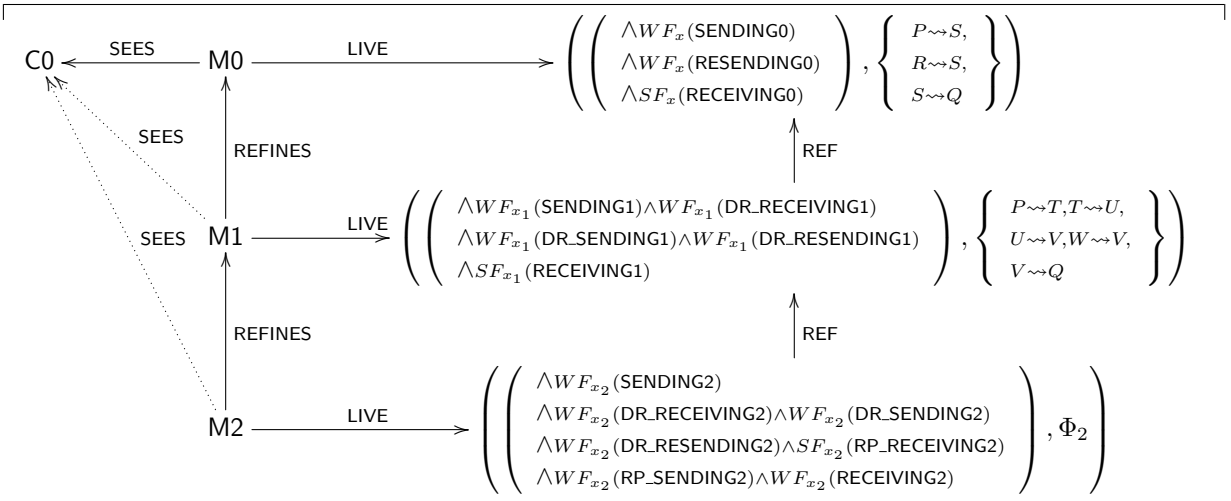


FIGURE 5.6 – Deuxième raffinement du protocole ANYCAST RP

Ce diagramme nous montre également que les propriétés de vivacité de Φ_1 sont préservées lors du raffinement (voir l'annexe). Nous verrons dans la section suivante, qui introduit les identités des nœuds sources, destinations, des routeurs DRs et RPs, comment la *décomposition* du protocole ANYCAST RP en phases est plus accentuée.

5.3.2.4 Troisième raffinement : identités des différents acteurs

Lors des précédents modèles, nous nous sommes contentés d'introduire les différentes étapes de l'algorithme, sans identifier explicitement les différents acteurs (routeurs) prenant part à l'algorithme de routage ANYCAST RP : nous avons seulement exprimé qu'un paquet était envoyé par une source quelconque, non-déterminée, puis reçu par un routeur DR, lui aussi quelconque ; le paquet étant ensuite transmis par ce routeur DR à un routeur RP quelconque qui le diffuse à un groupe de destinataires. L'objectif de ce raffinement M3 est d'identifier ces différents nœuds sources, routeurs, destinations, acteurs du protocole ANYCAST RP.

Nous définissons un contexte C1 qui raffine le contexte C0 précédent et qui est utilisé par la machine M3.

CONTEXT C1 EXTENDS C0
SETS
$NODES$
CONSTANTS
$HOSTS, DRs, RPs, SOURCES, TARGETS, source, group\_target$
AXIOMS
$axm1$: $NODES \neq \varnothing$
$axm2$: $partition(NODES, HOSTS, DR, RP)$
$axm3$: $HOSTS \neq \varnothing$
$axm4$: $DRs \neq \varnothing$
$axm5$: $RPs \neq \varnothing$
$axm6$: $SOURCES \cup TARGETS \subset HOSTS$
$axm7$: $SOURCES \neq \varnothing$
$axm8$: $TARGETS \neq \varnothing$
$axm9$: $source \in PCK \rightarrow SOURCES$
$axm10$: $group\_target \in PCK \rightarrow \mathbb{P}1(TARGETS)$
$axm11$: $\forall p \in PCK \Rightarrow source(p) \notin group\_target(p)$
END

Nous introduisons les différents nœuds du système réparti mis en jeu lors du protocole de routage ANYCAST RP, ainsi que certaines propriétés s'y rapportant :

— Les nœuds sont modélisés par un ensemble $NODES$, non-vide ($axm1$), partitionné ($axm2$) en trois sous-ensembles également non-vides ($axm3$, $axm4$, $axm5$) : DRs, RPs, $HOSTS$, modélisant respectivement les Routeurs Désignés, les Points de Rendez-vous et les nœuds différents de ces deux types de routeurs ($axm6$), dont les sources ($SOURCES$) des paquets, leurs destinataires ($TARGETS$), eux aussi non-vides ($axm7$, $axm8$), ainsi que les nœuds qui ne sont ni sources, ni destinataires.
— Une fonction totale *source* associe chaque paquet à un nœud source ($axm9$).
— Une fonction totale *group_target* associe chaque paquet à un sous-ensemble (groupe) non-vide de destinataires ($axm10$).
— Nous introduisons aussi une propriété ($axm11$) exprimant que pour chaque paquet p, la source du paquet n'est pas incluse dans son groupe de destinataires.

Ce raffinement comporte un ensemble de variables x_3 défini comme suit :

— Les variables abstraites *sent*, *dr_rcvd*, *dr_sent*, *rp_rcvd* et *rp_sent* sont remplacées par des variables plus concrètes :
  — Une variable *sent_by_s* remplace *sent*. Elle associe à chaque source les paquets envoyés par cette dernière.
  — Une variable *rcvd_by_dr* remplace *dr_rcvd*. Elle associe à chaque routeur DR les paquets reçus par le routeur.
  — Une variable *sent_by_dr* remplace *dr_sent*. Elle associe à chaque routeur DR les paquets envoyés par le routeur.

- Une variable *rcvd_by_rp* remplace *rp_rcvd*. Elle associe à chaque routeur RP les paquets reçus par le routeur.
- Une variable *sent_by_rp* remplace *rp_sent*. Elle associe à chaque routeur RP les paquets envoyés par le routeur.

— De nouvelles variables sont introduites :
- *dr_r_pck* contient des couples (dr, p) où dr est un routeur DR choisi par une source pour recevoir un paquet p qu'elle va diffuser.
- *dr_s_pck* contient des couples (dr, p) où dr est un routeur DR ayant reçu un paquet p et qui va le diffuser à un routeur RP.
- *rp_r_pck* contient des couples (rp, p) où rp est un routeur RP choisi par un routeur DR pour recevoir un paquet p qu'il va lui diffuser.

appeler

L'invariant de la machine M3, noté $I_3(x_3)$ et exprimant des propriétés de sûreté contraignant ces variables, est défini de la manière suivante :

— Il permet de typer les variables :

$$\begin{array}{l} inv1 \ : \ sent\_by\_s \in SOURCES \leftrightarrow PCK \\ inv2 \ : \ rcvd\_by\_dr \in DRs \leftrightarrow PCK \\ inv3 \ : \ sent\_by\_dr \in DRs \leftrightarrow PCK \\ inv4 \ : \ rcvd\_by\_rp \in RPs \leftrightarrow PCK \\ inv5 \ : \ sent\_by\_rp \in RPs \leftrightarrow PCK \\ inv6 \ : \ dr\_r\_pck \in DRs \leftrightarrow PCK \\ inv7 \ : \ dr\_s\_pck \in DRs \leftrightarrow PCK \\ inv8 \ : \ rp\_s\_pck \in RPs \leftrightarrow PCK \end{array}$$

Ces propriétés associent selon le cas, soit des sources, soit des routeurs DRs, soit des routeurs RPs aux paquets diffusés et/ou reçus.

— Il définit les liens les variables d'états abstraites des machines précédentes et celles concrètes (invariant de collage) introduites dans M3 :

$$\begin{array}{l} inv9 \ : \ \forall s, p.s \in NODES \wedge p \in PCK \wedge s \mapsto p \in sent\_by\_s \Rightarrow s = source(p) \\ inv10 \ : \ \forall s1, s2, p \cdot \left(\begin{array}{l} \wedge s1 \in NODES \wedge s2 \in NODES \wedge p \in PCK \\ \wedge s1 \mapsto p \in sent\_by\_s \wedge s2 \mapsto p \in sent\_by\_s \end{array} \right) \Rightarrow s1 = s2 \\ inv11 \ : \ ran(sent\_by\_s) = sent \end{array}$$

- *inv*9 et *inv*10 expriment que pour chaque paquet p diffusé, l'expéditeur est unique (*inv*10). Il s'agit notamment de la source ($source(p)$) du paquet p (*inv*9).
- *inv*11 fait le lien entre la variable concrète *sent_by_s* et celle abstraite *sent* : chaque paquet p envoyé par sa source au niveau concret (*sent_by_s*), est un paquet envoyé au niveau abstrait (*sent*).

$$\begin{array}{l} inv12 \ : \ ran(dr\_r\_pck) = ran(sent\_by\_s) \\ inv13 \ : \ \forall dr1, dr2, p \cdot \left(\begin{array}{l} \wedge dr1 \in DRs \wedge dr2 \in DRs \wedge p \in PCK \\ \wedge dr1 \mapsto p \in dr\_r\_pck \wedge dr2 \mapsto p \in dr\_r\_pck \end{array} \right) \Rightarrow dr1 = dr2 \\ inv14 \ : \ rcvd\_by\_dr \subseteq dr\_r\_pck \\ inv15 \ : \ ran(rcvd\_by\_dr) = dr\_rcvd \end{array}$$

- *inv*12 exprime que chaque paquet qui est acheminé vers un routeur DR a été envoyé par une source.
- *inv*13 et *inv*14 expriment que pour chaque paquet p diffusé, le récepteur est unique (*inv*13), il s'agit notamment du routeur DR vers qui le paquet p est acheminé (*inv*14).
- *inv*15 fait le lien entre la variable concrète *rcvd_by_dr* et celle abstraite *dr_rcvd* : chaque paquet p reçu par un routeur DR, choisi par la source du paquet p, au niveau concret (*rcvd_by_dr*), est paquet reçu par un routeur DR au niveau abstrait (*dr_rcvd*).

$$\begin{array}{l} inv16 \ : \ dr\_s\_pck \subseteq rcvd\_by\_dr \\ inv17 \ : \ sent\_by\_dr \subseteq dr\_s\_pck \\ inv18 \ : \ ran(sent\_by\_dr) = dr\_sent \end{array}$$

- *inv*16 exprime que chaque paquet marqué comme devant être acheminé à partir d'un routeur DR a d'abord été au préalable reçu par ce dernier.
- *inv*17 exprime que chaque paquet envoyé par un routeur DR a été marqué au préalable comme devant être acheminé à partir de ce dernier.
- *inv*16 et *inv*17, couplés à *inv*13 et *inv*14, permettent d'exprimer l'unicité du routeur DR diffusant un paquet p.

— L'invariant $inv18$ fait le lien entre la variable concrète *sent_by_dr* et celle abstraite *dr_sent* : chaque paquet p associé à un routeur DR diffuseur de paquets, dans la variable *sent_by_dr*, est contenu dans la variable abstraite *dr_sent*.

$$
\begin{array}{l}
inv19 \ : \ ran(rp\_r\_pck) = ran(sent\_by\_dr) \\
inv20 \ : \ \forall rp1, rp2, p \cdot \left(\begin{array}{l} \wedge rp1 \in RPs \wedge rp2 \in RPs \wedge p \in PCK \\ \wedge rp1 \mapsto p \in rp\_r\_pck \wedge rp2 \mapsto p \in rp\_r\_pck \end{array} \right) \Rightarrow rp1 = rp2 \\
inv21 \ : \ rcvd\_by\_rp \subseteq rp\_r\_pck \\
inv22 \ : \ ran(rcvd\_by\_rp) = rp\_rcvd
\end{array}
$$

— $inv19$ exprime que chaque paquet marqué comme devant être acheminé vers un routeur RP est un paquet envoyé par un routeur DR à ce routeur RP.

— $inv20$ et $inv21$ expriment que pour chaque paquet p diffusé par un routeur DR, le récepteur est unique ($inv20$), il s'agit notamment du routeur RP vers qui le paquet p est acheminé ($inv21$).

— $inv22$ fait le lien entre la variable concrète *rcvd_by_rp* et celle abstraite *rp_rcvd* : chaque paquet p associé à un routeur RP récepteur, dans la variable *rcvd_by_rp*, est contenu dans la variable abstraite *rp_rcvd*.

$$
\begin{array}{l}
inv23 \ : \ sent\_by\_rp \subseteq rcvd\_by\_rp \\
inv24 \ : \ ran(sent\_by\_rp) = rp\_sent
\end{array}
$$

— $inv23$ exprime que chaque paquet envoyé par un routeur RP a d'abord été au préalable reçu par ce dernier.

— $inv23$, couplé à $inv20$ et $inv21$, exprime l'unicité du routeur RP diffusant un paquet p.

— $inv24$ fait le lien entre la variable concrète *sent_by_rp* et celle abstraite *rp_sent* : chaque paquet p associé à un routeur RP diffuseur, dans la variable *sent_by_rp*, est contenu dans la variable abstraite *rp_sent*.

Nous introduisons les événements : SENDING3, DR_RECEIVING3, DR_SENDING3, DR_RESENDING3, RP_RECEIVING3, RP_SENDING3, RECEIVING3, LOSING3, raffinements des événements de la machine M2 précédent. Comme précédemment, une liste de propriétés de vivacité Φ_3 est définie (cf. annexe) et nous guide dans la construction de la machine M3. Nous commençons par définir les états initiaux des variables introduites dans ce modèle :

```
INITIALISATION ≙
  BEGIN
    sent_by_s, rcvd_by_dr, sent_by_dr := ∅
    rcvd_by_rp, dr_r_pck, dr_s_pck, rp_r_pck := ∅
    lost, got := ∅
  END
```

L'absence de diffusion, de réception et de perte de paquets à l'état initial justifie l'initialisation des variables à l'aide de l'ensemble vide ($\varnothing$).

Les autres événements de M3 sont définis de la manière suivante :

— L'événement SENDING3 identifie maintenant clairement la source s ($grd5$) qui diffuse ($act1$) un paquet p non encore envoyé ($grd6$) à un Routeur Désigné dr ($grd7$), lui aussi clairement identifié. Le paquet p est marqué comme devant être acheminé au routeur dr ($act2$).

```
EVENT SENDING3 REFINES SENDING2 ≙
  ANY
    p, s, dr, gdr
  WHERE
    grd1 : p ∈ PCK
    grd2 : s ∈ SOURCES
    grd3 : dr ∈ DRs
    grd4 : gdr ⊆ DRs
    grd5 : s = source(p)
    grd6 : s ↦ p ∉ sent_by_s
    grd7 : gdr = {dr}
  THEN
    act1 : sent_by_s := sent_by_s ∪ {s ↦ p}
    act2 : dr_r_pck := dr_r_pck ∪ (gdr × {p})
  END
```

— L'événement DR_RECEIVING3 identifie maintenant le Routeur Désigné dr qui reçoit ($act1$) un paquet p expédié par une sour : ran(sent_by_rp) = rp_sentce et pas encore reçu ($grd4$) : il s'agit du Routeur Désigné auquel le paquet p doit être acheminé ($grd3$).

EVENT DR_RECEIVING3 REFINES DR_RECEIVING2 $\widehat{=}$
ANY
p, dr
WHERE
$grd1 : p \in PCK$
$grd2 : dr \in DRs$
$grd3 : dr \mapsto p \in dr\_r\_pck$
$grd4 : dr \mapsto p \notin rcvd\_by\_dr$
THEN
$act1 : rcvd\_by\_dr := rcvd\_by\_dr \cup \{dr \mapsto p\}$
END

— L'événement DR_SENDING3 modélise le fait qu'un paquet p reçu par un Routeur Désigné dr ($grd5$), mais non encore diffusé par ce dernier ($grd6$), est maintenant diffusé par le routeur dr ($act1$). Le paquet est marqué comme ayant pour origine le routeur dr ($act2$) et comme ayant pour destination un Point de Rendez-vous rp ($grd7$ et $act3$).

EVENT DR_SENDING3 REFINES DR_SENDING2 $\widehat{=}$
ANY
p, rp, dr, grp
WHERE
$grd1 : p \in PCK$
$grd2 : dr \in DRs$
$grd3 : rp \in RPs$
$grd4 : grp \subseteq RPs$
$grd5 : dr \mapsto p \in rcvd\_by\_dr$
$grd6 : dr \mapsto p \notin sent\_by\_dr$
$grd7 : grp = \{rp\}$
THEN
$act1 : sent\_by\_dr := sent\_by\_dr \cup \{dr \mapsto p\}$
$act2 : dr\_s\_pck := dr\_s\_pck \cup \{dr \mapsto p\}$
$act3 : rp\_r\_pck := rp\_r\_pck \cup (grp \times \{p\})$
END

— Un paquet p déjà envoyé par un Routeur Désigné dr ($grd3$) est envoyé de nouveau par ce dernier. Si le paquet p est perdu lors du renvoi, il est alors ôté de l'ensemble des paquets perdus ($act1$).

EVENT DR_RESENDING3 REFINES DR_RESENDING2 $\widehat{=}$
ANY
p, dr
WHERE
$grd1 : p \in PCK$
$grd2 : dr \in DRs$
$grd3 : dr \mapsto p \in sent\_by\_dr$
THEN
$act1 : lost := lost \setminus \{p\}$
END

— L'événement RP_RECEIVING3 identifie maintenant le routeur Point de Rendez-vous rp qui reçoit ($act1$) un paquet p, qui n'est pas perdu et dont le routeur rp est un des récepteurs intermédiaires ($grd3$).

EVENT RP_RECEIVING3 REFINES RP_RECEIVING2 $\widehat{=}$
ANY
p, rp
WHERE
$grd1 : p \in PCK$
$grd2 : rp \in RPs$
$grd3 : rp \mapsto p \in rp\_r\_pck$
$grd4 : p \notin lost$
THEN
$act1 : rcvd\_by\_rp := rcvd\_by\_rp \cup \{rp \mapsto p\}$
END

— L'événement RP_SENDING3 identifie clairement le routeur Point de Rendez-vous rp qui a reçu ($grd3$) un paquet p, qui le diffuse ensuite ($act1$).

EVENT RP_SENDING3 REFINES RP_SENDING2 $\widehat{=}$
ANY
p, rp
WHERE
$grd1$: $p \in PCK$
$grd2$: $rp \in RPs$
$grd3$: $rp \mapsto p \in rcvd\_by\_rp$
THEN
$act1$: $sent\_by\_rp := sent\_by\_rp \cup \{rp \mapsto p\}$
END

— Nous faisons maintenant figurer dans l'événement RECEIVING3 les identités du groupe (gt) de destinataires ($grd3$) qui reçoivent un paquet p ($act1$).

EVENT RECEIVING3 REFINES RECEIVING2 $\widehat{=}$
ANY
p, gt
WHERE
$grd1$: $p \in PCK$
$grd2$: $gt \subseteq TARGETS$
$grd3$: $gt = group\_target(p)$
$grd4$: $p \in ran(sent\_by\_rp)$
$grd5$: $p \notin got$
THEN
$act1$: $got := got \cup \{p\}$
END

— L'événement LOSING3 exprime qu'un paquet p envoyé à un routeur RP ($grd1$) mais non encore reçu par ce dernier (et non perdu) ($grd2$) peut être perdu ($act1$).

EVENT LOSING3 REFINES LOSING2 $\widehat{=}$
ANY
p
WHERE
$grd1$: $p \in PCK \wedge p \in ran(rp\_r\_pck)$
$grd2$: $p \notin (ran(rcvd\_by\_rp) \cup lost)$
THEN
$act1$: $lost := lost \cup \{p\}$
END

Les modèles M2 et M3 mettent en avant une identification de trois phases algorithmiques du protocole ANYCAST RP : **(1)** la première phase est la diffusion d'un paquet d'une source à un routeur DR, **(2)** la seconde est la diffusion *anycast* d'un paquet par un routeur DR à un routeur RP et **(3)** la dernière est la diffusion *multicast* d'un paquet par un routeur RP au groupe de destinataires. Nous avons aussi introduit le fait que des informations sont ajoutés aux paquets diffusés lors de leur acheminement : les identités de leurs sources, des routeurs DRs qui les reçoivent en premier, des routeurs RPs qui se chargent de le diffuser aux destinataires.

Ce troisième raffinement est résumé par le diagramme 5.7 suivant :

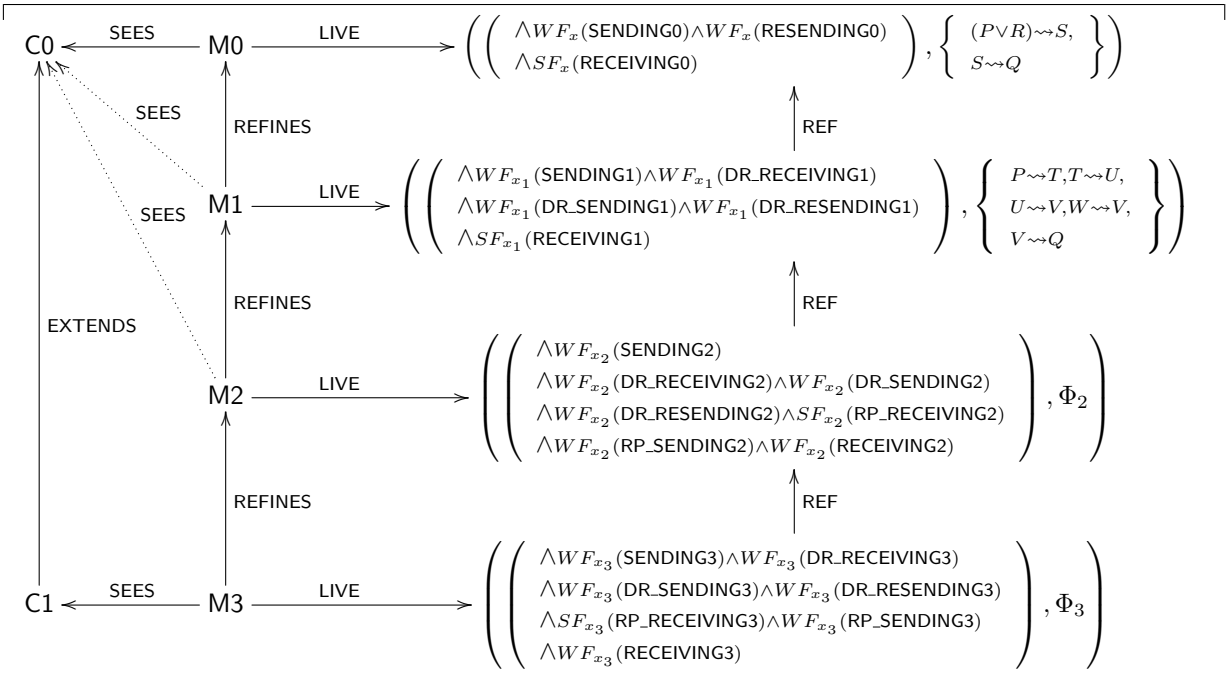


FIGURE 5.7 – Troisième raffinement du protocole ANYCAST RP

Ce diagramme met également en avant le fait que les propriétés de vivacité de Φ_2 sont satisfaites par M3 (pour la preuve, voir l'annexe). Nous nous concentrons dans la section suivante sur la phase **(3)** : nous détaillons la diffusion *multicast* entre le routeur RP et le groupes de destinataires.

5.3.2.5 Quatrième raffinement : le multicast RPs - destinataires

Nous présentons dans cette section le raffinement M4 de la machine M3. Ce raffinement détaille le mécanisme de la diffusion *multicast* entre le routeur RP, choisi par un routeur DR pour recevoir un paquet, et les destinataires du paquet.

Cette phase algorithmique peut s'abstraire et se résumer de la manière suivante :

Algorithm 2 Anycast RP - Abstraction de la Phase 3

Paquet : p
RPs : $\{rp_1, \ldots, rp_m\}$

$\{rp_1$ est le **RP** choisi pour recevoir le paquet p des routeurs DRs$\}$

1: rp_1 reçoit p
2: rp_1 envoie p aux t_i qui l'ont rejoint
3: rp_1 envoie p à tous les rp_i $(i \neq 1)$
4: **for** rp_i $(i \neq 1)$ **do**
5: recevoir p
6: envoyer p aux t_i qui l'ont rejoint
7: **end for**

La première étape consiste pour le routeur RP, ayant reçu un paquet d'un routeur DR, à **(1)** le diffuser tout d'abord aux destinataires qui l'ont rejoint, c'est-à-dire aux destinataires qui lui sont connectés et qui ont choisi de recevoir du traffic de lui. Ensuite, le routeur RP **(2)** diffuse le paquet aux autres routeurs RPs, qui **(3)** le diffusent aux destinataires qui les ont rejoints.

Pour prendre en compte cette notion de *« destinataire qui rejoint un Point de Rendez-vous »*, nous définissons un contexte C2 étendant le contexte C1 vu précédemment. Nous y définissons une fonction totale *join* de l'ensemble $TARGETS$ des destinataires à l'ensemble RPs des Points de Rendez-vous.

CONTEXT C2 EXTENDS C1
CONSTANTS
$join$
AXIOMS
$axm1 : join \in TARGETS \rightarrow RPs$
END

Ce contexte C2 est utilisé par la machine M4.

L'ensemble des variables de M4, que nous notons x_4, est constitué des variables x_3 du raffinement précédent et des nouvelles variables suivantes :

— *sent_to_t* modélise les paquets envoyés par les routeurs RPs aux destinataires. Elle contient des couples du type (rp, p), où rp est le routeur RP expéditeur et p le paquet expédié.
— *t_r_pck* indique vers quels destinataires des paquets envoyés par des routeurs RPs transitent. Elle contient des couples du type (t, p) où t est le destinataire à qui le paquet p est adressé.
— *rcvd_by_t* modélise les paquets reçus individuellement par leurs destinataires. Elle contient des couples du type (t, p) où t est le destinataire qui a reçu le paquet p.
— *sent_to_o_rp* indique quels paquets ont été envoyés aux routeurs RPs non choisis par les routeurs DRs. Elle contient des couples du type (rp, p), où rp est le routeur RP choisi par les routeurs DRs pour recevoir un paquet p.
— *o_rp_r_pck* indique vers quels autres routeurs RPs les paquets envoyés par un routeur RP, choisi par un routeur DR, transitent. Elle contient des couples du type (rp, p), où rp est un routeur RP non-choisi par les routeurs DRs pour recevoir un paquet p.
— *rcvd_by_o_rp* contient les paquets reçus par les routeurs RPs non choisis par les routeurs DRs (pour recevoir des paquets). Elle contient des couples du type (rp, p), où rp est un routeur RP (non-choisi par les routeurs DRs) ayant reçu un paquet p.

Un invariant noté $I_4(x_4)$ et contraignant ces variables x_4 est défini comme suit :

— $inv1$ à $inv6$ définissent le typage.

$inv1 : rcvd\_by\_o\_rp \in RPs \leftrightarrow PCK$
$inv2 : rcvd\_by\_t \in TARGETS \leftrightarrow PCK$
$inv3 : sent\_to\_o\_rp \in RPs \leftrightarrow PCK$
$inv4 : sent\_to\_t \in RPs \leftrightarrow PCK$
$inv5 : t\_r\_pck \in TARGETS \leftrightarrow PCK$
$inv6 : o\_rp\_r\_pck \in RPs \leftrightarrow PCK$

Ces propriétés associent soient des routeurs RPs soit des destinataires, aux paquets diffusés, en cours d'acheminement ou reçus, selon les cas.

— Nous avons ensuite des propriétés de sûreté.

$inv7 : rcvd\_by\_t \subseteq t\_r\_pck$
$inv8 : ran(t\_r\_pck) \subseteq ran(sent\_to\_t)$
$inv9 : ran(rcvd\_by\_t) \subseteq ran(sent\_to\_t)$
$inv10 : ran(sent\_to\_t) \subseteq ran(sent\_by\_rp)$
$inv11 : ran(rcvd\_by\_o\_rp) \subseteq ran(o\_rp\_r\_pck)$
$inv12 : ran(o\_rp\_r\_pck) \subseteq ran(sent\_to\_o\_rp)$
$inv13 : ran(sent\_to\_o\_rp) \subseteq ran(sent\_by\_rp)$
$inv14 : ran(sent\_to\_o\_rp) \subseteq ran(sent\_to\_t)$
$inv15 : \forall rp, p \cdot rp \in RPs \wedge p \in PCK \wedge rp \mapsto p \in o\_rp\_r\_pck \Rightarrow rp \notin sent\_by\_rp^{-1}[\{p\}]$
$inv16 : \forall rp, p \cdot rp \in RPs \wedge p \in PCK \wedge rp \mapsto p \in rcvd\_by\_o\_rp \Rightarrow rp \notin sent\_by\_rp^{-1}[\{p\}]$
$inv17 : \forall t, p \cdot t \in TARGETS \wedge p \in PCK \wedge t \mapsto p \in t\_r\_pck \Rightarrow t \in group\_target(p)$
$inv18 : \forall t, p \cdot t \in TARGETS \wedge p \in PCK \wedge t \mapsto p \in rcvd\_by\_t \Rightarrow t \in group\_target(p)$

— $inv7$ exprime que chaque paquet reçu par un destinataire a été au préalable acheminé vers ce dernier.
— $inv8$ exprime que chaque paquet acheminé vers un destinataire a été auparavant envoyé à ce dernier par un routeur RP.
— $inv9$ est déduit de $inv8$ et $inv7$: chaque paquet reçu par un destinataire a été auparavant envoyé à ce dernier par un routeur RP.
— $inv10$ exprime que chaque paquet envoyé aux destinataires est passé au préalable par le routeur RP qui a été choisi par les routeurs DRs pour le recevoir en premier.

— $inv11$ exprime que chaque paquet reçu par un routeur RP non-choisi par les routeurs DRs, a été au préalable acheminé vers ce dernier.
— $inv12$ exprime que chaque paquet acheminé vers un routeur RP non-choisi par les routeurs DRs, a été au préalable envoyé par le routeur RP choisi par les routeurs DRs à destination du routeur RP non-choisi.
— $inv13$ exprime que chaque paquet envoyé vers un RP non-choisi par les routeurs DRs, est passé au préalable par le routeur RP qui a été choisi par les routeurs DRs pour le recevoir en premier.
— $inv14$ exprime que si un paquet a été envoyé vers un RP non-choisi par les routeurs DRs, cela veut dire qu'il a été au préalable envoyé par le routeur RP choisi par les routeurs DRs aux destinataires qui lui rejoint (qui lui sont connectés et qui ont choisi de recevoir du traffic de sa part).
— $inv15$ exprime qu'un routeur RP rp vers qui un paquet p est acheminé lors de cette troisième phase de l'algorithme ANYCAST RP, n'est pas le routeur RP choisi par les routeurs DRs pour recevoir le paquet p en premier.
— $inv16$ exprime qu'un routeur RP rp qui reçoit un paquet p lors de cette troisième phase de l'algorithme ANYCAST RP, n'est pas le routeur RP choisi par les routeurs DRs pour recevoir le paquet p en premier.
— $inv17$ exprime que si un paquet p est acheminé vers un destinataire t, alors cela signifie que ce destinataire t fait partie du groupe de destinataires ayant demandé à recevoir le paquet p.
— $inv18$ exprime que si un paquet p est acheminé vers un destinataire t, alors cela signifie que ce destinataire t est bien un membre du groupe de destinataires ayant demandé à recevoir le paquet p. Cette propriété est déduite de $inv7$ et $inv17$.

Nous introduisons également les événements de la machine M4 : SENDING4, DR_RECEIVING4, DR_SEN - DING4, DR_RESENDING4, RP_RECEIVING4, RP_SENDING4, RECEIVING4, LOSING4 sont des raffinements des événements de la machine M3 précédente ; de nouveaux événements sont définis : RP_SENDING_ - OTHER_RPS4, O_RP_RECEIVING4, RP_SENDING_TGTS4, TGT_RECEIVING4 modélisent respectivement l'envoi de paquets par les routeurs RPs choisis par les routeurs DRs aux routeurs RPs non-choisis, la réception de paquets par les routeurs RPs non-choisis par les routeurs DRs, l'envoi de paquets par les routeurs RPs aux destinataires qui leurs sont connectés, et la réception d'un paquet par un destinataire.

Nous détaillons maintenant la machine M4 à l'aide d'une liste Φ_4 de propriétés de vivacité, détaillée en annexe. Pour cela, nous commençons par l'initialisation des variables ajoutées lors de ce raffinement :

INITIALISATION $\widehat{=}$
BEGIN
...
$\oplus$ $sent\_to\_t, t\_r\_pck, rcvd\_by\_t, sent\_to\_o\_rp, o\_rp\_r\_pck, rcvd\_by\_o\_rp := \varnothing$
END

Aucun paquet ne circule à l'état initial entre les routeurs RPs et les destinataires : les variables $sent\_to\_t$, t_r_pck, $rcvd\_by\_t$, $sent\_to\_o\_rp$, $o\_rp\_r\_pck$, $rcvd\_by\_o\_rp$ sont donc initialisées à l'aide de l'ensemble vide ($\varnothing$).

Nous détaillons ensuite les événements de la machine M4. Les événements RP_SENDING_TGTS4, TGT_RECEIVING4, RP_SENDING_OTHER_RPS4, O_RP_RECEIVING4 et RECEIVING4 étant nouveaux ou étant les seuls proposant des modifications après raffinement, nous ne détaillons que ces derniers :

— L'événement RP_SENDING_TGTS4 modélise l'envoi d'un paquet p à un groupe de destinataires grt par un routeur RP rp. Le routeur rp ($grd1$), qui a reçu le paquet p d'un routeur DR ou qui a reçu le paquet p d'un autre routeur RP ($grd4$), diffuse le paquet p ($act1$) à un groupe grt composé des destinataires du paquet p qui ont joint le routeur rp ($grd3$ et $grd5$). Le paquet p est marqué comme étant acheminé à destination des membres du groupe grt ($act2$).

EVENT RP_SENDING_TGTS4 $\widehat{=}$
ANY
rp, p, grt
WHERE
$grd1 : rp \in RPs$
$grd2 : p \in PCK$
$grd3 : grt \subseteq TARGETS$
$grd4 : (rp \mapsto p \in sent\_by\_rp) \vee (rp \mapsto p \in rcvd\_by\_o\_rp)$
$grd5 : grt = join^{-1}[\{rp\}] \cap group\_target(p)$
THEN
$act1 : sent\_to\_t := sent\_to\_t \cup \{rp \mapsto p\}$
$act2 : t\_r\_pck := t\_r\_pck \cup (grt \times \{p\})$
END

— L'événement TGT_RECEIVING4 modélise la réception d'un paquet p par un destinataire t ($act1$) vers lequel le paquet p a été acheminé ($grd3$).

EVENT TGT_RECEIVING4 $\widehat{=}$
ANY
t, p
WHERE
$grd1 : t \in TARGETS$
$grd2 : p \in PCK$
$grd3 : t \mapsto p \in t\_r\_pck$
THEN
$act1 : rcvd\_by\_t := rcvd\_by\_t \cup \{t \mapsto p\}$
END

— L'événement RP_SENDING_OTHER_RPS4 modélise l'envoi d'un paquet p, reçu d'un routeur DR ($grd5$), par un Point de Rendez-vous rp ($act1$) à un groupe grp qui contient tous les routeurs RPs différents de rp ($grd6$). Pour que l'envoi puisse se faire, le paquet p doit avoir été au préalable envoyé par le routeur rp aux destinataires qui l'ont rejoint ($grd4$). Lors de l'envoi, le paquet p est aussi marqué comme étant acheminé aux membres du groupe grp ($act2$).

EVENT RP_SENDING_OTHER_RPS4 $\widehat{=}$
ANY
rp, p, grp
WHERE
$grd1 : rp \in RPs$
$grd2 : p \in PCK$
$grd3 : grp \subseteq RPs$
$grd4 : rp \mapsto p \in sent\_to\_t$
$grd5 : rp \mapsto p \in sent\_by\_rp$
$grd6 : grp = RPs \setminus \{rp\}$
THEN
$act1 : sent\_to\_orp := sent\_to\_o\_rp \cup \{rp \mapsto p\}$
$act2 : o\_rp\_r\_pck := o\_rp\_r\_pck \cup (grp \times \{p\})$
END

— L'événement O_RP_RECEIVING4 modélise la réception d'un paquet p ($act1$), transitant vers un routeur RP rp ($grd3$) (non-choisi par les routeurs DRs pour recevoir le paquet p en premier), par ce dernier.

EVENT O_RP_RECEIVING4 $\widehat{=}$
ANY
rp, grp
WHERE
$grd1 : rp \in RPs$
$grd2 : p \in PCK$
$grd3 : rp \mapsto p \in o\_rp\_r\_pck$
THEN
$act1 : rcvd\_by\_o\_rp := rcvd\_by\_o\_rp \cup \{rp \mapsto p\}$
END

— L'événement RECEIVING4 raffine l'événement RECEIVING3 de la machine M3 vue précédemment. Nous exprimons ici le fait qu'un paquet p est considéré comme étant reçu ($act1$) par tous ses destinaires membres du groupe gt ($grd3$), si :
— Chaque destinataire membre du groupe grt a reçu le paquet p individuellement ($grd4$).

— Le paquet p a transité par tous les routeurs RPs ($grd5$).

EVENT RECEIVING4
REFINES RECEIVING3 $\widehat{=}$
  ANY
    gt, p
  WHERE
    $grd1$: $gt \subseteq TARGETS$
    $grd2$: $p \in PCK$
    $grd3$: $gt = group\_target(p)$
    $grd4$: $rcvd\_by\_t^{-1}[\{p\}] = gt$
    $grd5$: $rcvd\_by\_o\_rp^{-1}[\{p\}] \cup rcvd\_by\_rp^{-1}[\{p\}] = RPs$
    $grd6$: $p \notin got$
  THEN
    $act1$: $got := got \cup \{p\}$
  END

Ce troisième raffinement est résumé par le diagramme 5.8. Ce raffinement et les propriétés de vivacité le caractérisant nous ont permis de mieux détailler les trois principales phases algorithmiques du protocole Anycast RP :

1. Une première diffusion d'un paquet p, par une source s à un Routeur Désigné dr.
2. Une diffusion *anycast* du paquet p par le Routeur Désigné dr à un Point de Rendez-vous rp choisi pour son optimalité (e.g. charge du routeur, proximité géographique, performance, etc).
3. Une diffusion *multicast* finale du paquet p par le routeur rp aux destinataires, caractérisée par **(a)** une diffusion *multicast* par le Point de Rendez-vous rp, choisi par le Routeur Désigné dr responsable de l'acheminement du paquet, aux autres Points de Rendez-vous et par **(b)** une diffusion *multicast* du paquet par les Points de Rendez-vous aux destinataires qui les ont « rejoint ». C'est cette phase qui a été détaillée principalement lors de ce raffinement.

Le diagramme met également en avant le fait que les propriétés de vivacité de Φ_3 sont satisfaites par M4 (pour la preuve, voir l'annexe). Dans les prochains raffinements, nous enrichissons et détaillons les phases identifiées à l'aide des propriétés de vivacité et du raffinement, jusqu'à l'obtention d'un modèle local proche de l'algorithme 1 donné dans la deuxième section. Nous introduisons dans la section suivante l'encapsulation des paquets émis lors du protocole Anycast RP en messages appelés « register ».

5.3.2.6 Cinquième raffinement : messages « register »

Durant le protocole Anycast RP, les paquets sont encapsulés dans des messages appelés « register » ([18], 1), après leur réception par les Routeurs Désignés, lors de leur envoi à destination des Points de Rendez-vous. Ces messages « register » servent notamment à la découverte des routes entre les Routeurs Désignés et les Points de Rendez-vous, à la maintenance et à la reconstruction de ces routes, le cas échéant. Lors que les messages arrivent au niveau des routeurs RPs, ces derniers désencapsulent les messages « register » et transmettent les paquets ainsi obtenus aux destinataires qui les ont « rejoint ».

Pour prendre en compte l'encapsulation de paquets, nous définissons un contexte C3 étendant le précédent contexte C2 :

CONTEXT C3 EXTENDS C2
  SETS
    REG
  CONSTANTS
    $enc, desenc$
  AXIOMS
    $axm1$: $enc \in PCK \rightarrowtail REG$
    $axm2$: $desenc \in REG \rightarrowtail PCK$
    $axm3$: $desenc = enc^{-1}$
  END

Nous introduisons dans le contexte C3 une bijection *enc* de l'ensemble des paquets PCK vers l'ensemble des messages « register » REG. Nous y définissons aussi une bijection *desenc*, inverse de la fonction *enc* et modélisant la désencapsulation d'un message « register » en paquet. Ce contexte C3 est utilisé par le cinquième raffinement M5.

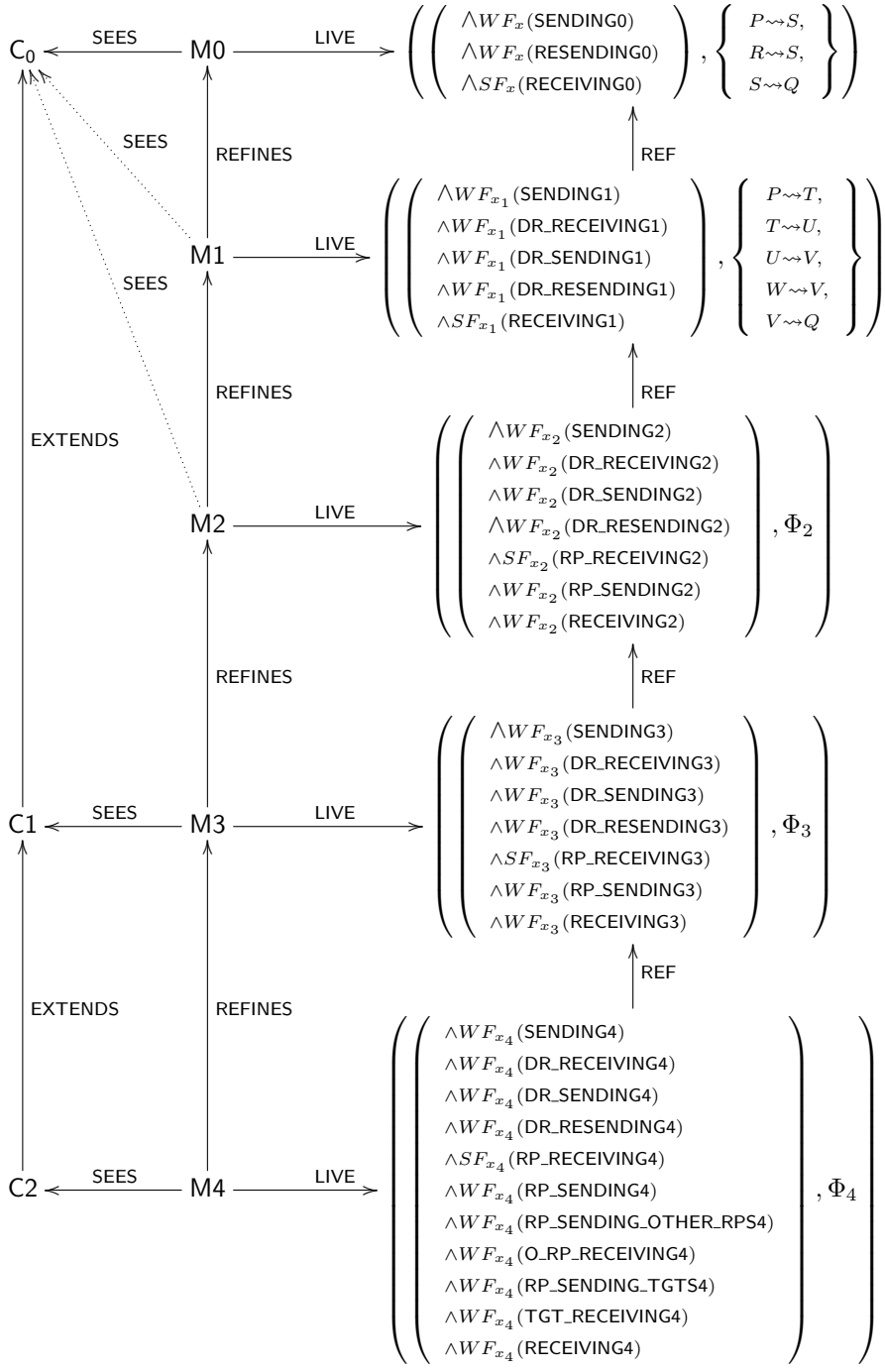


FIGURE 5.8 – Quatrième raffinement du protocole ANYCAST RP

Les variables x_5 de cette machine sont définies comme suit : des variables prenant en compte les échanges de messages « register » entre les routeurs DRs et routeurs RPs sont introduites dans le raffinement M5 et remplacent les anciennes variables ne prenant en compte que les paquets :

— *reg_sent_by_dr* contient les messages « register » envoyé par les routeurs DRs et remplace *sent_by_dr*.
— *reg_dr_s_pck* contient les messages « register » dont la source est un routeur DR et remplace *dr_s_pck*.
— *reg_rp_r_pck* contient les messages « register » transitant vers les routeurs RPs (choisis par les routeurs DRs) et remplace *rp_r_pck*.
— *reg_rcvd_by_rp* contient les messages « register » reçus par les routeurs RPs (choisis par les routeurs DRs). Elle remplace *rcvd_by_rp*.
— *reg_lost* remplace *lost*. Ce ne sont plus des paquets qui sont maintenant perdus, ce sont des messages.
— *reg_sent_by_rp* remplace *sent_by_rp*. Les paquets transitant par les routeurs RPs (choisis par les routeurs DRs) sont maintenant encapsulés dans des messages « register ».
— *reg_sent_to_o_rp* remplace *sent_to_o_rp*. Les paquets envoyés par les routeurs RPs (choisis par les routeurs DRs) aux autres routeurs RPs sont maintenant encapsulés dans des messages « register ».
— *reg_o_rp_r_pck* contient les messages « register » transitant vers les routeurs RPs non-choisis par les routeurs DRs et remplace *o_rp_r_pck*
— *reg_rcvd_by_o_rp* remplace *rcvd_by_o_rp*. Les paquets reçus par les routeurs RPs non-choisis par les routeurs DRs sont encapsulés dans des messages « register ».

Des invariants $I_5(x_5)$ contraignent ces variables :

— Nous avons des propriétés qui typent les nouvelles variables.

$$
\begin{array}{ll}
inv1 & : reg\_sent\_by\_dr \in DRs \leftrightarrow REG \\
inv2 & : reg\_dr\_s\_pck \in REG \nrightarrow DRs \\
inv3 & : reg\_rcvd\_by\_rp \in RPs \leftrightarrow REG \\
inv4 & : reg\_rp\_r\_pck \in RPs \leftrightarrow REG \\
inv5 & : reg\_sent\_by\_rp \in RPs \leftrightarrow REG \\
inv6 & : reg\_sent\_to\_o\_rp \in RPs \leftrightarrow REG \\
inv7 & : reg\_rcvd\_by\_o\_rp \in RPs \leftrightarrow REG \\
inv8 & : reg\_o\_rp\_r\_pck \in RPs \leftrightarrow REG \\
inv9 & : reg\_lost \subseteq REG
\end{array}
$$

— Les propriétés de $inv10$ à $inv27$ définissent les relations entre les variables abstraites (celles que nous avons remplacées) et les variables concrètes (celles qui remplacent les variables abstraites). Nous exprimons avec ces invariants que pour chaque paquet qui est soit envoyé, soit cheminant vers un routeur intermédiaire, soit reçu au niveau abstrait, nous avons au niveau concret le message « register » équivalent, qui est lui aussi soit envoyé, soit cheminant vers un routeur intermédiaire ou bien reçu.

$$
\begin{array}{ll}
inv10 & : \forall r, dr \cdot dr \in DRs \wedge r \in REG \wedge dr \mapsto r \in reg\_sent\_by\_dr \Rightarrow dr \mapsto desenc(r) \in sent\_by\_dr \\
inv11 & : \forall p, dr \cdot dr \in DRs \wedge p \in PCK \wedge dr \mapsto p \in sent\_by\_dr \Rightarrow dr \mapsto enc(p) \in reg\_sent\_by\_dr \\
inv12 & : \forall r, dr \cdot dr \in DRs \wedge r \in REG \wedge r \mapsto dr \in reg\_dr\_s\_pck \Rightarrow dr \mapsto desenc(r) \in dr\_s\_pck \\
inv13 & : \forall p, dr \cdot dr \in DRs \wedge p \in PCK \wedge dr \mapsto p \in dr\_s\_pck \Rightarrow enc(p) \mapsto dr \in reg\_dr\_s\_pck \\
inv14 & : \forall r, rp \cdot rp \in RPs \wedge r \in REG \wedge rp \mapsto r \in reg\_rp\_r\_pck \Rightarrow rp \mapsto desenc(r) \in rp\_r\_pck \\
inv15 & : \forall p, rp \cdot rp \in RPs \wedge p \in PCK \wedge rp \mapsto p \in rp\_r\_pck \Rightarrow rp \mapsto enc(p) \in reg\_rp\_r\_pck \\
inv16 & : \forall r, rp \cdot rp \in RPs \wedge r \in REG \wedge rp \mapsto r \in reg\_rcvd\_by\_rp \Rightarrow rp \mapsto desenc(r) \in rcvd\_by\_rp \\
inv17 & : \forall p, rp \cdot rp \in RPs \wedge p \in PCK \wedge rp \mapsto p \in rcvd\_by\_rp \Rightarrow rp \mapsto enc(p) \in reg\_rcvd\_by\_rp \\
inv18 & : \forall r, rp \cdot rp \in RPs \wedge r \in REG \wedge rp \mapsto r \in reg\_sent\_by\_rp \Rightarrow rp \mapsto desenc(r) \in sent\_by\_rp \\
inv19 & : \forall p, rp \cdot rp \in RPs \wedge p \in PCK \wedge rp \mapsto p \in sent\_by\_rp \Rightarrow rp \mapsto enc(p) \in reg\_sent\_by\_rp \\
inv20 & : \forall r, rp \cdot rp \in RPs \wedge r \in REG \wedge rp \mapsto r \in reg\_sent\_to\_o\_rp \Rightarrow rp \mapsto desenc(r) \in sent\_to\_o\_rp \\
inv21 & : \forall p, rp \cdot rp \in RPs \wedge p \in PCK \wedge rp \mapsto p \in sent\_to\_o\_rp \Rightarrow rp \mapsto enc(p) \in reg\_sent\_to\_o\_rp \\
inv22 & : \forall r, rp \cdot rp \in RPs \wedge r \in REG \wedge rp \mapsto r \in reg\_o\_rp\_r\_pck \Rightarrow rp \mapsto desenc(r) \in o\_rp\_r\_pck \\
inv23 & : \forall p, rp \cdot rp \in RPs \wedge p \in PCK \wedge rp \mapsto p \in o\_rp\_r\_pck \Rightarrow rp \mapsto enc(p) \in reg\_o\_rp\_r\_pck \\
inv24 & : \forall r, rp \cdot rp \in RPs \wedge r \in REG \wedge rp \mapsto r \in reg\_rcvd\_by\_o\_rp \Rightarrow rp \mapsto desenc(r) \in rcvd\_by\_o\_rp \\
inv25 & : \forall p, rp \cdot rp \in RPs \wedge p \in PCK \wedge rp \mapsto p \in rcvd\_by\_o\_rp \Rightarrow rp \mapsto enc(p) \in reg\_rcvd\_by\_o\_rp \\
inv26 & : \forall p \cdot p \in PCK \wedge p \in lost \Rightarrow enc(p) \in reg\_lost \\
inv27 & : \forall r \cdot r \in REG \wedge r \in reg\_lost \Rightarrow desenc(r) \in lost
\end{array}
$$

— Nous avons des propriété de sûreté.

$$
\begin{array}{ll}
inv28 & : \ dom(reg\_dr\_s\_pck) \subseteq ran(reg\_sent\_by\_dr) \\
inv29 & : \ ran(reg\_rp\_r\_pck) \subseteq dom(reg\_dr\_s\_pck) \\
inv30 & : \ reg\_rcvd\_by\_rp \subseteq reg\_rp\_r\_pck \\
inv31 & : \ ran(reg\_rp\_r\_pck) \subseteq ran(reg\_sent\_by\_dr) \\
inv32 & : \ ran(reg\_rcvd\_by\_rp) \subseteq ran(reg\_sent\_by\_dr) \\
inv33 & : \ ran(reg\_rcvd\_by\_rp) \subseteq dom(reg\_dr\_s\_pck) \\
inv34 & : \ reg\_sent\_by\_rp \subseteq reg\_rcvd\_by\_rp \\
inv35 & : \ ran(reg\_rcvd\_by\_o\_rp) \subseteq ran(reg\_o\_rp\_r\_pck) \\
inv36 & : \ ran(reg\_o\_rp\_r\_pck) \subseteq ran(reg\_sent\_to\_o\_rp)
\end{array}
$$

1. $inv28$ exprime que chaque message « register » dont la source est un routeur DR a été envoyé par ce dernier.
2. $inv29$ exprime que chaque message « register » à destination d'un routeur RP choisi, a pour source un routeur DR.
3. $inv30$ exprime que chaque message « register » reçu par un routeur RP, choisi par un routeur DR, a transité à destination du routeur RP.
4. $inv31$ exprime que chaque message « register » transitant vers un routeur RP, choisi par un routeur DR, a été envoyé par un routeur DR.
5. $inv32$ et $inv33$ expriment que chaque message « register » reçu par un routeur RP, choisi par un routeur DR, a été envoyé au routeur RP par le routeur DR.
6. $inv34$ exprime que chaque message « register » en instance d'envoi par un routeur RP, choisi par un routeur DR, a été au préalable reçu par le routeur RP.
7. $inv35$ et $inv36$ expriment que chaque message « register » reçu ou à destination de routeurs RPs, non-choisis par les routeurs DRs, a été au préalable envoyé à ces routeurs RPs par le routeur RP choisi par les routeurs DRs.

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
$sent\_by\_dr$	$reg\_sent\_by\_dr$
dr_s_pck	$reg\_dr\_s\_pck$
$rcvd\_by\_rp$	$reg\_rcvd\_by\_rp$
$lost$	reg_lost
$sent\_by\_rp$	$reg\_sent\_by\_rp$
$sent\_to\_o\_rp$	$reg\_sent\_to\_o\_rp$
$o\_rp\_r\_pck$	$reg\_o\_rp\_r\_pck$
$rcvd\_by\_o\_r$	$reg\_rcvd\_by\_o\_rp$

Les axiomes $axm1$, $axm2$ et $axm3$ du contexte C3, ainsi que l'invariant de collage I_5 nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M4 en M5, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M5, ainsi que pour justifier la préservation des propriétés Φ_4 lors du raffinement.

Une liste Φ_5 de propriétés de vivacité satisfaites par la machine M5, détaillée en annexe nous permet ensuite de guider le développement de la machine M5. Les événements DR_SENDING4, DR_RESENDING4, LOSING4, RP_RECEIVING4, RP_SENDING4, RP_SENDING_TGTS4, RP_SENDING_OTHER_RPS4, O_RP_-RECEIVING4, RECEIVING4 sont impactés par l'introduction des encapsulations de paquets et désencapsulations de messages « register ». Nous les modifions donc par raffinement dans la machine M5.

— Aucune diffusion ou réception de messages « register » n'ayant lieu à l'état initial, les variables de M5 sont initialisées à l'aide de l'ensemble vide ($\varnothing$).

INITIALISATION $\widehat{=}$
BEGIN
$\ominus sent\_by\_dr := \varnothing$
$\ominus dr\_s\_pck := \varnothing$
$\ominus rcvd\_by\_rp := \varnothing$
$\ominus rp\_r\_pck := \varnothing$
$\ominus lost := \varnothing$
$\ominus sent\_by\_rp := \varnothing$
$\ominus rcvd\_by\_o\_rp := \varnothing$
$\ominus o\_rp\_r\_pck := \varnothing$
$\ominus sent\_to\_o\_rp := \varnothing$
$\oplus reg\_sent\_by\_dr, \quad \oplus reg\_dr\_s\_pck := \varnothing$
$\oplus reg\_rcvd\_by\_rp := \varnothing$
$\oplus reg\_rp\_r\_pck, \quad \oplus reg\_lost := \varnothing$
$\oplus reg\_sent\_by\_rp := \varnothing$
$\oplus reg\_rcvd\_by\_o\_rp := \varnothing$
$\oplus reg\_sent\_to\_o\_rp := \varnothing$
$\oplus reg\_o\_rp\_r\_pck := \varnothing$
END

— L'événement DR_SENDING5 prend maintenant en compte les messages « register » : un paquet p reçu (d'une source s) par un routeur DR dr est encapsulé dans un message « register » reg ($grd8$), et est ensuite envoyé par le routeur dr ($act1$, $act3$) à destination d'un routeur RP rp, unique membre du groupe de routeurs destinataires grp ($act2$).

EVENT DR_SENDING5
REFINES DR_SENDING4 $\widehat{=}$
ANY
...
$\oplus$ reg
WHERE
...
...
$\ominus$ $grd6$: $dr \mapsto p \notin sent\_by\_dr$
$\oplus$ $grd6$: $reg \in REG$
$\oplus$ $grd8$: $reg = enc(p)$
$\oplus$ $grd9$: $dr \mapsto reg \notin reg\_sent\_by\_dr$
THEN
$\ominus$ $act1, act2, act3$
$\oplus$ $act1$: $reg\_sent\_by\_dr := reg\_sent\_by\_dr \cup \{dr \mapsto reg\}$
$\oplus$ $act2$: $reg\_rp\_r\_pck := reg\_rp\_r\_pck \cup (grp \times \{reg\})$
$\oplus$ $act3$: $reg\_dr\_s\_pck := reg\_dr\_s\_pck \cup \{reg \mapsto dr\}$
END

— Nous remplaçons dans l'événement DR_RESENDING5 le paquet p à renvoyer par le message « register » reg correspondant ($p = desenc(reg)$). Si le message reg était perdu, l'événement DR_RESENDING5 permet de le récupérer.

EVENT DR_RESENDING5
REFINES DR_RESENDING4 $\widehat{=}$
ANY
...
$\ominus$ p
$\oplus$ reg
WHERE
...
$\ominus$ $grd1, grd3$
$\oplus$ $grd1$: $reg \in REG$
...
$\oplus$ $grd3$: $dr \mapsto reg \in reg\_sent\_by\_dr$
WITH
p : $p = desenc(reg)$
THEN
$\ominus$ $act1$
$\oplus$ $act1$: $reg\_lost := reg\_lost \setminus \{reg\}$
END

— La modification que nous apportons à l'événement LOSING5 est la suivante : désormais ce ne sont plus des paquets p qui sont perdus, mais les messages « register » correspondants reg ($p = desenc(reg)$).

EVENT LOSING5
REFINES LOSING4 $\widehat{=}$
  ANY
    $\ominus\ p$
    $\oplus\ reg$
  WHERE
    $\ominus\ grd1, grd2, grd3$
    $\oplus\ grd1\ :\ reg \in REG$
    $\oplus\ grd2\ :\ reg \in ran(reg\_rp\_r\_pck)$
    $\oplus\ grd3\ :\ reg \notin (ran(reg\_rcvd\_by\_rp) \cup reg\_lost)$
  WITH
    $p\ :\ p = desenc(reg)$
  THEN
    $\ominus\ act1$
    $\oplus\ act1\ :\ reg\_lost := reg\_lost \cup \{reg\}$
  END

— L'événement RP_RECEIVING5 permet maintenant la prise en compte des messages « register » : nous remplaçons la réception, par un routeur RP rp, d'un paquet p par la réception du message « register » reg correspondant.

EVENT RP_RECEIVING5
REFINES RP_RECEIVING4 $\widehat{=}$
  ANY
    ...
    $\ominus\ p$
    $\oplus\ reg$
  WHERE
    ...
    $\ominus\ grd1, grd3, grd4$
    $\oplus\ grd1\ :\ reg \in REG$
    $\oplus\ grd3\ :\ rp \mapsto reg \in reg\_rp\_r\_pck$
    $\oplus\ grd4\ :\ reg \notin reg\_lost$
  WITH
    $p\ :\ p = desenc(reg)$
  THEN
    $\ominus\ act1$
    $\oplus\ act1\ :\ reg\_rcvd\_by\_rp := reg\_rcvd\_by\_rp \cup \{rp \mapsto reg\}$
  END

— Nous raffinons comme précédemment les événements RP_SENDING4, RP_SENDING_OTHER_RPS4, O_RP_RECEIVING4 : les routeurs RPs ne communiquent plus par l'intermédiaire de paquets p, mais par l'intermédiaire des messages « register » reg correspondants.

EVENT RP_SENDING5
REFINES RP_SENDING4 $\widehat{=}$
  ANY
    ...
    $\ominus\ p$
    $\oplus\ reg$
  WHERE
    ...
    $\ominus\ grd1, grd3$
    $\oplus\ grd1\ :\ reg \in REG$
    $\oplus\ grd3\ :\ rp \mapsto reg \in reg\_rcvd\_by\_rp$
  WITH
    $p\ :\ p = desenc(reg)$
  THEN
    $\ominus\ act1$
    $\oplus\ act1\ :\ reg\_sent\_by\_rp := reg\_sent\_by\_rp \cup \{rp \mapsto reg\}$
  END

— Nous remarquons pour l'événement RP_SENDING_OTHER_RPS5 que nous remplaçons la garde $grd4$ par une nouvelle garde : l'envoi d'un message « register » reg, par un routeur RP rp, choisi par un routeur DR, aux autres routeurs RPs, a lieu seulement après que le message « register » reg ait été désencapsulé et envoyé aux destinataires ayant joint le routeur rp.

EVENT RP_SENDING_OTHER_RPS5
REFINES RP_SENDING_OTHER_RPS4 $\widehat{=}$
  ANY
    ...
    $\ominus\ p$
    $\oplus\ reg$
  WHERE
    ...
    $\ominus\ grd2, grd4, grd5$
    $\oplus\ grd2\ :\ reg \in REG$
    $\oplus\ grd4\ :\ rp \mapsto desenc(reg) \in sent\_to\_t$
    $\oplus\ grd5\ :\ rp \mapsto reg \in reg\_sent\_by\_rp$
    ...
  WITH
    $p\ :\ p = desenc(reg)$
  THEN
    $\ominus\ act1, act2$
    $\oplus\ act1\ :\ reg\_sent\_to\_o\_rp := reg\_sent\_to\_o\_rp \cup \{rp \mapsto reg\}$
    $\oplus\ act2\ :\ reg\_o\_rp\_r\_pck := reg\_o\_rp\_r\_pck \cup (grp \times \{reg\})$
  END

EVENT O_RP_RECEIVING5
REFINES O_RP_RECEIVING4 $\widehat{=}$
  ANY
    ...
    $\ominus\ p$
    $\oplus\ reg$
  WHERE
    ...
    $\ominus\ grd2, grd3$
    $\oplus\ grd2\ :\ reg \in REG$
    $\oplus\ grd3\ :\ rp \mapsto reg \in reg\_o\_rp\_r\_pck$
  WITH
    $p\ :\ p = desenc(reg)$
  THEN
    $\ominus\ act1$
    $\oplus\ act1\ :\ reg\_rcvd\_by\_o\_rp := reg\_rcvd\_by\_o\_rp \cup \{rp \mapsto reg\}$
  END

— L'événement RP_SENDING_TGTS5 modélise la désencapsulation, par un routeur RP rp, d'un message « register » en un paquet, avant l'envoi de ce dernier ($act1$) à un groupe de destinataires grt ($act2$), ayant joint le routeur RP rp ($grd5$).

EVENT RP_SENDING_TGTS5
REFINES RP_SENDING_TGTS4 $\widehat{=}$
  ANY
    ...
    $\ominus\ p$
    $\oplus\ reg$
  WHERE
    ...
    $\ominus\ grd2, grd4, grd5$
    $\oplus\ grd2\ :\ reg \in REG$
    ...
    $\oplus\ grd4\ :\ (rp \mapsto reg \in reg\_by\_rp) \vee (rp \mapsto reg \in reg\_rcvd\_by\_o\_rp)$
    $\oplus\ grd5\ :\ grt = join^{-1}[\{rp\}] \cap group\_target(desenc(reg))$
    ...
  WITH
    $p\ :\ p = desenc(reg)$
  THEN
    $\ominus\ act1, act2$
    $\oplus\ act1\ :\ sent\_to\_t := sent\_to\_t \cup \{rp \mapsto desenc(reg)\}$
    $\oplus\ act2\ :\ t\_r\_pck := t\_r\_pck \cup (grt \times \{desenc(reg)\})$
  END

— La garde $grd5$ de l'événement RECEIVING5 est modifiée. Un paquet est considéré comme reçu par tous ses destinataires, s'il a été au préalable reçu individuellement par ces derniers et si le message « register » reg correspondant a été reçu par tous les routeurs RPs ($grd5$).

EVENT RECEIVING5
REFINES RECEIVING4 $\widehat{=}$
  ANY
    gt, p
  WHERE
    $\ominus$ $grd5$
    $\oplus$ $grd5$: $reg\_rcvd\_by\_o\_rp^{-1}[\{enc(p)\}] \cup reg\_rcvd\_by\_rp^{-1}[\{enc(p)\}] = RPs$
    ...
  THEN
    $act1$: $got := got \cup \{p\}$
  END

Le diagramme 5.9 résume ce raffinement de M4 en M5. Un autre type de message, appelé « register-stop » est renvoyé par les routeurs RPs à ceux qui leur ont diffusé des messages « register ». Il s'agit d'un message similaire à un message *d'aquittement* : les routeurs RPs signifient à l'émetteur des messages « register » que ce dernier a été reçu et qu'il n'est donc plus nécessaire de le renvoyer. Le prochain raffinement introduit ce nouveau message.

5.3.2.7 Sixième raffinement : messages « register-stop »

Ce sixième raffinement propose de prendre en compte un autre type de message, envoyé par les routeurs RPs en réponse aux messages « register » : il s'agit des messages « register-stop » [18, 17]. Ce message est envoyé par les routeurs RPs à deux destinations :

1. Au routeur DR ayant diffusé des messages « register », à destination d'un routeur RP choisi. Le routeur RP demande ainsi au routeur DR d'arrêter la diffusion de messages « register » encapsulant un paquet à destination d'un groupe donné de destinataires. Il permet ainsi au routeur DR de savoir qu'il a pu transmettre le paquet encapsulé dans le message « register » aux destinataires de ce dernier qui l'ont rejoint et qu'une route menant de la source du paquet à (une partie de) ses destinataires a été découverte.
2. Les routeurs RPs non-choisis par le routeur DR envoient un message « register-stop » au routeur RP choisi, pour les mêmes raisons que citées précédemment, à savoir, l'arrêt de la diffusion par le RP choisi de messages « register » et pour signaler qu'une route entre les sources et les destinataires a été découverte.

Nous commençons par définir un contexte C4 étendant le contexte C3 vu dans la section précédente et introduisant un ensemble de messages « register-stop » REG_STOP non-vide. Ce contexte C4 est utilisé par la machine M6.

CONTEXT C4 EXTENDS C3
  SETS
    REG_STOP
  AXIOMS
    $axm1$: $REG\_STOP \neq \varnothing$
  END

Les variables x_6 de ce raffinement sont composées des variables x_5 de la machine M5 précédente et de nouvelles variables, qui prenent en compte les messages « register-stop » :

— $rs\_sent\_by\_rp$ contient les messages « register-stop » envoyés par un routeur RP choisi par un routeur DR, en réponse à un message « register » reçu de ce routeur DR.
— rs_dr_r contient les messages « register-stop » envoyés par un routeur RP, choisi par un routeur DR, transitant à destination de ce routeur DR.
— Les messages « register-stop » envoyés en réponse à un message « register » et reçus par un Routeur Désigné (DR) sont contenus dans la variable $rs\_rcvd\_by\_dr$.
— $dr\_reg\_ack\_rs$ contient les messages « register » pour lesquels un routeur DR a reçu un message « register-stop ».

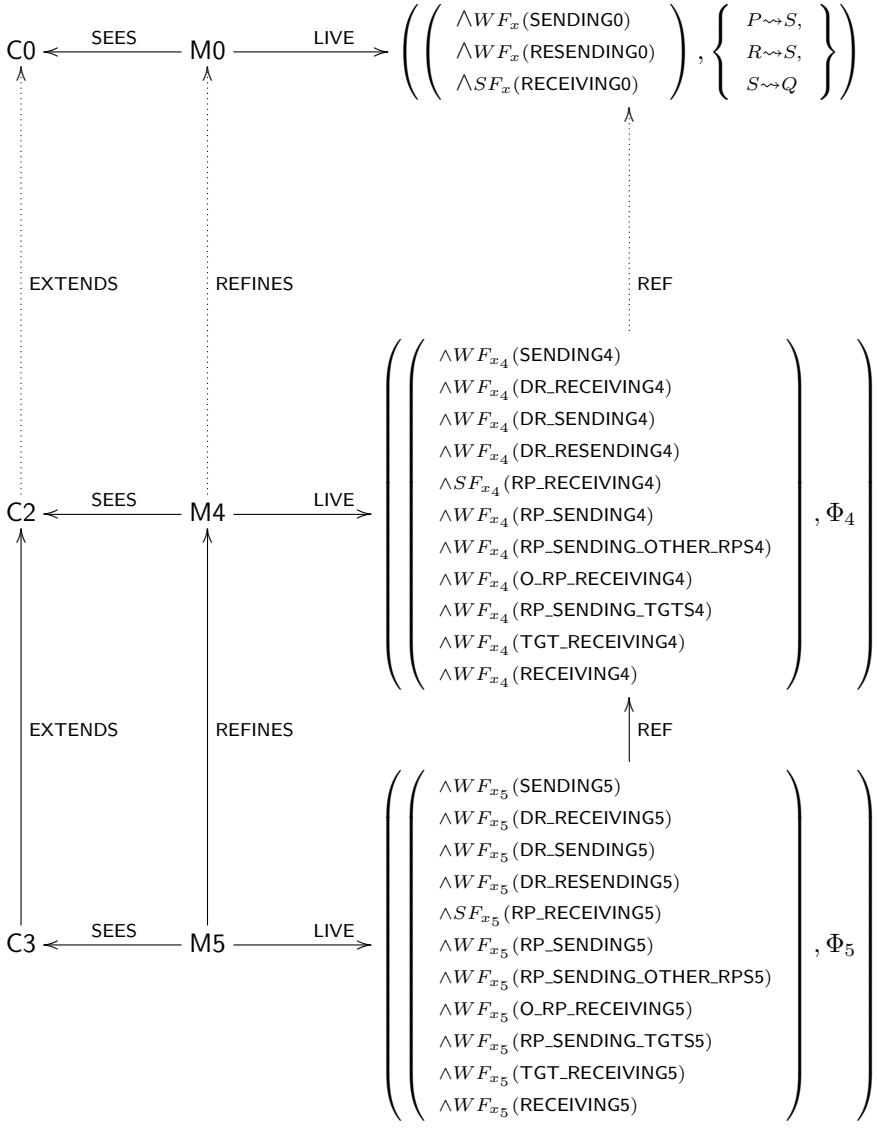


FIGURE 5.9 – Cinquième raffinement du protocole ANYCAST RP

— *rs_sent_by_o_rp* contient les messages « register-stop » envoyés par les routeurs RP non-choisis par un routeur DR, en réponse à un message « register » reçu du routeur RP choisi.
— *rs_rp_r* contient les messages « register-stop » envoyés par les routeurs RP non-choisis par un routeur DR, transitant à destination du routeur RP choisi.
— Les messages « register-stop » envoyés en réponse à un message « register » et reçus par un Point de Rendez-vous (RP) choisi par un routeur DR sont contenus dans la variable *rs_rcvd_by_rp*.
— *rp_reg_ack_rs* contient pour un routeur RP choisi par un routeur DR, le message « register » pour lequel il a reçu un message « register-stop », ainsi que l'identité du routeur RP non-choisi duquel il a reçu le message « register-stop ».

L'invariant $I_6(x_6)$ contraignant ces variables est défini comme suit :

— Les propriétés de $inv1$ à $inv8$ typent les variables, conformément à leurs descriptions données un peu plus haut.

$$
\begin{array}{l}
inv1 \ : \ rs\_sent\_by\_rp \in RPs \leftrightarrow (REG\_STOP \times REG) \\
inv2 \ : \ rs\_rcvd\_by\_dr \in DRs \leftrightarrow (REG\_STOP \times REG) \\
inv3 \ : \ rs\_dr\_r \in DR \leftrightarrow (REG\_STOP \times REG) \\
inv4 \ : \ dr\_reg\_ack\_rs \in DRs \leftrightarrow REG \\
inv5 \ : \ rs\_sent\_by\_o\_rp \in RPs \leftrightarrow (REG\_STOP \times REG) \\
inv6 \ : \ rs\_rp\_r \in RP \leftrightarrow (REG\_STOP \times REG) \\
inv7 \ : \ rs\_rcvd\_by\_rp \in RPs \leftrightarrow (REG\_STOP \times REG) \\
inv8 \ : \ rp\_reg\_ack\_rs \in RPs \leftrightarrow (RP \times REG)
\end{array}
$$

— Des propriétés de sûreté caractérisant les échanges de messages « register-stop » entre un routeur RP choisi par un routeur DR et ce dernier.

$$
\begin{array}{l}
inv9 \ : \ ran(rs\_dr\_r) \subseteq ran(rs\_sent\_by\_rp) \\
inv10 \ : \ rs\_rcvd\_by\_dr \subseteq rs\_dr\_r \\
inv11 \ : \ \forall rp1, rp2, rs \cdot \left(\begin{array}{l} \wedge rp1 \in RPs \wedge rp2 \in RPs \wedge rs \in (REG\_STOP \times REG) \\ \wedge rp1 \mapsto rs \in rs\_sent\_by\_rp \wedge rp2 \mapsto rs \in rs\_sent\_by\_rp \end{array} \right) \Rightarrow rp1 = rp2 \\
inv12 \ : \ \forall dr1, dr2, rs \cdot \left(\begin{array}{l} \wedge dr1 \in DRs \wedge dr2 \in DRs \wedge rs \in (REG\_STOP \times REG) \\ \wedge dr1 \mapsto rs \in rs\_dr\_r \wedge dr2 \mapsto rs \in rs\_dr\_r \end{array} \right) \Rightarrow dr1 = dr2 \\
inv13 \ : \ \forall rp, rs \cdot \left(\begin{array}{l} \wedge rp \in RPs \wedge rs \in (REG\_STOP \times REG) \\ \wedge rp \mapsto rs \in rs\_sent\_by\_rp \end{array} \right) \Rightarrow rp \mapsto prj_2(rs) \in reg\_rp\_r\_pck \\
inv14 \ : \ \forall dr, rs \cdot \left(\begin{array}{l} \wedge dr \in DRs \wedge rs \in (REG\_STOP \times REG) \\ \wedge dr \mapsto rs \in rs\_dr\_r \end{array} \right) \Rightarrow prj_2(rs) \mapsto dr \in reg\_dr\_s\_pck \\
inv15 \ : \ \forall dr, rs \cdot \left(\begin{array}{l} \wedge dr \in DRs \wedge rs \in (REG\_STOP \times REG) \\ \wedge dr \mapsto rs \in rs\_rcvd\_by\_dr \end{array} \right) \Rightarrow dr \mapsto prj_2(rs) \in dr\_reg\_ack\_rs \\
inv16 \ : \ \forall dr, r \cdot dr \in DRs \wedge r \in REG \wedge dr \mapsto r \in dr\_reg\_ack\_rs \Rightarrow r \mapsto dr \in reg\_dr\_s\_pck
\end{array}
$$

— *inv*9 exprime que chaque message « register-stop » à destination d'un routeur DR a été envoyé par un routeur RP choisi par ce dernier, en réponse à un message « register ».
— *inv*10 exprime que chaque message « register-stop » reçu par un routeur DR a au préalable transité vers ce dernier.
— *inv*11 et *inv*12 expriment respectivement l'unicité du routeur RP émetteur d'un message « register-stop » à destination d'un routeur DR et l'unicité du routeur DR vers lequel le message transite et qui va le recevoir.
— *inv*13 exprime que si un message « register-stop » est envoyé par un routeur RP choisi par un routeur DR, en réponse à un message « register », cela signifie que le routeur RP a reçu auparavant le message « register ».
— *inv*14 exprime que si un message « register-stop », envoyé en réponse à un message « register », transite vers un routeur DR, alors ce routeur DR est celui qui a diffusé le message « register ».
— *inv*15 exprime que si un message « register-stop », envoyé en réponse à un message « register », a été reçu par un routeur DR, alors ce dernier a reçu un acquittement pour le message « register » envoyé.
— *inv*16 exprime que si un routeur DR a reçu un acquittement pour un message « register », alors ce routeur DR est le routeur par l'intermédiaire duquel le message « register » a été diffusé.

— Des propriétés de sûreté caractérisant les échanges de messages « register-stop » entre les routeurs RPs non-choisi par un routeur DR et le routeur RP choisi par ce dernier.

$$
\begin{array}{l}
inv17 \ : \ ran(rs\_rp\_r) \subseteq ran(rs\_sent\_by\_o\_rp) \\
inv18 \ : \ rs\_rcvd\_by\_rp \subseteq rs\_rp\_r \\
inv19 \ : \ \forall rp, rs \cdot \left(\begin{array}{l} \wedge rp \in RPs \wedge rs \in (REG\_STOP \times REG) \\ \wedge rp \mapsto rs \in rs\_sent\_by\_o\_rp \end{array} \right) \Rightarrow rp \mapsto prj_2(rs) \notin reg\_sent\_by\_rp \\
inv20 \ : \ \forall rp, rs \cdot \left(\begin{array}{l} \wedge rp \in RPs \wedge rs \in (REG\_STOP \times REG) \\ \wedge rp \mapsto rs \in rs\_sent\_by\_o\_rp \end{array} \right) \Rightarrow prj_2(rs) \in ran(reg\_sent\_by\_rp) \\
inv21 \ : \ \forall rp, rs \cdot \left(\begin{array}{l} \wedge rp \in RPs \wedge rs \in (REG\_STOP \times REG) \\ \wedge rp \mapsto rs \in rs\_rp\_r \end{array} \right) \Rightarrow rp \mapsto prj_2(rs) \in reg\_sent\_by\_rp \\
inv22 \ : \ \forall rp1, rp2, rs \cdot \left(\begin{array}{l} \wedge rp1 \in RPs \wedge rp2 \in RPs \wedge rs \in (REG\_STOP \times REG) \\ \wedge rp1 \mapsto rs \in rs\_rp\_r \wedge rp2 \mapsto rs \in rs\_rp\_r \end{array} \right) \Rightarrow rp1 = rp2 \\
inv23 \ : \ \forall rp, rs \cdot \left(\begin{array}{l} \wedge rp \in RPs \wedge rs \in (REG\_STOP \times REG) \\ \wedge rp \mapsto rs \in rs\_rcvd\_by\_rp \end{array} \right) \Rightarrow prj_2(rs) \in ran(rp\_reg\_ack\_rs[\{rp\}])
\end{array}
$$

— $inv17$ exprime que chaque message « register-stop » à destination d'un routeur RP choisi par un routeur DR, a été envoyé par des routeurs RPs non-choisis, en réponse à un message « register ».

— $inv18$ exprime chaque message « register-stop » reçu par un routeur RP, choisi par un un routeur DR, a au préalable transité vers le routeur RP.

— $inv19$ exprime que si un message « register-stop » a été envoyé par des routeurs RPs à destination d'un autre routeur RP, cela signifie que les routeurs RPs émetteurs ne sont pas des routeurs RPs choisis un routeur DR.

— $inv20$ et $inv21$ expriment que si un message « register-stop »,en réponse à un message « register », a été envoyé par un routeur RP à un autre routeur RP rp, cela veut dire que le second routeur rp est le routeur qui a reçu le message d'un routeur DR et qui l'a ensuite diffusé aux autres routeurs RPs.

— $inv22$ exprime l'unicité du routeur RP récepteur d'un message « register-stop », envoyé en réponse à un message « register ».

— $inv23$ exprime que si un message « register-stop », envoyé en réponse à un message « register », a été reçu par un routeur RP, alors cela signifie que le message « register »a été acquitté au moins par un des autres routeurs RPs.

Une liste Φ_6 des propriétés de vivacité détaillée en annexe et satisfaite par M6, nous permet de construire la machine M6 :

— Evénement INITIALISATION : les nouvelles variables de la machine M6 sont initialisées à l'aide de l'ensemble vide ($\varnothing$) parce qu'aucun message « register » ou « register-stop » ne circule à l'état initial.

INITIALISATION $\widehat{=}$
BEGIN
...
$\oplus$ $rs\_sent\_by\_rp, rs\_rcvd\_by\_dr, rs\_dr\_r, dr\_reg\_ack\_rs := \varnothing$
$\oplus$ $rs\_sent\_by\_o\_rp, rs\_rp\_r, rs\_rcvd\_by\_rp, rp\_reg\_ack\_rs := \varnothing$
END

— Nous modifions par raffinement l'événement DR_RESENDING5 et introduisons maintenant le fait qu'un routeur DR dr renvoie un message « register » à un routeur RP qu'il a choisi, tant qu'il n'a pas reçu de ce dernier un acquittement du message « register », sous la forme d'un message « register-stop » ($grd4$).

EVENT DR_RESENDING6
REFINES DR_RESENDING5 $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd4 \ : \ dr \mapsto reg \notin dr\_reg\_ack\_rs$
THEN
...
END

— Nous raffinons de la même manière l'événement RP_SENDING_OTHER_RPS5 : le raffinement RP_SENDING_OTHER_RPS6 modélise maintenant le fait qu'un message « register », est renvoyé par le routeur rp, choisi par un routeur DR, tant que le routeur rp n'a pas reçu de tous les autres routeurs RPs un message « register-stop » en acquittement du message « register » ($grd7$).

EVENT RP_SENDING_OTHER_RPS6
REFINES RP_SENDING_OTHER_RPS5 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\oplus$ $grd7$: $\exists otrp \cdot otrp \in RPs \land (rp \mapsto (otrp \mapsto reg) \notin rp\_reg\_ack\_rs)$
  THEN
    ...
  END

Nous introduisons aussi dans ce modèle M6 de nouveaux événements. Les événements RP_REG_STOP_SENDING6, DR_REG_STOP_RECEIVING6, DR_REG_STOP_LOSING6 modélisent les échanges de messages « register-stop » entre les routeurs RP choisis par les routeurs DRs et ces derniers.

— L'événement RP_REG_STOP_SENDING6 modélise l'envoi d'un message d'acquittement « register-stop » par un routeur RP rp à un routeur DR dr. Le routeur rp est le routeur RP ayant reçu le message « register » reg ($grd6$) du routeur dr ($grd7$) : le routeur rp envoie ($act1$) donc au routeur dr ($act2$) un message « register-stop » stp en réponse au message « register » reg.

EVENT RP_REG_STOP_SENDING6 $\widehat{=}$
  ANY
    reg, rp, dr, stp, gdr
  WHERE
    $grd1$: $rp \in RPs$
    $grd2$: $dr \in DRs$
    $grd3$: $reg \in REG$
    $grd4$: $stp \in REG\_STOP$
    $grd5$: $gdr = \{dr\}$
    $grd6$: $rp \mapsto reg \in reg\_rcvd\_by\_rp$
    $grd7$: $dr = reg\_dr\_s\_pck(reg)$
  THEN
    $act1$: $rs\_sent\_by\_rp := rs\_sent\_by\_rp \cup \{rp \mapsto (stp \mapsto reg)\}$
    $act2$: $rs\_dr\_r := rs\_dr\_r \cup (gdr \times \{stp \mapsto reg\})$
  END

— L'événement DR_REG_STOP_RECEIVING6 modélise la réception d'un message « register-stop » par un routeur DR dr : un message « register-stop » stp, en réponse à un message message « register » reg, transite à destination du routeur dr ($grd4$). Ce dernier peut ainsi recevoir le message « register-stop » stp ($act2$) et noter qu'un acquittement a été reçu pour le message « register » reg ($act1$).

EVENT DR_REG_STOP_RECEIVING6 $\widehat{=}$
  ANY
    reg, dr, stp
  WHERE
    $grd1$: $dr \in DRs$
    $grd2$: $reg \in REG$
    $grd3$: $stp \in REG\_STOP$
    $grd4$: $dr \mapsto (stp \mapsto reg) \in rs\_dr\_r$
  THEN
    $act1$: $dr\_reg\_ack\_rs := dr\_reg\_ack\_rs \cup \{dr \mapsto reg\}$
    $act2$: $rs\_rcvd\_by\_dr := rs\_rcvd\_by\_dr \cup \{dr \mapsto (stp \mapsto reg)\}$
  END

— L'événement DR_REG_STOP_LOSING6 modélise la possible perte ($act1$, $act2$) d'un message « register-stop » stp envoyé par un routeur RP rp ($grd5$) à destination d'un routeur DR dr ($grd7$), en réponse à un message « register » reg. La perte est possible tant que le routeur dr n'a pas reçu le message stp, c'est-à-dire, tant que le message reg n'a pas été acquitté ($grd6$).

EVENT DR_REG_STOP_LOSING6 $\widehat{=}$
ANY
reg, rp, dr, stp
WHERE
$grd1 : rp \in RPs$
$grd2 : dr \in DRs$
$grd3 : reg \in REG$
$grd4 : stp \in REG\_STOP$
$grd5 : rp \mapsto (stp \mapsto reg) \in rs\_sent\_by\_rp$
$grd6 : dr \mapsto reg \notin dr\_reg\_ack\_rs$
$grd7 : dr \mapsto (stp \mapsto reg) \in rs\_dr\_r$
THEN
$act1 : rs\_sent\_by\_rp := rs\_sent\_by\_rp \setminus \{rp \mapsto (stp \mapsto reg)\}$
$act2 : rs\_dr\_r := rs\_dr\_r \setminus \{dr \mapsto (stp \mapsto reg)\}$
END

Les événements O_RP_REG_STOP_SENDING6, RP_REG_STOP_RECEIVING6, RP_REG_STOP_LOSING6 modélisent les échanges possibles de messages « register-stop » entre les routeurs RPs non-choisis par les routeurs DRs et les routeurs RPs choisis.

— L'événement O_RP_REG_STOP_SENDING6 modélise l'envoi d'un message « register-stop » par un routeur RP rp non-choisi par un routeur DR. Le routeur rp a reçu du routeur rp_chs ($grd7$, $grd8$) un message « register » reg et le routeur rp a envoyé le paquet encapsulé par le message reg aux destinataires qui l'ont rejoint ($grd9$). Le routeur rp peut par conséquent envoyer au routeur rp_chs un message « register-stop » en réponse message « register » reg reçu ($act1$, $act2$).

EVENT O_RP_REG_STOP_SENDING6 $\widehat{=}$
ANY
$reg, rp, stp, rp\_chs, grp$
WHERE
$grd1 : rp \in RPs$
$grd2 : rp\_chs \in RPs$
$grd3 : grp \subseteq RPs$
$grd4 : reg \in REG$
$grd5 : stp \in REG\_STOP$
$grd6 : grp = \{rp\_chs\}$
$grd7 : rp \mapsto reg \in reg\_rcvd\_by\_o\_rp$
$grd8 : rp\_chs \mapsto reg \in reg\_sent\_by\_rp$
$grd9 : rp \mapsto desenc(reg) \in sent\_to\_t$
THEN
$act1 : rs\_sent\_by\_o\_rp := rs\_sent\_by\_o\_rp \cup \{rp \mapsto (stp \mapsto reg)\}$
$act2 : rs\_rp\_r := rs\_rp\_r \cup (grp \times \{stp \mapsto reg\})$
END

— L'événement RP_REG_STOP_RECEIVING6 modélise la réception d'un message « register-stop » par un routeur RP rp_chs (choisi par un routeur DR). Le routeur rp_chs est le destinataire d'un message « register-stop » stp ($grd6$), envoyé par un routeur rp en réponse à un message « register » reg ($grd5$). Le routeur rp_chs reçoit le message stp ($act2$) et le marque comme étant reçu du routeur rp et acquittant le message « register » reg ($act1$).

EVENT RP_REG_STOP_RECEIVING6 $\widehat{=}$
ANY
reg, rp, rp_chs
WHERE
$grd1 : rp \in RPs$
$grd2 : rp\_chs \in RPs$
$grd3 : reg \in REG$
$grd4 : stp \in REG\_STOP$
$grd5 : rp \mapsto (stp \mapsto reg) \in rs\_sent\_by\_o\_rp$
$grd6 : rp\_chs \mapsto (stp \mapsto reg) \in rs\_rp\_r$
THEN
$act1 : rp\_reg\_ack\_rs := rp\_reg\_ack\_rs \cup \{rp\_chs \mapsto (rp \mapsto reg)\}$
$act2 : rs\_rcvd\_by\_rp := rs\_rcvd\_by\_rp \cup \{rp\_chs \mapsto (stp \mapsto reg)\}$
END

— L'événement RP_REG_STOP_LOSING6 modélise la possible perte ($act1$, $act2$) d'un message « register-stop » stp envoyé par un routeur RP rp ($grd5$) à destination d'un autre routeur RP rp_chs ($grd7$), en réponse à un message « register » reg. La perte est possible tant que le routeur rp_chs n'a pas reçu le message stp ($grd8$), c'est-à-dire, tant que le message reg n'a pas été acquitté ($grd6$).

EVENT RP_REG_STOP_LOSING6 $\widehat{=}$
ANY
reg, rp, rp_chs
WHERE
$grd1 : rp \in RPs$
$grd2 : rp\_chs \in RPs$
$grd3 : reg \in REG$
$grd4 : stp \in REG\_STOP$
$grd5 : rp \mapsto (stp \mapsto reg) \in rs\_sent\_o\_by\_rp$
$grd6 : rp\_chs \mapsto (rp \mapsto reg) \notin rp\_reg\_ack\_rs$
$grd7 : rp\_chs \mapsto (stp \mapsto reg) \in rs\_rp\_r$
$grd8 : rp\_chs \mapsto (stp \mapsto reg) \notin rs\_rcvd\_by\_rp$
THEN
$act1 : rs\_sent\_by\_o\_rp := rs\_sent\_by\_rp \setminus \{rp \mapsto (stp \mapsto reg)\}$
$act2 : rs\_rp\_r := rs\_rp\_r \setminus \{rp\_chs \mapsto (stp \mapsto reg)\}$
END

Le diagramme 5.10 suivant résume ce raffinement de M5 en M6 :

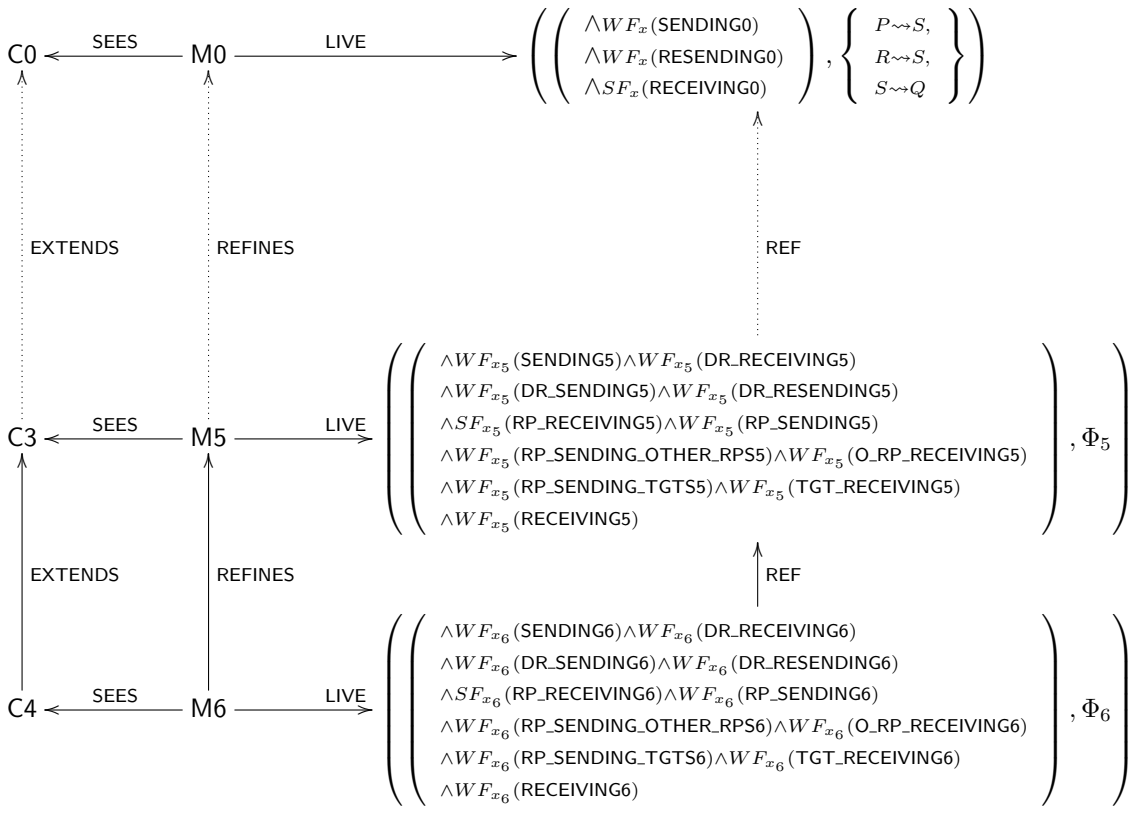


FIGURE 5.10 – Sixième raffinement du protocole ANYCAST RP

Le diagramme met également en avant le fait que les propriétés de vivacité de Φ_5 sont satisfaites par M6 (pour la preuve, voir l'annexe). Les routeurs RPs créent aussi, pour chaque message « register » qu'ils reçoivent, un état appelé « (S,G) » : cet état signifie qu'ils peuvent être utilisés par la source S du paquet encapsulé dans le message « register » pour joindre des membres du groupe G de destinataires du paquet et que si d'autres destinataires deviennent intéressés par le traffic provenant de la source S, ils peuvents rejoindre (se connecter et faire savoir qu'ils veulent recevoir du traffic en provenance de S) les routeurs RPs ayant créé un état « (S,G) ». Le prochain raffinement nous introduit la création de cet état « (S,G) ».

5.3.2.8 Septième raffinement : états « (S,G) » des RPs

Les routeurs RPs créent un état appelé « (S,G) », lors de l'algorithme ANYCAST RP, après avoir reçu des messages « register » : ils indiquent ainsi qu'ils peuvent servir de relais intermédiaires entre une source S et son groupe G de destinataires, que d'autres destinataires intéressés par le flux de données émis par la source S peuvent rejoindre via les routeurs RPs. Les routeurs RPs maintiennent ainsi des canaux de communication ayant une structure arborescente, entre une source d'un paquet et ses destinataires, avec la source en tant que racine, et les destinataires et routeurs RPs formant des sous-arbres, avec les RPs en tant que racines et les destinataires aux feuilles [18, 17].

Nous introduisons, dans ce raffinement M7, une variable *state_sg* indiquant pour quel message « register » un routeur RP a créé un état « (S,G) ». Une autre variable *reg_rs_o_rp* est aussi introduite, car pour créer un état « (S,G) » par rapport à un message « register », un routeur RP non-choisi par un DR doit savoir pour quels messages « register » il a envoyé des message « register-stop ». L'ensemble des variables x_7 de la machine est constitué des variables x_6 du raffinement précédent et de ces deux variables *state_sg* et *reg_rs_o_rp*.

INITIALISATION $\widehat{=}$
BEGIN
...
$\oplus$ $reg\_rs\_o\_rp, state\_sg := \varnothing$
END

A l'état initial, ces deux nouvelles variables sont initialisées à l'aide de l'ensemble vide ($\varnothing$), parce qu'aucun routeur n'a encore créé d'état « (S,G) » et aucun message « register » n'a encore été acquitté.

Nous définissons, dans le raffinement M7, l'invariant $I_7(x_7)$ caractérisant ces variables x_7 :

— *inv*1 et *inv*2 permettent de typer *state_sg* et *reg_rs_o_rp*.

$$inv1 \ : \ state\_sg \in REG \leftrightarrow RP$$
$$inv2 \ : \ reg\_rs\_o\_rp \in RP \leftrightarrow REG$$

  — *inv*1 associe à chaque message « register », les routeurs RPs qui ont créé un état « (S,G) » par rapport au message « register ».
  — *inv*2 associe à chaque routeur RP non-choisi par un DR, les messages « register » reçu par le routeur et pour lesquels le routeur a envoyé un message « register-stop ».

— *inv*3, *inv*4 et *inv*5 expriment des propriétés de sûreté :

$$inv3 \ : \ dom(state\_sg) \subseteq ran(reg\_sent\_to\_o\_rp)$$
$$inv4 \ : \ \forall r \cdot r \in ran(reg\_rs\_o\_rp) \Rightarrow r \in ran(reg\_sent\_to\_o\_rp)$$
$$inv5 \ : \ \forall r \cdot r \in ran(ran(rs\_sent\_by\_o\_rp)) \Rightarrow r \in ran(reg\_sent\_to\_o\_rp)$$

  — *inv*3 exprime que chaque message « register », pour lequel un état « (S,G) » a été créé, a été au préalable envoyé par un routeur RP (choisi par un routeur DR ayant émis le message « register ») aux autres routeurs RPs.
  — *inv*4 exprime que chaque message « register », pour lequel un message « register-stop » a été émis, a été au préalable envoyé par un routeur RP (choisi par un routeur DR ayant émis le message « register ») aux autres routeurs RPs.
  — *inv*5 exprime que chaque message « register », pour lequel des routeurs RPs non-choisis par les routeurs DRs ont émis des messages « register-stop », on été au préalable envoyés à ces routeurs RPs.

Nous reprenons, dans cette machine M7, les événements de M6, hormis O_RP_REG_STOP_SENDING6 qui est modifié par un raffinement O_RP_REG_STOP_SENDING7, de telle sorte que le routeur point de rendez-vous acquittant un message « register », reçu d'un routeur désigné, mémorise le fait qu'il a envoyé un acquittement. Nous ajoutons aussi un nouvel événement RP_CREATING_STATE_SG7 modélisant la création d'un état « (S,G) » par un routeur point de rendez-vous.

Nous détaillons les événements de la machine M7 qui sont nouveaux ou modifiés par raffinement :

— O_RP_REG_STOP_SENDING7 prend maintenant en compte que le routeur *rp*, qui envoie un message « register-stop », en réponse à un message « register » *reg* reçu, sauvegarde le fait qu'il ait envoyé un acquittement au message *reg* (*act*3).

EVENT O_RP_REG_STOP_SENDING7
REFINES O_RP_REG_STOP_SENDING6 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
  THEN
    ...
    $\oplus act3$: $reg\_rs\_o\_rp := reg\_rs\_o\_rp \cup \{rp \mapsto reg\}$
  END

— RP_CREATING_STATE_SG7 modélise la création d'un état « (S,G) » ($act1$), par rapport à un message « register » reg, par un routeur RP rp. Cette création diffère, selon le routeur RP rp :
  — Si le routeur rp est un routeur choisi par un routeur DR, alors il doit au préalable avoir diffusé le message « register » reg aux autres RPs (première partie de $grd5$).
  — Sinon, le routeur rp doit avoir au préalable acquitté le message « register » reg par un message « register-stop » (deuxième partie de $grd5$).

  Mais dans les deux cas, le message « register » reg doit auparavant avoir transité par le routeur RP choisi par un routeur DR ($grd4$) et il ne doit pas encore avoir créé d'état « (S,G) » pour le message « register » reg ($grd6$).

EVENT RP_CREATING_STATE_SG7 $\widehat{=}$
  ANY
    rp, rp_chs, reg
  WHERE
    $grd1$: $rp \in RPs$
    $grd2$: $rp\_chs \in RPs$
    $grd3$: $reg \in REG$
    $grd4$: $rp\_chs \mapsto reg \in reg\_sent\_by\_rp$
    $grd5$: $\vee(rp = rp\_chs \wedge rp\_chs \mapsto reg \in reg\_sent\_to\_o\_rp)$
             $\vee(rp \neq rp\_chs \wedge rp \mapsto reg \in reg\_rs\_o\_rp)$
    $grd6$: $reg \mapsto rp \notin state\_sg$
  THEN
    $act1$: $state\_sg := state\_sg \cup \{reg \mapsto rp\}$
  END

Ce modèle M7 présente les trois grandes étapes de l'algorithme ANYCAST RP : **(1)** la transmission des paquets des sources aux Routeurs Désignés DRs, **(2)** la diffusion des paquets encapsulés dans des messages « register » par les routeurs DRs aux routeurs Points de Rendez-vous RPs et **(3)** la diffusion des paquets par les routeurs RPs aux groupes de destinataires. Nous avons aussi introduit des mécanismes d'acquittement, par l'intermédiaire des messages « register-stop », ainsi que la création d'un état appelé « (S,G) » (par les routeurs RPs), pour la gestion et la maintenance des routes reliant une source S et un groupe G de destinataires.

Le diagramme 5.11 suivant résume ce raffinement de M6 en M7 :

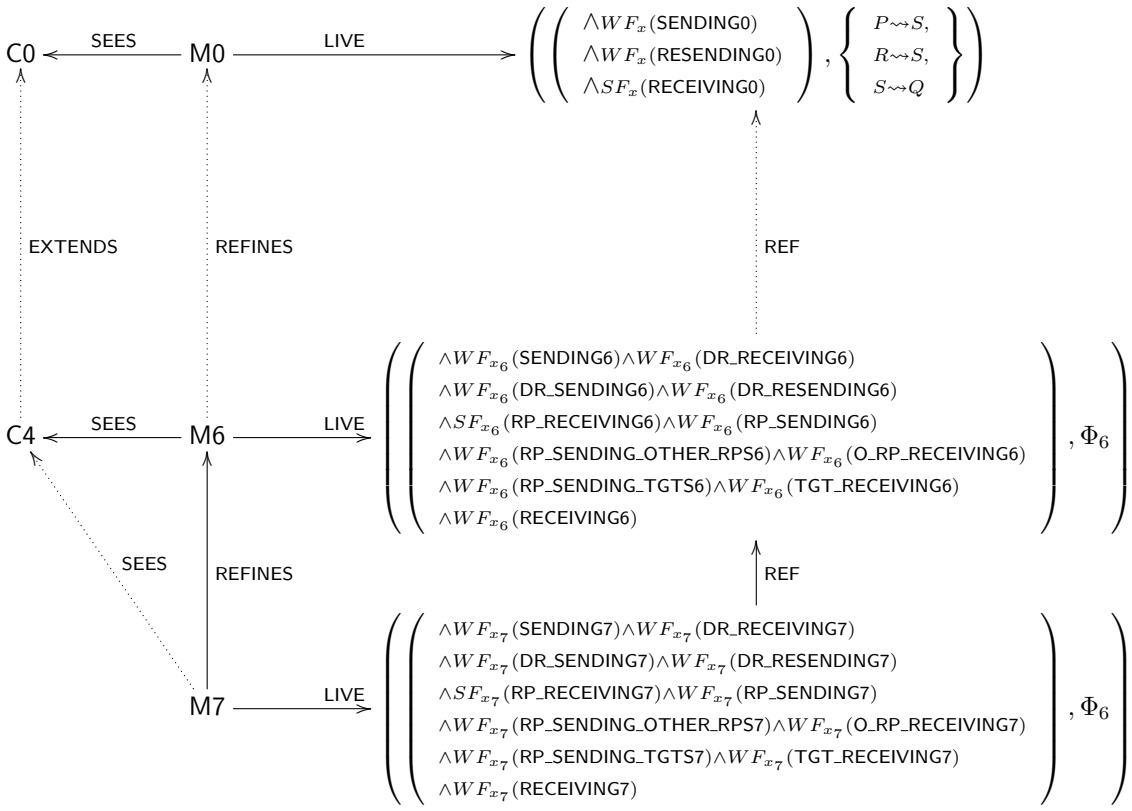


FIGURE 5.11 – Septième raffinement du protocole ANYCAST RP

Nous remarquons dans ce raffinement que les hypothèses d'équité posées sur les événements de M7 raffinant ceux de M6 demeurent les mêmes, et que nous ne modifions pas les gardes de ces derniers : les conditions d'équité sur ces événements sont ainsi conservées lors du raffinement. Les nouvelles variables (en particulier *state_sg*) sont modifiées par les nouveaux événements ou ceux ne participant pas à l'algorithme. Nous pouvons alors appliquer la première règle de raffinement automatique vue en page 54 et nous déduisons que cette machine M7 satisfait une liste de propriétés de vivacité, qui est la même (Φ_6)que celle du raffinement M6 précédent.

Le prochain raffinement introduit le graphe reliant les différents nœuds et routeurs, formant ainsi le système réparti support de l'algorithme ANYCAST RP.

5.3.2.9 Huitième raffinement : système réparti

Ce huitième raffinement introduit le système réparti support de l'algorithme ANYCAST RP. Nous proposons ici un réseau mobile, dont les modifications sont soumises à quelques contraintes que nous expliquerons plus bas.

Nous commençons par définir un contexte C5 étendant le contexte C4 vu précédemment. Nous définissons dans ce contexte des constantes g, $closure$, GR :

CONTEXT C5 EXTENDS C4
CONSTANTS
$g, closure, GR$
AXIOMS
$axm1$: $closure \in (NODES \leftrightarrow NODES) \rightarrow (NODES \leftrightarrow NODES)$
$axm2$: $\forall r, s \cdot ((r \subseteq closure(r)) \land (closure(r); r \subseteq closure(r)) \land (r \subseteq s \land s; r \subseteq s \Rightarrow closure(r) \subseteq s))$
$axm3$: $\forall g1, g2 \cdot g1 \subseteq g2 \Rightarrow closure(g1) \subseteq closure(g2)$
$axm4$: $GR = \left\{ gr \mid \left(\begin{array}{l} \land gr \subseteq NODES \times NODES \\ \land gr \neq \varnothing \\ \land NODES \lhd id \cap gr = \varnothing \\ \land gr = gr^{-1} \\ \land (SOURCES \times DRs \subseteq gr) \\ \land (\forall dr1, rp2 \cdot \left(\begin{array}{l} \land dr1 \in DRs \\ \land rp2 \in RPs \end{array} \right) \Rightarrow dr1 \mapsto rp2 \in closure(gr)) \\ \land (\forall rp1, rp2 \cdot \left(\begin{array}{l} \land rp1 \in RPs \\ \land rp2 \in RPs \end{array} \right) \Rightarrow rp1 \mapsto rp2 \in closure(gr)) \\ \land (\forall t, rp \cdot \left(\begin{array}{l} \land t \in TARGETS \\ \land rp \in RPs \land join(t) = rp \end{array} \right) \Rightarrow rp \mapsto t \in closure(gr)) \end{array} \right) \right\}$
$axm5$: $g \subseteq NODES \times NODES$
$axm6$: $g \in GR$
END

— *closure* est une fonction qui associe à tous les graphes possibles ($axm1$) sur l'ensemble des nœuds $NODES$, leurs fermetures transitives réflexives. En résumé, *closure* calcule pour chaque nœud d'un graphe donné, quels sont les autres nœuds atteignables à partir de ce dernier, y compris lui-même ($axm2$). Nous ajoutons dans le contexte une propriété $axm3$ caractérisant la fonction *closure* : soit deux graphes $g1$ et $g2$; si $g1$ est inclus dans $g2$, alors la fermeture transitive réflexive de $g1$ est contenue dans $g2$.
— GR définit l'ensemble des graphes/réseaux gr possibles lors du déroulement du protocole ANYCAST RP ($axm4$) :
  — gr n'est pas vide.
  — gr n'est pas réflexif.
  — gr est symétrique.
  — Dans le graphe gr, les sources sont toujours reliés directement aux routeurs DRs ; les routeurs RPs sont atteignables à partir des routeurs DRs ; les routeurs RPs sont interconnectés et les destinataires qui ont rejoint ces routeurs RPs sont atteignables à partir de ces derniers.

  GR permet ainsi déjà de définir les contraintes qui restreindront les modifications possibles à apporter au graphe/réseau support de l'algorithme ANYCAST RP.
— g est le graphe initial utilisé dans ce raffinement. Il respecte les contraintes définies par GR ($axm6$).

Le contexte C5 est utilisé par la machine M8 raffinant la machine M7, vue dans la section précédente.

Nous commençons par introduire dans la machine M8 de nouvelles variables relatives à la notion de réseau/système réparti :

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus$ $reg\_lost := \varnothing$
$\oplus$ $graph := g$
$\oplus$ $dr\_rp\_store, rp\_t\_store, s\_dr\_store := \varnothing$
$\oplus$ $rp\_dr\_store\_stp, rp\_rp\_store\_stp := \varnothing$
END

— *graph* représente le réseau/graphe courant et relie des nœuds entre eux.
— *s_dr_store*, *dr_rp_store*, *rp_t_store* permettent de représenter où se situent (à quel nœud du réseau) les paquets (cas de *s_dr_store* et *rp_t_store*) ou les messages « register » (*dr_rp_store*) transitant dans le réseau. Nous avons ici trois variables différentes, correspondant aux différentes phases identifiées de l'algorithmes ANYCAST RP : *s_dr_store* permet de localiser les paquets circulant sur la portion du réseau entre les sources et les routeurs DRs, *dr_rp_store* sert à la localisation des paquets encapsulés dans des messages « register » transitant sur la portion entre les routeurs DRs et les routeurs RPs, et finalement *rp_t_store* donne la possibilité de localiser les paquets transitant sur la portion du réseau entre les routeurs RPs et les destinataires.
— De la même manière *rp_dr_store_stp* et *rp_rp_store_stp* permettent de savoir les positions dans le réseau des messages « register-stop », envoyés en réponse à des messages « register ». Nous avons

ici deux variables car nous avons souhaité séparer clairement la communication entre un routeur DR et le routeur RP qu'il a choisi (*rp_dr_store_stp*), ainsi que la communication entre le routeur RP choisi et les routeurs RPs non-choisis (*rp_rp_store_stp*).

— Nous notons ici que l'introduction de la variable *dr_rp_store* nous permet de supprimer la variable *reg_lost* du modèle : en effet, nous considérons un message « register » perdu s'il disparaît du réseau (plus explicitement, s'il disparaît de la variable *dr_rp_store*).

Les variables x_8 de ce raffinement est donc constitué de ces nouvelles variables et de celles x_7 du raffinement M7 précédent, desquelles *reg_lost* a été retirée. Le réseau *graph* est initialisé à l'aide de la constante g définie dans le contexte C5, tandis que les autres variables sont initialisées à l'aide de l'ensemble vide ($\varnothing$), ce qui est justifié par le fait qu'à l'état initial, aucun paquet, message ne circule dans le réseau.

Un invariant noté $I(x_8)$ contraint ces variables x_8 et est défini comme suit :

— Les propriétés de $inv1$ à $inv6$ permettent de typer les nouvelles variables conformément à leurs descriptions données précédemment.

$$
\begin{array}{l}
inv1 \ : \ graph \in NODES \leftrightarrow NODES \\
inv2 \ : \ s\_dr\_store \in NODES \leftrightarrow PCK \\
inv3 \ : \ dr\_rp\_store \in NODES \leftrightarrow REG \\
inv4 \ : \ rp\_t\_store \in NODES \leftrightarrow MESSAGES \\
inv5 \ : \ rp\_dr\_store\_stp \in NODES \leftrightarrow (REG\_STOP \times REG) \\
inv6 \ : \ rp\_rp\_store\_stp \in NODES \leftrightarrow (REG\_STOP \times REG)
\end{array}
$$

— Des propriétés de sûreté ($inv7$ à $inv8$) permettent de caractériser ces variables :

$$
\begin{array}{l}
inv7 \ : \ graph \in GR \\
inv8 \ : \ ran(s\_dr\_store) \subseteq ran(sent\_by\_s) \\
inv9 \ : \ ran(dr\_rp\_store) \subseteq ran(reg\_sent\_by\_dr) \\
inv10 \ : \ reg\_lost \cap ran(dr\_rp\_store) = \varnothing \\
inv11 \ : \ \forall rs \cdot rs \in ran(rp\_dr\_store\_stp) \Rightarrow prj_2(sr) \in ran(reg\_sent\_by\_dr) \\
inv12 \ : \ ran(rp\_dr\_store\_stp) \subseteq ran(rs\_sent\_by\_rp) \\
inv13 \ : \ \forall rs \cdot rs \in ran(rp\_rp\_store\_stp) \Rightarrow prj_2(sr) \in ran(reg\_sent\_by\_rp) \\
inv14 \ : \ ran(rp\_rp\_store\_stp) \subseteq ran(rs\_sent\_by\_o\_rp)
\end{array}
$$

— $inv7$ exprime que le réseau courant *graph* respecte les contraintes définies par l'ensemble GR des graphes/réseaux gr possibles.

— $inv8$ exprime que chaque paquet mis sur le réseau, dans la portion entre les sources et les routeurs DRs, a été envoyé par une source.

— $inv9$ exprime que chaque message « register » mis sur le réseau, dans la portion entre les routeurs DRs et les routeurs RPs, a été envoyé par un routeur DR.

— $inv10$ exprime qu'un message « register » est soit circulant sur le réseau, soit perdu.

— $inv11$ exprime que chaque message « register-stop » circulant sur la portion du réseau, entre un routeur DR et le routeur RP choisi par le routeur DR, et envoyé par le routeur RP, est une réponse à un message « register », envoyé par le routeur DR.

— $inv12$ exprime que chaque message « register-stop » circulant sur la portion du réseau, entre un routeur DR et le routeur RP choisi par le routeur DR, a été envoyé par le routeurs RP en réponse à un message « register ».

— $inv13$ exprime que chaque message « register-stop » circulant sur la portion du réseau, entre les routeurs RPs non-choisis par un routeur DR et le routeur RP choisi par le routeur DR, et envoyé par les routeurs RPs non-choisis, est une réponse à un message « register », envoyé par le routeur RP choisi.

— $inv14$ exprime que chaque message « register-stop » circulant sur la portion du réseau, entre les routeurs RPs non-choisis par un routeur DR et le routeur RP choisi par le routeur DR, a été envoyé par les routeurs RPs non-choisis.

Les événements présents dans cette machine M8 sont les suivants :

— Les raffinements des événements de la machine M7 précédente.

— Des événements modélisant les acheminements de paquets, de messages, entre nœuds du réseau :

  — MOVING_PCK8 : déplacement d'un paquet d'une source à un routeur DR.

  — DR_RP_FWD8 : acheminement d'un message « register » entre un routeur DR et le routeur RP auquel il a choisi de transmettre le message.

  — RP_O_FWD8 : acheminement d'un message « register » entre un routeur DR choisi et les autres routeurs RPs.

- RP_T_FWD8 : acheminement d'un paquet d'un routeur RP aux destinataires qui l'ont « rejoint » (lui sont connectés, ont choisi de recevoir du traffic de lui exclusivement).
- RP_DR_STOP_FWD8 : acheminement d'un message « register-stop » entre un routeur RP choisi et le routeur DR lui ayant transmis un message de type « register ».
- O_RP_RP_REG_STOP_FWD8 : acheminement de messages « register-stop » entre les routeurs RPs non-choisis et le routeur RP choisi leur ayant transmis un message de type « register ».

— Des événements de modification du réseau :
- ADD_LINK8 : une connexion entre deux nœud non-existante dans le réseau est créée.
- REMOVE_LINK8 : une connexion entre deux nœud existante dans le réseau est détruite.

Une liste Φ_8 de propriétés de vivacité caractérisant la machine M8, et détaillée en annexe, nous permet de construire le système d'événements de ce raffinement :

1. **Modélisation de la communication *unicast* entre la source d'un paquet p et le routeur désigné choisi pour son acheminement.**
   - Evénement SENDING8 : la source s d'un paquet p dépose ce dernier dans le réseau.

```
EVENT SENDING8
REFINES SENDING7 ≙
  ANY
    ...
  WHERE
    ...
  THEN
    ...
    ⊕ act3 : s_dr_store := s_dr_store ∪ {s ↦ p}
  END
```

   - Evénement MOVING_PCK8 : un paquet p se déplace ($act1$) de sa source s à un routeur désigné dr adjacent ($grd5$) choisi ($grd8$), après avoir été placé par sa source ($grd6$ et $grd7$) dans le réseau.

```
EVENT MOVING_PCK8 ≙
  ANY
    p, s, dr
  WHERE
    grd1 : p ∈ PCK
    grd2 : s ∈ SOURCES
    grd3 : dr ∈ DRs
    grd4 : s = source(p)
    grd5 : s ↦ dr ∈ graph
    grd6 : s ↦ p ∈ s_dr_store
    grd7 : dr ↦ p ∉ s_dr_store
    grd8 : dr ↦ p ∈ dr_r_pck
  THEN
    act1 : s_dr_store := (s_dr_store \ {s ↦ p}) ∪ {dr ↦ p}
  END
```

   - Evénement DR_RECEIVING8 : un routeur désigné dr adjacent à la source d'un paquet p le reçoit, car le paquet p se trouve à son niveau dans le réseau ($grd5$). Le paquet p est ensuite supprimé de la portion de réseau entre la source et le routeurs désigné dr, car le premier destinataire intermédiaire du paquet est unique (un seul routeur désigné).

```
EVENT DR_RECEIVING8
REFINES DR_RECEIVING7 ≙
  ANY
    ...
  WHERE
    ...
    ⊕ grd5 : dr ↦ p ∈ s_dr_store
  THEN
    ...
    ⊕ act2 : s_dr_store := s_dr_store \ {dr ↦ p}
  END
```

2. **Modélisation des communications entre le routeur désigné choisi pour l'acheminement d'un paquet p et les routeurs points de rendez-vous.**

   (a) **Communication *anycast* entre routeur désigné et le point de rendez-vous choisi.**

— Evénement DR_SENDING8 : un routeur désigné dr ayant reçu un paquet p l'encapsule dans un message « register » reg, et dépose ce dernier dans la portion du réseau entre les routeurs désignés et les points de rendez-vous ($act4$), à destination du point de rendez-vous choisi.

EVENT DR_SENDING8 $\widehat{=}$
REFINES DR_SENDING7 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
  THEN
    ...
    $\oplus$ $act4$: $dr\_rp\_store := dr\_rp\_store \cup \{dr \mapsto reg\}$
  END

— Evénement DR_SENDING8 : un routeur désigné dr renvoie un message « register » reg (en cas de perte, etc), en le redéposant dans la portion du réseau entre les routeurs désignés et les points de rendez-vous ($act4$), à destination du point de rendez-vous choisi.

EVENT DR_RESENDING8 $\widehat{=}$
REFINES DR_RESENDING7 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
  THEN
    ...
    $\oplus$ $act4$: $dr\_rp\_store := dr\_rp\_store \cup \{dr \mapsto reg\}$
  END

— Evénement DR_RP_FWD8 : un message « register » reg est acheminé à travers le réseau : le message se déplace d'un nœud x, au niveau duquel il se trouve, vers un nœud y ($grd8$, $grd9$ et $act1$) adjacent/voisin, tant que sa destination d (qui est le routeur point de rendez-vous choisi) n'est pas atteinte ($grd5$ et $grd6$).

EVENT DR_RP_FWD8 $\widehat{=}$
  ANY
    x, y, reg, d
  WHERE
    $grd1$: $x \in NODES$
    $grd2$: $y \in NODES$
    $grd3$: $d \in RPs$
    $grd4$: $reg \in REG$
    $grd5$: $x \neq d$
    $grd6$: $d \mapsto reg \in reg\_rp\_r\_pck$
    $grd7$: $x \mapsto y \in graph$
    $grd8$: $x \mapsto reg \in dr\_rp\_store$
    $grd9$: $y \mapsto reg \notin dr\_rp\_store$
  THEN
    $act1$: $dr\_rp\_store := (dr\_rp\_store \setminus \{x \mapsto reg\}) \cup \{y \mapsto reg\}$
  END

— Evénement RP_RECEIVING8 : un message « register » reg se trouve dans le réseau au niveau d'un routeur rp qui est le point de rendez-vous choisi ($grd4$) : ce dernier peut donc le recevoir.

EVENT RP_RECEIVING8
REFINES RP_RECEIVING7 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd4$: $reg \notin reg\_lost$
    $\oplus$ $grd4$: $rp \mapsto reg \in dr\_rp\_store$
  THEN
    ...
  END

(b) **Communication *multicast* entre le point de rendez-vous choisi et les autres points de rendez-vous.**

— Evénement RP_SENDING8 : un message « register » reg transite au niveau d'un routeur rp qui est le point de rendez-vous choisi ($grd4$).

EVENT RP_SENDING8
REFINES RP_SENDING7 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\oplus$ $grd4$: $rp \mapsto reg \in dr\_rp\_store$
  THEN
    ...
  END

— Evénement RP_SENDING_OTHER_RPS8 : un message « register » reg est transmis par un routeur point de rendez-vous rp choisi, aux autres points de rendez-vous, lorsque le message se trouve à son niveau dans le réseau ($grd8$).

EVENT RP_SENDING_OTHER_RPS8
REFINES RP_SENDING_OTHER_RPS7 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\oplus$ $grd8$: $rp \mapsto reg \in dr\_rp\_store$
  THEN
    ...
  END

— Evénement RP_O_RP_FWD8 : un message « register » reg est acheminé à travers la portion du réseau reliant un point de rendez-vous choisi aux autres point de rendez-vous : le message se déplace d'un nœud x, au niveau duquel il se trouve, vers un nœud y ($grd8$, $grd9$ et $act1$) adjacent/voisin, tant que sa destination d (qui est l'un des routeurs RPs non-choisi) n'est pas atteinte ($grd5$ et $grd6$). Nous avons ici un groupe de destinataires (soit un ensemble de routeurs RPs non-choisis par un routeur DR). Aussi, nous ne retirons pas du nœud x l'élément transitant dans le réseau. Nous le faisons juste passer au nœud y. Nous considérons ainsi la possibilité qu'à partir du nœud x, un autre voisin différent du nœud y puisse mener vers un autre destinataire.

EVENT RP_O_RP_FWD8 $\widehat{=}$
  ANY
    x, y, reg, d
  WHERE
    $grd1$: $x \in NODES$
    $grd2$: $y \in NODES$
    $grd3$: $d \in RPs$
    $grd4$: $reg \in REG$
    $grd5$: $x \neq d$
    $grd6$: $d \mapsto reg \in reg\_o\_rp\_r\_pck$
    $grd7$: $x \mapsto y \in graph$
    $grd8$: $x \mapsto reg \in dr\_rp\_store$
    $grd9$: $y \mapsto reg \notin dr\_rp\_store$
  THEN
    $act1$: $dr\_rp\_store := dr\_rp\_store \cup \{y \mapsto reg\}$
  END

— Evénement O_RP_RECEIVING8 : un message « register » reg, qui se trouve au niveau d'un routeur point de rendez-vous rp non-choisi ($grd8$), est reçu par ce dernier.

EVENT O_RP_RECEIVING8
REFINES O_RP_RECEIVING7 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\oplus$ $grd4$: $rp \mapsto reg \in dr\_rp\_store$
  THEN
    ...
  END

3. **Modélisation de la communication *multicast* entre les routeurs points de rendez-vous et les destinataires.**

— Evénement RP_SENDING_TGTS8 : un message « register » reg reçu par un point de rendez-vous rp et se trouvant à son niveau dans le réseau ($grd8$), est désencapsulé par le point de rendez-vous et le paquet ainsi obtenu est déposé par le routeur rp dans le réseau ($act3$), vers les destinataires du paquet p.

EVENT RP_SENDING_TGTS8
REFINES RP_SENDING_TGTS7 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\oplus$ $grd8$: $rp \mapsto reg \in dr\_rp\_store$
  THEN
    ...
    $\oplus$ $act3$: $rp\_t\_store := rp\_t\_store \cup \{rp \mapsto desenc(reg)\}$
  END

— Evénement RP_T_FWD8 : un paquet p est acheminé à travers la portion du réseau reliant les points de rendez-vous aux destinataires : le paquet se déplace d'un nœud x, au niveau duquel il se trouve, vers un nœud y ($grd8$, $grd9$ et $act1$) adjacent/voisin, tant que sa destination d (qui est l'un des des destinataires finaux de p) n'est pas atteinte ($grd5$ et $grd6$). Nous avons ici un groupe de destinataires, alors, nous ne retirons pas du nœud x l'élément transitant dans le réseau. Nous le faisons juste passer au nœud y. Nous considérons ainsi la possibilité qu'à partir du nœud x, un autre voisin différent du nœud y puisse mener vers un autre destinataire.

EVENT RP_T_FWD8 $\widehat{=}$
  ANY
    x, y, p, d
  WHERE
    $grd1$: $x \in NODES$
    $grd2$: $y \in NODES$
    $grd3$: $d \in RPs$
    $grd4$: $p \in PCK$
    $grd5$: $x \neq d$
    $grd6$: $d \mapsto p \in t\_r\_pck$
    $grd7$: $x \mapsto y \in graph$
    $grd8$: $x \mapsto p \in rp\_t\_store$
    $grd9$: $y \mapsto p \notin rp\_t\_store$
  THEN
    $act1$: $rp\_t\_store := rp\_t\_store \cup \{y \mapsto p\}$
  END

— Evénement TGT_RECEIVING8 : un paquet p, qui se trouve au niveau d'un destinataire t ($grd4$), est reçu par ce dernier.

EVENT TGT_RECEIVING8
REFINES TGT_RECEIVING7 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\oplus$ $grd4$: $t \mapsto p \in rp\_t\_store$
  THEN
    ...
  END

Le système réparti considéré étant mobile, nous introduisons dans ce raffinement deux événements permettant de modifier le système réparti courant $graph$:

— L'événement ADD_LINK8 modélise l'ajout, dans le réseau courant $graph$, de nouveaux liens qui n'existaient pas encore entre deux nœuds x et y ($grd4$) différents ($grd3$) : un lien direct de x vers y est donc établi, ainsi qu'un lien de y vers x ($act1$). En d'autres termes, x devient voisin de y et vice versa.

EVENT ADD_LINK8 $\widehat{=}$
ANY
x, y
WHERE
$grd1 : x \in NODES$
$grd2 : y \in NODES$
$grd3 : x \neq y$
$grd4 : x \mapsto y \notin graph$
THEN
$act1 : graph := graph \cup \{x \mapsto y, y \mapsto x\}$
END

— L'événement ADD_LINK8 modélise le retrait ($act1$), dans le réseau courant $graph$, des liens existants entre deux nœuds x et y voisins ($grd4$). La suppression des liens entre x et y est effective, si le graphe/réseau résultant de la suppression respecte les contraintes relatives aux réseaux possibles définies par l'ensemble GR ($grd5$).

EVENT REMOVE_LINK8 $\widehat{=}$
ANY
x, y
WHERE
$grd1 : x \in NODES$
$grd2 : y \in NODES$
$grd3 : x \neq y$
$grd4 : x \mapsto y \in graph$
$grd5 : (graph \setminus \{x \mapsto y, y \mapsto x\}) \in GR$
THEN
$act1 : graph := graph \setminus \{x \mapsto y, y \mapsto x\}$
END

Nous modifions aussi par raffinement les événements relatifs aux envois, acheminements et réceptions des messages d'acquittement « register-stop » :

1. **Communications entre le routeur point de rendez-vous choisi et le routeur désigné qui l'a choisi.**
   — Evénement RP_REG_STOP_SENDING8 : un message « register » reg, se trouve au niveau d'un routeur point de rendez-vous rp choisi ($grd4$) et a été reçu par ce dernier. Le routeur rp dépose alors dans le réseau, en direction du routeur désigné qui l'a choisi, un message « register-stop » correspondant au message reg reçu.

EVENT RP_REG_STOP_SENDING8
REFINES RP_REG_STOP_SENDING7 $\widehat{=}$
ANY
...
WHERE
...
$\oplus\ grd8 : rp \mapsto reg \in dr\_rp\_store$
THEN
...
$\oplus\ act3 : rp\_dr\_store\_stp := rp\_dr\_store\_stp \cup \{rp \mapsto (stp \mapsto reg)\}$
END

   — Evénement RP_DR_STOP_FWD8 : un message « register-stop » rs, correspondant à un message « register » reçu par un routeur point de rendez-vous choisi, est acheminé à travers la portion du réseau reliant les points de rendez-vous aux routeurs désignés : le message « register-stop » se déplace d'un nœud x, au niveau duquel il se trouve, vers un nœud y ($grd8$, $grd9$ et $act1$) adjacent/voisin, tant que sa destination d (qui est un routeur désigné) n'est pas atteinte ($grd5$ et $grd6$). Le message « register-stop » rs qui transite dans le réseau est retiré du nœud x et passe au nœud y : cette action est justifié par le fait que le destinataire (ici un routeur DR) et l'émetteur (un routeur RP choisi par un routeur DR) sont uniques.

EVENT RP_DR_STOP_FWD8 $\widehat{=}$
ANY
x, y, rs, d
WHERE
$grd1 : x \in NODES$
$grd2 : y \in NODES$
$grd3 : d \in DRs$
$grd4 : rs \in (STOP\_REG \times REG)$
$grd5 : x \neq d$
$grd6 : d \mapsto rs \in rs\_dr\_r$
$grd7 : x \mapsto y \in graph$
$grd8 : x \mapsto rs \in rp\_dr\_store\_stp$
$grd9 : y \mapsto rs \notin rp\_dr\_store\_stp$
THEN
$act1 : rp\_dr\_store\_stp := (rp\_dr\_store\_stp \setminus \{x \mapsto rs\}) \cup \{y \mapsto rs\}$
END

— Evénement DR_REG_STOP_RECEIVING8 : un message « register-stop » acquittant « register » reg envoyé, se trouve au niveau du routeur désigné dr ($grd5$) ayant envoyé le message reg. Le message « register-stop » est alors retiré du réseau ($act3$).

EVENT DR_REG_STOP_RECEIVING8
REFINES DR_REG_STOP_RECEIVING7 $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd5 : dr \mapsto (stp \mapsto reg) \in rp\_dr\_store\_stp$
THEN
...
$\oplus$ $act3 : rp\_dr\_store\_stp := rp\_dr\_store\_stp \setminus \{dr \mapsto (stp \mapsto reg)\}$
END

2. **Communications entre les routeurs points de rendez-vous non-choisis et le routeur point de rendez-vous choisi.**

— Evénement O_RP_REG_STOP_SENDING8 : un message « register » reg, se trouve au niveau d'un routeur point de rendez-vous rp non-choisi et a été reçu par ce dernier. Le routeur rp dépose alors dans le réseau, en direction du routeur point de rendez-vous choisi, un message « register-stop » correspondant au message reg reçu.

EVENT O_RP_REG_STOP_SENDING8
REFINES O_RP_REG_STOP_SENDING7 $\widehat{=}$
ANY
...
WHERE
...
THEN
...
$\oplus$ $act4 : rp\_rp\_store\_stp := rp\_rp\_store\_stp \cup \{rp \mapsto (stp \mapsto reg)\}$
END

— Evénement O_RP_RP_REG_STOP_FWD8 : un message « register-stop » rs, correspondant à un message « register » reçu par un routeur point de rendez-vous non-choisi, est acheminé à travers la portion du réseau reliant les points de rendez-vous non-choisis à celui choisi : le message « register-stop » se déplace d'un nœud x, au niveau duquel il se trouve, vers un nœud y ($grd8$, $grd9$ et $act1$) adjacent/voisin, tant que sa destination d (qui est un routeur désigné) n'est pas atteinte ($grd5$ et $grd6$). Le message « register-stop » rs qui transite dans le réseau est retiré du nœud x et passe au nœud y : cette action est justifié par le fait que le destinataire (ici un routeur RP choisi par un routeur DR) et l'émetteur (un routeur DR) sont uniques.

EVENT O_RP_RP_REG_STOP_FWD8 $\widehat{=}$
ANY
x, y, rs, d
WHERE
$grd1 : x \in NODES$
$grd2 : y \in NODES$
$grd3 : d \in RPs$
$grd4 : rs \in (STOP\_REG \times REG)$
$grd5 : x \neq d$
$grd6 : d \mapsto rs \in rs\_rp\_r$
$grd7 : x \mapsto y \in graph$
$grd8 : x \mapsto rs \in rp\_rp\_store\_stp$
$grd9 : y \mapsto rs \notin rp\_rp\_store\_stp$
THEN
$act1 : rp\_rp\_store\_stp := (rp\_rp\_store\_stp \setminus \{x \mapsto rs\}) \cup \{y \mapsto rs\}$
END

— Evénement RP_REG_STOP_RECEIVING8 : un message « register-stop » acquittant « register » *reg* envoyé, se trouve au niveau du routeur point de rendez-vous choisi *rp_chs* ($grd7$) ayant transmis le message *reg*. Le message « register-stop » est alors retiré du réseau ($act3$).

EVENT RP_REG_STOP_RECEIVING8
REFINES RP_REG_STOP_RECEIVING7 $\widehat{=}$
ANY
...
WHERE
...
$\oplus\ grd7 : rp\_chs \mapsto (stp \mapsto reg) \in rp\_rp\_store\_stp$
THEN
...
$\oplus\ act3 : rp\_rp\_store\_stp := rp\_rp\_store\_stp \setminus \{rp\_chs \mapsto (stp \mapsto reg)\}$
END

Les événements modélisant les possibles pertes de paquets et de messages sont aussi modifiés par raffinement. Les pertes de messages et/ou paquets sont modélisées par leur retrait du réseau ($act1$ de LOSING8, $act3$ de DR_REG_STOP_LOSING8 et RP_REG_STOP_LOSING7). Nous notons que nous éliminons toute référence à la variable *reg_lost*, car nous la remplaçons maintenant par la variable *dr_rp_store*.

EVENT LOSING8
REFINES LOSING7 $\widehat{=}$
ANY
...
WHERE
...
$\ominus\ grd3 : reg \notin (ran(reg\_rcvd\_by\_rp) \cup reg\_lost)$
$\oplus\ grd3 : reg \in ran(dr\_rp\_store)$
$\oplus\ grd4 : reg \notin ran(reg\_rcvd\_by\_rp)$
THEN
...
$\ominus\ act1 : reg\_lost := reg\_lost \cup \{reg\}$
$\oplus\ act1 : dr\_rp\_store := dr\_rp\_store \rhd \{reg\}$
END

EVENT DR_REG_STOP_LOSING8 REFINES DR_REG_STOP_LOSING7 $\widehat{=}$
...
THEN
...
$\oplus\ act3 : rp\_dr\_store\_stp := rp\_dr\_store\_stp \rhd \{stp \mapsto reg\}$
END

EVENT RP_REG_STOP_LOSING8 REFINES RP_REG_STOP_LOSING7 $\widehat{=}$
...
THEN
...
$\oplus\ act3 : rp\_rp\_store\_stp := rp\_rp\_store\_stp \rhd \{stp \mapsto reg\}$
END

Nous avons introduit dans ce raffinement la notion de système réparti mobile/dynamique. Le diagramme 5.12 suivant résume ce raffinement de M7 en M8 :

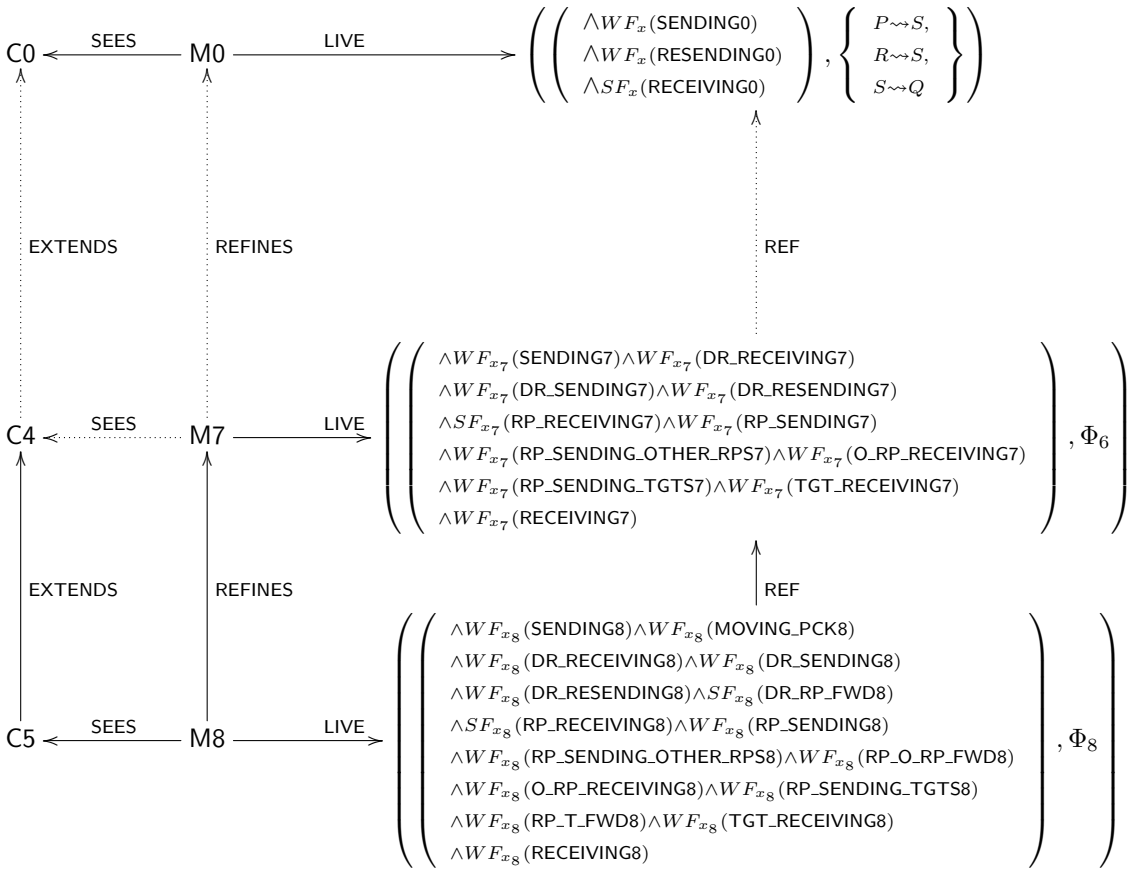


FIGURE 5.12 – Huitième raffinement du protocole ANYCAST RP

Ce diagramme nous montre aussi que les propriétés de vivacité de Φ_6, satisfaites par M7, sont aussi satisfaites par M8 (pour la preuve, voir l'annexe). Nous obtenons lors de ce huitième raffinement les détails des différentes phases du protocole ANYCAST RP. Dans les sous-sections qui suivent, nous localisons ces étapes.

5.3.2.10 Localisation de la première phase

Dans cette sous-section, nous localisons les parties des modèles relatives à la première phase de l'algorithme ANYCAST RP, c'est-à-dire celles qui se rapportent à la communication entre la source d'un paquet et le routeur DR choisi pour recevoir ce paquet. Nous commençons pour cela, par définir un contexte C6 étendant le context C5 vu dans les sections précédentes. Nous y définissons des constantes qui seront utilisées pour l'initialisation des variables que nous allons introduire dans la machine M9_P1, raffinant la machine M8 précédente :

CONTEXT C6 EXTENDS C5
CONSTANTS
$init\_sent\_by\_s, init\_rcvd\_by\_dr, init\_dr\_dst\_pck$
AXIOMS
$axm1$: $init\_sent\_by\_s \in SOURCES \rightarrow \mathbb{P}(PCK)$
$axm2$: $init\_dr\_dst\_pck \in PCK \rightarrow \mathbb{P}(DRs)$
$axm3$: $init\_rcvd\_by\_dr \in DRs \rightarrow \mathbb{P}(PCK)$
$axm4$: $\forall s \cdot s \in SOURCES \Rightarrow init\_sent\_by\_s(s) = \varnothing$
$axm5$: $\forall p \cdot p \in PCK \Rightarrow init\_dr\_dst\_pck(p) = \varnothing$
$axm6$: $\forall dr \cdot dr \in DRs \Rightarrow init\_rcvd\_by\_dr(dr) = \varnothing$
END

— $init\_sent\_by\_s$ associe à chaque source un sous-ensemble vide de paquets ($axm1$, $axm4$).

— *init_dr_dst_pck* associe à chaque paquet un sous-ensemble de vide routeurs DRs ($axm2$, $axm5$).
— *init_rcvd_by_dr* associe à chaque routeur DR un sous ensemble vide de paquets ($axm3$, $axm6$).

Les variables abstraites *sent_by_s*, *rcvd_by_dr*, *dr_r_pck* sont supprimées et remplacées par les variables concrètes *l_sent_by_s*, *l_rcvd_by_dr* et *l_dr_dst_pck* dans la machine M9_P1.

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus$ $sent\_by\_s, rcvd\_by\_dr, dr\_r\_pck := \varnothing$
$\oplus$ $l\_sent\_by\_s := init\_sent\_by\_s$
$\oplus$ $l\_rcvd\_by\_dr := init\_rcvd\_by\_dr$
$\oplus$ $l\_dr\_dst\_pck := init\_dr\_dst\_pck$
END

L'ensemble des variables de ce raffinement, que nous notons x_{9_1}, est constitué de ces nouvelles variables, des variables x_8 du niveau précédent (sans les variables abstraites remplacées par les nouvelles).

L'invariant noté $I_{9_1}(x_{9_1})$ caractérisant ces variables est défini comme suit :

— Des propriétés nous permettent de typer les nouvelles variables.
  — *l_sent_by_s* est une fonction totale associant à chaque source de paquets, un ensemble de paquets que la source a envoyés ($inv1$). Cette variable est initialisée à l'aide de la constante *init_sent_by_s*.
  — *l_dr_dst_pck* est une fonction totale associant à chaque paquet envoyé par une source les informations suivantes : les identités des routeurs DRs vers lesquels il est expédié ($inv2$). Cette variable est initialisée à l'aide de la constante *init_dr_dst_pck*.
  — *l_rcvd_by_dr* est une fonction totale associant à chaque routeur DR, un ensemble de paquets que le routeur aura reçu des sources ($inv3$). Cette variable est initialisée à l'aide de la constante *init_dr_dst_pck*.

$inv1 : l\_sent\_by\_s \in SOURCES \rightarrow \mathbb{P}(PCK)$
$inv2 : l\_dr\_dst\_pck \in PCK \rightarrow \mathbb{P}(DRs)$
$inv3 : l\_rcvd\_by\_dr \in DRs \rightarrow \mathbb{P}(PCK)$

— Les propriétés $inv4$, $inv5$, $inv6$ établissent les relations entre les variables abstraites et concrètes : pour chaque paquet envoyé par une source s ou reçu par un routeur DR dr au niveau abstrait, le même paquet est envoyé ($inv4$) ou reçu ($inv6$) au niveau concret ; chaque paquet p transitant vers un routeur DR au niveau abstrait transite aussi au niveau concret vers ce même routeur DR ($inv5$).

$inv4 : \forall s \cdot s \in SOURCES \Rightarrow l\_sent\_by\_s(s) = sent\_by\_s[\{s\}]$
$inv5 : \forall p \cdot p \in PCK \Rightarrow l\_dr\_dst\_pck(p) = dr\_r\_pck^{-1}[\{p\}]$
$inv6 : \forall dr \cdot dr \in DRs \Rightarrow l\_rcvd\_by\_dr(dr) = rcvd\_by\_dr[\{dr\}]$

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
sent_by_s	*l_sent_by_s*
dr_r_pck	*l_dr_dst_pck*
rcvd_by_dr	*l_rcvd_by_dr*

Les axiomes du contexte C6, ainsi que l'invariant de collage I_{9_1} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M8 en M9_P1, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M9_P1, ainsi que pour justifier la préservation des propriétés Φ_8 lors du raffinement.

Les événements de la machine M9_P1 sont des raffinements des événements de la machine M8 précédente : pour chaque événement X8, nous avons un raffinement correspondant noté X9_P1. Une liste Φ_{9_1} des propriétés de vivacité satisfaites par M9_P1, et détaillée en annexe, nous permet de raffiner les événements modélisant la première phase du protocole ANYCAST RP.

EVENT SENDING9_P1 REFINES SENDING8 ≙
ANY
...
WHERE
...
⊖ $grd6 \ : \ s \mapsto p \notin sent\_by\_s$
⊕ $grd6 \ : \ p \notin l\_sent\_by\_s(s)$
THEN
...
⊖ $act1, act2$
⊕ $act1 \ : \ l\_sent\_by\_s(s) := l\_sent\_by\_s(s) \cup \{p\}$
⊕ $act2 \ : \ l\_dr\_dst\_pck(p) := l\_dr\_dst\_pck(p) \cup gdr$
...
END

EVENT DR_RECEIVING9_P1 REFINES DR_RECEIVING8 ≙
ANY
...
WHERE
...
⊖ $grd3, grd4$
⊕ $grd3 \ : \ dr \in l\_dr\_dst\_pck(p)$
⊕ $grd4 \ : \ p \notin l\_rcvd\_by\_dr(dr)$
...
THEN
⊖ $act1$
⊕ $act1 \ : \ l\_rcvd\_by\_dr(dr) := l\_rcvd\_by\_dr(dr) \cup \{p\}$
...
END

EVENT MOVING_PCK9_P1
REFINES MOVING_PCK8 ≙
ANY
...
WHERE
...
⊖ $grd8 \ : \ dr \mapsto p \in dr\_r\_pck$
⊕ $grd8 \ : \ dr \in l\_dr\_dst\_pck(p)$
...
THEN
...
END

EVENT DR_SENDING9_P1
REFINES DR_SENDING8 ≙
ANY
...
WHERE
...
⊖ $grd5 \ : \ dr \mapsto p \in rcvd\_by\_dr$
⊕ $grd5 \ : \ p \in l\_rcvd\_by\_dr(dr)$
...
THEN
...
END

— L'événement SENDING9_P1 est localisé sur la source s et un paquet p que cette source s veut envoyer : le paquet p est envoyé s'il ne fait pas partie des paquets diffusés par la source s ($grd6$). Dans ce cas, la source s met le paquet p dans l'ensemble des paquets qu'elle a envoyé ($act1$) et l'information suivante est attachée au paquet p : sa prochaine destination est le routeur dr (contenu dans le groupe gdr) ($act2$).

— L'événement MOVING_PCK9_P1 est localisé sur les nœuds source s et routeur DR dr dans lesquels le paquet p transite, ainsi que sur le paquet p. Les informations du paquet p permettent de savoir qu'il transite vers le routeur dr ($grd8$).

— L'événement DR_RECEIVING9_P1 est localisé sur le routeur DR dr recevant un paquet p, ainsi que sur ce dernier. Le paquet p apporte l'information que le routeur DR dr est le destinataire qu'il cherche à atteindre ($grd3$) et il ne fait pas encore partie de l'ensemble des paquets reçus par le routeur DR dr. Ce dernier peut ainsi le mettre dans sa liste de paquets reçus ($act1$).

— L'événement DR_SENDING9_P1 ne fait pas partie des événements modélisant la première phase de l'algorithme Anycast RP, cependant nous le modifions aussi par raffinement, pour remplacer la variable abstraite $rcvd\_by\_dr$ par la variable concrète $l\_rcvd\_by\_dr$.

Nous avons localisé dans ce raffinement la première phase de l'algorithme Anycast RP, qui comprend les communications entre les sources des paquets et les routeurs désignés (DRs). Le diagramme 5.13 suivant résume ce raffinement de M8 en M9_P1 :

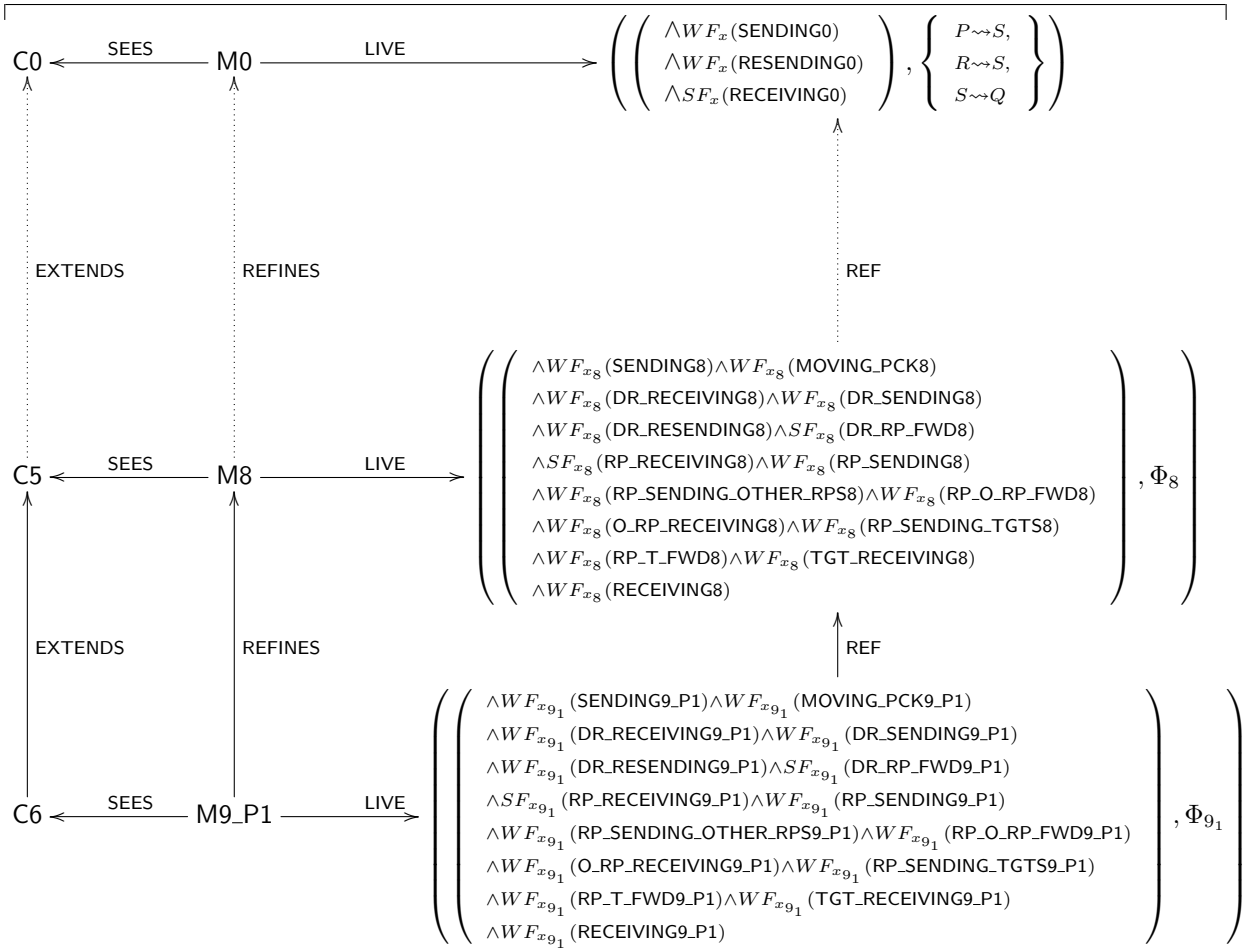


FIGURE 5.13 – Neuvième raffinement du protocole ANYCAST RP

Nous présentons dans la sous-section suivante la localisation de la seconde phase du protocole ANYCAST RP, c'est-à-dire, les étapes de communications entre les routeurs désignés (DRs) et les points de rendez-vous (RPs).

5.3.2.11 Localisation de la seconde phase

Cette sous-section présente la localisation des éléments des modèles relatifs à la seconde phase de l'algorithme ANYCAST RP, c'est-à-dire à la phase décrivant les communications entre un routeur désigné (DR) choisi par une source pour diffuser un paquet et le router point de rendez-vous (RP) choisi par le routeur DR pour recevoir en premier un message « register » encapsulant le paquet. Nous nous intéressons ici particulièrement à trois étapes algorithmiques que nous pouvons trouver dans cette phase :

— La diffusion d'un message « register » encapsulant un paquet par un routeur DR à un routeur RP, qu'il a sélectionné selon des critères d'optimalité (distance, charge, etc.).
— La réponse envoyée par le routeur RP au routeur DR, sous la forme d'un message « register-stop ».
— L'entrée du routeur RP choisi dans un état « (S,G) » relatif au message « register » qu'il a reçu.

Nous définissons un contexte C7 étendant le contexte C6 vu précédemment. Nous y définissons des constantes, qui seront utilisées lors de l'initialisation des variables d'état de la machine M9_P2, raffinement de la machine M9_P1 vue dans la sous-section précédente :

CONTEXT C7 EXTENDS C6
CONSTANTS
$init\_sent\_by\_dr, init\_rcvd\_by\_rp, init\_reg\_rp\_dst, init\_rs\_sent\_by\_rp$
$init\_rs\_dr\_dst, init\_rs\_rcvd\_by\_dr, init\_reg\_ack\_by\_dr, init\_state\_sg$
AXIOMS
$axm1 : init\_sent\_by\_dr \in DRs \to \mathbb{P}(REG)$
$axm2 : init\_reg\_rp\_dst \in REG \to \mathbb{P}(RPs)$
$axm3 : init\_rcvd\_by\_rp \in RPs \to \mathbb{P}(REG)$
$axm4 : init\_rs\_sent\_by\_rp \in RPs \to \mathbb{P}(REG\_STOP \times REG)$
$axm5 : init\_rs\_dr\_dst \in (REG\_STOP \times REG) \to \mathbb{P}(DRs)$
$axm6 : init\_rs\_rcvd\_by\_dr \in DRs \to \mathbb{P}(REG\_STOP \times REG)$
$axm7 : init\_reg\_ack\_dr \in DRs \to \mathbb{P}(REG)$
$axm8 : init\_state\_sg \in REG \to \mathbb{P}(RPs)$
$axm9 : \forall s \cdot s \in DRs \Rightarrow init\_sent\_by\_dr(dr) = \varnothing$
$axm10 : \forall r \cdot r \in REG \Rightarrow init\_reg\_rp\_dst(r) = \varnothing$
$axm11 : \forall rp \cdot rp \in RPs \Rightarrow init\_rcvd\_by\_rp(rp) = \varnothing$
$axm12 : \forall rp \cdot rp \in RPs \Rightarrow init\_rs\_sent\_by\_rp(rp) = \varnothing$
$axm13 : \forall rs \cdot rs \in (REG\_STOP \times REG) \Rightarrow init\_rs\_dr\_dst(rs) = \varnothing$
$axm14 : \forall dr \cdot dr \in DRs \Rightarrow init\_rs\_rcvd\_by\_dr(dr) = \varnothing$
$axm15 : \forall dr \cdot dr \in DRs \Rightarrow init\_reg\_ack\_dr(dr) = \varnothing$
$axm16 : \forall r \cdot r \in REG \Rightarrow init\_state\_sg(r) = \varnothing$
END

— *init_sent_by_dr* associe chaque élément de l'ensemble des routeurs DRs à un sous-ensemble vide de messages « register » ($axm1$, $axm9$).
— *init_reg_rp_dst* ($axm2$, $axm10$) et *init_state_sg* ($axm8$, $axm16$) associent chaque message « register » à un sous-ensemble vide de routeurs RPs .
— *init_rcvd_by_rp* associe chaque routeur RP à un sous-ensemble vide de messages « register » ($axm7$, $axm15$). ($axm3$, $axm11$).
— *init_rs_sent_by_rp* associe chaque routeur RP à un sous-ensemble vide de messages « register-stop », envoyés en réponses à des messages « register » ($axm4$, $axm12$).
— *init_rs_dr_dst* associe à chaque combinaison possible de messages « register-stop » et messages « register », un sous-ensemble de routeurs DRs vide ($axm5$, $axm13$).
— *init_rs_rcvd_by_dr* associe chaque routeur DR à un sous-ensemble vide de messages « register-stop », envoyés en réponses à des messages « register » ($axm6$, $axm14$).
— *init_reg_ack_dr* associe chaque routeurs DR à un sous-ensemble vide de messages « register » ($axm7$, $axm15$).

Des variables abstraites de la machine M9_P1 sont remplacées, dans ce raffinement M9_P2, par des variables concrètes :

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus\ reg\_sent\_by\_dr, reg\_rcvd\_by\_rp, reg\_rp\_r\_pck, rs\_sent\_by\_rp := \varnothing$
$\ominus\ rs\_dr\_r, rs\_rcvd\_by\_dr, dr\_reg\_ack\_rs, state\_sg := \varnothing$
$\oplus\ l\_reg\_sent\_by\_dr := init\_sent\_by\_dr$
$\oplus\ l\_rcvd\_by\_rp := init\_rcvd\_by\_rp$
$\oplus\ l\_reg\_rp\_dst := init\_reg\_rp\_dst$
$\oplus\ l\_rs\_sent\_by\_rp := init\_rs\_sent\_by\_rp$
$\oplus\ l\_rs\_dr\_dst := init\_rs\_dr\_dst$
$\oplus\ l\_rs\_rcvd\_by\_dr := init\_rs\_rcvd\_by\_dr$
$\oplus\ l\_reg\_ack\_dr := init\_reg\_ack\_dr$
$\oplus\ l\_state\_sg := init\_state\_sg$
END

L'ensemble des variables de la machine M9_P2, que nous notons x_{9_2} est composé des variables x_{9_1} du modèle précédent, dont certaines variables abstraites ont été remplacées par ces nouvelles variables :
— *l_sent_by_dr* remplace *reg_sent_by_dr*. Elle associe à chaque routeur DR un ensemble de messages « register » envoyés. Elle est initialisée à l'aide de la constante *init_sent_by_dr*.
— *l_reg_rp_dst* remplace *reg_rp_r_pck*. Elle associe à chaque message « register » une information : il s'agit de l'ensemble des routeurs RPs vers lesquels le message « register » est envoyé. Techniquement, cet ensemble de routeurs RPs contient tout au plus un routeur, il s'agit du routeur RP choisi par le routeur DR émetteur du message « register ». Cette variable est initialisée à l'aide de la constante *init_reg_rp_dst*.
— *l_rcvd_by_rp* remplace *reg_rcvd_by_rp*. Elle couple chaque routeur RP à un ensemble de messages « register » reçus. Cette variable est initialisée à l'aide de la constante *init_rcvd_by_rp*.

— $l\_rs\_sent\_by\_rp$ remplace $rs\_sent\_by\_rp$. Elle couple chaque routeur RP à un ensemble de messages « register-stop » envoyés en réponse à des messages « register » reçus. Cette variable est initialisée à l'aide de la constante $init\_rs\_sent\_by\_rp$.
— $l\_rs\_dr\_dst$ remplace rs_dr_r. Elle associe à chaque couple possible de messages « register-stop » et messages « register » (le message « register-stop » peut être envoyé en réponse au message « register »), un ensemble de routeurs DRs destinataires des messages « register-stop ». Techniquement, cet ensemble de routeurs DRs contient tout au plus un routeur, il s'agit de celui qui a diffusé le message « register » à un routeur RP de sont choix. Cette variable est initialisée à l'aide de la constante $init\_rs\_dr\_dst$.
— $l\_rs\_rcvd\_by\_dr$ remplace $rs\_rcvd\_by\_dr$. Elle associe à chaque routeur DR un ensemble de messages « register-stop » (envoyés en réponse à des messages « register ») reçus. Cette variable est initialisée à l'aide de la constante $init\_rs\_rcvd\_by\_dr$.
— $l\_reg\_ack\_dr$ remplace $dr\_reg\_ack\_rs$. Elle associe à chaque routeur DR un ensemble de messages « register », pour lesquels le routeur DR a reçu un acquittement sous la forme d'un message « register-stop ». Cette variable est initialisée à l'aide de la constante $init\_reg\_ack\_dr$.
— l_state_sg remplace $state\_sg$. Elle associe à chaque message « register » un ensemble de routeurs RPs qui sont entrés dans l'état « (S,G) », suite à la réception du message « register ». Cette variable est initialisée à l'aide de la constante $init\_state\_sg$.

L'invariant de ce raffinement, noté $I_{9_2}(x_{9_2})$ est défini comme suit :

— Les propriétés de $inv1$ à $inv8$ permettent de typer les nouvelles variables définies dans ce raffinement.

$$
\begin{array}{l}
inv1 \ : \ l\_sent\_by\_dr \in DRs \rightarrow \mathbb{P}(REG) \\
inv2 \ : \ l\_reg\_rp\_dst \in REG \rightarrow \mathbb{P}(RPs) \\
inv3 \ : \ l\_rcvd\_by\_rp \in RPs \rightarrow \mathbb{P}(REG) \\
inv4 \ : \ l\_rs\_sent\_by\_rp \in RPs \rightarrow \mathbb{P}(REG\_STOP \times REG) \\
inv5 \ : \ l\_rs\_dr\_dst \in (REG\_STOP \times REG) \rightarrow \mathbb{P}(DRs) \\
inv6 \ : \ l\_rs\_rcvd\_by\_dr \in DRs \rightarrow \mathbb{P}(REG\_STOP \times REG) \\
inv7 \ : \ l\_reg\_ack\_dr \in DRs \rightarrow \mathbb{P}(REG) \\
inv8 \ : \ l\_state\_sg \in REG \rightarrow \mathbb{P}(RPs)
\end{array}
$$

— Des invariants de collage permettent d'établir les relations entre les variables abstraites et les variables concrètes : si un message est soit acquitté ($inv15$), soit transitant dans le graphe ($inv10$, $inv13$), soit envoyé ($inv9$, $inv12$), soit reçu par ($inv11$, $inv14$) un routeur, au niveau concret, alors le même message a reçu le même traitement au niveau abstrait ; si routeur RP a décidé d'entrer dans un état « (S,G) » au niveau concret, ce même routeur RP l'a aussi décidé au niveau abstrait ($inv16$). Ces invariants nous permettent de supprimer les variables abstraites du modèle et de les remplacer par les variables concrètes.

$$
\begin{array}{l}
inv9 \ : \ \forall dr \cdot dr \in DRs \Rightarrow l\_sent\_by\_dr(dr) = reg\_sent\_by\_dr[\{dr\}] \\
inv10 \ : \ \forall r \cdot r \in REG \Rightarrow l\_reg\_rp\_dst(r) = reg\_rp\_r\_pck^{-1}[\{r\}] \\
inv11 \ : \ \forall rp \cdot rp \in RPs \Rightarrow l\_rcvd\_by\_rp(rp) = rcvd\_by\_rp[\{rp\}] \\
inv12 \ : \ \forall rp \cdot rp \in RPs \Rightarrow l\_rs\_sent\_by\_rp(rp) = rs\_sent\_by\_rp[\{rp\}] \\
inv13 \ : \ \forall rs \in (REG\_STOP \times REG) \Rightarrow l\_rs\_dr\_dst(rs) = rs\_dr\_r^{-1}[\{rs\}] \\
inv14 \ : \ \forall dr \cdot dr \in DRs \Rightarrow l\_rs\_rcvd\_by\_dr(dr) = rs\_rcvd\_by\_dr[\{dr\}] \\
inv15 \ : \ \forall dr \cdot dr \in DRs \Rightarrow l\_reg\_ack\_dr(dr) = dr\_reg\_ack\_rs[\{dr\}] \\
inv16 \ : \ \forall r \cdot r \in REG \Rightarrow l\_state\_sg(r) = state\_sg[\{r\}]
\end{array}
$$

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
$sent\_by\_dr$	$l\_sent\_by\_dr$
$reg\_rp\_r\_pck$	$l\_reg\_rp\_dst$
$rcvd\_by\_rp$	$l\_rcvd\_by\_rp$
$rs\_sent\_by\_rp$	$l\_rs\_sent\_by\_rp$
rs_dr_r	$l\_rs\_dr\_dst$
$rs\_rcvd\_by\_dr$	$l\_rs\_rcvd\_by\_dr$
$dr\_reg\_ack\_rs$	$l\_reg\_ack\_dr$
$state\_sg$	l_state_sg

Les axiomes du contexte C7, ainsi que l'invariant de collage I_{9_2} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M9_P1 en M9_P2, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M9_P2, ainsi que pour justifier la préservation des propriétés Φ_{9_1} lors du raffinement.

La machine M9_P2 contient les raffinements des événements de la machine M9_P1 précédente : un événement de M9_P1 noté X9_P1 est raffiné en une version X9_P2. Similairement à ce qui a été vu dans les machine précédente, une liste Φ_{9_2} des propriétés de vivacité satisfaites par la machine M9_P2, détaillée en annexe, nous permet de raffiner les événements de M9_P1 en ceux de M9_P2 :

— L'événement DR_SENDING9_P2 est localisé sur un routeur DR dr qui diffuse un message « register » reg et sur ce dernier : lors de l'envoi, le message reg est ajouté dans la liste des messages « register » envoyé par le routeur dr ($act1$) et une information indiquant sa destination est ajouté au message reg ($act2$: il est envoyé à un groupe grp de routeurs RPs contenant un seul membre).

EVENT DR_SENDING9_P2
REFINES DR_SENDING9_P1 ≙
  ANY
    ...
  WHERE
    ...
    ⊖ $grd9 \ : \ dr \mapsto reg \notin reg\_sent\_by\_dr$
    ⊕ $grd9 \ : \ reg \notin l\_sent\_by\_dr(dr)$
    ...
  THEN
    ⊖ $act1, act2$
    ⊕ $act1 \ : \ l\_sent\_by\_dr(dr) := l\_sent\_by\_dr(dr) \cup \{reg\}$
    ⊕ $act2 \ : \ l\_reg\_rp\_dst(r) := l\_reg\_rp\_dst(r) \cup grp$
    ...
  END

.

— L'événement DR_RESENDING9_P2 est localisé sur un routeur DR dr qui va rediffuser un message « register » reg : ce dernier est rediffusé par le routeur dr, s'il fait partie des messages déjà diffusés par le routeur ($grd3$) et s'il ne figure pas dans la liste des messages « register » pour lesquels le routeur a reçu un acquittement sous la forme d'un message « register-stop » ($grd4$).

EVENT DR_RESENDING9_P2
REFINES DR_RESENDING9_P1 ≙
  ANY
    ...
  WHERE
    ...
    ⊖ $grd3, grd4$
    ⊕ $grd3 \ : \ reg \in l\_sent\_by\_dr(dr)$
    ⊕ $grd4 \ : \ reg \notin l\_reg\_ack\_dr(dr)$
    ...
  THEN
    ...
  END

— L'événement DR_RP_FWD9_P2 est localisé sur les nœuds x, y dans lesquels transitent le message « register » reg et sur ce dernier : le message reg circule dans le réseau tant que sa destination d (contenue dans les informations encapsulées dans le message) n'est pas atteinte ($grd6$).

EVENT DR_RP_FWD9_P2
REFINES DR_RP_FWD9_P1 ≙
  ANY
    ...
  WHERE
    ...
    ⊖ $grd6 \ : \ d \mapsto reg \in reg\_rp\_r\_pck$
    ⊕ $grd6 \ : \ d \in l\_reg\_rp\_dst(reg)$
    ...
  THEN
    ...
  END

— L'événement RP_RECEIVING9_P2 est localisé sur un routeur RP rp choisi pour recevoir un message « register » reg par un routeur DR : le routeur rp met le message reg dans la liste des messages « register » qu'il a reçu ($act1$), si ce dernier se trouve au niveau du routeur dans le réseau et si le routeur rp fait partie des destinations intermédiaire du message reg ($grd3$).

EVENT RP_RECEIVING9_P2
REFINES RP_RECEIVING9_P1 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd3$: $rp \mapsto reg \in reg\_rp\_r\_pck$
    $\oplus$ $grd3$: $rp \in l\_reg\_rp\_dst(reg)$
    ...
  THEN
    $\ominus$ $act1$: $reg\_rcvd\_by\_rp := reg\_rcvd\_by\_rp \cup \{rp \mapsto reg\}$
    $\oplus$ $act1$: $l\_rcvd\_by\_rp(rp) := l\_rcvd\_by\_rp(rp) \cup \{reg\}$
  END

— L'événement RP_SENDING9_P2 ne fait pas partie de la seconde phase de l'algorithme ANYCAST RP, cependant, il utilise dans sa garde la variable abstraite $reg\_rcvd\_by\_rp$ de cette seconde phase. Nous remplaçons donc cette variable par sa version concrète $l\_rcvd\_by\_rp$.

EVENT RP_SENDING9_P2
REFINES RP_SENDING9_P1 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd3$: $rp \mapsto reg \in reg\_rcvd\_by\_rp$
    $\oplus$ $grd3$: $reg \in l\_rcvd\_by\_rp(rp)$
    ...
  THEN
    ...
  END

— L'événement RECEIVING9_P2ne fait pas partie de la seconde phase de l'algorithme ANYCAST RP, cependant, nous le transformons par raffinement car la première partie de la garde $grd5$ utilise maintenant la nouvelle variable $l\_rcvd\_by\_rp$, pour obtenir l'ensemble des RPs ayant reçu, d'un routeur DR, le message « register » encapsulant le paquet p.

EVENT RECEIVING9_P2
REFINES RECEIVING9_P1 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd5$
    $\oplus$ $grd5$: $\{rp | rp \in RPs \wedge enc(p) \in l\_rcvd\_by\_rp(rp)\} \cup rcvd\_by\_o\_rp^{-1}[\{enc(p)\}] = RPs$
    ...
  THEN
    ...
  END

— L'événement LOSING9_P2 modélise une action de l'environnement (perte d'un message « register » envoyé à destination d'un routeur RP). Nous nous contentons donc de le raffiner pour remplacer les variables abstraites mises en jeu dans cet événement par les variables concrètes.

EVENT LOSING9_P2
REFINES LOSING9_P1 $\widehat{=}$
  ANY
    reg, dr, rp
  WHERE
    $grd1$: $dr \in DR$
    $grd2$: $rp \in RP$
    $grd3$: $reg \in REG$
    $grd4$: $reg \in l\_sent\_by\_dr(dr)$
    $grd5$: $rp \in l\_reg\_rp\_dst(reg)$
    $grd6$: $reg \notin l\_rcvd\_by\_rp(rp)$
    $grd7$: $reg \in ran(dr\_rp\_store)$
  THEN
    $act1$: $dr\_rp\_store := dr\_rp\_store \rhd \{reg\}$
  END

Nous modifions aussi certains événements ne participant pas à l'acheminement d'un paquet p de sa source à sa destination par raffinement :

— L'événement RP_REG_STOP_SENDING9_P2 est localisé sur un routeur RP rp, sur le message « register-stop » stp qu'il envoie en réponse à message « register » reg et sur ce dernier : Le routeur rp, destination intermédiaire du message reg, a ce dernier dans la liste des messages qu'il a reçus ($grd6$) ; le routeur rp met le message « register-stop » stp (en réponse au message reg) dans la liste des messages « register-stop » qu'il a envoyés ($act1$) et y ajoute l'information suivante : le message stp (en réponse au message reg) est envoyé à destination d'un routeur dr, seul élément du groupe gdr ($act2$).

EVENT RP_REG_STOP_SENDING9_P2
REFINES RP_REG_STOP_SENDING9_P1 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd6$: $rp \mapsto reg \in reg\_rcvd\_by\_rp$
    $\oplus$ $grd6$: $reg \in l\_rcvd\_by\_rp(rp)$
    ...
  THEN
    $\ominus$ $act1, act2$
    $\oplus$ $act1$: $l\_rs\_sent\_by\_rp(rp) := l\_rs\_sent\_by\_rp(rp) \cup \{stp \mapsto reg\}$
    $\oplus$ $act2$: $l\_rs\_dr\_dst(rs) := l\_rs\_dr\_dst(rs) \cup gdr$
  END

— L'événement RP_DR_STOP_FWD9_P2 est localisé sur les nœuds x, y dans lesquels transitent un message « register-stop » (en réponse à un message « register ») et sur le couple rs (message « register-stop », message « register ») : le message « register-stop », acquittant le message « register », circule dans le réseau tant que sa destination d (contenue dans les informations encapsulées dans le message) n'est pas atteinte ($grd5$).

EVENT RP_DR_STOP_FWD9_P2
REFINES RP_DR_STOP_FWD9_P1 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd5$: $d \mapsto rs \in rs\_dr\_r$
    $\oplus$ $grd5$: $d \in l\_rs\_dr\_dst(rs)$
  THEN
    ...
  END

— L'événement DR_REG_STOP_RECEIVING9_P2 est localisé sur un routeur DR dr, sur le message « register-stop » stp qu'il a reçu en réponse à un message « register » reg qu'il a envoyé et sur le message « register » reg. Le routeur dr est le destinataire du message stp ($grd6$) et ce message se trouve au niveau du routeur dr dans le réseau : le routeur dr met le message reg dans la liste des messages « register » pour lesquels il a reçu un acquittement ($act1$) et il met le message stp, acquittant le message reg, dans la liste des messages « register-stop » qu'il a reçus ($act2$).

EVENT DR_REG_STOP_RECEIVING9_P2
REFINES DR_REG_STOP_RECEIVING9_P1 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd6$: $dr \mapsto (stp \mapsto reg) \in rs\_dr\_r$
    $\oplus$ $grd6$: $dr \in l\_rs\_dr\_dst(stp \mapsto reg)$
    ...
  THEN
    $\ominus$ $act1, act2$
    $\oplus$ $act1$: $l\_reg\_ack\_dr(dr) := l\_reg\_ack\_dr(dr) \cup \{reg\}$
    $\oplus$ $act2$: $l\_rs\_rcvd\_by\_dr(dr) := l\_rs\_rcvd\_by\_dr(dr) \cup \{stp \mapsto reg\}$
    ...
  END

— L'événement DR_REG_STOP_LOSING9_P2 modélise une action de l'environnement (perte d'un message « register-stop » envoyé à destination d'un routeur DR). Nous nous contentons donc de le raffiner pour remplacer les variables abstraites mises en jeu dans cet événement par les variables concrètes.

EVENT DR_REG_STOP_LOSING9_P2
REFINES DR_REG_STOP_LOSING9_P1 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd5, grd6, grd7$
    $\oplus$ $grd5$: $stp \mapsto reg \in l\_rs\_sent\_by\_rp(rp)$
    $\oplus$ $grd6$: $reg \notin l\_reg\_ack\_dr(dr)$
    $\oplus$ $grd7$: $dr \in l\_rs\_dr\_dst(stp \mapsto reg)$
  THEN
    $\ominus$ $act1, act2$
    $\oplus$ $act1$: $l\_rs\_sent\_by\_rp(rp) := l\_rs\_sent\_by\_rp(rp) \setminus \{stp \mapsto reg\}$
    $\oplus$ $act2$: $l\_rs\_dr\_dst(stp \mapsto reg) := l\_rs\_dr\_dst(stp \mapsto reg) \setminus \{dr\}$
    ...
  END

— Nous localisons l'événement RP_CREATING_STATE_SG9_P2 sur un message « register » reg, ainsi que sur le routeur rp qui a décidé d'entrer dans un état « (S,G) », après avoir reçu le message reg, l'avoir désencapsulé et envoyé aux destinataires qui l'ont rejoint.

EVENT RP_CREATING_STATE_SG9_P2
REFINES RP_CREATING_STATE_SG9_P1 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd6$
    $\oplus$ $grd6$: $rp \notin l\_state\_sg(reg)$
  THEN
    $\ominus$ $act1$
    $\oplus$ $act1$: $l\_state\_sg(reg) := l\_state\_sg(reg) \cup \{rp\}$
  END

Nous avons localisé dans ce raffinement la deuxième phase de l'algorithme ANYCAST RP, qui comprend les communications entre les routeurs désignés (DRs) et les routeurs points de rendez-vous (RPs). Le diagramme 5.14 suivant résume ce raffinement de M9_P1 en M9_P2 :

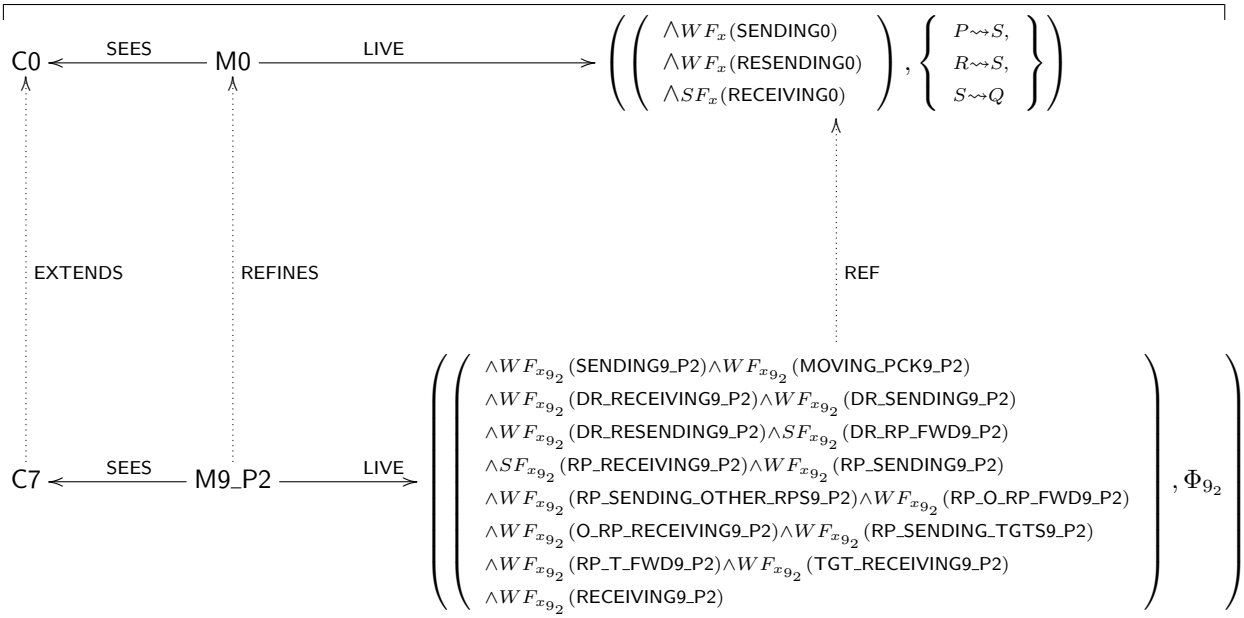


FIGURE 5.14 – Dixième raffinement du protocole ANYCAST RP

Nous présentons dans la sous-section suivante la localisation de la troisième phase du protocole ANYCAST RP, c'est-à-dire, les étapes de communications entre les routeurs points de rendez-vous (*RPs*) et les destinataires d'un paquet.

5.3.2.12 Localisation de la troisième phase

Nous nous intéressons maintenant à la localisation de la troisième phase de l'algorithme ANYCAST RP. Nous décrivons dans cette sous-section la localisation des éléments de nos modèles EVENT-B relatifs aux communications entre routeurs Points de Rendez-vous (RPs), ainsi que celles entre ces routeurs et les destinataires des paquets.

Nous nous focaliserons sur quelques étapes qui sont les suivantes :

— La diffusion d'un message « register », transmis par un routeur DR, par le routeur RP qu'il a choisi, à destination des autres routeurs RPs.
— La désencapsulation d'un message « register » par un routeur RP et la diffusion du paquet encapsulé par le message aux destinataires.
— L'envoi d'un message « register-stop » (en réponse à un message « register » reçu) par un routeur RP non choisi par un routeur DR au routeur RP choisi par le routeur DR.
— L'entrée des routeurs RPs non-choisis par des routeurs DRs dans un état « (S,G) », suite à la réception d'un message « register » et à la diffusion de ce dernier aux destinataires ayant rejoint les routeurs RPs.

Nous commençons par étendre le contexte C7 vu dans la sous-section précédente en un contexte C8 définissant des constantes qui sont utilisées dans le cadre de la localisation des modèles.

CONTEXT C8 EXTENDS C7
CONSTANTS
$init\_sent\_by\_rp, init\_sent\_to\_o\_rp, init\_reg\_o\_rp\_dst, init\_rcvd\_by\_o\_rp,$
$init\_sent\_to\_t, init\_t\_dst, init\_rcvd\_by\_t, init\_rs\_sent\_by\_o\_rp,$
$init\_rs\_rp\_dst, init\_rs\_rcvd\_by\_rp, init\_rp\_reg\_ack\_rs, init\_reg\_rs\_o\_rp,$
$join\_rp$
AXIOMS
$axm1 : init\_sent\_by\_rp \in RPs \to \mathbb{P}(REG)$
$axm2 : init\_sent\_to\_o\_rp \in RPs \to \mathbb{P}(REG)$
$axm3 : init\_reg\_o\_rp\_dst \in REG \to \mathbb{P}(RPs)$
$axm4 : init\_rcvd\_by\_o\_rp \in RPs \to \mathbb{P}(REG)$
$axm5 : init\_sent\_to\_t \in RPs \to \mathbb{P}(PCK)$
$axm6 : init\_t\_dst \in PCK \to \mathbb{P}(TARGETS)$
$axm7 : init\_rcvd\_by\_t \in TARGETS \to \mathbb{P}(PCK)$
$axm8 : init\_rs\_sent\_by\_o\_rp \in RPs \to \mathbb{P}(REG\_STOP \times REG)$
$axm9 : init\_rs\_rp\_dst \in (REG\_STOP \times REG) \to \mathbb{P}(RPs)$
$axm10 : init\_rs\_rcvd\_by\_rp \in RPs \to \mathbb{P}(REG\_STOP \times REG)$
$axm11 : init\_rp\_reg\_ack\_rs \in RPs \to \mathbb{P}(RPs \times REG)$
$axm12 : init\_reg\_rs\_o\_rp \in RPs \to \mathbb{P}(REG)$
$axm13 : join\_rp \in RPs \to \mathbb{P}(TARGETS)$
$axm14 : \forall rp \cdot rp \in RPs \Rightarrow init\_sent\_by\_rp(rp) = \varnothing$
$axm15 : \forall rp \cdot rp \in RPs \Rightarrow init\_sent\_to\_o\_rp(rp) = \varnothing$
$axm16 : \forall rp \cdot rp \in RPs \Rightarrow init\_reg\_o\_rp\_dst(rp) = \varnothing$
$axm17 : \forall reg \cdot reg \in REG \Rightarrow init\_rcvd\_by\_o\_rp(reg) = \varnothing$
$axm18 : \forall rp \cdot rp \in RPs \Rightarrow init\_sent\_to\_t(rp) = \varnothing$
$axm19 : \forall p \cdot p \in PCK \Rightarrow init\_t\_dst(p) = \varnothing$
$axm20 : \forall t \cdot t \in TARGETS \Rightarrow init\_rcvd\_by\_t(t) = \varnothing$
$axm21 : \forall rp \cdot rp \in RPs \Rightarrow init\_rs\_sent\_by\_o\_rp(rp) = \varnothing$
$axm22 : \forall rs \cdot rs \in (REG\_STOP \times REG) \Rightarrow init\_rs\_rp\_dst(rs) = \varnothing$
$axm23 : \forall rp \cdot rp \in RPs \Rightarrow init\_rs\_rcvd\_by\_rp(rp) = \varnothing$
$axm24 : \forall rp \cdot rp \in RPs \Rightarrow init\_rp\_reg\_ack\_rs(rp) = \varnothing$
$axm25 : \forall rp \cdot rp \in RPs \Rightarrow init\_reg\_rs\_o\_rp(rp) = \varnothing$
$axm26 : \forall rp \cdot rp \in RPs \Rightarrow join\_rp(rp) = join^{-1}[\{rp\}]$
END

— *init_sent_by_rp* ($axm1$, $axm14$) , *init_sent_to_o_rp* ($axm2$, $axm15$), *init_rcvd_by_o_rp* ($axm4$, $axm17$) et *init_reg_rs_o_rp* ($axm12$, $axm25$) associent chaque RP à un sous-ensemble vide de messages « register ».
— *init_reg_o_rp_dst* associe chaque message « register » à un sous-ensemble vide de routeurs RPs ($axm3$, $axm16$).
— *init_sent_to_t* associe chaque routeur RP à un sous-ensemble vide de paquets ($axm5$, $axm18$).
— *init_t_dst* associe chaque paquet à un sous-ensemble vide de destinataires ($axm6$, $axm19$).
— *init_rcvd_by_t* associe à chaque destinataire, un sous-ensemble vide de paquets ($axm7$, $axm20$).
— *init_rs_sent_by_o_rp* ($axm8$, $axm21$) et *init_rs_rcvd_by_rp* ($axm10$, $axm23$) associent à chaque routeur RP, un sous-ensemble vide de messages « register-stop », envoyés en réponse à des messages « register ».
— *init_rs_rp_dst* associe à chaque combinaison possible (message « register-stop », message « register »), un ensemble vide de routeurs RPs ($axm9$, $axm22$).
— *init_rp_reg_ack_rs* associe à chaque routeur RP, un sous-ensemble vide de couples (routeur RPs, message « register ») ($axm11$, $axm24$).
— *join_rp* est une fonction totale associant à chaque routeur RP un ensemble contenant les destinataires qui l'ont rejoint ($axm13$, $axm26$).

Ce contexte C8 est utilisé par la machine M9_P3, raffinement de la machine M9_P2 vue dans la section précédente. La machine M9_P3 introduit de nouvelles variables qui serviront à remplacer certaines variables abstraites du modèle.

INITIALISATION $\widehat{=}$
BEGIN
$\ldots$
$\ominus$ $reg\_sent\_by\_rp, reg\_sent\_to\_o\_rp, reg\_rcvd\_by\_o\_rp := \varnothing$
$\ominus$ $reg\_o\_rp\_r\_pck, sent\_to\_t, t\_r\_pck, rcvd\_by\_t := \varnothing$
$\ominus$ $rs\_sent\_by\_o\_rp, rs\_rp\_r, rs\_rcvd\_by\_rp, rp\_reg\_ack\_rs, reg\_rs\_o\_rp := \varnothing$
$\oplus$ $l\_reg\_sent\_by\_rp := init\_sent\_by\_rp$
$\oplus$ $l\_sent\_to\_o\_rp := init\_sent\_to\_o\_rp$
$\oplus$ $l\_reg\_o\_rp\_dst := init\_reg\_o\_rp\_dst$
$\oplus$ $l\_rcvd\_by\_o\_rp := init\_rcvd\_by\_o\_rp$
$\oplus$ $l\_sent\_to\_t := init\_sent\_to\_t$
$\oplus$ $l\_t\_dst := init\_t\_dst$
$\oplus$ $l\_rcvd\_by\_t := init\_rcvd\_by\_t$
$\oplus$ $l\_rs\_sent\_by\_o\_rp := init\_rs\_sent\_by\_o\_rp$
$\oplus$ $l\_rs\_rp\_dst := init\_rs\_rp\_dst$
$\oplus$ $l\_rs\_rcvd\_by\_rp := init\_rs\_rcvd\_by\_rp$
$\oplus$ $l\_rp\_reg\_ack\_rs := init\_rp\_reg\_ack\_rs$
$\oplus$ $l\_reg\_rs\_o\_rp := init\_reg\_rs\_o\_rp$
END

— *l_sent_by_rp* remplace *reg_sent_by_rp*. Cette variable associe les routeurs RPs choisis par les routeurs DRs à un ensemble de messages « register » ayant transité dans ces routeurs. Elle initialisée à l'aide de la constante *init_sent_by_rp*.
— *l_sent_to_o_rp* est associe les routeurs RPs choisis par les routeurs DRs à un ensemble de messages « register » que ces routeurs RPs ont diffusé aux autres RPs. Cette variable remplace *reg_sent_to_o_rp* et son état initial est donné par la constante *init_sent − _to_o_rp*.
— *l_reg_o_rp_dst* associe à chaque message « register », les routeurs RPs non-choisis par les routeurs DRs vers lesquels il est diffusé. Cette variable remplace la variable *reg_o_rp_r_pck*. Elle est initialisée à l'aide de la constante *init_reg_o_rp_dst*.
— *l_rcvd_by_o_rp* associe à chaque routeur RP non-choisi par un routeur DR, l'ensemble des messages « register » qu'il a reçus du routeur RP choisi par le routeur DR. Elle remplace la variable *reg_rcvd_by_o_rp* et est initialisée à l'aide de la constante *init_rcvd_by_o_rp*.
— *l_sent_to_t* remplace *sent_to_t* et associe à chaque routeur RP, l'ensemble des paquets qu'il a transmis aux destinataires de ces derniers qui l'ont rejoint. Cette variable est initialisée à l'aide de la constante *init_sent_to_t*.
— *l_t_dst* associe à chaque paquet, envoyé par un routeur RP, la liste des destinataires vers lesquels le paquet transite. Cette variable remplace la variable abstraite *t_r_pck* et elle initialisée à l'aide de la constante *init_t_dst*.
— *l_rcvd_by_t* donne pour chaque destinataire, l'ensemble des paquets qu'il a reçus. Cette variable remplace la variable abstraite *rcvd_by_t* et elle initialisée à l'aide de la constante *init_rcvd_by_t*.
— *l_rs_sent_by_o_rp* associe à chaque routeur RP non-choisi par un routeur DR, un ensemble des messages « register-stop » que le routeur RP a envoyé en réponse à des messages « register ». Cette variable remplace la variable *rs_sent_by_o_rp* et est initialisée à l'aide de la constante *init_rs_sent_by_o_rp*.
— *l_rs_rp_dst* associe à chaque message « register-stop », envoyé par un routeur RP non-choisi par un routeur DR, en réponse à un message « register » reçu, le routeur RP choisi vers lequel le message « register-stop » transite. Cette variable remplace *rs_rp_r* et est initialisée à l'aide de la constante *init_rs_rp_dst*.
— *l_rs_rcvd_by_rp* associe à chaque routeur RP choisi par un routeur DR, l'ensemble des messages « register-stop », reçus en réponse à un message « register » envoyé. Elle remplace la variable *rs_rcvd_by_rp* et est initialisée à l'aide de la constante *init_rs_rcvd_by_rp*.
— *l_rp_reg_ack_rs* associe chaque routeur RP choisi par un routeur DR, l'ensemble des routeurs RP non-choisis desquels il a reçu des messages « register-stop » et les messages « register » pour lesquels ces messages « register-stop » ont été envoyés.
— *l_reg_rs_o_rp* associe à chaque routeur RP non-choisi par un routeur DR, un ensemble des messages « register » pour lesquels le routeur RP a envoyé un acquittement sous la forme d'un message « register-stop ».

Nous notons l'ensemble des variables de cette machine M9_P3 x_{9_3}. L'invariant de ce raffinement, noté $I_{9_3}(x_{9_3})$ est défini comme suit :
— Les propriétés de $inv1$ à $inv12$ permettent de typer les nouvelles variables définies dans ce raffine-

ment.

$$
\begin{array}{l}
inv1 \ : \ l\_sent\_by\_rp \in RPs \rightarrow \mathbb{P}(REG) \\
inv2 \ : \ l\_sent\_to\_o\_rp \in RPs \rightarrow \mathbb{P}(REG) \\
inv3 \ : \ l\_reg\_o\_rp\_dst \in REG \rightarrow \mathbb{P}(RPs) \\
inv4 \ : \ l\_rcvd\_by\_o\_rp \in RPs \rightarrow \mathbb{P}(REG) \\
inv5 \ : \ l\_sent\_to\_t \in RPs \rightarrow \mathbb{P}(PCK) \\
inv6 \ : \ l\_t\_dst \in PCK \rightarrow \mathbb{P}(TARGETS) \\
inv7 \ : \ l\_rcvd\_by\_t \in TARGETS \rightarrow \mathbb{P}(PCK) \\
inv8 \ : \ l\_rs\_sent\_by\_o\_rp \in RPs \rightarrow \mathbb{P}(REG\_STOP \times REG) \\
inv9 \ : \ l\_rs\_rp\_dst \in (REG\_STOP \times REG) \rightarrow \mathbb{P}(RPs) \\
inv10 \ : \ l\_rs\_rcvd\_by\_rp \in RPs \rightarrow \mathbb{P}(REG\_STOP \times REG) \\
inv11 \ : \ l\_rp\_reg\_ack\_rs \in RPs \rightarrow \mathbb{P}(RPs \times REG) \\
inv12 \ : \ l\_reg\_rs\_o\_rp \in RPs \rightarrow \mathbb{P}(REG)
\end{array}
$$

— Les autres propriétés sont appelés « invariants de collage » et permettent de faire le lien entre les états concrets et les états abstraits, et de remplacer les variables abstraites par leurs versions concrètes.

$$
\begin{array}{l}
inv13 \ : \ \forall rp \cdot rp \in RPs \Rightarrow l\_sent\_by\_rp(rp) = reg\_sent\_by\_rp[\{rp\}] \\
inv14 \ : \ \forall rp \cdot rp \in RPs \Rightarrow l\_sent\_to\_o\_rp(rp) = reg\_sent\_to\_o\_rp[\{rp\}] \\
inv15 \ : \ \forall reg \cdot reg \in REG \Rightarrow l\_reg\_o\_rp\_dst(reg) = reg\_o\_rp\_r\_pck^{-1}[\{reg\}] \\
inv16 \ : \ \forall rp \cdot rp \in RPs \Rightarrow l\_rcvd\_by\_o\_rp(rp) = reg\_rcvd\_by\_o\_rp[\{rp\}] \\
inv17 \ : \ \forall rp \cdot rp \in RPs \Rightarrow l\_sent\_to\_t(rp) = sent\_to\_t[\{rp\}] \\
inv18 \ : \ \forall p \cdot p \in PCK \Rightarrow l\_t\_dst(p) = t\_r\_pck^{-1}[\{p\}] \\
inv19 \ : \ \forall t \cdot t \in TARGETS \Rightarrow l\_rcvd\_by\_t(t) = rcvd\_by\_t[\{t\}] \\
inv20 \ : \ \forall rp \cdot rp \in RPs \Rightarrow l\_rs\_sent\_by\_o\_rp(rp) = rs\_sent\_by\_o\_rp[\{rp\}] \\
inv21 \ : \ \forall rs \cdot rs \in (REG\_STOP \times REG) \Rightarrow l\_rs\_rp\_dst(rs) = rs\_rp\_r^{-1}[\{rs\}] \\
inv22 \ : \ \forall rp \cdot rp \in RPs \Rightarrow l\_rs\_rcvd\_by\_rp(rp) = rs\_rcvd\_by\_rp[\{rp\}] \\
inv23 \ : \ \forall rp \cdot rp \in RPs \Rightarrow l\_rp\_reg\_ack\_rs(rp) = rp\_reg\_ack\_rs[\{rp\}] \\
inv24 \ : \ \forall rp \cdot rp \in RPs \Rightarrow l\_reg\_rs\_o\_rp(rp) = reg\_rs\_o\_rp[\{rp\}]
\end{array}
$$

— Les propriétés de $inv13$ à $inv24$ expriment que pour chaque message, paquet envoyé ($inv13$, $inv14$, $inv17$, $inv20$), reçu ($inv16$, $inv19$, $inv22$) par un nœud du réseau, au niveau abstrait, le même message ou paquet a été soit envoyé, soit reçu, par le même nœud du réseau, au niveau concret. Il en est de même pour les messages ($inv15$, $inv21$), paquets ($inv18$) transitant à destination d'un nœud destinataire, ainsi que pour les messages acquittés par un destinataire ($inv24$).

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
$reg\_sent\_by\_rp$	$l\_sent\_by\_rp$
$reg\_sent\_to\_o\_rp$	$l\_sent\_to\_o\_rp$
$reg\_o\_rp\_r\_pck$	$l\_reg\_o\_rp\_dst$
$reg\_rcvd\_by\_o\_rp$	$l\_rcvd\_by\_o\_rp$
$sent\_to\_t$	$l\_sent\_to\_t$
t_r_pck	l_t_dst
$rcvd\_by\_t$	$l\_rcvd\_by\_t$
$rs\_sent\_by\_o\_rp$	$l\_rs\_sent\_by\_o\_rp$
rs_rp_r	$l\_rs\_rp\_dst$
$rs\_rcvd\_by\_rp$	$l\_rs\_rcvd\_by\_rp$
$rp\_reg\_ack\_rs$	$l\_rp\_reg\_ack\_rs$
$reg\_rs\_o\_rp$	$l\_reg\_rs\_o\_rp$

Les axiomes du contexte C8, ainsi que l'invariant de collage I_{9_3} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M9_P2 en M9_P3, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M9_P3, ainsi que pour justifier la préservation des propriétés Φ_{9_2} lors du raffinement.

La machine M9_P3 contient les raffinements des événements de la machine M9_P2 précédente : un événement de M9_P2 noté X9_P2 est raffiné en une version X9_P3. Nous utilisons une liste Φ_{9_3} de propriétés de vivacité (cf. annexe), caractérisant M9_P3, pour guider le raffinement de M9_P2 en M9_P3, principalement pour la localisation de la troisième phase du protocole ANYCAST RP :

— Nous localisons l'événement RP_SENDING9_P3 sur un routeur RP rp choisi par un routeur DR et sur un message « register » reg qui transite par ce routeur DR ($act1$).

EVENT RP_SENDING9_P3 REFINES RP_SENDING9_P2 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
  THEN
    $\ominus$ $act1$
    $\oplus$ $act1$: $l\_sent\_by\_rp(rp) := l\_sent\_by\_rp(rp) \cup \{reg\}$
  END

— L'événement RP_SENDING_TGTS9_P3 est localisé sur un routeur RP rp et le message « register » reg dont il va envoyer le contenu aux destinataires grt qui l'ont rejoint : le paquet encapsulé dans le message reg est envoyé ($act1$) par le routeur rp aux destinataires grt ($act2$) qui l'ont rejoint ($grd5$), si le message reg ($grd4$) fait partie de l'ensemble des messages ayant transité par le routeur rp (rp a été choisi par un routeur DR) ou s'il a été reçu par le routeur rp (rp est un routeur non-choisi par un routeur DR).

EVENT RP_SENDING_TGTS9_P3 REFINES RP_SENDING_TGTS9_P2 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd4, grd5$
    $\oplus$ $grd4$: $(reg \in l\_sent\_by\_rp(rp)) \vee (reg \in l\_rcvd\_by\_o\_rp(rp))$
    $\oplus$ $grd5$: $grt = join\_rp(rp) \cap group\_target(desenc(reg))$
    ...
  THEN
    $\ominus$ $act1, act2$
    $\oplus$ $act1$: $l\_sent\_to\_t(rp) := l\_sent\_to\_t(rp) \cup \{desenc(reg)\}$
    $\oplus$ $act2$: $l\_t\_dst(desenc(reg)) := l\_t\_dst(desenc(reg)) \cup grt$
    ...
  END

— L'événement RP_T_FWD9_P3 est localisé sur les nœuds x et y dans lesquels un paquet p transite et sur ce paquet p, qui apporte comme information l'identité du destinataire d qu'il doit atteindre ($grd6$).

EVENT RP_T_FWD9_P3 REFINES RP_T_FWD9_P2 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd6$
    $\oplus$ $grd6$: $d \in l\_t\_dst(p)$
    ...
  THEN
    ...
  END

— L'événement TGT_RECEIVING9_P3 est localisé sur un destinataire t d'un paquet p et sur ce paquet p, qui apporte comme information que t fait partie des destinataires qu'il doit atteindre ($grd3$). Le paquet p est alors mis par le destinataire t dans la liste des paquets qu'il a reçus ($act1$).

EVENT TGT_RECEIVING9_P3 REFINES TGT_RECEIVING9_P2 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd3$
    $\oplus$ $grd3$: $t \in l\_t\_dst(p)$
    ...
  THEN
    $\ominus$ $act1$
    $\oplus$ $act1$: $l\_rcvd\_by\_t(t) := l\_rcvd\_by\_t(t) \cup \{p\}$
  END

— L'événement RP_SENDING_OTHER_RPS9_P3 est localisé sur un routeur RP rp choisi par un routeur DR et sur un message « register » reg qu'il diffuse aux autres routeurs RPs. Le message reg est diffusé ($act1$) par le routeur rp aux autres routeurs RPs contenus dans le groupe grp ($act2$), si le paquet qu'il encapsule fait partie des paquets envoyés par le routeur rp à des destinataires ($grd4$), si le message reg a transité par le routeur rp ($grd5$) et tant que le routeur rp n'a pas reçu tous les acquittements pour le message reg des autres routeurs RPs ($grd7$).

EVENT RP_SENDING_OTHER_RPS9_P3 REFINES RP_SENDING_OTHER_RPS9_P2 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd4, grd5, grd7$
$\oplus$ $grd4$: $desenc(reg) \in l\_sent\_to\_t(rp)$
$\oplus$ $grd5$: $reg \in l\_sent\_by\_rp(rp)$
...
$\oplus$ $grd7$: $\exists otrp \cdot otrp \in RPs \wedge ((otrp \mapsto reg) \notin l\_rp\_reg\_ack\_rs(rp))$
...
THEN
$\ominus$ $act1, act2$
$\oplus$ $act1$: $l\_sent\_to\_o\_rp(rp) := l\_sent\_to\_o\_rp(rp) \cup \{reg\}$
$\oplus$ $act2$: $l\_reg\_o\_rp\_dst(reg) := l\_reg\_o\_rp\_dst(reg) \cup grp$
...
END

— L'événement RP_O_RP_FWD9_P3 est localisé sur les nœuds x et y dans lesquels un message « register » reg transite et sur ce message reg, qui apporte comme information l'identité du destinataire d qu'il doit atteindre ($grd6$).

EVENT RP_O_RP_FWD9_P3 REFINES RP_T_FWD9_P2 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd6$
$\oplus$ $grd6$: $d \in l\_reg\_o\_rp\_dst(reg)$
...
THEN
...
END

— L'événement O_RP_RECEIVING9_P3 est localisé sur un routeur RP rp non-choisi par un routeur DR et un message « register » reg qu'il reçoit. Le message reg donne comme information l'identité du destinataire intermédiaire rp qu'il doit atteindre ($grd3$). Lorsque le message reg arrive au niveau du routeur rp dans le réseau, ce dernier l'ajoute à l'ensemble des messages « register » qu'il a reçus ($act1$).

EVENT O_RP_RECEIVING9_P3 REFINES O_RP_RECEIVING9_P2 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd3$
$\oplus$ $grd3$: $rp \in l\_reg\_o\_rp\_dst(reg)$
...
THEN
$\ominus$ $act1$
$\oplus$ $act1$: $l\_rcvd\_by\_o\_rp(rp) := l\_rcvd\_by\_o\_rp(rp) \cup \{reg\}$
END

— Nous modifions par raffinement les gardes $grd4$ et $grd5$ de l'événement RECEIVING9_P2 : le raffinement RECEIVING9_P3 prend maintenant en compte les nouvelles variables $l\_rcvd\_by\_t$ et $l\_rcvd\_by\_o\_rp$. Un paquet p est considéré comme étant reçu par tous ses destinataires gt, s'il a été reçu individuellement par tous les membres t de gt ($grd4$) et si le message « register » qui l'encapsule a été reçu par tous les routeurs RPs ($grd5 : enc(p) \in l\_rcvd\_by\_o\_rp(rp)$ si rp est un routeur RP non choisi par un routeur DR et $grd5 : enc(p) \in l\_rcvd\_by\_rp(rp)$ si rp est un routeur RP choisi par un routeur DR).

EVENT RECEIVING9_P3 REFINES RECEIVING9_P2 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd4, grd5$
$\oplus$ $grd4$: $\{t|t \in TARGETS \wedge p \in l\_rcvd\_by\_t(t)\} = gt$
$\oplus$ $grd5$: $\{rp|rp \in RPs \wedge enc(p) \in (l\_rcvd\_by\_o\_rp(rp) \cup l\_rcvd\_by\_rp(rp))\} = RPs$
...
THEN
...
END

Nous modifions aussi par raffinement les autres événements impactés par l'ajout des nouvelles variables :

— L'événement O_RP_REG_STOP_SENDING9_P3 est localisé sur un routeur RP rp non-choisi par un routeur DR et sur le message « register » reg, reçu d'un routeur RP choisi rp_chs ($grd7$, $grd8$), en réponse auquel il envoie un message « register-stop » stp ($act1$, $act3$) à destination du routeur RP choisi rp_chs ($act2$).

EVENT O_RP_REG_STOP_SENDING9_P3 REFINES O_RP_REG_STOP_SENDING9_P2 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd7, grd8, grd9$
$\oplus$ $grd7$: $reg \in rcvd\_by\_o\_rp(rp)$
$\oplus$ $grd8$: $reg \in l\_sent\_by\_rp(rp\_chs)$
$\oplus$ $grd9$: $desen(reg) \in l\_sent\_to\_t(rp)$
...
THEN
$\ominus$ $act1, act2, act3$
$\oplus$ $act1$: $l\_rs\_sent\_by\_o\_rp(rp) := l\_rs\_sent\_by\_o\_rp(rp) \cup \{stp \mapsto reg\}$
$\oplus$ $act2$: $l\_rs\_rp\_dst(rs) := l\_rs\_rp\_dst(rs) \cup grp$
$\oplus$ $act3$: $l\_reg\_rs\_o\_rp(rp) := l\_reg\_rs\_o\_rp(rp) \cup \{reg\}$
...
END

— L'événement O_RP_REG_STOP_FWD9_P3 est localisé sur les nœuds x et y dans lesquels un message « register-stop », envoyé en réponse à un message « register », transite et sur ce message « register-stop », qui apporte comme informations l'identité du destinataire d qu'il doit atteindre ($grd6$), ainsi que celle du message « register » qu'il acquitte.

EVENT O_RP_REG_STOP_FWD9_P3 REFINES O_RP_REG_STOP_FWD9_P2 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd6$
$\oplus$ $grd6$: $d \in l\_rs\_rp\_dst(rs)$
...
THEN
END

— L'événement RP_REG_STOP_RECEIVING9_P3 est localisé sur un routeur RP rp_chs choisi par un routeur DR, sur un message « register-stop » stp reçu d'un routeur RP rp non-choisi ($grd5$), en réponse à un message « register » reg diffusé. Le routeur rp_chs est le destinataire du message « register-stop » stp ($grd6$). Lorsque le message stp arrive au niveau du routeur rp_chs dans le réseau, le routeur rp_chs ajoute le routeur rp dans la liste des routeurs desquels il a reçu un acquittement pour le message reg ($act1$) et il ajoute le message stp dans la liste des messages « register-stop » reçus en réponse au message « register » reg ($act2$).

EVENT RP_REG_STOP_RECEIVING9_P3 REFINES RP_REG_STOP_RECEIVING9_P2 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd5, grd6$
$\oplus$ $grd5$: $stp \mapsto reg \in l\_rs\_sent\_by\_o\_rp(rp)$
$\oplus$ $grd6$: $rp\_chs \in l\_rs\_rp\_dst(stp \mapsto reg)$
...
THEN
$\ominus$ $act1, act2$
$\oplus$ $act1$: $l\_rp\_reg\_ack\_rs(rp\_chs) := l\_rp\_reg\_ack\_rs(rp\_chs) \cup \{rp \mapsto reg\}$
$\oplus$ $act2$: $l\_rs\_rcvd\_by\_rp(rp\_chs) := l\_rs\_rcvd\_by\_rp(rp\_chs) \cup \{stp \mapsto reg\}$
...
END

— L'événement RP_CREATING_STATE_SG9_P3 est localisé sur un routeur RP rp et un message « register » reg reçu par ce routeur rp. Un état « (S,G) » est créé par le routeur rp, par rapport au message reg, si ce dernier a transité dans le routeur RP rp_chs choisi par les routeurs DRs ($grd4$) et s'il a ($grd5$) soit été envoyé par le routeur rp aux autres routeurs RPs (cas où le routeur rp est le routeur choisi rp_chs), ou s'il est un message pour lequel le routeur rp a envoyé un message « register-stop » (cas où le routeur rp est différent du routeur choisi rp_chs).

EVENT RP_CREATING_STATE_SG9_P3 REFINES RP_CREATING_STATE_SG9_P2 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd4, grd5$
$\oplus$ $grd4$: $reg \in l\_sent\_by\_rp(rp\_chs)$
$\oplus$ $grd5$: $\vee(rp = rp\_chs \wedge reg \in l\_sent\_to\_o\_rp(rp\_chs))$
$\vee(rp \neq rp\_chs \wedge reg \in l\_reg\_rs\_o\_rp(rp))$
...
THEN
...
END

Nous avons localisé dans ce raffinement la troisième phase de l'algorithme ANYCAST RP, qui comprend les communications entre les routeurs points de rendez-vous (RPs) et les destinataires finaux des paquets. Le diagramme 5.15 suivant résume ce raffinement de M9_P2 en M9_P3 :

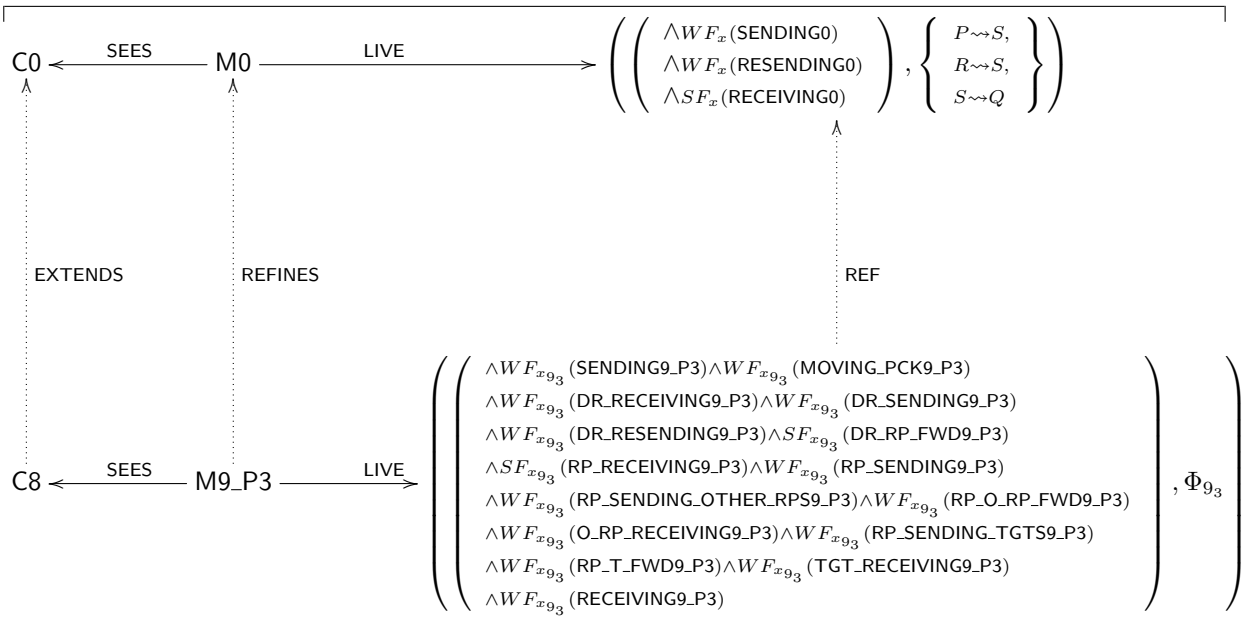


FIGURE 5.15 – Onzième raffinement du protocole ANYCAST RP

Nous présentons dans la sous-section suivante l'introduction des tables de routage dans le protocole ANYCAST RP.

5.3.2.13 Dernier modèle : introduction du routage

Cette sous-section introduit les tables de routage dans nos modèles du protocole ANYCAST RP. Nous avons choisi pour de représenter la table de routage d'un nœud n du système réparti support du protocole ANYCAST RP de la manière suivante :

— Le nœud n maintient dans sa table de routage la liste des nœuds atteignables à partir du nœud n : il s'agit de lui-même et des nœuds auxquels il est relié dans le graphe.

— Le nœud n maintient aussi dans sa table de routage la liste des voisins par lesquels il faut passer pour arriver aux nœuds atteignables à partir du nœud n.

Nous commençons par définir un contexte C9 qui contient une constante *init_routing_table* définissant les tables de routage pour chaque nœud n du graphe original g ($axm1$, $axm2$). La constante *init_routing_table* est définie en tenant compte de la représentation des tables de routage décrite dans le paragraphe précédent.

CONTEXT C9 EXTENDS C8
  CONSTANTS
    $init\_routing\_table$
  AXIOMS
    $axm1$: $init\_routing\_table \in NODES \rightarrow (NODES \leftrightarrow NODES)$
    $axm2$: $\forall n \in NODES \Rightarrow init\_routing\_table(n) = (g[\{n\}] \cup \{n\}) \lhd closure(g)$
  END

Ce contexte C9 sera utilisé par le raffinement M9_R de la machine M9_P3 précédente.

Deux nouvelles variables sont introduites dans la machine M9_R.

INITIALISATION $\widehat{=}$
  BEGIN
    ...
    $\oplus$ $routing\_table := init\_routing\_table$
    $\oplus$ $stabilize := FALSE$
  END

L'ensemble des variables de ce raffinement est noté x_{9_4} et est composé des variables x_{9_3} du raffinement précédent et de ces nouvelles variables.

L'invariant noté $I_{9_4}(x_{9_4})$, caractérisant les variables x_{9_4} est défini comme suit :

— Des propriétés permettent de typer les nouvelles variables :

$inv1$: $routing\_table \in NODES \rightarrow (NODES \leftrightarrow NODES)$
$inv2$: $stabilize \in BOOL$

  — *routing_table* modélise pour chaque nœud sa table de routage. Elle est initialisée à l'aide de la constante *init_routing_table*.
  — *stabilize* est un booléen. Cette variable permet de stabiliser les tables de routage et le réseau *graph*. Si la valeur de *stabilise* est $TRUE$, alors la modification des tables de routage et du réseau *graph* n'est pas autorisée. Dans le cas contraire, les tables de routage peuvent être mises à jour et le réseau peut changer. La valeur initiale de *stabilize* est $FALSE$: nous supposons qu'à l'état initial, les tables de routage et le réseau *graph* peuvent changer.

— Des propriétés de sûreté contraignent ces variables :

$$inv3 : stabilize = TRUE \Rightarrow \left(\begin{array}{l} \forall n \cdot n \in NODES \\ \Rightarrow \\ closure(graph)[\{n\}] = ran(routing\_table(n)) \end{array} \right)$$

$$inv4 : stabilize = TRUE \Rightarrow \left(\begin{array}{l} \forall n \cdot n \in NODES \\ \Rightarrow \\ routing\_table(n) = ((\{n\} \cup graph[\{n\}]) \lhd closure(graph)) \end{array} \right)$$

- La propriété *inv*3 exprime que si la valeur de *stabilize* est vraie, alors pour tout nœud n, les nœuds atteignables à partir du nœud n dans le graphe *graph* sont marqués comme atteignables dans la table de routage de n.
- La propriété *inv*4 exprime que si la valeur de *stabilize* est vraie, alors pour tout nœud n, le contenu de sa table de routage est conforme au graphe *graph* :
  - La table de routage du nœud n contient tous les nœuds réellement atteignables à partir du nœud n dans le réseau *graph*.
  - La table de routage du nœud n contient les voisins par lesquels il faut passer dans le graphe pour arriver aux nœuds atteignables à partir de n.

Nous trouvons dans la machine M9_R les événements suivants :

- Les raffinements des événements de la machine M9_P3 précédente : un événement de M9_P3 noté X9_P3 est raffiné en une version X9_R.
- De nouveaux événements relatifs aux tables de routage : UPDATE_TABLE9_R qui modélise la mise à jour des tables de routage des nœuds entre un routeur émetteur et un routeur récepteur ; STABILIZE9_R qui permet de modéliser un moment durant lequel le système réparti mobile reste stable ; UNSTABILIZE9_R qui met fin à la stabilité du système réparti.

Nous nous intéressons maintenant aux propriétés de vivacité satisfaites par la machine M9_R. Avant de présenter ces propriétés, nous commençons par définir :

- Nous posons x comme la position courante d'un message « register » $enc(p)$ encapsulant un paquet p dans le réseau, sur la portion entre les routeurs DRs et RPs. Soit orp, un routeur point de rendez-vous (non choisi par un routeur DR) auquel $enc(p)$ est destiné. Le routeur orp est atteignable à partir du nœud x, selon la table de routage de ce dernier ($x \mapsto orp \in routing\_table(x)$). Nous notons $dist_9(x, orp, enc(p))$ la distance entre la position courante x du message $enc(p)$ et le routeur orp, sur un chemin donné, avec $dist_9(x, orp, enc(p)) \in \mathbb{N}$. Il s'agit du nombre de nœuds entre la position courante x et le routeur orp, sur le chemin donné. Cette distance décroît au fur et à mesure que le message $enc(p)$ progresse dans le réseau, vers le routeur orp, jusqu'à ce qu'elle soit égale à 0, c'est-à-dire lorsque le message arrive au niveau du routeur destination orp. Nous posons $OD_9(x, orp, enc(p), o) \ \widehat{=}\ (dist_9(x, orp, enc(p)) = o)$, où $o \in \mathbb{N}$. Nous précisons que quelque soit la valeur de o, le message $enc(p)$ n'est encore reçu par le routeur orp : $orp \mapsto enc(p) \notin reg\_rcvd\_by\_o\_rp$.
- $dist_9(x, t, p)$ et $TD_9(x, t, p, i)$ Nous posons x comme la position courante d'un paquet p dans le réseau, sur la portion entre les routeurs RPs et les destinataires finaux du paquet p. Soit t, un destinataire final du paquet p. Le destinataire t est atteignable à partir du nœud x, selon la table de routage de ce dernier ($x \mapsto t \in routing\_table(x)$). Nous notons $dist_9(x, t, p)$ la distance entre la position courante x du paquet p et le destinataire t, sur un chemin donné, avec $dist_9(x, t, p) \in \mathbb{N}$. Il s'agit du nombre de nœuds entre la position courante x et le destinataire t, sur le chemin donné. Cette distance décroît au fur et à mesure que le paquet p progresse dans le réseau, vers le destinataire t, jusqu'à ce qu'elle soit égale à 0, c'est-à-dire lorsque le paquet arrive au niveau de la destination t. Nous posons $TD_9(x, t, p, i) \ \widehat{=}\ (dist_9(x, t, p) = i)$, où $i \in \mathbb{N}$. Nous précisons que quelque soit la valeur de i, le paquet p n'est encore reçu par le destinataire t : $t \mapsto p \notin rcvd\_by\_t$.
- $A_9(p) \ \widehat{=}\ \left\{ orp \;\middle|\; \begin{array}{l} \wedge orp \in RPs \\ \wedge orp \neq rp \\ \wedge dist_9(x, orp, enc(p)) > 0 \\ \wedge x \mapsto orp \in routing\_table(x) \end{array} \right\}$

  La propriété $A_9(p)$ exprime qu'un message « register » $enc(p)$ encapsulant un paquet p se trouve au niveau d'un nœud x, distant d'un routeur point de rendez-vous non-choisi orp, accessible à partir de x selon sa table de routage.
- $B_9(p) \ \widehat{=}\ \left\{ t \;\middle|\; \begin{array}{l} \wedge t \in TARGETS \\ \wedge t \in group\_target(p) \\ \wedge dist_9(x, t, p) > 0 \\ \wedge x \mapsto t \in routing\_table(x) \end{array} \right\}$

  La propriété $B_9(p)$ exprime qu'un paquet p se trouve au niveau d'un nœud x, distant d'un destinataire t de p, accessible à partir de x selon sa table de routage.
- Nous posons :

$$R_{90}(p) \;\widehat{=}\; \left\{ orp \;\middle|\; \begin{array}{l} \wedge orp \in RPs \setminus \{rp\} \\ \wedge enc(p) \notin l\_rcvd\_by\_o\_rp(orp) \\ \wedge OD_9(x, orp, enc(p), 0) \end{array} \right\}$$

$R_{90}(p)$ est l'ensemble contenant chaque routeur orp non choisi (différent de rp), n'ayant pas encore reçu le message $enc(p)$ qui se trouve à son niveau dans le réseau. Nous posons $R_{90}(l, p) \;\widehat{=}\; (card(R_{90}(p)) = l)$.

— Nous posons :

$$S_{90}(p) \;\widehat{=}\; \left\{ orp \;\middle|\; \begin{array}{l} \wedge orp \in RPs \setminus \{rp\} \\ \wedge enc(p) \in l\_rcvd\_by\_o\_rp(orp) \\ \wedge OD_9(x, orp, enc(p), 0) \\ \wedge p \notin l\_sent\_to\_t(orp) \\ \wedge orp \mapsto p \notin rp\_t\_store \end{array} \right\}$$

$S_{90}(p)$ est l'ensemble contenant chaque routeur orp non choisi (différents de rp), ayant reçu le message $enc(p)$, mais ne l'ayant pas encore envoyé et déposé dans le réseau à destination du groupe de destinataires finaux du paquet p. Nous posons $S_{90}(n, p) \;\widehat{=}\; (card(S_{90}(p)) = n)$.

— Nous posons :

$$T_{91}(p) \;\widehat{=}\; \left\{ t \;\middle|\; \begin{array}{l} \wedge t \in group\_target(p) \\ \wedge TD_9(x, t, p, 0) \\ \wedge p \notin l\_rcvd\_by\_t(t) \end{array} \right\}$$

$T_{91}(p)$ est l'ensemble contenant chaque destinataire t, membre du groupe $group\_target(p)$, n'ayant pas encore reçu le paquet p, qui se trouve à son niveau dans le réseau. Nous posons $T_{91}(k, p) \;\widehat{=}\; (card(T_{91}(p)) = k)$.

Ces propriétés nous permettent de définir les diagrammes 5.16 et 5.17 décrivant visuellement la liste Φ_{9_4} des propriétés de vivacité satisfaites par M9_R.

Nous modifions ensuite par raffinement les événements d'envoi, de transfert de paquets pour prendre en compte les tables de routage :

— Pour les événements $< \mathsf{X} > \in \{\mathsf{SENDING}, \mathsf{MOVING\_PCK}\}$, nous nous assurons que le routeur DR dr qui va recevoir le paquet p diffusé par la source s est atteignable à partir de cette dernière, c'est-à-dire que la table de routage de la source s indique qu'un chemin existe de s au routeur dr ($grdR$).

```
EVENT < X >9_R
REFINES < X >9_P3 ≙
  ANY
    ...
  WHERE
    ...
    ⊕ grdR : s ↦ dr ∈ routing_table(s)
  THEN
    ...
  END
```

— Pour DR_SENDING9_R, RP_REG_STOP_SENDING9_R et O_RP_REG_STOP_SENDING9_R, nous rajoutons une garde $grdR$ qui exprime que l'envoi d'un message « register » ou « register-stop » se fait seulement si la table de routage de l'expéditeur indique qu'un chemin existe de l'expéditeur au destinataire.

```
EVENT DR_SENDING9_R
REFINES DR_SENDING9_P3 ≙
  ANY
    ...
  WHERE
    ...
    ⊕ grdR : dr ↦ rp ∈ routing_table(dr)
  THEN
    ...
  END
```

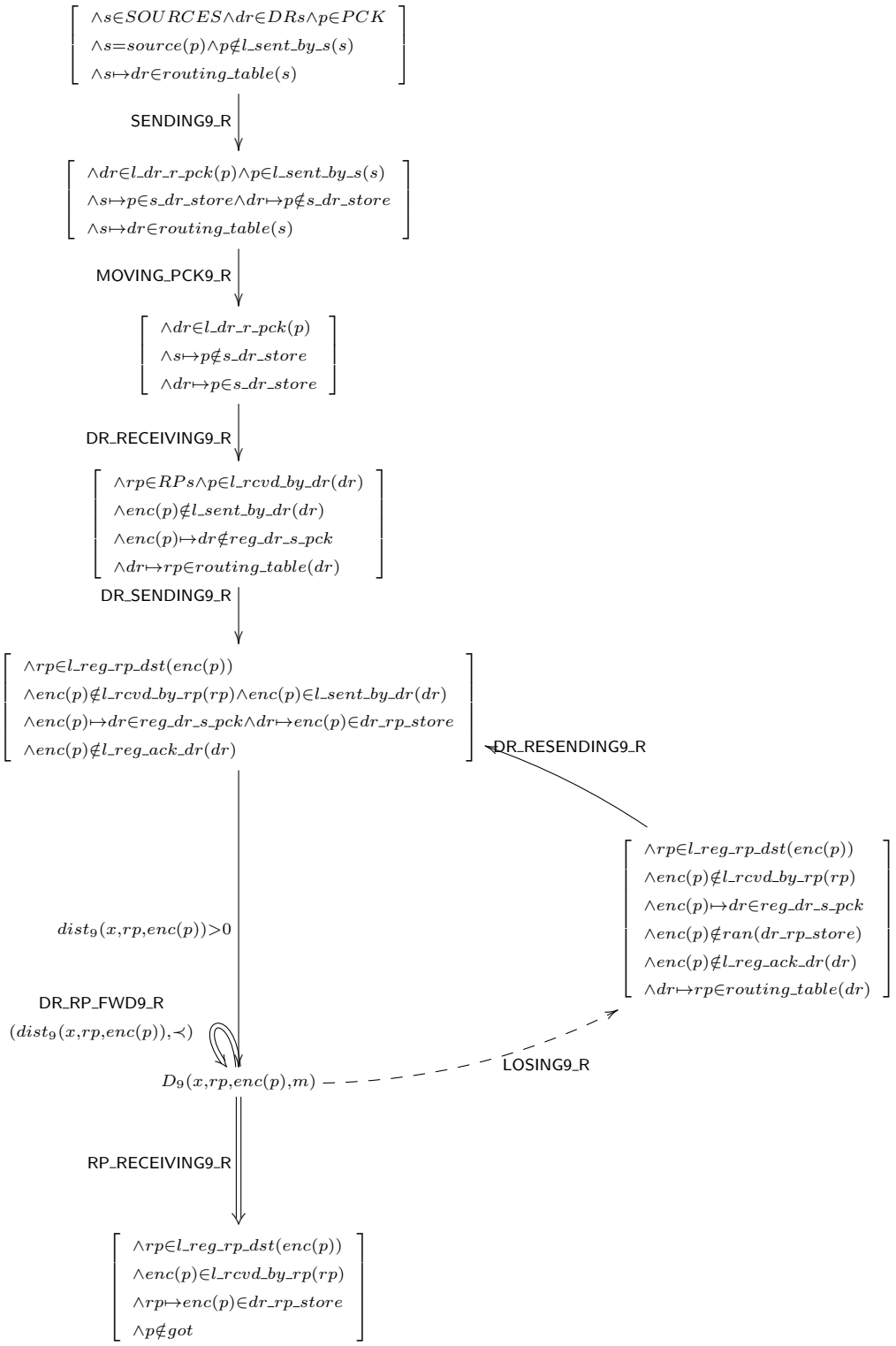


FIGURE 5.16 – Diagramme d'assertions pour M9_R (Partie 1)

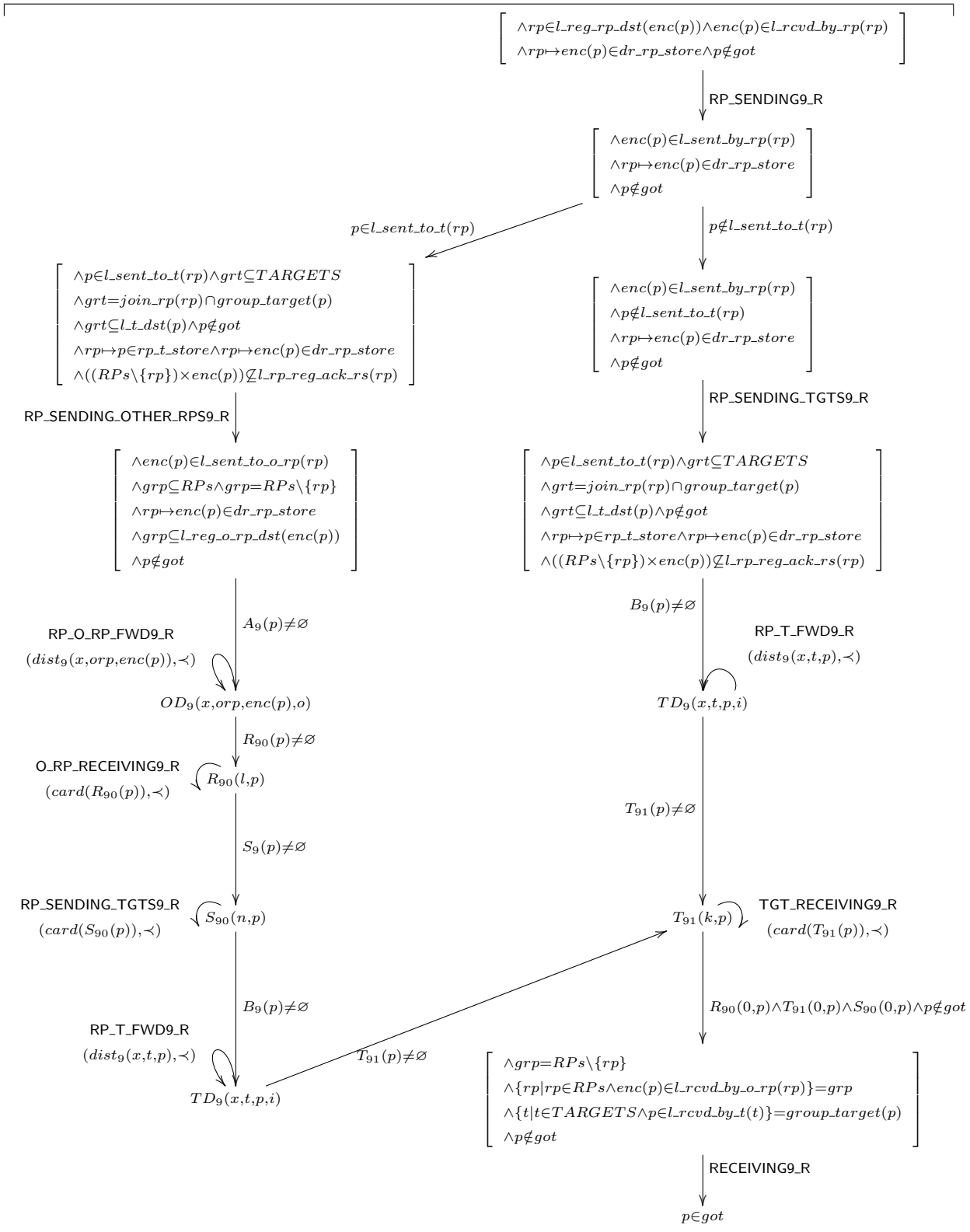


FIGURE 5.17 – Diagramme d'assertions pour M9_R (Partie 2)

```
EVENT RP_REG_STOP_SENDING9_R
REFINES RP_REG_STOP_SENDING9_P3 ≙
  ANY
    ...
  WHERE
    ...
    ⊕ grdR : rp ↦ dr ∈ routing_table(rp)
  THEN
    ...
  END
```

```
EVENT O_RP_REG_STOP_SENDING9_R
REFINES O_RP_REG_STOP_SENDING9_P3 ≙
  ANY
    ...
  WHERE
    ...
    ⊕ grdR : rp ↦ rp_chs ∈ routing_table(rp)
  THEN
    ...
  END
```

— Pour les événements $< \mathsf{Y} >$ de transfert de messages ou de paquets :

$$< \mathsf{Y} > \in \left\{ \begin{array}{l} \mathsf{DR\_RP\_FWD}, \mathsf{RP\_O\_RP\_FWD}, \mathsf{RP\_T\_FWD}, \\ \mathsf{RP\_DR\_STOP\_FWD}, \mathsf{O\_RP\_RP\_REG\_STOP\_FWD} \end{array} \right\},$$

nous rajoutons le fait que le tranfert d'un paquet ou d'un message d'un nœud x à un nœud voisin y ne se fait que si la table de routage du nœud x indique qu'un chemin vers la destination d, du paquet ou du message, existe à partir du nœud y ($grdR$).

```
EVENT < Y >9_R
REFINES < Y >9_P3 ≙
  ANY
    ...
  WHERE
    ...
    ⊕ grdR : y ↦ d ∈ routing_table(x)
  THEN
    ...
  END
```

Nous prenons maintenant en compte la stabilité du système. Nous raffinons les événements $< \mathsf{Z} > \in \{\mathsf{ADD\_LINK}, \mathsf{REMOVE\_LINK}\}$ de modification du graphe courant $graph$, de telle manière qu'un ajout ou un retrait de liens entre deux nœuds x et y n'est possible que si le système n'est pas considéré comme étant stable, c'est-à-dire $stabilize = FALSE$ ($grdR$).

```
EVENT < Z >9_R
REFINES < Z >9_P3 ≙
  ANY
    ...
  WHERE
    ...
    ⊕ grdR : stabilize = FALSE
  THEN
    ...
  END
```

De la même manière, nous n'autorisons la mise à jour des tables de routage que si le système n'est pas stable ($stabilize = FALSE$). L'événement UPDATE_TABLE9_R est observé si un chemin entre un nœud source s et un nœud destination t n'existe pas dans la table de routage du nœud s ($grd3$) et évidemment, si le système n'est pas stable ($grd5$) : dans ce cas, nous prenons un sous-ensemble $UPDATE$ de nœuds, constitué des nœuds atteignables à partir du nœud s ($grd4$) et nous mettons à jour les tables de routage de ces nœuds ($act1$).

EVENT UPDATE_TABLE9_R $\widehat{=}$
ANY
$s, d, UPDATE$
WHERE
$grd1$: $s \in NODES$
$grd2$: $d \in NODES$
$grd3$: $s \mapsto d \notin routing\_table(s)$
$grd4$: $UPDATE \subseteq \{x | x \in NODES \wedge s \mapsto x \in closure(graph)\}$
$grd5$: $stabilize = FALSE$
THEN
$act1$: $routing\_table := routing\_table \Leftarrow (\lambda n \cdot n \in UPDATE | ((graph[\{n\}] \cup \{n\}) \lhd closure(graph)))$
END

Nous introduisons aussi dans ce modèle deux événements STABILIZE9_R et UNSTABILIZE9_R, relatifs à la stabilité du système réparti représenté par le graphe *graph*.

— L'événement STABILIZE9_R nous permet de modéliser la stabilité du système :
  — Les deux premières gardes ($grd1$ et $grd2$) de l'événement STABILIZE9_R expriment que chaque noeud x du graphe a dans sa table de routage locale une vue correcte de tous les nœuds u qu'il peut atteindre dans le graphe.
  — La troisième garde ($grd3$) exprime que tout noeud x, t, u, si t est différent de x, et qu'un chemin de t à u existe dans la table de routage de x, alors t est un voisin de x et un chemin existe réellement de t à u dans le graphe, et vice-versa.
  En résumé, si dans le réseau *graph*, les contenus de toutes les tables de routages sont conformes au graphe *graph*, c'est-à-dire si tous les nœuds du graphe ont une vue locale correcte du réseau, alors nous pouvons considérer que le système est stable ($act1$).

EVENT STABILIZE9_R $\widehat{=}$
WHEN
$grd1$: $\forall x, u \cdot u \in ran(routing\_table(x)) \Leftrightarrow x \mapsto u \in closure(graph)$
$grd2$: $\forall x, u \cdot x \mapsto u \in routing\_table(x) \Leftrightarrow x \mapsto u \in closure(graph)$
$grd3$: $\forall x, t, u \cdot ((t \neq x \wedge t \mapsto u \in routing\_table(x)) \Leftrightarrow (x \mapsto t \in graph \wedge t \mapsto u \in closure(graph)))$
THEN
$act1$: $stabilize := TRUE$
END

— L'événement UNSTABILIZE9_R peut être observé uniquement lorsque le système réparti représenté par le graphe *graph* est stable ($grd1$) : en mettant la valeur de la variable *stabilize* à $FALSE$ ($act1$), nous autorisons les modifications du graphe *graph* ainsi que des tables de routage.

EVENT UNSTABILIZE9_R $\widehat{=}$
WHEN
$grd1$: $stabilize = TRUE$
THEN
$act1$: $stabilize := FALSE$
END

Le diagramme 5.18 suivant résume cette dernière étape de raffinement de M9_P3 en M9_R :

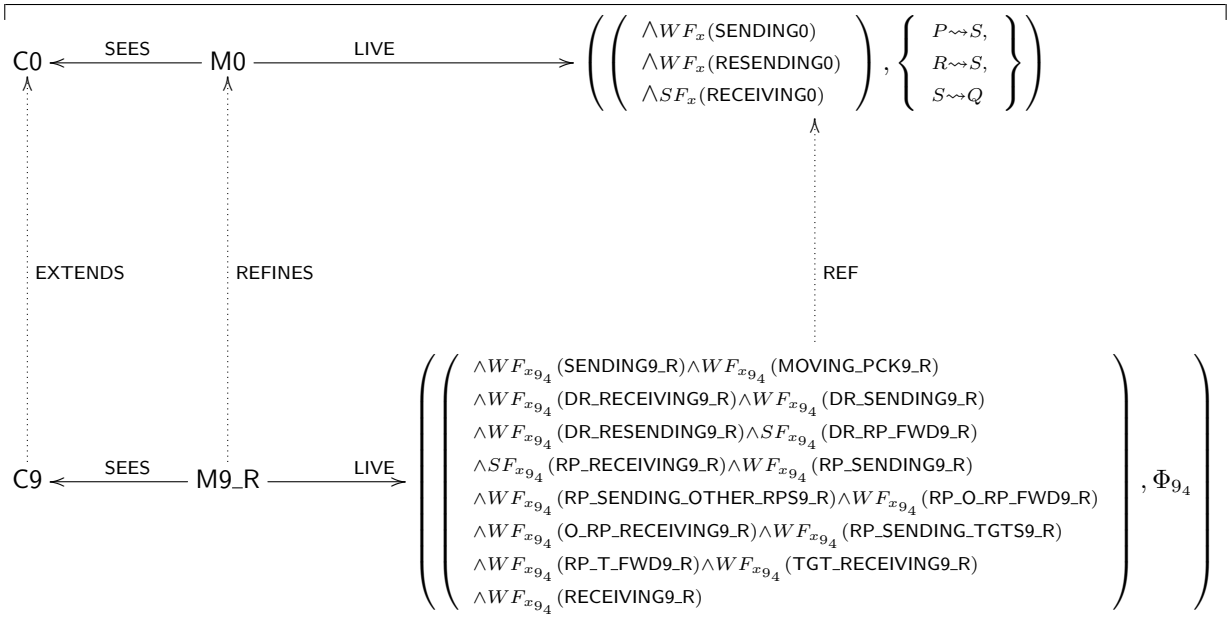


FIGURE 5.18 – Dernier raffinement du protocole ANYCAST RP

Ce raffinement M9_R nous a permis d'introduire les tables de routage dans l'algorithme ANYCAST RP. Les techniques de routage et la constitution des tables de routage des nœuds n'étant pas décrites explicitement dans [17, 18], nous avons préféré modélisé les tables de routage de manière simple, en considérant les domaines/nœuds pouvant être atteints à partir d'un routeur et les chemins menant à ces domaines/nœuds. Ce raffinement nous a aussi permis de considérer un état stable pour un système réparti mobile [14] : si un système réparti est stable, alors tous les nœuds du système réparti ont une vue locale correcte du système réparti global.

5.4 Conclusion

Nous avons présenté dans ce chapitre la modélisation par raffinement du protocole de communication ANYCAST RP. Nous nous sommes intéressés à cet algorithme, parce qu'il s'agit d'un protocole réel, industriel mis au point par CISCO Systems [17, 18], et pouvant être utilisé par des applications distribuées critiques, et requérant de la robustesse, fiabilité [17]. Nous nous sommes focalisés sur l'analyse de cet algorithme de routage dans un système distribué mobile, c'est-à-dire, un système changeant, que des nœuds ou processus peuvent rejoindre ou quitter à tout moment.

L'objectif principal du protocole ANYCAST RP est une communication de « un à plusieurs », aussi appelée *multicast* : il s'agit pour une source de diffuser un paquet vers un groupe de destinataires. Nous avons appliqué le paradigme *service-as-event* [4, 13], pour modéliser ce service de routage et de diffusion proposé par le protocole ANYCAST RP :

— Une propriété de vivacité a permis de modéliser de manière abstraite ce service, par des événements simple d'envoi de paquets par une source et de réception des messages envoyés par les destinataires finaux.

— Les règles d'inférences relatives aux propriétés de vivacités, nous ont ensuite permis détailler la propriété de vivacité abstraite et d'en dériver un raffinement du modèle abstrait identifiant trois phases algorithmiques séquentielles, pouvant être considérées comme des sous-services de diffusion de paquets :

1. Une première diffusion d'un paquet par sa source à un premier routeur intermédiaire appelé Routeur Désigné (DR).

2. Une seconde diffusion du paquet, cette fois-ci du Routeur Désigné (DR) à un second intermédiaire appelé Point de Rendez-vous (RP).

3. Puis finalement, une troisième diffusion du paquet du routeur Point de Rendez-vous (RP) aux destinataires finaux.

En outre, l'utilisation des propriétés de vivacité et des règles d'inférence qui y sont reliées, à travers le paradigme *service-as-event* nous a permis de détailler par raffinement la troisième phase, qui se décompose en une diffusion du paquet par le routeur Point de Rendez-vous choisi par le routeur DR aux autres RPs et la diffusion par chaque routeur RP du message qu'il a reçu aux destinataires qui l'ont rejoint.

L'identification des phases de l'algorithme, qui se sont avérées répétitives et similaires nous a poussés à réfléchir à la notion de *patron de conception en* Event-B [13, 10, 1, 16] et à élaborer le patron de conception vu dans la section précédente et présentant une solution (un développement formel Event-B) au routage et à la diffusion de paquets/messages d'une source aux destinataires dans un système distribué.

Cette étude de cas nous a aussi permis de modéliser un algorithme réparti, pouvant fonctionner dans un environnement distribué mobile, sous réserve de certaines hypothèses :

— Les pertes de messages/paquets peuvent avoir lieu seulement sur la partie du réseau entre les routeurs DRs et les routeurs RPs. C'est seulement sur cette section du réseau qu'un mécanisme de récupération des messages perdus/paquets, par renvoi par les routeurs DRs, existe [18].
— Le réseau est mobile, cependant, lors des modifications de ce dernier, nous veillons à maintenir toujours au moins une connexion :
  — Directe entre les sources et les Routeurs Désignés (DRs).
  — Entre les Routeurs Désignés (DRs) et les Points de Rendez-vous (RPs).
  — Entre les Points de Rendez-vous (RPs) et les destinataires des paquets diffusés.

Ces hypothèses nous permettent de nous assurer qu'un paquet diffusé par sa source à un groupe de destinataires sera un jour reçu par ces derniers.

Des travaux existent sur la modélisation en Event-B d'algorithmes de routage permettant la découverte de topologie de systèmes répartis dynamiques, notamment ceux de Méry et Singh [14] et Hoang et al [11]. Ces travaux introduisent la notion de stabilité d'un système réparti dynamique : il s'agit d'un état dans lequel chaque nœud du système distribué a une vue locale correcte de l'état global du système distribué. Nous étendons cette notion en y ajoutant le fait que le système peut rester stable durant un temps fini, au cours duquel ni les informations de routage locales des nœuds, ni la topologie du système distribué support ne sont modifiées.

Le tableau 5.1 suivant présente la difficulté des étapes de modélisation en donnant les statistiques des obligations de preuves engendrées lors de la modélisation du protocole de routage Anycast RP et prouvées soit automatiquement, soit interactivement. Nous ne présentons ici que les statistiques obtenues en utilisant uniquement les prouveurs de l'Atelier B (nous n'utilisons pas de solveurs SMT, ou d'autres prouveurs).

Modèles	Total	Automatiques		Interactives	
C0	0	0	100%	0	0%
C1	1	1	100%	0	0%
C2	0	0	100%	0	0%
C3	0	0	100%	0	0%
C4	0	0	100%	0	0%
C5	3	3	100%	0	0%
C6	3	3	100%	0	0%
C7	11	11	100%	0	0%
C8	15	15	100%	0	0%
C9	2	2	100%	0	0%
M0	9	9	100%	0	0%
M1	14	14	100%	0	0%
M2	16	16	100%	0	0%
M3	100	76	76%	24	24%
M4	48	33	68.75%	15	31.25%
M5	149	110	73.83%	39	26.17%
M6	93	50	53.76%	43	46.24%
M7	14	5	35.72%	9	64.28%
M8	53	33	62.26%	20	37.74%
M9_P1	34	18	82.35%	16	47.06%
M9_P2	106	60	72.64%	46	43.40%
M9_P3	140	88	69.29%	52	37.14%
M9_R	40	35	87.5%	5	12.5%
Total	851	582	68.39%	269	31.61%

TABLE 5.1 – Protocole ANYCAST RP : Bilan des POs

Nous remarquons que le nombre des obligations de preuves manuelles commence à devenir assez élevé à partir de la machine M3 : ces obligations sont dues, dans un premier temps, aux faits que nous identifions les routeurs impliqués dans le protocole ; que nous définissons des propriétés sur ces routeurs (unicité d'une source, d'un DR, d'un RP récepteurs et émetteurs d'un paquet, etc) et que nous exprimons à l'aide d'invariants la séquentialité des trois phases du protocole. Les obligations de preuves manuelles introduites par les machines de M4 à M7 concernent des propriétés relatives aux particularités du protocole ANYCAST RP : encapsulation des paquets reçus des sources par le DR ; la troisième phase entre le RP et le destinataires caractérisée par plusieurs phases de relais de messages et de désencapsulation de ces derniers ; l'envoi de messages d'acquittement des paquets reçus et l'entrée des routeurs dans des états spéciaux lors de la réception de paquets. Les obligations de preuves engendrées pour M8, M9_P1, M9_P2 et M9_P3 sont dues notamment à l'introduction du système réparti et à la localisation de chaque phase du protocole (transformation par raffinement de données de relations en fonctions).

Ce protocole nous a permis d'appliquer le paradigme de service-as-event [4] : nous avons utilisé les propriétés de vivacité pour caractériser le protocole à l'aide des services qu'il fournit et appliqué les règles d'inférence pour identifier et développer les différentes phases du protocole Anycast RP. Les propriétés de vivacité de type *leadsto*, les règles d'inférence relatives à ces propriétés, nous ont permis de guider nos différentes étapes de raffinement, à partir d'un niveau très abstrait n'exprimant seulement que la transmission de messages/paquets dans un réseau, jusqu'à un modèle concret local, faisant apparaître les nœuds/processus participant au protocole, les différentes étapes de ce dernier, ainsi que des éléments de routages (tables de routages, mises à jour de ces dernières, etc).

Nous avons étudié ici un algorithme complexe, mais pouvant être décomposé en phases simples d'envoi et de réception de messages/paquets. La similarité des phases du protocole Anycast RP (envoi d'un message/paquet par une source, réception par un ou plusieurs destinataires), la similarité de leurs développements formels, des propriétés de sûreté et de vivacité les caractérisant, nous ont aussi amenés à nous intéresser aux patrons de conception. Nous avons ainsi pu mettre au point un patron de conception [1, 10] pour le routage, c'est-à-dire un développement formel générique, que nous instancierons notamment pour d'autres études de cas dans cette thèse. Cet algorithme nous a ainsi permis de poser des bases pour nos autres études de cas, à travers notamment la gestion des hypothèses à poser pour assurer le fonctionnement correct des algorithmes, la modélisation de réseaux mobiles et la notion de stabilisation. Ces notions ont été réutilisées notamment pour l'étude des *« réseaux-sur-puces »* (NoCs) [3] et des systèmes auto-stabilisants [5].

Bibliographie

[1] J.-R. Abrial and T. S. Hoang. Using design patterns in formal methods : An event-b approach. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigün, editors, *ICTAC*, volume 5160 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2008.

[2] B. Adams, J. Nicholas, and W. Siadiac. Bidirectional PIM (BIDIR PIM). `http://tools.ietf.org/html/rfc3973`.

[3] M. B. Andriamiarina, H. Daoud, M. Belarbi, D. Méry, and C. Tanougast. Formal Verification of Fault Tolerant NoC-based Architecture. In *First International Workshop on Mathematics and Computer Science (IWMCS2012)*, Tiaret, Algérie, Dec. 2012.

[4] M. B. Andriamiarina, D. Méry, and N. K. Singh. Integrating proved state-based models for constructing correct distributed algorithms. In E. B. Johnsen and L. Petre, editors, *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 2013.

[5] M. B. Andriamiarina, D. Méry, and N. K. Singh. Analysis of self-* and p2p systems using refinement. In Y. Aït-Ameur and K.-D. Schewe, editors, *ABZ*, volume 8477 of *Lecture Notes in Computer Science*, pages 117–123. Springer, 2014.

[6] S. Bhattacharyga. An overview of Source-Specific Multicast (SSM). `http://tools.ietf.org/html/rfc3569`.

[7] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas. Protocol Indepent Multicast - Sparse Mode (PIM-SM) :Protocol Specification. `http://tools.ietf.org/html/rfc4601`.

[8] M. Handley, I. Kouvelas, T. Speakman, and L. Vicisano. Protocol Indepent Multicast - Dense Mode (PIM-DM) :Protocol Specification. `http://tools.ietf.org/html/rfc5015`.

[9] D. Hardy, G. Malléus, and J. Méreur. *Réseaux : Internet, téléphonie, multimédia. Convergences et complémentarités*. De Boeck Université, 2002.

[10] T. S. Hoang, A. Furst, and J.-R. Abrial. Event-b patterns and their tool support. *Software Engineering and Formal Methods, International Conference on*, 0 :210–219, 2009.

[11] T. S. Hoang, H. Kuruma, D. Basin, and J.-R. Abrial. Developing topology discovery in event-b. *Science of Computer Programming*, 74(11–12) :879 – 899, 2009.

[12] J. Kang, J. Sucec, V. Kaul, S. Samtani, and M. A. Fecko. Robust pim-sm multicasting using anycast rp in wireless ad hoc networks. In *Proceedings of the 2009 IEEE international conference on Communications*, ICC'09, pages 5139–5144, Piscataway, NJ, USA, 2009. IEEE Press.

[13] D. Méry. Refinement-based guidelines for algorithmic systems. *International Journal of Software and Informatics*, 3(2-3) :197–239, June/September 2009.

[14] D. Méry and N. K. Singh. Analysis of DSR protocol in event-B. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, SSS'11, pages 401–415, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] D. Meyer and D. Fenner. Multicast source discovery protocol MSDP. `http://tools.ietf.org/html/rfc3618`.

[16] J. Rehm. *Gestion du temps par le raffinement*. Thèse, Université Henri Poincaré - Nancy I, Dec. 2009.

[17] C. Systems. Anycast RP. `http://www.cisco.com/en/US/docs/ios/solutions_docs/ip_multicast/White_papers`.

[18] C. Systems. Anycast RP using PIM. `http://tools.ietf.org/html/draft-ietf-pim-anycast-rp-07`.

6

Patrons de conception et « Réseaux-sur-Puce »

Sommaire

6.1 Introduction

Nous avons étudié dans le chapitre précédent un algorithme de routage industriel, appelé Anycast RP, mis au point par CISCO Systems [15, 26, 27] : il s'agit d'un protocole un protocole robuste, fiable, pouvant passer à l'échelle et être utilisé par des applications réseaux critiques, impliquant différents types de routeurs (Sources, Routeurs Désignés, Points de Rendez-vous, Destinataires), permettant de diviser le protocoles en trois phases séquentielles similaires :

— Nous avons une première diffusion d'un paquet, d'une source à un Routeur Désigné (DR).
— Nous avons ensuite une deuxième diffusion du paquet, du routeur DR à un des routeurs Points de Rendez-vous (RPs), qui a été choisi comme destinataire intermédiaire, à l'aide de critères d'optimalité (routeur RP géographiquement proche du routeur DR ou charge du routeur minimale, etc.)
— Nous avons enfin une diffusion du paquet, du routeur RP au groupe de destinataires du paquet.

Ce protocole nous a permis d'appliquer le paradigme de service-as-event [5] : nous avons utilisé les propriétés de vivacité pour caractériser le protocole à l'aide des services qu'il fournit et appliqué les règles d'inférence pour identifier et développer les différentes phases du protocole Anycast RP. Les propriétés de vivacité de type *leadsto*, les règles d'inférence relatives à ces propriétés, nous ont permis de guider nos différentes étapes de raffinement.

La similarité des phases du protocole Anycast RP (envoi d'un message/paquet par une source, réception par un ou plusieurs destinataires), la similarité de leurs développements formels, des propriétés de sûreté et de vivacité les caractérisant, nous ont aussi amenés à nous intéresser à la notion de patrons de conception, que nous développons dans ce chapitre. Nous avons ainsi pu mettre au point un patron de conception [2, 13] pour le routage, c'est-à-dire un développement formel générique, que nous instancierons notamment pour l'analyse et l'étude des réseaux-sur-puce (NoC) [4].

6.2 Patrons de conception pour le routage

Nous nous sommes intéressés dans la section précédente au protocole Anycast RP. Nous avons vu que ce protocole pouvaient être décomposé en plusieurs étapes d'envoi et de réception de paquets et/ou de messages dans un système réparti, dont notamment (voir figure 6.1) :

1. Un envoi d'un paquet par une source et sa réception par un Routeur Désigné (DR).

2. La diffusion d'un message « register » encapsulant un paquet par un Routeur Désigné (DR) à un Point de Rendez-vous (RP) choisi.

3. La diffusion d'un message « register » et du paquet qu'il encapsule par un Point de Rendez-vous (RP) choisi par un routeur DR aux destinataires du paquet. Cette étape peut être décomposée en deux sous-étapes plus précises :

   (a) L'envoi par le Point de Rendez-vous (RP) choisi par un routeur DR, d'un message « register » aux autres routeurs RPs.

   (b) La désencapsulation d'un message « register » et la diffusion du paquet qui y a été encapsulé, par un routeur RP aux destinataires qui l'ont rejoint.

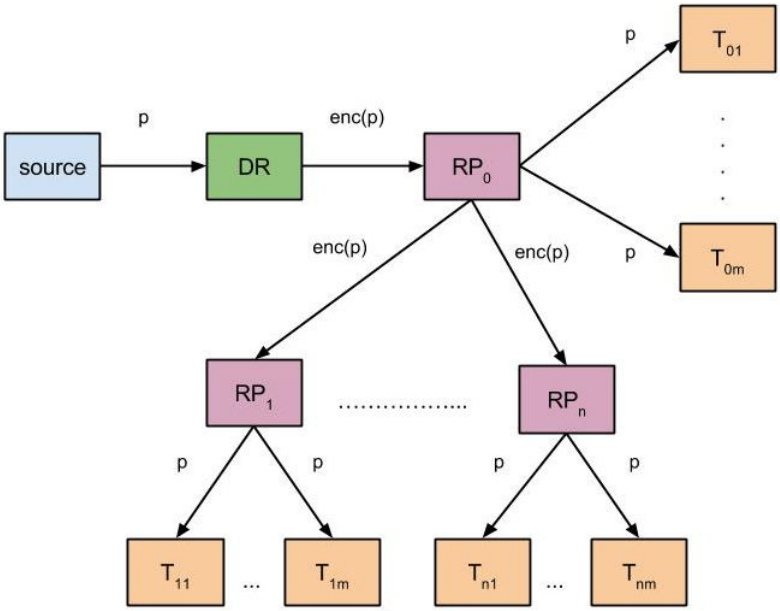


FIGURE 6.1 – Schéma : protocole ANYCAST RP

Pour modéliser ces étapes, nous avons utilisé les mêmes éléments de modèles pour les abstraire, à savoir :

- Des ensembles (messages/paquets, nœuds, etc), constantes (sources et destinations de paquets/messages, etc), axiomes similaires.
- Un événement (RE-)SENDING modélisant l'envoi (le renvoi) d'un paquet/message par une source à un groupe de destinataires.
- Un événement RECEIVING modélisant la réception d'un paquet/message envoyé par une source par ses destinataires.
- Un événement LOSING modélisant la possible perte d'un paquet/message envoyé par une source, mais pas encore reçu par ses destinataires.
- Des propriétés de sûreté relatives aux envois et réceptions de paquets et/ou messages, telles que : un paquet/message ne peut être envoyé que par sa source, la source d'un paquet/message est unique, un paquet/message ne peut être reçu que par ses destinataires, etc.

Les développements par raffinement de chacune de ces étapes sont aussi similaires :

- Il s'agit principalement d'introduire le système réparti support de l'algorithme de routage.
- De modifier les événements (RE-)SENDING, RECEIVING et LOSING vus précédemment pour prendre le réseau en compte.
- D'ajouter un événement FORWARD modélisant la circulation des messages/paquets dans le réseau.
- Et dans le cas où le réseau est mobile, d'ajouter des événements de modification de ce dernier : ADD_LINK (création de nouveaux liens entre dans le réseau), REMOVE_LINK (suppression de liens existants dans le réseau).

L'emploi d'un même schéma de développement formel, c'est-à-dire des mêmes éléments de modèles (événements, invariants, etc) et d'étapes de raffinement similaires, pour modéliser trois cas de communications par messages (envois et réceptions) nous a emmené à nous interroger sur la réutilisation de modèles, ainsi que des propriétés et preuves de correction qui y sont attachées. Nous nous intéressons particulièrement à associer à un type de problème (ici, le routage), une solution ou une partie d'une solution sous la forme d'un développement formel pouvant être (ré)utilisée de manière systématique à chaque fois que nous nous trouvons face à une occurence du problème étudié, ce qui est finalement identique à la notion de patrons de conception en EVENT-B, introduite par Abrial et al dans [2].

Un patron de conception peut être considéré comme une solution générique à un problème commun, récurrent (par exemple, les actions/réactions dans les systèmes réactifs, la transmission de messages, etc)

[2, 22, 7]. L'objectif d'un patron de conception est de faciliter le développement d'un problème étudié. Pour illustrer la notion de patron de conception, nous donnons les exemples suivant :

— Dans le domaine de l'architecture des bâtiments, Alexander propose dans [3] des solutions par rapport à la conception de bâtiments.

— Dans le domaine des systèmes orientés objets, Gamma et al présentent dans [12] des solutions de conception et de codage permettant de résoudre des problèmes spécifiques à la conception de systèmes orientés objets.

En résumé, nous pouvons définir un patron de conception de la manière suivante : un patron de conception présente un problème (typique) à résoudre, une solution générique à ce problème, guide et donne des indices sur l'utilisation de la solution pour la résolution du problème [17]. L'objectif d'un patron de conception est de rendre le développement d'une solution à un problème donné, méthodique et systématique [2, 22].

L'utilisation et l'application d'un patron de conception peuvent se décrire comme suit : il s'agit d'*adapter* et *incorporer* une solution prédéfinie (un modèle, du code, etc) dans un projet plus large [2]. Il s'agit d'une étape qui peut être assez ardue, car les problèmes à résoudre, auxquels s'adressent les patrons de conception, peuvent possèder une grande variété de formes, rendant les adaptations et incorporations compliquées.

Dans notre cas, nous considérons comme patrons de conception des modèles Event-B (ensembles, constantes, variables, invariants, événements) et leurs raffinements, formalisant des problème récurrents, typiques [2, 13]. L'utilisation de ces patrons de conception consiste dans un premier temps à identifier dans un développement formel plus large, les problèmes auxquels les solutions proposées dans les patrons de conception permettent de répondre, puis d'appliquer les solutions : il faut d'abord adapter les solutions, c'est-à-dire renommer les ensembles, constantes, variables, événements apportés par le patron de conception, par rapport aux éléments du développement plus large (pour éviter les conflits, utiliser des éléments déjà existants, etc.) et peut être même les modifier pour se conformer aux exigences et contraintes du développement plus large ; puis l'incorporation se fait en intégrant les éléments des modèles définis dans les patrons de conception aux modèles du développement formel plus large. Des raffinements des modèles utilisant les éléments fournis par les patrons de conceptions peuvent être ainsi créés. La figure 6.2 suivante, tirée de [13] résume la description que nous avons faite des patrons de conception, ainsi que leur utilisation :

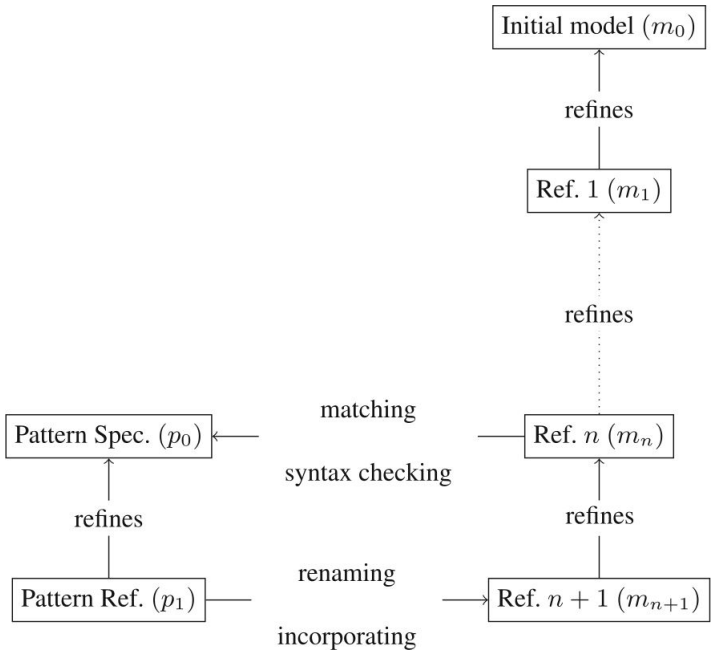


Figure 6.2 – Patron de conception en Event-B

Un patron de conception composé de deux modèles p_0 et p_1 est incorporé à un développement formel

plus large, comportant $n+1$ modèles, au niveau du modèle $\mathsf{m_n}$.

Nous proposons ici un patron de conception s'adressant à un problème typique et récurrent rencontré dans les systèmes répartis : il s'agit de la communication entre des nœuds distants et du routage de messages/paquets entre un nœud source et des nœuds destinations. Ce patron de conception que nous présentons ici a été extrait des modèles formels du protocole Anycast RP définis dans les sections précédentes : il s'agit d'éléments récurrents du développement formel du protocole Anycast RP que nous avons choisis pour élaborer un patron de conception.

6.2.1 Développement du patron de conception

Nous décrivons dans cette section le développement formel du patron de conception que nous proposons pour répondre aux problèmes de la communication entre des nœuds distants et du routage de messages/paquets entre un nœud source et des nœuds destinations dans un système réparti.

Nous avons distingué lors du développement formel du protocole Anycast RP deux cas :

— La communication entre une source et un destinataire unique.

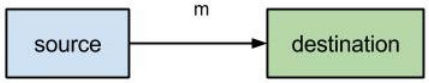


Figure 6.3 – Communication : source - destinataire

— La communication entre une source et un groupe de destinataires.

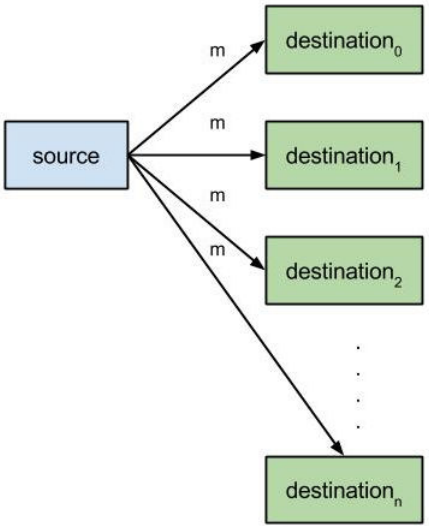


Figure 6.4 – Communication : source - groupe

6.2.1.1 Plan de développement

Le développement formel que nous proposons pour ce patron de conception prend en compte les deux cas cités ci-dessus et est constitué de trois niveaux d'abstraction :

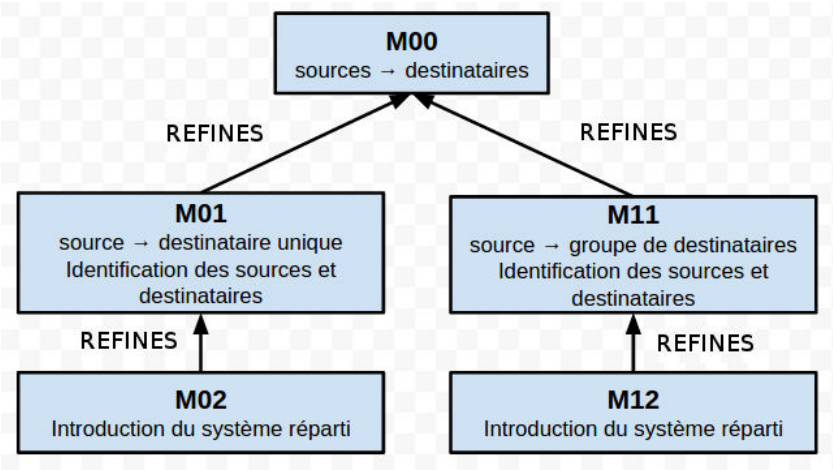


FIGURE 6.5 – Plan de développement du patron

1. Une abstraction M00 qui décrit le problème posé : la diffusion d'un message d'une source à des destinataires.
2. Un premier niveau de raffinement qui introduit les acteurs du problème (les nœuds sources et destinataires des messages) :
   — M01 modélise le cas où le destinataire du message diffusé par la source est unique.
   — M11 modélise le cas où la source diffuse un message pour un groupe de destinataires.
3. Un second niveau de raffinement qui définit le réseau réparti reliant les nœuds définis précédemment et dans lequel les messages circulent.
   — M02 raffine M01 et modélise l'acheminement dans le réseau d'un message de sa source à sa destination (unique).
   — M12 raffine M11 et modélise l'acheminement dans le réseau ud d'un message de sa source à son groupe de destinataires.

Nous détaillons chaque étape de ce développement formel dans les sous-sections suivantes.

6.2.1.2 Abstraction

L'abstraction présente l'idée générale du routage : il s'agit de diffuser un message, d'une source à un ou des destinataires. Nous commençons par introduire un contexte C00 qui définit un ensemble de messages MSG non-vide ($axm1$).

CONTEXT C00
SETS
MSG
AXIOMS
$axm1$: $MSG \neq \varnothing$
END

Ce contexte C00 est utilisé par une machine abstraite M00 dans laquelle nous définissons des événements modélisant la diffusion d'un message par sa source et sa réception par les destinataires.

Nous introduisons dans la machine M00 trois variables, toutes sous-ensembles de MSG :

INITIALISATION $\widehat{=}$
BEGIN
$act1$: $sent := \varnothing$
$act2$: $got := \varnothing$
$act3$: $lost := \varnothing$
END

$inv1$: $sent \subseteq MSG$
$inv2$: $rcvd \subseteq MSG$
$inv3$: $lost \subseteq MSG$

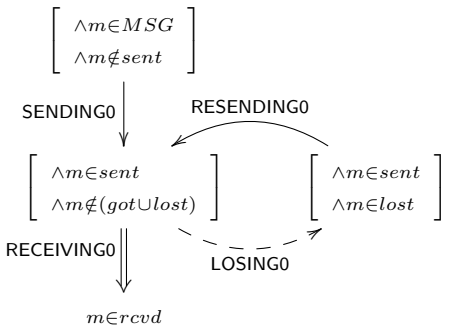


FIGURE 6.6 – Diagramme d'assertions pour l'abstraction M00

— La variable *sent* contient les messages envoyés par les sources.
— La variable *rcvd* contient les messages reçus par les destinataires.
— La variable *lost* contient les messages perdus.

Le fait qu'aucun message ne circule à l'état initial justifie les initialisations de ces variables à l'aide de l'ensemble vide ($\varnothing$). Nous notons x les variables de la machine M00.

Des propriétés de sûreté contraignent ces variables :

$$\begin{array}{l} inv3 \ : \ rcvd \cup lost \subseteq sent \\ inv4 \ : \ lost \cap rcvd = \varnothing \end{array}$$

— $inv3$ exprime que chaque message reçu ou perdu est un message qui a été envoyé.
— $inv4$ exprime qu'un message peut être soit perdu, soit reçu. Il ne peut pas être dans les deux états « reçu » et « perdu » en même temps.

La machine M00 contient les événements suivants que nous détaillerons après :
— SENDING0 modélise l'envoi d'un message m par une source.
— RESENDING0 modélise le renvoi d'un message m par une source.
— RECEIVING0 modélise la réception d'un message m, envoyé par une source, par son(ses) destinataire(s).
— LOSING0 modélise la perte d'un message m envoyé par une source, mais pas encore reçu par son(ses) destinataire(s).

L'hypothèse d'équité L_0 que nous posons sur le modèle M00 est la suivante :

$$L_0 \ \widehat{=} \ WF_x(\mathsf{SENDING0}) \wedge WF_x(\mathsf{RESENDING0}) \wedge SF_x(\mathsf{RECEIVING0})$$

Nous posons des hypothèses d'équité faible sur les événements SENDING0, RESENDING0 et une hypothèse d'équité forte sur l'événement RECEIVING0. Par contre, nous ne posons aucune hypothèse d'équité sur l'événement LOSING0, qui représente une action de l'environnement (perte de messages).

Nous définissons aussi la relation NEXT :

$$\begin{array}{rl} \mathsf{NEXT} \ \widehat{=} & \vee BA(\mathsf{SENDING0})(x, x') \\ & \vee BA(\mathsf{LOSING0})(x, x') \\ & \vee BA(\mathsf{RESENDING0})(x, x') \\ & \vee BA(\mathsf{RECEIVING0})(x, x') \\ & \vee (x = x') \end{array}$$

et définissons les propriétés suivantes :
— $P \ \widehat{=} \ (m \in MSG \wedge m \notin sent)$, qui exprime qu'un message m n'a pas encore été envoyé par une source.
— $R \ \widehat{=} \ (m \in sent \wedge m \in lost)$, exprimant qu'un message m a été envoyé par une source et est perdu.
— $S \ \widehat{=} \ (m \in sent \wedge m \notin (rcvd \cup lost))$, qui exprime qu'un message m a été envoyé par une source et n'est ni perdu, ni reçu par son(ses) destinataire(s) : il transite entre sa source et son(ses) destinataire(s).

— $Q \ \widehat{=}\ (m \in rcvd)$, exprimant qu'un message m est reçu par son(ses) destinataire(s).

Le diagramme 6.6 nous permet d'exprimer simplement la diffusion d'un message m par une source à sa(ses) destination(s), à l'aide des propriétés de vivacité suivantes :

1. $P \rightsquigarrow S$ qui exprime qu'un message m non-encore envoyé par sa source, transitera fatalement entre cette dernière et les destinataires.
2. $R \rightsquigarrow S$ qui exprime qu'un message m encore envoyé par sa source et perdu, sera fatalement renvoyé à ses destinataires.
3. $S \rightsquigarrow Q$ qui exprime qu'un message m transitant vers ses destinataires sera fatalement reçu par ces derniers.

L'ensemble Φ_0 des propriétés caractérisant M00 est donc : $\Phi_0 \ \widehat{=}\ \{P \rightsquigarrow S, R \rightsquigarrow S, S \rightsquigarrow Q\}$.

Démonstration. Nous montrons maintenant que la machine M00 satisfait les propriétés de vivacité de Φ_0.

⟨1⟩1. Nous montrons que M00 satisfait $P \rightsquigarrow S$. Une hypothèse d'équité faible est posée sur l'événement SENDING0 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de SENDING0 sont observables lorsque P est vrai, leurs observations ne falsifient pas le prédicat P, condition d'observation de SENDING0, ou conduisent à S.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :
$$P \wedge [\mathsf{NEXT}]_x \Rightarrow (P' \vee S')$$
Soit :
— $P \wedge BA(\mathsf{SENDING0})(x, x') \Rightarrow (P' \vee S')$
— $P \wedge BA(\mathsf{RESENDING0})(x, x') \Rightarrow (P' \vee S')$
— $P \wedge BA(\mathsf{RECEIVING0})(x, x') \Rightarrow (P' \vee S')$
— $P \wedge BA(\mathsf{LOSING0})(x, x') \Rightarrow (P' \vee S')$
— $P \wedge (x = x') \Rightarrow (P' \vee S')$
La propriété suivante **(3)** : $P(x) \wedge BA(\mathsf{SENDING0})(x, x') \Rightarrow S(x')$, et la condition de faisabilité **(4)** : $P(x) \Rightarrow (\exists x' \cdot BA(\mathsf{SENDING0})(x, x'))$ sont satisfaites par l'événement SENDING0. La propriété **(3)** nous permet de déduire $P \wedge \langle \mathsf{NEXT} \wedge \mathsf{SENDING0} \rangle_x \Rightarrow S'$ et **(4)** nous permet de déduire $P \Rightarrow ENABLED\langle \mathsf{SENDING0} \rangle_x$, où $ENABLED\langle \mathsf{SENDING0} \rangle_x \ \widehat{=}\ (\exists y \cdot BA(\mathsf{SENDING0})(x, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M00}) \vdash P \rightsquigarrow S$, donc que M00 satisfait $P \rightsquigarrow S$. □

⟨1⟩2. Nous montrons que M00 satisfait $R \rightsquigarrow S$. L'événement LOSING0 peut nous conduire d'un état satisfaisant S à un état satisfaisant R. Nous montrons qu'une opération de renvoi modélisée par l'événement RESENDING0 permet de retourner à un état satisfaisant S. Une hypothèse d'équité faible est posée sur l'événement RESENDING0 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de RESENDING0 sont observables lorsque R est vrai, leurs observations ne falsifient pas le prédicat R, condition d'observation de RESENDING0, ou conduisent à S.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :
$$R \wedge [\mathsf{NEXT}]_x \Rightarrow (R' \vee S')$$
Soit :
— $R \wedge BA(\mathsf{RESENDING0})(x, x') \Rightarrow (R' \vee S')$
— $R \wedge BA(\mathsf{SENDING0})(x, x') \Rightarrow (P' \vee S')$
— $R \wedge BA(\mathsf{RECEIVING0})(x, x') \Rightarrow (R' \vee S')$
— $R \wedge BA(\mathsf{LOSING0})(x, x') \Rightarrow (R' \vee S')$
— $R \wedge (x = x') \Rightarrow (R' \vee S')$
La propriété **(3)** : $R(x) \wedge BA(\mathsf{RESENDING0})(x, x') \Rightarrow S(x')$, et la condition de faisabilité **(4)** : $R(x) \Rightarrow (\exists x' \cdot BA(\mathsf{RESENDING0})(x, x'))$ sont satisfaites par l'événement RESENDING0. La propriété **(3)** permet de déduire $R \wedge \langle \mathsf{NEXT} \wedge \mathsf{RESENDING0} \rangle_x \Rightarrow S'$ et **(4)** permet de déduire $R \Rightarrow ENABLED\langle \mathsf{RESENDING0} \rangle_x$, où $ENABLED\langle \mathsf{RESENDING0} \rangle_x \ \widehat{=}\ (\exists y \cdot BA(\mathsf{RESENDING0})(x, y))$. La règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M00}) \vdash R \rightsquigarrow S$, donc que M00 satisfait $R \rightsquigarrow S$. □

⟨1⟩3. Nous montrons que M0 satisfait $S \rightsquigarrow Q$. Pour cela, nous allons prouver que M00 satisfait $(R \vee S) \rightsquigarrow Q$. Une hypothèse d'équité forte est posée sur l'événement RECEIVING0 et nous nous trouvons dans le cas suivant **(1)** : un événement LOSING0 peut nous ramener, d'un état satisfaisant

S, à un état satisfaisant R (tel que $R \rightsquigarrow S$), sinon les observations des autres événements, différents de LOSING0 et RECEIVING0, ne falsifient pas $(R \vee S)$ ou peuvent nous conduire à Q.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(R \vee S) \wedge [\mathsf{NEXT}]_x \Rightarrow ((R' \vee S') \vee Q')$$

Soit :
— $(R \vee S) \wedge BA(\mathsf{SENDING0})(x, x') \Rightarrow ((R' \vee S') \vee Q')$
— $(R \vee S) \wedge BA(\mathsf{RESENDING0})(x, x') \Rightarrow ((R' \vee S') \vee Q')$
— $(R \vee S) \wedge BA(\mathsf{LOSING0})(x, x') \Rightarrow ((R' \vee S') \vee Q')$
— $(R \vee S) \wedge BA(\mathsf{RECEIVING0})(x, x') \Rightarrow ((R' \vee S') \vee Q')$
— $(R \vee S) \wedge (x = x') \Rightarrow ((R' \vee S') \vee Q')$
La propriété **(3)** : $S(x) \wedge BA(\mathsf{RECEIVING0})(x, x') \Rightarrow Q(x')$, ainsi que la condition de faisabilité **(4)** : $S(x) \Rightarrow (\exists x' \cdot BA(\mathsf{RECEIVING0})(x, x'))$ sont satisfaites par RECEIVING0. **(3)** nous permet de déduire $(R \vee S) \wedge \langle \mathsf{NEXT} \wedge \mathsf{RECEIVING0} \rangle_x \Rightarrow Q'$. Les propriétés **(3)**, **(4)** et $R \rightsquigarrow S$ nous permettent de déduire que tant que Q n'est pas satisfait, alors RECEIVING0 sera fatalement activable : $(\neg Q) \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING0} \rangle_x$, où $ENABLED\langle \mathsf{RECEIVING0} \rangle_x \ \widehat{=}\ (\exists y \cdot BA(\mathsf{RECEIVING0})(x, y))$. Nous pouvons en déduire, en tenant compte de $R \rightsquigarrow S : \Box(R \vee S) \wedge \Box[\mathsf{NEXT}]_x \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING0} \rangle_x$. L'application de la règle SF1 donne $(\mathcal{S}pec(\mathsf{M00}), R \rightsquigarrow S) \vdash ((R \vee S) \rightsquigarrow Q)$. La machine M00 satisfait $(R \vee S) \rightsquigarrow Q$, donc $S \rightsquigarrow Q$. □

⟨1⟩4. Les étapes ⟨1⟩1, ⟨1⟩2 et ⟨1⟩3 nous permettent de déduire que M00 satisfait les propriétés de vivacité de Φ_0. QED.

□

Nous nous aidons de ces propriétés de vivacité de Φ_0 et par conséquent du diagramme 6.6 pour détailler le modèle Event-B abstrait M00 du patron de conception :

— Un événement SENDING0 modélise l'envoi d'un message : un message m qui n'a pas encore été envoyé est diffusé par une source ($act1$).

```
EVENT SENDING0 ≙
  ANY
    m
  WHERE
    grd1 : m ∈ MSG
    grd2 : m ∉ sent
  THEN
    act1 : sent := sent ∪ {m}
  END
```

— Un événement RESENDING0 modélise le renvoi d'un message : si un message m renvoyé par sa source se trouve dans la variable *lost* contenant les messages perdus, il est retiré de cette dernière lors du nouvel envoi ($act1$).

```
EVENT RESENDING0 ≙
  ANY
    m
  WHERE
    grd1 : m ∈ MSG
    grd2 : m ∈ sent
    grd3 : m ∈ lost
  THEN
    act1 : lost := lost \ {m}
  END
```

— RECEIVING0 modélise la réception d'un message m par un ou plusieurs destinataires : le message m a été envoyé par une source ($grd2$), il n'a pas encore été reçu par son (ses) destinataire(s) et il n'est pas perdu ($grd3$) ; il peut alors être reçu par son (ses) destinataire(s) ($act1$).

EVENT RECEIVING0 $\widehat{=}$
ANY
m
WHERE
$grd1$: $m \in MSG$
$grd2$: $m \in sent$
$grd3$: $m \notin (rcvd \cup lost)$
THEN
$act1$: $rcvd := rcvd \cup \{m\}$
END

— Un événement LOSING0 modélise la perte d'un message m lors de sa diffusion : la perte d'un message m est possible ($act1$), s'il a été envoyé par une source ($grd2$), n'a pas encore été reçu par son (ses) destinataire(s) et n'est pas perdu ($grd3$).

EVENT LOSING0 $\widehat{=}$
ANY
m
WHERE
$grd1$: $m \in MSG$
$grd2$: $m \in sent$
$grd2$: $m \notin (rcvd \cup lost)$
THEN
$act1$: $lost := lost \cup \{m\}$
END

Nous remarquons que cette abstraction est une copie conforme de l'abstraction du protocole de routage Anycast RP. Les deux modèles sont les mêmes (aux noms de constantes et variables près), car ils modélisent tous les deux l'objectif d'un protocole de routage, c'est-à-dire la transmission d'un élément (paquet ou message) d'une source à ses destinataires. Nous pouvons résumer cette abstraction M00 par le diagramme 6.7 suivant :

$$\text{C00} \xleftarrow{\text{SEES}} \text{M00} \xrightarrow{\text{LIVE}} \left(\left(\begin{array}{l} \wedge WF_x(\text{SENDING0}) \\ \wedge WF_x(\text{RESENDING0}) \\ \wedge SF_x(\text{RECEIVING0}) \end{array} \right), \left\{ \begin{array}{l} P \rightsquigarrow S, \\ R \rightsquigarrow S, \\ S \rightsquigarrow Q \end{array} \right\} \right)$$

Figure 6.7 – Abstraction du patron

Le raffinement suivant introduit les identités des sources et destinataires mis en jeu lors du routage d'un message de sa sources à ses destinations.

6.2.1.3 Premier raffinement

Lors de l'abstraction, nous nous sommes contentés de dire qu'un message était acheminé d'un point source à un point destination, tous les deux anonymes. L'objectif de ce premier raffinement est d'identifier maintenant les différents acteurs lors de la diffusion d'un message, c'est-à-dire les sources et les destinataires.

Cas où le destinataire d'un message est unique. Nous commençons par définir un contexte C01 étendant le contexte C00 vu dans la section précédente.

CONTEXT C01 EXTENDS C00
SETS
$NODES$
CONSTANTS
src, dst
AXIOMS
$axm1$: $NODES \neq \varnothing$
$axm2$: $src \in MSG \rightarrow NODES$
$axm3$: $dst \in MSG \rightarrow NODES$
$axm4$: $\forall m \cdot m \in MSG \Rightarrow src(m) \neq dst(m)$
END

— Nous y définissons un ensemble non-vide de nœuds $NODES$ participant à la réalisation du routage d'un message ($axm1$).
— Nous associons chaque message à envoyer à une source, à l'aide d'une fonction totale src associant chaque message à un nœud source ($axm2$).
— Nous associons chaque message à une destination, à l'aide d'une fonction totale dst associant chaque message à un nœud destination ($axm3$).
— Nous exprimons ($axm4$) aussi le fait que la source ($src(m)$) d'un message m est différente de son destinataire ($dst(m)$).

Ce contexte C01 est utilisé par un raffinement M01 de la machine M00.

Nous introduisons dans ce premier raffinement de nouvelles variables :

INITIALISATION $\widehat{=}$
BEGIN
$\oplus$ $act1$: $sent\_by\_s := \varnothing$
$\oplus$ $act2$: $rcvd\_by\_d := \varnothing$
$\oplus$ $act3$: $m\_dst := \varnothing$
$\ominus$ $act1$: $sent := \varnothing$
$\ominus$ $act2$: $rcvd := \varnothing$
...
END

$inv1$: $sent\_by\_s \in NODES \leftrightarrow MSG$
$inv2$: $rcvd\_by\_d \in NODES \leftrightarrow MSG$
$inv3$: $m\_dst \in NODES \leftrightarrow MSG$

— $sent\_by\_s$ associe à chaque message envoyé, la source qui l'a diffusé. Elle remplace la variable abstraite $sent$.
— $rcvd\_by\_d$ associe à chaque message reçu, le destinataire qui l'a reçu.
— m_dst associe à chaque message envoyé, le destinataire vers lequel il est acheminé.

Ces variables sont initialisées à l'aide de l'ensemble vide ($\varnothing$), car à l'état initial, aucun message ne circule. Nous notons x_1 l'ensemble des variables de la machine M01.

Des invariants contraignent ces variables :

$inv4$: $\forall s, m.s \in NODES \land m \in MSG \land s \mapsto m \in sent\_by\_s \Rightarrow s = source(m)$
$inv5$: $\forall s1, s2, m \cdot \left(\begin{array}{l} \land s1 \in NODES \land s2 \in NODES \land m \in MSG \\ \land s1 \mapsto m \in sent\_by\_s \land s2 \mapsto m \in sent\_by\_s \end{array} \right) \Rightarrow s1 = s2$
$inv6$: $ran(sent\_by\_s) = sent$
$inv7$: $ran(m\_dst) \subseteq ran(sent\_by\_s)$
$inv8$: $rcvd\_by\_d \subseteq m\_dst$
$thm1$: $ran(rcvd\_by\_d) \subseteq ran(sent\_by\_s)$
$inv9$: $\forall n, m.n \in NODES \land m \in MSG \land n \mapsto m \in m\_dst \Rightarrow n = dst(m)$
$inv10$: $\forall n1, n2, m \cdot \left(\begin{array}{l} \land n1 \in NODES \land n2 \in NODES \land m \in MSG \\ \land n1 \mapsto m \in m\_dst \land n2 \mapsto m \in m\_dst \end{array} \right) \Rightarrow n1 = n2$
$inv11$: $ran(rcvd\_by\_d) = rcvd$

— $inv4$, $inv5$ et $inv6$ sont des invariants de collage et permettent de faire le lien entre la variable concrète $sent\_by\_s$ et celle abstraite $sent$.
  — $inv4$ exprime que si un message m est envoyé par un nœud s, alors s est la source du message m.
  — $inv5$ exprime que l'expéditeur d'un message m est unique.
  — $inv6$ exprime enfin que pour chaque message envoyé par un nœud au niveau concret, le même message est envoyé au niveau abstrait.

  Ces trois invariants permettent ainsi de remplacer $sent$ par $sent\_by\_s$.
— $inv7$ exprime que chaque message acheminé vers une destination a été envoyé par une source.
— $inv8$ exprime que chaque message reçu par une destination a été acheminé vers cette destination.
— $thm1$ exprime que chaque message reçu par une destination a été envoyé par une source. Cette propriété est déduite à partir de $inv7$ et $inv8$.
— $inv9$, $inv10$, $inv11$ sont invariants de collage, établissent la relation entre $rcvd\_by\_d$ et $rcvd$.
  — $inv9$ exprime que si un message m est acheminé vers un nœud n, alors ce nœud n est le destinataire de m.
  — $inv10$ exprime que la destination d'un message m est unique.

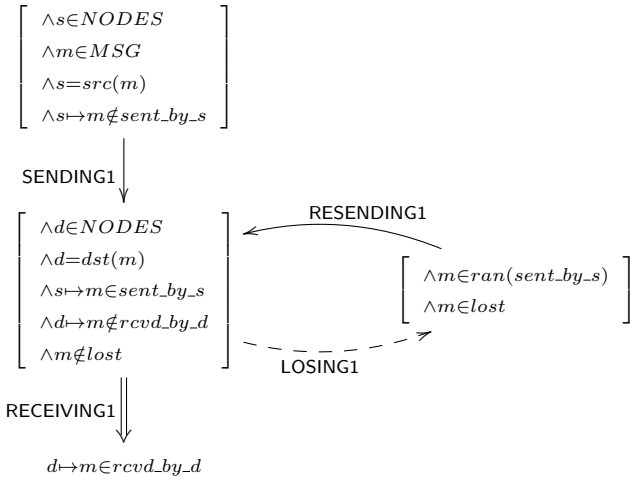


FIGURE 6.8 – Diagramme d'assertions pour le premier raffinement M01

— $inv11$ exprime que chaque message reçu par un destinataire au niveau concret, l'est aussi au niveau abstrait.

Ces invariants, ainsi que l'invariant $inv8$, donnent la possibilité de supprimer la variable abstraite $rcvd$ et de la remplacer par $rcvd\_by\_d$.

La machine M01 contient les raffinements des événements de la machine M00, soit SENDING1, RESENDING1, RECEIVING1 et LOSING1. L'hypothèse d'équité L_1 que nous posons sur le modèle M01 est la suivante :

$$L_0 \ \widehat{=}\ WF_{x_1}(\mathsf{SENDING1}) \wedge WF_{x_1}(\mathsf{RESENDING1}) \wedge SF_{x_1}(\mathsf{RECEIVING1})$$

Nous posons des hypothèses d'équité faible sur les événements SENDING1, RESENDING1 et une hypothèse d'équité forte sur l'événement RECEIVING1. Par contre, nous ne posons aucune hypothèse d'équité sur l'événement LOSING1, qui représente une action de l'environnement (perte de messages).

Nous définissons aussi la relation NEXT :

$$\begin{array}{rl} \mathsf{NEXT} \ \widehat{=} & \vee BA(\mathsf{SENDING1})(x_1, x_1') \\ & \vee BA(\mathsf{LOSING1})(x_1, x_1') \\ & \vee BA(\mathsf{RESENDING1})(x_1, x_1') \\ & \vee BA(\mathsf{RECEIVING1})(x_1, x_1') \\ & \vee (x_1 = x_1') \end{array}$$

et définissons les propriétés suivantes :

— $P_1 \ \widehat{=}\ \begin{pmatrix} \wedge s \in NODES \\ \wedge m \in MSG \\ \wedge s = src(m) \\ \wedge s \mapsto m \notin sent\_by\_s \end{pmatrix}$

Cette propriété exprime qu'un message m n'a pas encore été envoyé par sa source s.

— $R_1 \ \widehat{=}\ (m \in ran(sent\_by\_s) \wedge m \in lost)$,

Cette propriété exprime qu'un message m a été envoyé par une source et est perdu.

— $S_1 \ \widehat{=}\ \begin{pmatrix} \wedge d \in NODES \\ \wedge d = dst(m) \\ \wedge s \mapsto m \in sent \\ \wedge d \mapsto m \notin rcvd\_by\_d \\ \wedge m \notin lost \end{pmatrix}$

Cette propriété exprime qu'un message m a été envoyé par sa source s et n'est ni perdu, ni reçu par son destinataire d : il transite entre sa source et son destinataire.

— $Q_1 \ \widehat{=}\ (d \mapsto m \in rcvd\_by\_d)$, exprimant qu'un message m est reçu par son destinataire d.

Le diagramme 6.8 nous permet d'exprimer simplement la diffusion d'un message m par une source à une destination, à l'aide des propriétés de vivacité suivantes :

1. $P_1 \rightsquigarrow S_1$ qui exprime qu'un message m non-encore envoyé par sa source s, transitera fatalement entre cette dernière et son destinataire d.
2. $R_1 \rightsquigarrow S_1$ qui exprime qu'un message m encore envoyé par sa source s et perdu, sera fatalement renvoyé à son destinataire d.
3. $S_1 \rightsquigarrow Q_1$ qui exprime qu'un message m transitant vers son destinataire d sera fatalement reçu par ce dernier.

L'ensemble Φ_1 des propriétés caractérisant M01 est donc : $\Phi_1 \ \widehat{=}\ \{P_1 \rightsquigarrow S_1, R_1 \rightsquigarrow S_1, S_1 \rightsquigarrow Q_1\}$.

Démonstration. Nous montrons maintenant que la machine M01 satisfait les propriétés de vivacité de Φ_1.

⟨1⟩1. Nous montrons que M01 satisfait $P_1 \rightsquigarrow S_1$. Une hypothèse d'équité faible est posée sur l'événement SENDING1 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de SENDING1 sont observables lorsque P_1 est vrai, leurs observations ne falsifient pas le prédicat P_1, condition d'observation de SENDING1, ou conduisent à S_1.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :
$$P_1 \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow (P_1' \vee S_1')$$
Soit :
— $P_1 \wedge BA(\mathsf{SENDING1})(x_1, {x_1}') \Rightarrow (P_1' \vee S_1')$
— $P_1 \wedge BA(\mathsf{RESENDING1})(x_1, {x_1}') \Rightarrow (P_1' \vee S_1')$
— $P_1 \wedge BA(\mathsf{RECEIVING1})(x_1, {x_1}') \Rightarrow (P_1' \vee S_1')$
— $P_1 \wedge BA(\mathsf{LOSING1})(x_1, {x_1}') \Rightarrow (P_1' \vee S_1')$
— $P_1 \wedge (x_1 = {x_1}') \Rightarrow (P_1' \vee S_1')$
La propriété suivante **(3)** : $P_1(x_1) \wedge BA(\mathsf{SENDING1})(x_1, {x_1}') \Rightarrow S_1({x_1}')$, et la condition de faisabilité **(4)** : $P_1(x_1) \Rightarrow (\exists {x_1}' \cdot BA(\mathsf{SENDING1})(x_1, {x_1}'))$ sont satisfaites par l'événement SENDING1. **(3)** nous permet de déduire $P_1 \wedge \langle \mathsf{NEXT} \wedge \mathsf{SENDING1} \rangle_{x_1} \Rightarrow S_1'$ et **(4)** permet de déduire $P_1 \Rightarrow ENABLED\langle \mathsf{SENDING1} \rangle_{x_1}$, où $ENABLED\langle \mathsf{SENDING1} \rangle_{x_1} \ \widehat{=}\ (\exists y \cdot BA(\mathsf{SENDING1})(x_1, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M01}) \vdash P_1 \rightsquigarrow S_1$, donc que M01 satisfait $P_1 \rightsquigarrow S_1$. □

⟨1⟩2. Nous montrons que M01 satisfait $R_1 \rightsquigarrow S_1$. L'événement LOSING0 peut nous conduire d'un état satisfaisant S_1 à un état satisfaisant R_1. Nous montrons qu'une opération de renvoi modélisée par l'événement RESENDING1 permet de retourner à un état satisfaisant S_1. Une hypothèse d'équité faible est posée sur l'événement RESENDING1 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de RESENDING1 sont observables lorsque R_1 est vrai, leurs observations ne falsifient pas le prédicat R_1, condition d'observation de RESENDING1, ou conduisent à S_1.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :
$$R_1 \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow (R_1' \vee S_1')$$
Soit :
— $R_1 \wedge BA(\mathsf{RESENDING1})(x_1, {x_1}') \Rightarrow (R_1' \vee S_1')$
— $R_1 \wedge BA(\mathsf{SENDING1})(x_1, {x_1}') \Rightarrow (R_1' \vee S_1')$
— $R_1 \wedge BA(\mathsf{RECEIVING1})(x_1, {x_1}') \Rightarrow (R_1' \vee S_1')$
— $R_1 \wedge BA(\mathsf{LOSING1})(x_1, {x_1}') \Rightarrow (R_1' \vee S_1')$
— $R_1 \wedge (x_1 = {x_1}') \Rightarrow (R_1' \vee S_1')$
La propriété **(3)** : $R_1(x_1) \wedge BA(\mathsf{RESENDING1})(x_1, {x_1}') \Rightarrow S_1({x_1}')$, et la condition de faisabilité **(4)** : $R_1(x_1) \Rightarrow (\exists {x_1}' \cdot BA(\mathsf{RESENDING1})(x_1, {x_1}'))$ sont satisfaites par l'événement RESENDING1. **(3)** permet de déduire $R_1 \wedge \langle \mathsf{NEXT} \wedge \mathsf{RESENDING1} \rangle_{x_1} \Rightarrow S_1'$ et **(4)** permet de déduire $R_1 \Rightarrow ENABLED\langle \mathsf{RESENDING1} \rangle_{x_1}$, où $ENABLED\langle \mathsf{RESENDING1} \rangle_{x_1} \ \widehat{=}\ (\exists y \cdot BA(\mathsf{RESENDING1})(x_1, y))$.
La règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M01}) \vdash R_1 \rightsquigarrow S_1$, donc que M01 satisfait $R_1 \rightsquigarrow S_1 S$. □

⟨1⟩3. Nous montrons que M01 satisfait $S_1 \rightsquigarrow Q_1$. Pour cela, nous allons prouver que M01 satisfait $(R_1 \vee S_1) \rightsquigarrow Q_1$. Une hypothèse d'équité forte est posée sur l'événement RECEIVING1 et nous

nous trouvons dans le cas suivant **(1)** : un événement LOSING1 peut nous ramener, d'un état satisfaisant S_1, à un état satisfaisant R_1 (tel que $R_1 \leadsto S_1$), sinon les observations des autres événements, différents de LOSING1 et RECEIVING1, ne falsifient pas $(R_1 \vee S_1)$ ou peuvent nous conduire à Q_1.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(R_1 \vee S_1) \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow ((R_1' \vee S_1') \vee Q_1')$$

Soit :
— $(R_1 \vee S_1) \wedge BA(\mathsf{SENDING1})(x_1, {x_1}') \Rightarrow ((R_1' \vee S_1') \vee Q_1')$
— $(R_1 \vee S_1) \wedge BA(\mathsf{RESENDING1})(x_1, {x_1}') \Rightarrow ((R_1' \vee S_1') \vee Q_1')$
— $(R_1 \vee S_1) \wedge BA(\mathsf{LOSING1})(x_1, {x_1}') \Rightarrow ((R_1' \vee S_1') \vee Q_1')$
— $(R_1 \vee S_1) \wedge BA(\mathsf{RECEIVING1})(x_1, {x_1}') \Rightarrow ((R_1' \vee S_1') \vee Q_1')$
— $(R_1 \vee S_1) \wedge (x_1 = {x_1}') \Rightarrow ((R_1' \vee S_1') \vee Q_1')$
La propriété **(3)** : $S_1(x_1) \wedge BA(\mathsf{RECEIVING1})(x_1, {x_1}') \Rightarrow Q_1({x_1}')$, ainsi que la condition de faisabilité **(4)** : $S_1(x_1) \Rightarrow (\exists {x_1}' \cdot BA(\mathsf{RECEIVING1})(x_1, {x_1}'))$ sont satisfaites par RECEIVING1. **(3)** nous permet de déduire $(R_1 \vee S_1) \wedge \langle \mathsf{NEXT} \wedge \mathsf{RECEIVING1} \rangle_{x_1} \Rightarrow Q_1'$. Les propriétés **(3)**, **(4)** et $R_1 \leadsto S_1$ nous permettent de déduire que tant que Q_1 n'est pas satisfait, alors RECEIVING1 sera fatalement activable : $(\neg Q_1) \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING1} \rangle_{x_1}$, où $ENABLED\langle \mathsf{RECEIVING1} \rangle_{x_1} \ \widehat{=}\ (\exists y \cdot BA(\mathsf{RECEIVING1})(x_1, y))$. Nous déduisons, en tenant compte de $R_1 \leadsto S_1$: $\Box(R_1 \vee S_1) \wedge \Box[\mathsf{NEXT}]_{x_1} \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING1} \rangle_{x_1}$. L'application de la règle SF1 donne $(\mathcal{S}pec(\mathsf{M01}), R_1 \leadsto S_1) \vdash ((R_1 \vee S_1) \leadsto Q_1)$. La machine M01 satisfait $(R_1 \vee S_1) \leadsto Q_1$, donc $S_1 \leadsto Q_1$. □

⟨1⟩4. Les étapes ⟨1⟩1, ⟨1⟩2 et ⟨1⟩3 nous permettent de déduire que M01 satisfait les propriétés de vivacité de Φ_1. QED.

□

Soit Φ_0 l'ensemble des propriétés de vivacité caractérisant M00 :

$$\Phi_0 \ \widehat{=}\ \{P \leadsto S, R \leadsto S, S \leadsto Q\}$$

Démonstration. Nous montrons que M01 satisfait les propriétés de Φ_0. Nous commençons par établir quelques équivalences :
— $P_1 \Leftrightarrow P$. Nous avons :

$$P_1 \ \widehat{=}\ \begin{pmatrix} \wedge s \in NODES \\ \wedge m \in MSG \\ \wedge s = src(m) \\ \wedge s \mapsto m \notin sent\_by\_s \end{pmatrix}$$

et

$$P \ \widehat{=}\ (m \in MSG \wedge m \notin sent)$$

Les invariants $inv4$, $inv5$ et $inv6$ nous permettent de déduire $P_1 \Leftrightarrow P$.
— $R_1 \Leftrightarrow R$. Nous avons :

$$R_1 \ \widehat{=}\ (m \in ran(sent\_by\_s) \wedge m \in lost)$$

et

$$R \ \widehat{=}\ (m \in sent \wedge m \in lost)$$

Les invariants $inv4$, $inv5$ et $inv6$ nous permettent de déduire $R_1 \Leftrightarrow R$.
— $S_1 \Leftrightarrow S$. Nous avons :

$$S_1 \ \widehat{=}\ \begin{pmatrix} \wedge d \in NODES \\ \wedge d = dst(m) \\ \wedge s \mapsto m \in sent\_by\_s \\ \wedge d \mapsto m \notin rcvd\_by\_d \\ \wedge m \notin lost \end{pmatrix}$$

et

$$S \ \widehat{=} \ (m \in sent \wedge m \notin (rcvd \cup lost))$$

Les invariants $inv4$, $inv5$, $inv6$, $inv8$, $inv10$ et $inv11$ nous permettent de déduire $S_1 \Leftrightarrow S$.

— $Q_1 \Leftrightarrow Q$. Nous avons :

$$Q_1 \ \widehat{=} \ (d \mapsto m \in rcvd\_by\_d)$$

et

$$Q \ \widehat{=} \ (m \in rcvd)$$

Les invariants $inv8$, $inv10$ et $inv11$ nous permettent de déduire $Q_1 \Leftrightarrow Q$.

$\langle 1 \rangle 1$. Nous montrons que M01 satisfait $P \rightsquigarrow S$. Nous savons que M01 satisfait $P_1 \rightsquigarrow S_1$; et les équivalences $P_1 \Leftrightarrow P$ et $S_1 \Leftrightarrow S$ nous permettent de déduire que M01 satisfait $P \rightsquigarrow S$.

$\langle 1 \rangle 2$. Nous montrons que M01 satisfait $R \rightsquigarrow S$. Nous savons que M01 satisfait $R_1 \rightsquigarrow S_1$; et les équivalences $R_1 \Leftrightarrow R$ et $S_1 \Leftrightarrow S$ nous permettent de déduire que M01 satisfait $R \rightsquigarrow S$.

$\langle 1 \rangle 3$. Nous montrons que M01 satisfait $S \rightsquigarrow Q$. Nous savons que M01 satisfait $S_1 \rightsquigarrow Q_1$; et les équivalences $S_1 \Leftrightarrow S$ et $Q_1 \Leftrightarrow Q$ nous permettent de déduire que M01 satisfait $S \rightsquigarrow Q$.

$\langle 1 \rangle 4$. Les étapes $\langle 1 \rangle 1$, $\langle 1 \rangle 2$ et $\langle 1 \rangle 3$ nous permettent de déduire que M01 satisfait les propriétés de vivacité de Φ_0. QED.

□

Nous guidons les raffinements des événements SENDING0, RESENDING0, RECEIVING0, LOSING0 à l'aide des propriétés de Φ_1, pour prendre en compte l'identification des sources et des destinataires des messages (Nous ne présentons ici que les événements ayant changé suite au raffinement et les nouveaux événements) :

— L'événement SENDING1 raffine SENDING0. Un message m est diffusé ($act1$) par sa source s ($grd4$), à un groupe grd de destinataires ($act2$) contenant un seul membre $dst(m)$ ($grd6$), s'il n'a pas encore été diffusé par sa source s ($grd5$).

EVENT SENDING1
REFINES SENDING0 $\widehat{=}$
ANY
s, m, grd
WHERE
$grd1$: $s \in NODES$
$grd2$: $m \in MSG$
$grd3$: $grd \subseteq NODES$
$grd4$: $s = src(m)$
$grd5$: $s \mapsto m \notin sent\_by\_s$
$grd6$: $grd = \{dst(m)\}$
THEN
$act1$: $sent\_by\_s := sent\_by\_s \cup \{s \mapsto m\}$
$act2$: $m\_dst := m\_dst \cup (grd \times \{m\})$
END

— L'événement RESENDING1 raffine RESENDING0. Un message m est renvoyé par une source s et après ce renvoi, le message m est ôté de la liste des messages perdus, s'il a été perdu ($act1$).

EVENT RESENDING1
REFINES RESENDING0 $\widehat{=}$
ANY
s, m
WHERE
$grd1$: $s \in NODES$
$grd2$: $m \in MSG$
$grd3$: $s \mapsto m \in sent\_by\_s$
$grd4$: $m \notin lost$
THEN
$act1$: $lost := lost \setminus \{m\}$
END

— L'événement RECEIVING1 raffine RECEIVING0. Un message m est reçu ($act1$) par son destinataire d, s'il est acheminé vers ce destinataire ($grd3$), si ce dernier ne l'a pas encore reçu ($grd4$) et s'il n'est pas perdu ($grd5$).

EVENT RECEIVING1
REFINES RECEIVING0 $\widehat{=}$
  ANY
    d, m
  WHERE
    $grd1$: $d \in NODES$
    $grd2$: $m \in MSG$
    $grd3$: $d \mapsto m \in m\_dst$
    $grd4$: $d \mapsto m \notin rcvd\_by\_d$
    $grd5$: $m \notin lost$
  THEN
    $act1$: $rcvd\_by\_d := rcvd\_by\_d \cup \{d \mapsto m\}$
  END

— L'événement LOSING1 raffine LOSING0. Un message m peut être perdu ($act1$), s'il est a été envoyé par une source ($grd2$), n'a pas encore été reçu par son destinataire et n'est pas perdu ($grd3$).

EVENT LOSING1
REFINES LOSING0 $\widehat{=}$
  ANY
    m
  WHERE
    $grd1$: $m \in MSG$
    $grd2$: $m \in ran(sent\_by\_s)$
    $grd3$: $m \notin (ran(rcvd\_by\_d) \cup lost)$
  THEN
    $act1$: $lost := lost \cup \{m\}$
  END

Nous pouvons résumer ce raffinement M01 par le diagramme 6.9 suivant :

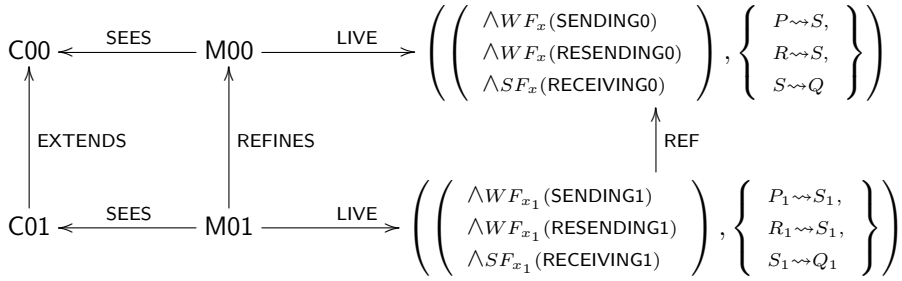


FIGURE 6.9 – Patron (destinataire unique) : premier raffinement

Le raffinement suivant introduit le système réparti support du routage que nous analysons ici. Mais avant de décrire cette nouvelle étape, nous présentons l'autre raffinement M11 de M00, modélisant le cas où un message possède des destinataires multiples.

Cas où un message est diffusé à un groupe de destinataires. Nous commençons par définir un contexte C11 étendant le contexte C00 vu dans la section précédente.

CONTEXT C11 EXTENDS C00
  SETS
    $NODES$
  CONSTANTS
    src, dst
  AXIOMS
    $axm1$: $NODES \neq \varnothing$
    $axm2$: $src \in MSG \rightarrow NODES$
    $axm3$: $dst \in MSG \rightarrow \mathbb{P}1(NODES)$
    $axm4$: $\forall m \cdot m \in MSG \Rightarrow src(m) \notin dst(m)$
  END

— Nous y définissons un ensemble non-vide de nœuds $NODES$ participant à la réalisation du routage d'un message ($axm1$).

— Nous associons chaque message à envoyer à une source, à l'aide d'une fonction totale *src* associant chaque message à un nœud source ($axm2$).
— Nous associons chaque message à un groupe non-vide de destinataire, à l'aide d'une fonction totale *dst* associant chaque message à un sous-ensemble de $NODES$ non-vide ($axm3$).
— Nous exprimons ($axm4$) aussi le fait que la source ($src(m)$) d'un message m ne fait pas partie de son groupe ($dst(m)$) de destinataires.

Ce contexte C11 est utilisé par un raffinement M11 de la machine M00.

Nous introduisons dans ce premier raffinement de nouvelles variables :

INITIALISATION $\widehat{=}$
BEGIN
$\oplus$ $act1$: $sent\_by\_s := \varnothing$
$\oplus$ $act2$: $rcvd\_by\_d := \varnothing$
$\oplus$ $act3$: $m\_dst := \varnothing$
$\ominus$ $act1$: $sent := \varnothing$
...
END

$inv1$: $sent\_by\_s \in NODES \leftrightarrow MSG$
$inv2$: $rcvd\_by\_d \in NODES \leftrightarrow MSG$
$inv3$: $m\_dst \in NODES \leftrightarrow MSG$

— *sent_by_s* associe à chaque message envoyé, la source qui l'a diffusé. Elle remplace la variable abstraite *sent*.
— *rcvd_by_d* associe à chaque message reçu, le destinataire qui l'a reçu.
— *m_dst* associe à chaque message envoyé, les destinataires vers lesquels il est acheminé.

Ces variables sont initialisées à l'aide de l'ensemble vide ($\varnothing$), car à l'état initial, aucun message ne circule. Nous notons x_{11} l'ensemble des variables de la machine M11.

Des invariants contraignent ces variables :

$inv4$: $\forall s, m.s \in NODES \land m \in MSG \land s \mapsto m \in sent\_by\_s \Rightarrow s = source(m)$
$inv5$: $\forall s1, s2, m \cdot \left(\begin{array}{l} \land s1 \in NODES \land s2 \in NODES \land m \in MSG \\ \land s1 \mapsto m \in sent\_by\_s \land s2 \mapsto m \in sent\_by\_s \end{array} \right) \Rightarrow s1 = s2$
$inv6$: $ran(sent\_by\_s) = sent$
$inv7$: $ran(m\_dst) \subseteq ran(sent\_by\_s)$
$inv8$: $rcvd\_by\_d \subseteq m\_dst$
$thm1$: $ran(rcvd\_by\_d) \subseteq ran(sent\_by\_s)$
$inv9$: $\forall n, m.n \in NODES \land m \in MSG \land n \mapsto m \in m\_dst \Rightarrow n \in dst(m)$
$inv10$: $\forall m \cdot m \in MSG \land m \in rcvd \Rightarrow dst(m) \times \{m\} \subseteq rcvd\_by\_d$

— $inv4$, $inv5$ et $inv6$ sont des invariants de collage et permettent de faire le lien entre la variable concrète *sent_by_s* et celle abstraite *sent*.
  — $inv4$ exprime que si un message m est envoyé par un nœud s, alors s est la source du message m.
  — $inv5$ exprime que l'expéditeur d'un message m est unique.
  — $inv6$ exprime enfin que pour chaque message envoyé par un nœud au niveau concret, le même message est envoyé au niveau abstrait.

  Ces trois invariants permettent ainsi de remplacer *sent* par *sent_by_s*.
— $inv7$ exprime que chaque message acheminé vers une destination a été envoyé par une source.
— $inv8$ exprime que chaque message reçu par une destination a été acheminé vers cette destination.
— $thm1$ exprime que chaque message reçu par une destination a été envoyé par une source. Cette propriété est déduite à partir de $inv7$ et $inv8$.
— $inv9$ exprime que si un message m est acheminé vers un nœud n, alors ce nœud n est un des destinataires de m.
— $inv10$ exprime que si un message m est considéré comme ayant été reçu globalement par tous ses destinataires, cela veut dire qu'il a été reçu individuellement par tous les membres de son groupe de destinataires.

La machine M11 contient les raffinements des événements de la machine M00, soit SENDING1, RESENDING1, RECEIVING1 et LOSING1, ainsi qu'un nouvel événement D_RECEIVING1, modélisant la réception individuelle d'un message par un de ses destinataires. L'hypothèse d'équité L_{11} que nous posons sur le modèle M11 est la suivante :

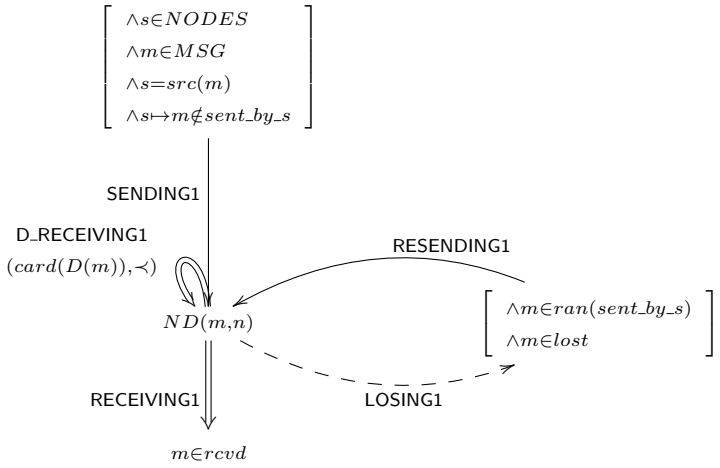


FIGURE 6.10 – Diagramme d'assertions pour le premier raffinement M11

$$\begin{array}{rcl} L_{11} & \widehat{=} & \wedge WF_{x_{11}}(\mathsf{SENDING1}) \\ & & \wedge WF_{x_{11}}(\mathsf{RESENDING1}) \\ & & \wedge SF_{x_{11}}(\mathsf{D\_RECEIVING1}) \\ & & \wedge SF_{x_{11}}(\mathsf{RECEIVING1}) \end{array}$$

Nous posons des hypothèses d'équité faible sur les événements SENDING1, RESENDING1 et des hypothèses d'équité forte sur les événements D_RECEIVING1 et RECEIVING1. Par contre, nous ne posons aucune hypothèse d'équité sur l'événement LOSING1, qui représente une action de l'environnement (perte de messages).

Nous définissons aussi la relation NEXT :

$$\begin{array}{rcl} \mathsf{NEXT} & \widehat{=} & \vee BA(\mathsf{SENDING1})(x_{11}, x_{11}') \\ & & \vee BA(\mathsf{LOSING1})(x_{11}, x_{11}') \\ & & \vee BA(\mathsf{RESENDING1})(x_{11}, x_{11}') \\ & & \vee BA(\mathsf{D\_RECEIVING1})(x_{11}, x_{11}') \\ & & \vee BA(\mathsf{RECEIVING1})(x_{11}, x_{11}') \\ & & \vee (x_{11} = x_{11}') \end{array}$$

Nous nous intéressons maintenant aux propriétés de vivacité satisfaites par la machine M11. Pour cela nous commençons par poser :

— $P_{11} \widehat{=} \left(\begin{array}{l} \wedge s \in NODES \\ \wedge m \in MSG \\ \wedge s = src(m) \\ \wedge s \mapsto m \notin sent\_by\_s \end{array} \right)$

Cette propriété exprime qu'un message m n'a pas encore été envoyé par sa source s.

— $R_{11} \widehat{=} (m \in ran(sent\_by\_s) \wedge m \in lost)$,

Cette propriété exprime qu'un message m a été envoyé par une source et est perdu.

— Nous posons :

$$D(m) \widehat{=} \left\{ d \;\middle|\; \begin{array}{l} \wedge d \in NODES \\ \wedge d \in m\_dst^{-1}[\{m\}] \\ \wedge d \notin rcvd\_by\_d^{-1}[\{m\}] \\ \wedge m \notin (rcvd \cup lost) \end{array} \right\}$$

$D(m)$ est un ensemble contenant les destinataires du message m (non perdu et non reçu globalement), vers lesquels il a été envoyé, mais qui ne l'ont pas encore reçu individuellement. Nous posons $ND(m,n) \widehat{=} (card(D(m)) = n)$, où $n \in \mathbb{N}$.

L'ensemble Φ_{11} des propriétés caractérisant M11 est donc :

$$\Phi_{11} \triangleq \left\{ \begin{array}{l} P_{11} \leadsto (\exists n \cdot n > 0 \wedge ND(m,n)), ND(m,n+1) \leadsto ND(m,n), \\ R_{11} \leadsto (\exists n \cdot n \in \mathbb{N} \wedge ND(m,n)), (\exists n \cdot n \in \mathbb{N} \wedge ND(m,n)) \leadsto ND(m,0) \\ (ND(m,0) \wedge m \notin (rcvd \cup lost)) \leadsto Q \end{array} \right\}$$

Démonstration. Nous montrons que M11 satisfait les propriétés de Φ_{11}.

⟨1⟩1. Nous montrons que M11 satisfait $P_{11} \leadsto (\exists n \cdot n > 0 \wedge ND(m,n))$. Nous notons $\gamma \triangleq (\exists n \cdot n > 0 \wedge ND(m,n))$. Cette propriété exprime qu'il existe un nombre n non-nul de destinataires d'un message m non perdu, ne l'ayant pas encore reçu. Une hypothèse d'équité faible est posée sur l'événement SENDING1 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de SENDING1 sont observables lorsque P_{11} est vrai, leurs observations ne falsifient pas le prédicat P_{11}, condition d'observation de SENDING1, ou conduisent à γ.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$P_{11} \wedge [\mathsf{NEXT}]_{x_{11}} \Rightarrow (P'_{11} \vee \gamma')$$

Soit :
— $P_{11} \wedge BA(\mathsf{SENDING1})(x_{11}, {x_{11}}') \Rightarrow (P'_{11} \vee \gamma')$
— $P_{11} \wedge BA(\mathsf{RESENDING1})(x_{11}, {x_{11}}') \Rightarrow (P'_{11} \vee \gamma')$
— $P_{11} \wedge BA(\mathsf{D\_RECEIVING1})(x_{11}, {x_{11}}') \Rightarrow (P'_{11} \vee \gamma')$
— $P_{11} \wedge BA(\mathsf{RECEIVING1})(x_{11}, {x_{11}}') \Rightarrow (P'_{11} \vee \gamma')$
— $P_{11} \wedge BA(\mathsf{LOSING1})(x_{11}, {x_{11}}') \Rightarrow (P'_{11} \vee \gamma')$
— $P_{11} \wedge (x_{11} = {x_{11}}') \Rightarrow (P'_{11} \vee \gamma')$
La propriété suivante **(3)** : $P_{11}(x_{11}) \wedge BA(\mathsf{SENDING1})(x_{11}, {x_{11}}') \Rightarrow \gamma({x_{11}}')$, et la condition de faisabilité **(4)** : $P_{11}(x_{11}) \Rightarrow (\exists {x_{11}}' \cdot BA(\mathsf{SENDING1})(x_{11}, {x_{11}}'))$ sont satisfaites par l'événement SENDING1. **(3)** nous permet de déduire $P_{11} \wedge \langle \mathsf{NEXT} \wedge \mathsf{SENDING1} \rangle_{x_{11}} \Rightarrow \gamma'$ et **(4)** permet de déduire $P_{11} \Rightarrow ENABLED\langle \mathsf{SENDING1} \rangle_{x_{11}}$, où $ENABLED\langle \mathsf{SENDING1} \rangle_{x_{11}} \triangleq (\exists y \cdot BA(\mathsf{SENDING1})(x_{11}, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M11}) \vdash P_{11} \leadsto \gamma$, donc que M11 satisfait $P_{11} \leadsto \gamma$. □

⟨1⟩2. Nous montrons que M11 satisfait $R_{11} \leadsto (\exists n \cdot n \in \mathbb{N} \wedge ND(m,n))$. Nous notons $\gamma \triangleq (\exists n \cdot n \in \mathbb{N} \wedge ND(m,n))$. Cette propriété exprime qu'il existe un nombre n de destinataires d'un message m non perdu, ne l'ayant pas encore reçu. Une hypothèse d'équité faible est posée sur l'événement RESENDING1 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de RESENDING1 sont observables lorsque R_{11} est vrai, leurs observations ne falsifient pas le prédicat R_{11}, condition d'observation de RESENDING1, ou conduisent à γ.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$R_{11} \wedge [\mathsf{NEXT}]_{x_{11}} \Rightarrow (R'_{11} \vee \gamma')$$

Soit :
— $R_{11} \wedge BA(\mathsf{SENDING1})(x_{11}, {x_{11}}') \Rightarrow (R'_{11} \vee \gamma')$
— $R_{11} \wedge BA(\mathsf{RESENDING1})(x_{11}, {x_{11}}') \Rightarrow (R'_{11} \vee \gamma')$
— $R_{11} \wedge BA(\mathsf{D\_RECEIVING1})(x_{11}, {x_{11}}') \Rightarrow (R'_{11} \vee \gamma')$
— $R_{11} \wedge BA(\mathsf{RECEIVING1})(x_{11}, {x_{11}}') \Rightarrow (R'_{11} \vee \gamma')$
— $R_{11} \wedge BA(\mathsf{LOSING1})(x_{11}, {x_{11}}') \Rightarrow (R'_{11} \vee \gamma')$
— $R_{11} \wedge (x_{11} = {x_{11}}') \Rightarrow (R'_{11} \vee \gamma')$
La propriété **(3)** : $R_{11}(x_{11}) \wedge BA(\mathsf{RESENDING1})(x_{11}, {x_{11}}') \Rightarrow \gamma({x_{11}}')$, et la condition de faisabilité **(4)** : $R_{11}(x_{11}) \Rightarrow (\exists {x_{11}}' \cdot BA(\mathsf{RESENDING1})(x_{11}, {x_{11}}'))$ sont satisfaites par RESENDING1. **(3)** permet de déduire $R_{11} \wedge \langle \mathsf{NEXT} \wedge \mathsf{RESENDING1} \rangle_{x_{11}} \Rightarrow \gamma'$ et **(4)** permet de déduire $R_{11} \Rightarrow ENABLED\langle \mathsf{RESENDING1} \rangle_{x_{11}}$, où $ENABLED\langle \mathsf{RESENDING1} \rangle_{x_{11}} \triangleq (\exists y \cdot BA(\mathsf{RESENDING1})(x_{11}, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M11}) \vdash R_{11} \leadsto \gamma$, donc que M11 satisfait $R_{11} \leadsto \gamma$. □

⟨1⟩3. Nous montrons que M11 satisfait $ND(m,n+1) \leadsto ND(m,n)$. Nous notons $\gamma_1 \triangleq ND(m,n+1)$ et $\gamma_2 \triangleq ND(m,n)$. Pour montrer que M11 satisfait $\gamma_1 \leadsto \gamma_2$, nous allons montrer que M11 satisfait $(R_{11} \vee \gamma_1) \leadsto \gamma_2$. Une hypothèse d'équité forte est posée sur l'événement D_RECEIVING1 et nous nous trouvons dans le cas suivant **(1)** : un événement LOSING1 peut nous ramener, d'un état satisfaisant γ_1, à un état satisfaisant R_{11} (tel que $R_{11} \leadsto \gamma_1$), sinon les observations des autres

événements, différents de LOSING1 et D_RECEIVING1, ne falsifient pas $(R_{11} \vee \gamma_1)$ ou peuvent nous conduire à γ_2.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(R_{11} \vee \gamma_1) \wedge [\mathsf{NEXT}]_{x_{11}} \Rightarrow ((R'_{11} \vee \gamma'_1) \vee \gamma'_2)$$

Soit :
— $(R_{11} \vee \gamma_1) \wedge BA(\mathsf{SENDING1})(x_{11}, x_{11}') \Rightarrow ((R'_{11} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{11} \vee \gamma_1) \wedge BA(\mathsf{RESENDING1})(x_{11}, x_{11}') \Rightarrow ((R'_{11} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{11} \vee \gamma_1) \wedge BA(\mathsf{LOSING1})(x_{11}, x_{11}') \Rightarrow ((R'_{11} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{11} \vee \gamma_1) \wedge BA(\mathsf{D\_RECEIVING1})(x_{11}, x_{11}') \Rightarrow ((R'_{11} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{11} \vee \gamma_1) \wedge BA(\mathsf{RECEIVING1})(x_{11}, x_{11}') \Rightarrow ((R'_{11} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{11} \vee \gamma_1) \wedge (x_{11} = x_{11}') \Rightarrow ((R'_{11} \vee \gamma'_1) \vee \gamma'_2)$

La propriété **(3)** : $\gamma_1(x_{11}) \wedge BA(\mathsf{D\_RECEIVING1})(x_{11}, x_{11}') \Rightarrow \gamma_2(x_{11}')$, ainsi que la condition de faisabilité **(4)** : $\gamma_1(x_{11}) \Rightarrow (\exists x_{11}' \cdot BA(\mathsf{D\_RECEIVING1})(x_{11}, x_{11}'))$ sont satisfaites par D_RECEIVING1. **(3)** nous permet de déduire $(R_{11} \vee \gamma_1) \wedge \langle \mathsf{NEXT} \wedge \mathsf{D\_RECEIVING1} \rangle_{x_{11}} \Rightarrow \gamma'_2$. Les propriétés **(3)**, **(4)** et $R_{11} \rightsquigarrow \gamma_1$ nous permettent de déduire que tant que γ_2 n'est pas satisfait, alors l'événement D_RECEIVING1 sera fatalement activable, ce qui nous donne : $(\neg\gamma_2) \Rightarrow \Diamond ENABLED\langle \mathsf{D\_RECEIVING1} \rangle_{x_{11}}$, où $ENABLED\langle \mathsf{D\_RECEIVING1} \rangle_{x_{11}} \ \widehat{=} \ (\exists y \cdot BA(\mathsf{D\_RECEIVING1})(x_{11}, y))$. Nous déduisons, en tenant compte de $R_{11} \rightsquigarrow \gamma_1$: $\Box(R_{11} \vee \gamma_1) \wedge \Box[\mathsf{NEXT}]_{x_{11}} \Rightarrow \Diamond ENABLED\langle \mathsf{D\_RECEIVING1} \rangle_{x_{11}}$. L'application de la règle SF1 donne $(\mathcal{S}pec(\mathsf{M11}), R_{11} \rightsquigarrow \gamma_1) \vdash ((R_{11} \vee \gamma_1) \rightsquigarrow \gamma_2)$. La machine M11 satisfait $(R_{11} \vee \gamma_1) \rightsquigarrow \gamma_2$, donc $\gamma_1 \rightsquigarrow \gamma_2$. □

⟨1⟩4. Nous montrons que M11 satisfait $(\exists n \cdot n \in \mathbb{N} \wedge ND(m, n)) \rightsquigarrow ND(m, 0)$. Nous avons montré que M11 satisfait $ND(m, n+1) \rightsquigarrow ND(m, n)$. En prenant comme relation d'ordre bien fondée $>$ sur l'ensemble des entiers naturels $\mathbb{N}$, soit $(\mathbb{N}, >)$ comme structure bien fondée, nous appliquons la règle d'inférence treillis (LATTICE), pour démontrer la convergence vers $ND(m, 0)$. Nous montrons ainsi $\mathcal{S}pec(\mathsf{M11}) \vdash (\exists n \cdot n \in \mathbb{N} \wedge ND(m, n)) \rightsquigarrow ND(m, 0)$. □

⟨1⟩5. Nous montrons que M11 satisfait $(ND(m, 0) \wedge m \notin (rcvd \cup lost)) \rightsquigarrow Q$. Nous notons $\rho \ \widehat{=} \ (ND(m, 0) \wedge m \notin (rcvd \cup lost))$. Pour montrer que M11 satisfait $\rho \rightsquigarrow Q$, nous allons montrer $(R_{11} \vee \rho) \rightsquigarrow Q$.Une hypothèse d'équité forte est posée sur l'événement RECEIVING1 et nous nous trouvons dans le cas suivant **(1)** : un événement LOSING1 peut nous ramener, d'un état satisfaisant ρ, à un état satisfaisant R_{11} (tel que $R_{11} \rightsquigarrow \rho$), sinon les observations des autres événements, différents de LOSING1 et RECEIVING1, ne falsifient pas $(R_{11} \vee \rho)$ ou peuvent nous conduire à Q.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(R_{11} \vee \rho) \wedge [\mathsf{NEXT}]_{x_{11}} \Rightarrow ((R'_{11} \vee \rho') \vee Q')$$

Soit :
— $(R_{11} \vee \rho) \wedge BA(\mathsf{SENDING1})(x_{11}, x_{11}') \Rightarrow ((R'_{11} \vee \rho') \vee Q')$
— $(R_{11} \vee \rho) \wedge BA(\mathsf{RESENDING1})(x_{11}, x_{11}') \Rightarrow ((R'_{11} \vee \rho') \vee Q')$
— $(R_{11} \vee \rho) \wedge BA(\mathsf{LOSING1})(x_{11}, x_{11}') \Rightarrow ((R'_{11} \vee \rho') \vee Q')$
— $(R_{11} \vee \rho) \wedge BA(\mathsf{D\_RECEIVING1})(x_{11}, x_{11}') \Rightarrow ((R'_{11} \vee \rho') \vee Q')$
— $(R_{11} \vee \rho) \wedge BA(\mathsf{RECEIVING1})(x_{11}, x_{11}') \Rightarrow ((R'_{11} \vee \rho') \vee Q')$
— $(R_{11} \vee \rho) \wedge (x_{11} = x_{11}') \Rightarrow ((R'_{11} \vee \rho') \vee Q')$

La propriété **(3)** : $\rho(x_{11}) \wedge BA(\mathsf{RECEIVING1})(x_{11}, x_{11}') \Rightarrow Q(x_{11}')$, ainsi que la condition de faisabilité **(4)** : $\rho(x_{11}) \Rightarrow (\exists x_{11}' \cdot BA(\mathsf{RECEIVING1})(x_{11}, x_{11}'))$ sont satisfaites par RECEIVING1. **(3)** nous permet de déduire $(R_{11} \vee \rho) \wedge \langle \mathsf{NEXT} \wedge \mathsf{RECEIVING1} \rangle_{x_{11}} \Rightarrow Q'$. Les propriétés **(3)**, **(4)** et $R_{11} \rightsquigarrow \rho$ nous permettent de déduire que tant que Q n'est pas satisfait, alors l'événement RECEIVING1 sera fatalement activable, ce qui nous donne : $(\neg Q) \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING1} \rangle_{x_{11}}$, où nous avons $ENABLED\langle \mathsf{RECEIVING1} \rangle_{x_{11}} \ \widehat{=} \ (\exists y \cdot BA(\mathsf{RECEIVING1})(x_{11}, y))$. Nous déduisons, en tenant compte de $R_{11} \rightsquigarrow \rho$: $\Box(R_{11} \vee \rho) \wedge \Box[\mathsf{NEXT}]_{x_{11}} \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING1} \rangle_{x_{11}}$. L'application de la règle SF1 donne $(\mathcal{S}pec(\mathsf{M11}), R_{11} \rightsquigarrow \rho) \vdash ((R_{11} \vee \rho) \rightsquigarrow Q)$. La machine M11 satisfait $(R_{11} \vee \rho) \rightsquigarrow Q$, donc $\rho \rightsquigarrow Q$. □

⟨1⟩6. Les étapes ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4 et ⟨1⟩5 nous permettent de déduire que M11 satisfait les propriétés de vivacité de Φ_{11}. QED.

□

Soit Φ_0 l'ensemble des propriétés de vivacité caractérisant M00 :

$$\Phi_0 \ \widehat{=} \ \{P \rightsquigarrow S, R \rightsquigarrow S, S \rightsquigarrow Q\}$$

Démonstration. Nous montrons que M11 satisfait les propriétés de Φ_0. Nous établissons auparavant quelques équivalences :

— $P_{11} \Leftrightarrow P$. Nous avons :

$$P_{11} \ \widehat{=} \ \left(\begin{array}{l} \wedge s \in NODES \\ \wedge m \in MSG \\ \wedge s = src(m) \\ \wedge s \mapsto m \notin sent\_by\_s \end{array} \right)$$

et

$$P \ \widehat{=} \ (m \in MSG \wedge m \notin sent)$$

Les invariants $inv4$, $inv5$ et $inv6$ nous permettent de déduire $P_{11} \Leftrightarrow P$.

— $R_{11} \Leftrightarrow R$. Nous avons :

$$R_{11} \ \widehat{=} \ (m \in ran(sent\_by\_s) \wedge m \in lost)$$

et

$$R \ \widehat{=} \ (m \in sent \wedge m \in lost)$$

Les invariants $inv4$, $inv5$ et $inv6$ nous permettent de déduire $R_{11} \Leftrightarrow R$.

— $S \Leftrightarrow \left(\begin{array}{l} \vee(\exists n \cdot n > 0 \wedge ND(m,n)) \\ \vee(ND(m,0) \wedge m \notin (rcvd \cup lost)) \end{array} \right)$.

La propriété :

$$S \ \widehat{=} \ \left(\begin{array}{l} \wedge m \in sent \\ \wedge m \notin (rcvd \cup lost) \end{array} \right)$$

et

la propriété :

$$\Gamma \ \widehat{=} \ \left(\begin{array}{l} \vee(\exists n \cdot n > 0 \wedge ND(m,n)) \\ \vee(ND(m,0) \wedge m \notin (rcvd \cup lost)) \end{array} \right)$$

expriment qu'un message m a été envoyé par une source aux destinataires du message et que ce dernier n'a pas encore été marqué comme étant reçu globalement : **(1)** tous les destinataires ne l'ont pas encore reçu individuellement ou **(2)** s'ils l'ont tous reçu individuellement, il n'a pas encore été marqué comme faisant partie de $rcvd$. Nous pouvons alors établir $S \Leftrightarrow \Gamma$, avec Γ une décomposition de S selon deux cas possibles **(1)** et **(2)**.

⟨1⟩1. Nous montrons que M11 satisfait $P \rightsquigarrow S$. Nous savons que M11 satisfait $P_{11} \rightsquigarrow (\exists n \cdot n > 0 \wedge ND(m,n))$; et les équivalences $P_{11} \Leftrightarrow P$ et $\Gamma \Leftrightarrow S$ nous permettent de déduire que M11 satisfait $P \rightsquigarrow S$.

⟨1⟩2. Nous montrons que M11 satisfait $R \rightsquigarrow S$. Nous savons que M01 satisfait $R_{11} \rightsquigarrow \Gamma$ (étape ⟨1⟩3 de la démonstration précédente) ; et les équivalences $R_{11} \Leftrightarrow R$ et $\Gamma \Leftrightarrow S$ nous permettent de déduire que M11 satisfait $R \rightsquigarrow S$.

⟨1⟩3. Nous montrons que M01 satisfait $S \rightsquigarrow Q$. Les étapes ⟨1⟩3, ⟨1⟩4 et ⟨1⟩5, ainsi que l'équivalence $S \Leftrightarrow \Gamma$ nous permettent de déduire que M11 satisfait $S \rightsquigarrow Q$.

⟨1⟩4. Les étapes ⟨1⟩1, ⟨1⟩2 et ⟨1⟩3 nous permettent de déduire que M11 satisfait les propriétés de vivacité de Φ_0. QED.

□

Nous guidons les raffinements des événements SENDING0, RESENDING0, RECEIVING0, LOSING0, ainsi que l'ajout du nouvel événement D_RECEIVING1 à l'aide des propriétés de Φ_{11}, pour prendre en compte l'identification des sources et des destinataires des messages (Nous ne présentons ici que les événements ayant changé suite au raffinement et les nouveaux événements) :

— L'événement SENDING1 raffine SENDING0. Un message m est diffusé ($act1$) par sa source s ($grd4$), à un groupe grd de destinataires ($act2$) contenant tous les destinataire $dst(m)$ de m ($grd6$), s'il n'a pas encore été diffusé par sa source s ($grd5$).

EVENT SENDING1 REFINES SENDING0 $\widehat{=}$
ANY
s, m, grd
WHERE
$grd1$: $s \in NODES$
$grd2$: $m \in MSG$
$grd3$: $grd \subseteq NODES$
$grd4$: $s = src(m)$
$grd5$: $s \mapsto m \notin sent\_by\_s$
$grd6$: $grd = dst(m)$
THEN
$act1$: $sent\_by\_s := sent\_by\_s \cup \{s \mapsto m\}$
$act2$: $m\_dst := m\_dst \cup (grd \times \{m\})$
END

— L'événement RESENDING1 raffine RESENDING0. Un message m est renvoyé par une source s et après ce renvoi, le message m est ôté de la liste des messages perdus, s'il a été perdu ($act1$).

EVENT RESENDING1 REFINES RESENDING0 $\widehat{=}$
ANY
s, m
WHERE
$grd1$: $s \in NODES$
$grd2$: $m \in MSG$
$grd3$: $s \mapsto m \in sent\_by\_s$
$grd4$: $m \notin lost$
THEN
$act1$: $lost := lost \setminus \{m\}$
END

— Un nouvel événement D_RECEIVING1 est introduit. Il modélise la réception individuel d'un message m par un de ses destinataires d : le message m est reçu par le destinataire d ($act1$), s'il a été acheminé vers ce dernier ($grd3$), s'il n'a pas encore été reçu par d ($grd4$) et s'il n'est pas perdu ($grd5$).

EVENT D_RECEIVING1 $\widehat{=}$
ANY
d, m
WHERE
$grd1$: $d \in NODES$
$grd2$: $m \in MSG$
$grd3$: $d \mapsto m \in m\_dst$
$grd4$: $d \mapsto m \notin rcvd\_by\_d$
$grd5$: $m \notin lost$
THEN
$act1$: $rcvd\_by\_d := rcvd\_by\_d \cup \{d \mapsto m\}$
END

— L'événement RECEIVING1 observe le fait qu'un message m a été reçu par tous ses destinataires ($act1$) : il est observé si le message m a été envoyé par une source ($grd2$), s'il n'a pas encore été marqué comme reçu globalement par tous ses destinataires, s'il n'est pas perdu ($grd3$) et si tous ses destinataires l'ont reçu individuellement ($grd4$).

EVENT RECEIVING1 REFINES RECEIVING0 $\widehat{=}$
ANY
m
WHERE
$grd1$: $m \in MSG$
$grd2$: $m \in ran(sent\_by\_s)$
$grd3$: $m \notin (rcvd \cup lost)$
$grd4$: $dst(m) \times \{m\} \subseteq rcvd\_by\_d$
THEN
$act1$: $rcvd := rcvd \cup \{m\}$
END

— L'événement LOSING1 raffine LOSING0. Un message m peut être perdu, s'il n'est pas encore considéré comme étant reçu globalement par tous ses destinataires et s'il n'est pas encore perdu.

EVENT LOSING1 REFINES LOSING0 $\widehat{=}$
ANY
m
WHERE
$grd1$: $m \in MSG$
$grd2$: $m \in ran(sent\_by\_s)$
$grd3$: $m \notin (rcvd \cup lost)$
THEN
$act1$: $lost := lost \cup \{m\}$
END

Nous pouvons résumer ce raffinement M11 par le diagramme 6.11 suivant :

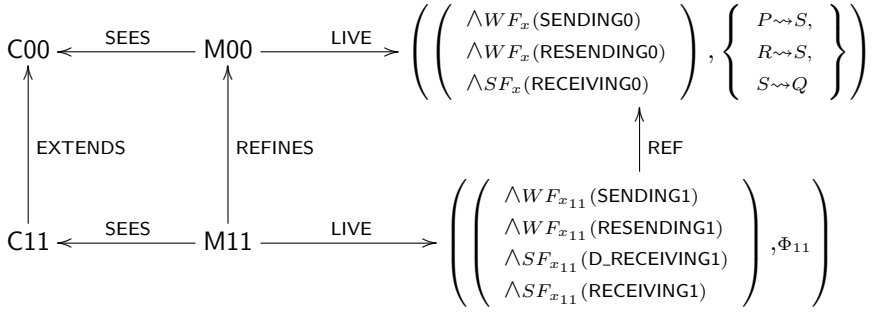


FIGURE 6.11 – Patron (groupe de destinataires) : premier raffinement

Le raffinement suivant introduit le système réparti support du routage, aussi bien pour le cas analysé ici (destinataires multiples d'un message) que pour l'autre cas existant (unique destinataire pour un message).

6.2.1.4 Second raffinement : introduction du réseau

Nous nous intéressons maintenant au système réparti support du routage. Nous l'introduisons dans ce second raffinement, aussi bien pour le cas dans lequel un message possède un destinataire unique, que pour le cas où il possède des destinataires multiples.

Cas où le destinataire d'un message est unique. Nous introduisons pour ce second raffinement un contexte C02 étendant le contexte C01 vu précédemment.

CONTEXT C02 EXTENDS C01
CONSTANTS
GR, g
AXIOMS
$axm1$: $GR = \left\{ gr \mid \left(\begin{array}{l} \wedge gr \subseteq NODES \times NODES \\ \wedge gr \neq \varnothing \\ \wedge NODES \lhd id \cap gr = \varnothing \\ \wedge gr^{-1} = gr \\ \wedge \ / * \ autres\ propriétés\ de\ gr * / \end{array} \right) \right\}$
$axm2$: $g \in GR$
END

— Ce contexte définit un ensemble GR des graphes gr possibles pour le raffinement et décrit leurs propriétés ($axm2$) : nous nous sommes focalisés ici sur le fait que gr relie des nœuds, est non-vide, n'est pas réflexif, mais est symétrique. Ces propriétés peuvent être complétées et même modifiées si besoin.
— Il introduit aussi une constante g qui représente le réseau original, respectant les propriétés décrites par GR.

Il est utilisé par la machine M02 raffinement de M01.

Le raffinement M02 introduit de nouvelles variables :

INITIALISATION $\widehat{=}$
BEGIN
$\ldots$
$\ominus\ lost := \varnothing$
$\oplus\ actg1\ :\ graph := g$
$\oplus\ actg2\ :\ store := \varnothing$
END

$inv1\ :\ graph \in NODES \leftrightarrow NODES$
$inv2\ :\ store \in NODES \leftrightarrow MSG$

— *graph* représente le réseau courant ($inv1$) et elle est initialisée à l'aide de la constante g.
— *store* permet de savoir dans quel(s) nœud(s) du réseau courant se trouve un message qui a été diffusé par sa source ($inv2$). Elle est initialisée à l'aide de l'ensemble vide ($\varnothing$).

Nous notons x_2 les variables de cette machine M02.

Des invariants contraignent ces variables :

$inv3\ :\ graph \in GR$
$inv4\ :\ ran(store) \subseteq ran(sent\_by\_s)$
$inv5\ :\ ran(store) \cap lost = \varnothing$

— $inv3$ exprime que le graphe courant est toujours un graphe vérifiant les propriétés définies par l'ensemble des graphes possibles GR.
— $inv4$ exprime que chaque message circulant dans le réseau a été envoyé par une source.
— $inv5$ exprime qu'un message envoyé est soit circulant dans le réseau, soit perdu. Nous avons ici un invariant de collage qui nous permet de supprimer la variable abstraite *lost* du modèle M02.

La machine M02 contient les raffinements des événements de la machine M01, soit SENDING2, RESENDING2, RECEIVING2 et LOSING2, ainsi qu'un nouvel événement FORWARD2, modélisant l'acheminement d'un message dans le système réparti *graph*. L'hypothèse d'équité L_2 que nous posons sur le modèle M02 est la suivante :

$$\begin{aligned} L_2 \quad \widehat{=} \quad & \wedge WF_{x_2}(\mathsf{SENDING2}) \\ & \wedge WF_{x_2}(\mathsf{RESENDING2}) \\ & \wedge SF_{x_2}(\mathsf{FORWARD2}) \\ & \wedge SF_{x_2}(\mathsf{RECEIVING2}) \end{aligned}$$

Nous posons des hypothèses d'équité faible sur les événements SENDING2, RESENDING2 et des hypothèses d'équité forte sur les événements FORWARD2 et RECEIVING2. Par contre, nous ne posons aucune hypothèse d'équité sur l'événement LOSING2, qui représente une action de l'environnement (perte de messages).

Nous définissons aussi la relation NEXT :

$$\begin{aligned} \mathsf{NEXT} \ \widehat{=} \quad & \vee BA(\mathsf{SENDING2})(x_2, {x_2}') \\ & \vee BA(\mathsf{LOSING2})(x_2, {x_2}') \\ & \vee BA(\mathsf{RESENDING2})(x_2, {x_2}') \\ & \vee BA(\mathsf{FORWARD2})(x_2, {x_2}') \\ & \vee BA(\mathsf{RECEIVING2})(x_2, {x_2}') \\ & \vee (x_2 = {x_2}') \end{aligned}$$

Nous nous intéressons maintenant aux propriétés de vivacité satisfaites par la machine M02. Pour cela nous commençons par poser :

— $R_2 \ \widehat{=}\ (minran(sent\_by\_s) \wedge m \notin ran(store) \wedge m \notin ran(rcvd\_by\_d))$,
Cette propriété exprime qu'un message m a été envoyé par une source et est perdu : il n'a pas encore été reçu par son destinataire et il ne circule pas dans le réseau.
— Nous posons $dist(x, d, m) \in \mathbb{N}$, la distance entre la position courante x du message m non-perdu ($m \in ran(store)$) et sa destination d : il s'agit du nombre de nœuds entre x et d. Nous posons $dist(x, d, m, i) \ \widehat{=}\ (dist(x, d, m) = i)$, où $i \in \mathbb{N}$.

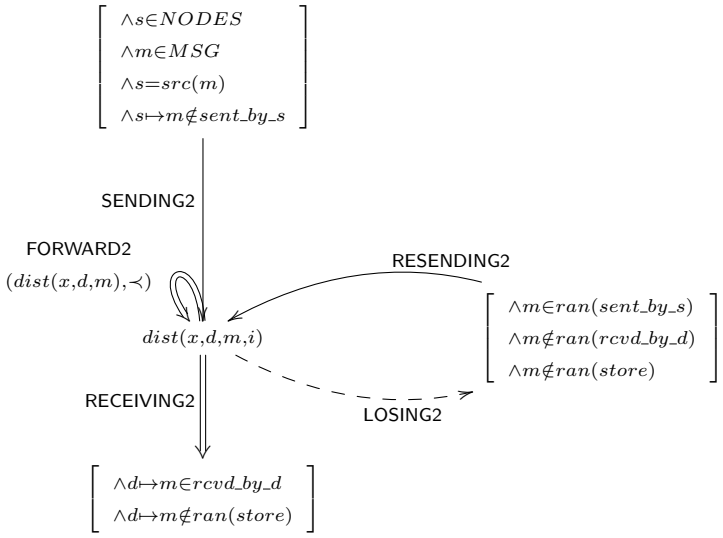


FIGURE 6.12 – Diagramme d'assertions pour le second raffinement M02

— $Q_2 \triangleq (d \mapsto m \in rcvd\_by\_d \wedge d \mapsto m \notin ran(store))$, exprimant qu'un message m est reçu par son destinataire d.

L'ensemble Φ_2 des propriétés caractérisant M2 est donc :

$$\Phi_2 \triangleq \left\{ \begin{array}{l} P_1 \rightsquigarrow (\exists i \cdot i > 0 \wedge dist(x,d,m,i)), dist(x,d,m,i+1) \rightsquigarrow dist(x,d,m,i), \\ R_2 \rightsquigarrow (\exists i \cdot i \in \mathbb{N} \wedge dist(x,d,m,i)), (\exists i \cdot i \in \mathbb{N} \wedge dist(x,d,m,i)) \rightsquigarrow dist(x,d,m,0) \\ (dist(x,d,m,0) \wedge d \mapsto m \notin rcvd\_by\_d \wedge d \mapsto m \in store) \rightsquigarrow Q_2 \end{array} \right\}$$

Démonstration. Nous montrons que M2 satisfait les propriétés de Φ_2.

⟨1⟩1. Nous montrons que M2 satisfait $P_1 \rightsquigarrow (\exists i \cdot i > 0 \wedge dist(x,d,m,i))$. Nous notons $\gamma \triangleq (\exists i \cdot i > 0 \wedge dist(x,d,m,i))$. Cette propriété exprime qu'il existe une distance i non-nulle entre la position courante x du message m et son destinataire d, qui ne l'a donc pas encore reçu. Une hypothèse d'équité faible est posée sur l'événement SENDING2 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de SENDING2 sont observables lorsque P_1 est vrai, leurs observations ne falsifient pas le prédicat P_1, condition d'observation de SENDING2, ou conduisent à γ.

Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$P_1 \wedge [\mathsf{NEXT}]_{x_2} \Rightarrow (P_1' \vee \gamma')$$

Soit :

— $P_1 \wedge BA(\mathsf{SENDING2})(x_2, x_2') \Rightarrow (P_1' \vee \gamma')$
— $P_1 \wedge BA(\mathsf{RESENDING2})(x_2, x_2') \Rightarrow (P_1' \vee \gamma')$
— $P_1 \wedge BA(\mathsf{FORWARD2})(x_2, x_2') \Rightarrow (P_1' \vee \gamma')$
— $P_1 \wedge BA(\mathsf{RECEIVING2})(x_2, x_2') \Rightarrow (P_1' \vee \gamma')$
— $P_1 \wedge BA(\mathsf{LOSING2})(x_2, x_2') \Rightarrow (P_1' \vee \gamma')$
— $P_1 \wedge (x_2 = x_2') \Rightarrow (P_1' \vee \gamma')$

La propriété suivante **(3)** : $P_1(x_2) \wedge BA(\mathsf{SENDING2})(x_2, x_2') \Rightarrow \gamma(x_2')$, et la condition de faisabilité **(4)** : $P_1(x_2) \Rightarrow (\exists x_2' \cdot BA(\mathsf{SENDING2})(x_2, x_2'))$ sont satisfaites par l'événement SENDING2. **(3)** nous permet de déduire $P_1 \wedge \langle \mathsf{NEXT} \wedge \mathsf{SENDING2} \rangle_{x_2} \Rightarrow \gamma'$ et **(4)** permet de déduire $P_1 \Rightarrow ENABLED\langle \mathsf{SENDING2} \rangle_{x_2}$, où $ENABLED\langle \mathsf{SENDING2} \rangle_{x_2} \triangleq (\exists y \cdot BA(\mathsf{SENDING2})(x_2, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M02}) \vdash P_1 \rightsquigarrow \gamma$, donc que M02 satisfait $P_1 \rightsquigarrow \gamma$. □

⟨1⟩2. Nous montrons que M02 satisfait $R_2 \rightsquigarrow (\exists i \cdot i \in \mathbb{N} \wedge dist(x,d,m,i))$. Nous notons $\gamma \triangleq (\exists i \cdot i \in \mathbb{N} \wedge dist(x,d,m,i))$. Cette propriété exprime qu'il existe une distance i entre la position courante

x du message m et son destinataire d. Une hypothèse d'équité faible est posée sur l'événement RESENDING2 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de RESENDING2 sont observables lorsque R_2 est vrai, leurs observations ne falsifient pas le prédicat R_2, condition d'observation de RESENDING2, ou conduisent à γ.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$R_2 \wedge [\mathsf{NEXT}]_{x_2} \Rightarrow (R_2' \vee \gamma')$$

Soit :
— $R_2 \wedge BA(\mathsf{SENDING2})(x_2, {x_2}') \Rightarrow (R_2' \vee \gamma')$
— $R_2 \wedge BA(\mathsf{RESENDING2})(x_2, {x_2}') \Rightarrow (R_2' \vee \gamma')$
— $R_2 \wedge BA(\mathsf{FORWARD2})(x_2, {x_2}') \Rightarrow (R_2' \vee \gamma')$
— $R_2 \wedge BA(\mathsf{RECEIVING2})(x_2, {x_2}') \Rightarrow (R_2' \vee \gamma')$
— $R_2 \wedge BA(\mathsf{LOSING2})(x_2, {x_2}') \Rightarrow (R_2' \vee \gamma')$
— $R_2 \wedge (x_2 = {x_2}') \Rightarrow (R_2' \vee \gamma')$

La propriété **(3)** : $R_2(x_2) \wedge BA(\mathsf{RESENDING2})(x_2, {x_2}') \Rightarrow \gamma({x_2}')$, et la condition de faisabilité **(4)** : $R_2(x_2) \Rightarrow (\exists {x_2}' \cdot BA(\mathsf{RESENDING2})(x_2, {x_2}'))$ sont satisfaites par RESENDING2. La propriété **(3)** nous permet de déduire $R_2 \wedge \langle \mathsf{NEXT} \wedge \mathsf{RESENDING2} \rangle_{x_2} \Rightarrow \gamma'$ et **(4)** permet de déduire $R_2 \Rightarrow ENABLED\langle \mathsf{RESENDING2} \rangle_{x_2}$, où $ENABLED\langle \mathsf{RESENDING1} \rangle_{x_2} \ \widehat{=}\ (\exists y \cdot BA(\mathsf{RESENDING2})(x_2, y))$.
L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M02}) \vdash R_2 \rightsquigarrow \gamma$, donc que M02 satisfait $R_2 \rightsquigarrow \gamma$. □

⟨1⟩3. Nous montrons que M02 satisfait $dist(x, d, m, i+1) \rightsquigarrow dist(x, d, m, i)$. Nous notons $\gamma_1 \ \widehat{=}\ dist(x, d, m, i+1)$ et $\gamma_2 \ \widehat{=}\ dist(x, d, m, i)$. Pour montrer que M02 satisfait $\gamma_1 \rightsquigarrow \gamma_2$, nous allons montrer que M02 satisfait $(R_2 \vee \gamma_1) \rightsquigarrow \gamma_2$. Une hypothèse d'équité forte est posée sur l'événement FORWARD2 et nous nous trouvons dans le cas suivant **(1)** : un événement LOSING2 peut nous ramener, d'un état satisfaisant γ_1, à un état satisfaisant R_2 (tel que $R_2 \rightsquigarrow \gamma_1$), sinon les observations des autres événements, différents de LOSING2 et FORWARD2, ne falsifient pas $(R_2 \vee \gamma_1)$ ou peuvent nous conduire à γ_2.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(R_2 \vee \gamma_1) \wedge [\mathsf{NEXT}]_{x_2} \Rightarrow ((R_2' \vee \gamma_1') \vee \gamma_2')$$

Soit :
— $(R_2 \vee \gamma_1) \wedge BA(\mathsf{SENDING2})(x_2, {x_2}') \Rightarrow ((R_2' \vee \gamma_1') \vee \gamma_2')$
— $(R_2 \vee \gamma_1) \wedge BA(\mathsf{RESENDING2})(x_2, {x_2}') \Rightarrow ((R_2' \vee \gamma_1') \vee \gamma_2')$
— $(R_2 \vee \gamma_1) \wedge BA(\mathsf{LOSING2})(x_2, {x_2}') \Rightarrow ((R_2' \vee \gamma_1') \vee \gamma_2')$
— $(R_2 \vee \gamma_1) \wedge BA(\mathsf{FORWARD2})(x_2, {x_2}') \Rightarrow ((R_2' \vee \gamma_1') \vee \gamma_2')$
— $(R_2 \vee \gamma_1) \wedge BA(\mathsf{RECEIVING2})(x_2, {x_2}') \Rightarrow ((R_2' \vee \gamma_1') \vee \gamma_2')$
— $(R_2 \vee \gamma_1) \wedge (x_2 = {x_2}') \Rightarrow ((R_2' \vee \gamma_1') \vee \gamma_2')$

La propriété **(3)** : $\gamma_1(x_2) \wedge BA(\mathsf{FORWARD2})(x_2, {x_2}') \Rightarrow \gamma_2({x_2}')$, ainsi que la condition de faisabilité **(4)** : $\gamma_1(x_2) \Rightarrow (\exists {x_2}' \cdot BA(\mathsf{FORWARD2})(x_2, {x_2}'))$ sont satisfaites par FORWARD2. **(3)** nous permet de déduire $(R_2 \vee \gamma_1) \wedge \langle \mathsf{NEXT} \wedge \mathsf{FORWARD2} \rangle_{x_2} \Rightarrow \gamma_2'$. Les propriétés **(3)**, **(4)** et $R_2 \rightsquigarrow \gamma_1$ nous permettent de déduire que tant que γ_2 n'est pas satisfait, alors l'événement FORWARD2 sera fatalement activable, ce qui nous donne la propriété : $(\neg\gamma_2) \Rightarrow \Diamond ENABLED\langle \mathsf{FORWARD2} \rangle_{x_2}$, où $ENABLED\langle \mathsf{FORWARD2} \rangle_{x_2} \ \widehat{=}\ (\exists y \cdot BA(\mathsf{FORWARD2})(x_2, y))$. Nous déduisons, en tenant compte de l'hypothèse $R_2 \rightsquigarrow \gamma_1$: $\Box(R_2 \vee \gamma_1) \wedge \Box[\mathsf{NEXT}]_{x_2} \Rightarrow \Diamond ENABLED\langle \mathsf{FORWARD2} \rangle_{x_2}$. L'application de la règle SF1 donne $(\mathcal{S}pec(\mathsf{M02}), R_2 \rightsquigarrow \gamma_1) \vdash ((R_2 \vee \gamma_1) \rightsquigarrow \gamma_2)$. La machine M02 satisfait $(R_2 \vee \gamma_1) \rightsquigarrow \gamma_2$, donc $\gamma_1 \rightsquigarrow \gamma_2$. □

⟨1⟩4. Nous montrons que M02 satisfait $(\exists i \cdot i \in \mathbb{N} \wedge dist(x, d, m, i)) \rightsquigarrow dist(x, d, m, 0)$. Nous avons montré que M02 satisfait $dist(x, d, m, i+1) \rightsquigarrow dist(x, d, m, i)$. En prenant comme relation d'ordre bien fondée $>$ sur l'ensemble des entiers naturels $\mathbb{N}$, soit $(\mathbb{N}, >)$ comme structure bien fondée, nous appliquons la règle d'inférence treillis (LATTICE), pour démontrer la convergence vers $dist(x, d, m, 0)$. Nous montrons ainsi $\mathcal{S}pec(\mathsf{M02}) \vdash (\exists i \cdot i \in \mathbb{N} \wedge dist(x, d, m, i)) \rightsquigarrow dist(x, d, m, 0)$. □

⟨1⟩5. Nous montrons que M02 satisfait $(dist(x, d, m, 0) \wedge d \mapsto m \notin rcvd\_by\_d \wedge d \mapsto m \in store) \rightsquigarrow Q_2$. Nous notons $\rho \ \widehat{=}\ (dist(x, d, m, 0) \wedge d \mapsto m \notin rcvd\_by\_d \wedge d \mapsto m \in store)$. Pour montrer que M02 satisfait $\rho \rightsquigarrow Q_2$, nous allons montrer $(R_2 \vee \rho) \rightsquigarrow Q_2$.Une hypothèse d'équité forte est posée sur l'événement RECEIVING2 et nous nous trouvons dans le cas suivant **(1)** : un événement LOSING2 peut nous ramener, d'un état satisfaisant ρ, à un état satisfaisant R_2 (tel que $R_2 \rightsquigarrow \rho$), sinon

les observations des autres événements, différents de LOSING2 et RECEIVING2, ne falsifient pas $(R_2 \vee \rho)$ ou peuvent nous conduire à Q_2.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(R_2 \vee \rho) \wedge [\mathsf{NEXT}]_{x_2} \Rightarrow ((R_2' \vee \rho') \vee Q_2')$$

Soit :
— $(R_2 \vee \rho) \wedge BA(\mathsf{SENDING1})(x_2, x_2') \Rightarrow ((R_2' \vee \rho') \vee Q_2')$
— $(R_2 \vee \rho) \wedge BA(\mathsf{RESENDING1})(x_2, x_2') \Rightarrow ((R_2' \vee \rho') \vee Q_2')$
— $(R_2 \vee \rho) \wedge BA(\mathsf{LOSING1})(x_2, x_2') \Rightarrow ((R_2' \vee \rho') \vee Q_2')$
— $(R_2 \vee \rho) \wedge BA(\mathsf{D\_RECEIVING1})(x_2, x_2') \Rightarrow ((R_2' \vee \rho') \vee Q_2')$
— $(R_2 \vee \rho) \wedge BA(\mathsf{RECEIVING1})(x_2, x_2') \Rightarrow ((R_2' \vee \rho') \vee Q_2')$
— $(R_2 \vee \rho) \wedge (x_2 = x_2') \Rightarrow ((R_2' \vee \rho') \vee Q_2')$
La propriété **(3)** : $\rho(x_2) \wedge BA(\mathsf{RECEIVING2})(x_2, x_2') \Rightarrow Q_2(x_2')$, ainsi que la condition de faisabilité **(4)** : $\rho(x_2) \Rightarrow (\exists x_2' \cdot BA(\mathsf{RECEIVING2})(x_2, x_2'))$ sont satisfaites par RECEIVING2. **(3)** nous permet de déduire $(R_2 \vee \rho) \wedge \langle \mathsf{NEXT} \wedge \mathsf{RECEIVING2} \rangle_{x_2} \Rightarrow Q_2'$. **(3)**, **(4)** et $R_2 \rightsquigarrow \rho$ permettent de déduire que tant que Q_2 n'est pas satisfait, alors RECEIVING2 sera fatalement activable, ce qui nous donne : $(\neg Q_2) \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING2} \rangle_{x_2}$, où $ENABLED\langle \mathsf{RECEIVING2} \rangle_{x_2} \mathrel{\widehat{=}} (\exists y \cdot BA(\mathsf{RECEIVING2})(x_2, y))$. Nous déduisons, en tenant compte de $R_2 \rightsquigarrow \rho : \Box(R_2 \vee \rho) \wedge \Box[\mathsf{NEXT}]_{x_2} \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING2} \rangle_{x_2}$. L'application de la règle SF1 donne $(\mathcal{S}pec(\mathsf{M02}), R_2 \rightsquigarrow \rho) \vdash ((R_2 \vee \rho) \rightsquigarrow Q_2)$. La machine M02 satisfait $(R_2 \vee \rho) \rightsquigarrow Q_2$, donc $\rho \rightsquigarrow Q_2$. □
⟨1⟩6. Les étapes ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4 et ⟨1⟩5 nous permettent de déduire que M02 satisfait les propriétés de vivacité de Φ_2. QED.

□

Soit Φ_1 l'ensemble des propriétés de vivacité caractérisant M01 :

$$\Phi_1 \mathrel{\widehat{=}} \{P_1 \rightsquigarrow S_1, R_1 \rightsquigarrow S_1, S_1 \rightsquigarrow Q_1\}$$

Démonstration. Nous montrons que M02 satisfait les propriétés de Φ_1. Nous établissons auparavant quelques équivalences :
— $R_1 \Leftrightarrow R_2$. Nous avons :

$$R_1 \mathrel{\widehat{=}} (m \in ran(sent\_by\_s) \wedge m \in lost)$$

et

$$R_2 \mathrel{\widehat{=}} (m \in ran(sent\_by\_s) \wedge m \notin ran(store) \wedge m \notin ran(rcvd\_by\_d))$$

Les invariants $inv4$, $inv5$, $inv6$ de M02, $inv11$ de M01 et $inv4$ de M00 et nous permettent de déduire $R_1 \Leftrightarrow R_2$.
— $S_1 \Leftrightarrow \begin{pmatrix} \vee(\exists n \cdot n > 0 \wedge ND(m, n)) \\ \vee(ND(m, 0) \wedge m \notin (rcvd \cup lost)) \end{pmatrix}$.
La propriété :

$$S_1 \mathrel{\widehat{=}} \begin{pmatrix} \wedge d \in NODES \\ \wedge d = dst(m) \\ \wedge s \mapsto m \in sent\_by\_s \\ \wedge d \mapsto m \notin rcvd\_by\_d \\ \wedge m \notin lost \end{pmatrix}$$

et
la propriété :

$$\Gamma \mathrel{\widehat{=}} \begin{pmatrix} \vee(\exists i \cdot i > 0 \wedge dist(x, d, m, i)) \\ \vee(dist(x, d, m, 0) \wedge d \mapsto m \notin rcvd\_by\_d \wedge d \mapsto m \in store) \end{pmatrix}$$

expriment qu'un message m a été envoyé par une source au destinataire d du message et que ce dernier n'a pas encore été reçu : **(1)** le message m est à une distance i non-nulle du destinataire d ou **(2)** il est dans le réseau au niveau de d (à une distance 0 de d), mais d ne l'a pas encore marqué comme reçu. Nous pouvons alors établir $S_1 \Leftrightarrow \Gamma$, avec Γ une décomposition de S_1 selon deux cas possibles **(1)** et **(2)**.

⟨1⟩1. Nous montrons que M02 satisfait $P_1 \rightsquigarrow S_1$. Nous savons que M02 satisfait $P_1 \rightsquigarrow (\exists i \cdot i > 0 \wedge dist(x, d, m, i))$; et l'équivalence $\Gamma \Leftrightarrow S_1$ nous permet de déduire que M11 satisfait $P_1 \rightsquigarrow S_1$.
⟨1⟩2. Nous montrons que M02 satisfait $R_1 \rightsquigarrow S_1$. Nous savons que M02 satisfait $R_2 \rightsquigarrow \Gamma$ (étape ⟨1⟩3 de la démonstration précédente) ; et les équivalences $R_2 \Leftrightarrow R_1$ et $\Gamma \Leftrightarrow S_1$ nous permettent de déduire que M11 satisfait $R_2 \rightsquigarrow S_1$.
⟨1⟩3. Nous montrons que M01 satisfait $S_1 \rightsquigarrow Q_1$. Les étapes ⟨1⟩3, ⟨1⟩4 et ⟨1⟩5, ainsi que l'équivalence $S_1 \Leftrightarrow \Gamma$ nous permettent de déduire que M02 satisfait $S_1 \rightsquigarrow Q_1$.
⟨1⟩4. Les étapes ⟨1⟩1, ⟨1⟩2 et ⟨1⟩3 nous permettent de déduire que M02 satisfait les propriétés de vivacité de Φ_1. QED.

□

Nous nous servons maintenant des propriétés de Φ_2 pour guider le raffinement de M01 par M02 (Nous ne présentons ici que les événements ayant changé suite au raffinement et les nouveaux événements) :

— Pour les raffinements des événements
$$< \mathsf{X} > \in \{\mathsf{SENDING}, \mathsf{RESENDING}\},$$
nous rajoutons juste le fait qu'à l'envoi, un message m est déposé par sa source s dans le réseau ($actg1$).

EVENT $< \mathsf{X} >2$ REFINES $< \mathsf{X} >1$ $\widehat{=}$
ANY
...
WHERE
...
THEN
...
$\oplus$ $actg1$: $store := store \cup \{s \mapsto m\}$
END

— Un nouvel événement FORWARD2 est introduit. Il modélise la transition ($act1$) d'un message m d'un nœud x à un voisin y ($grd5$, $grd6$, $grd7$) tant que le destinataire d du message n'est pas atteint ($grd8$, $grd9$).

EVENT FORWARD2 $\widehat{=}$
ANY
x, y, m, d
WHERE
$grd1 : x \in NODES$
$grd2 : y \in NODES$
$grd3 : d \in NODES$
$grd4 : m \in MSG$
$grd5 : x \mapsto m \in store$
$grd6 : y \mapsto m \notin store$
$grd7 : x \mapsto y \in graph$
$grd8 : d \in dst^{-1}[\{m\}]$
$grd9 : x \neq d$
THEN
$act1 : store := (store \setminus \{x \mapsto m\}) \cup \{y \mapsto m\}$
END

— L'événement RECEIVING1 est raffiné de telle manière qu'un message m n'est reçu par un destinataire d que s'il ne se trouve à son niveau dans le réseau ($grdg2$). Après la réception, le message m est supprimé du réseau ($actg1$).

EVENT RECEIVING2 REFINES RECEIVING1 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd3$
$\oplus$ $grdg1$: $d \mapsto m \notin rcvd\_by\_d$
$\oplus$ $grdg2$: $d \mapsto m \in store$
THEN
...
$\oplus$ $actg1$: $store := store \setminus \{d \mapsto m\}$
END

— L'événement LOSING2 est observé si un message m n'a pas encore été reçu par son destinataire et s'il se trouve encore dans le réseau. Dans ce cas, le message m est supprimé du réseau.

EVENT LOSING2 REFINES LOSING1 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd3$
$\oplus$ $grdg1$: $m \notin ran(rcvd\_by\_d)$
$\oplus$ $grdg2$: $m \in ran(store)$
THEN
...
$actg1$: $store := store \rhd \{m\}$
END

— Deux nouveaux événement ADD_LINK2 et REMOVE_LINK2 sont introduits :
  — ADD_LINK2 modélise l'ajout dans le graphe courant $graph$, de liens qui n'existaient pas auparavant entre des nœuds x et y ($grd4$), qui deviennent de ce fait adjacents ($act1$). L'événement ADD_LINK2 est observé si le graphe résultant respecte les conditions définies par GR ($grd5$).
  — REMOVE_LINK2 modélise la suppression ($act1$) dans le graphe courant $graph$, des liens existants ($grd3$) entre deux nœuds x et y. L'événement REMOVE_LINK2 est observé si le graphe résultant respecte les conditions définies par GR ($grd5$).

EVENT ADD_LINK2 $\widehat{=}$
ANY
x, y
WHERE
$grd1$: $x \in NODES$
$grd2$: $y \in NODES$
$grd3$: $x \neq y$
$grd4$: $x \mapsto y \notin graph$
$grd5$: $graph \cup \{x \mapsto y, y \mapsto x\} \in GR$
THEN
$act1$: $graph := graph \cup \{x \mapsto y, y \mapsto x\}$
END

EVENT REMOVE_LINK2 $\widehat{=}$
ANY
x, y
WHERE
$grd1$: $x \in NODES$
$grd2$: $y \in NODES$
$grd3$: $x \mapsto y \in graph$
$grd4$: $x \neq y$
$grd5$: $graph \setminus \{x \mapsto y, y \mapsto x\} \in GR$
THEN
$act1$: $graph := graph \setminus \{x \mapsto y, y \mapsto x\}$
END

Nous pouvons résumer ce raffinement M02 par le diagramme 6.13 suivant :

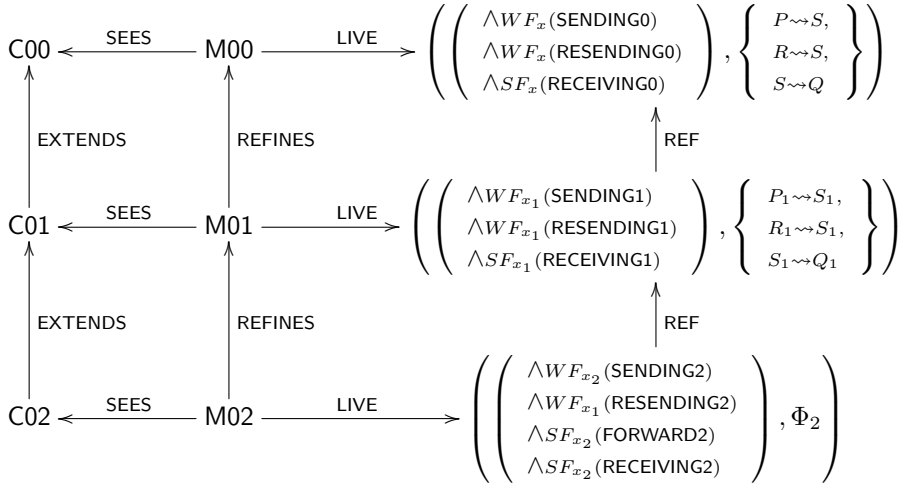


FIGURE 6.13 – Patron (destinataire unique) : deuxième raffinement

Nous traitons ensuite l'introduction du réseau, dans le cas où un message possède plusieurs destinataires.

Cas où un message est diffusé à un groupe de destinataires. Nous nous intéressons à l'introduction du routage, dans le cas où un message possède des destinataires multiples. Un contexte C12 est défini. Il reprend le contenu du contexte C02 vu précédemment :

— Il définit un ensemble GR des graphes gr possibles pour le raffinement et décrit leurs propriétés ($axm2$) : gr relie des nœuds, est non-vide, n'est pas réflexif, mais est symétrique. Ces propriétés peuvent être complétées et même modifiées si besoin.

— Il introduit aussi une constante g qui représente le réseau original, respectant les propriétés décrites par GR.

Une machine M12 est aussi définie et introduit les mêmes variables que celles introduites par M02 (vue précédemment) :

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus$ $lost := \varnothing$
$\oplus$ $actg1$: $graph := g$
$\oplus$ $actg2$: $store := \varnothing$
END

$inv1$: $graph \in NODES \leftrightarrow NODES$
$inv2$: $store \in NODES \leftrightarrow MSG$

— $graph$ représente le réseau courant ($inv1$) et elle est initialisée à l'aide de la constante g.

— $store$ permet de savoir dans quel(s) nœud(s) du réseau courant se trouve un message qui a été diffusé par sa source ($inv2$). Elle est initialisée à l'aide de l'ensemble vide ($\varnothing$).

Nous notons x_{12} les variables de cette machine M12.

Des invariants contraignent ces variables :

$inv3$: $graph \in GR$
$inv4$: $ran(store) \subseteq ran(sent\_by\_s)$
$inv5$: $ran(store) \cap lost = \varnothing$

— $inv3$ exprime que le graphe courant est toujours un graphe vérifiant les propriétés définies par l'ensemble des graphes possibles GR.

— $inv4$ exprime que chaque message circulant dans le réseau a été envoyé par une source.

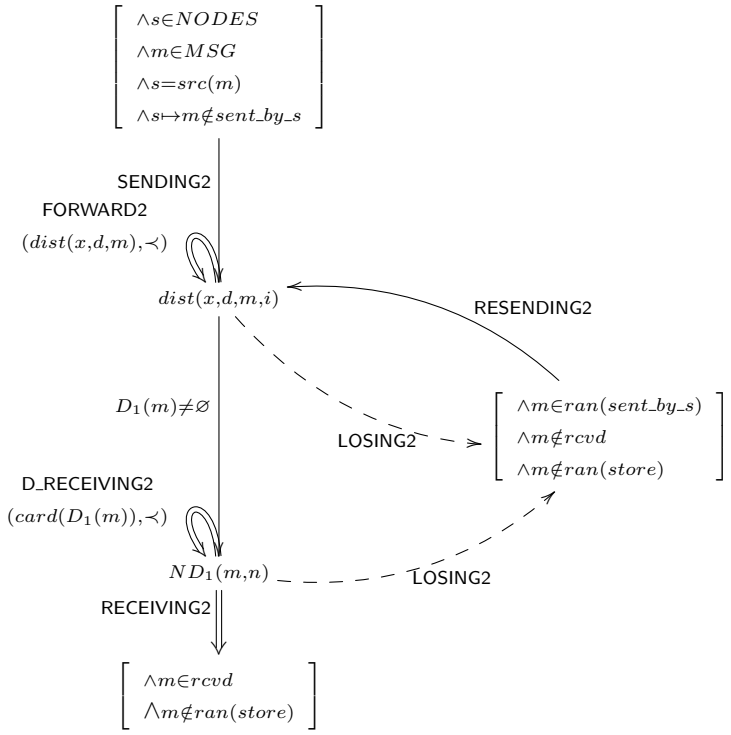


FIGURE 6.14 – Diagramme d'assertions pour le second raffinement M12

— *inv*5 exprime qu'un message envoyé est soit circulant dans le réseau, soit perdu. Nous avons ici un invariant de collage qui nous permet de supprimer la variable abstraite *lost* du modèle M12.

La machine M12 contient les raffinements des événements de la machine M11, soit SENDING2, RESENDING2, D_RECEIVING2, RECEIVING2 et LOSING2, ainsi qu'un nouvel événement FORWARD2, modélisant l'acheminement d'un message dans le système réparti *graph*. L'hypothèse d'équité L_{12} que nous posons sur le modèle M02 est la suivante :

$$\begin{aligned} L_{12} \quad \widehat{=} \quad & \wedge WF_{x_{12}}(\text{SENDING2}) \\ & \wedge WF_{x_{12}}(\text{RESENDING2}) \\ & \wedge SF_{x_{12}}(\text{D\_RECEIVING2}) \\ & \wedge SF_{x_{12}}(\text{FORWARD2}) \\ & \wedge SF_{x_{12}}(\text{RECEIVING2}) \end{aligned}$$

Nous posons des hypothèses d'équité faible sur les événements SENDING2, RESENDING2 et des hypothèses d'équité forte sur les événements FORWARD2,D_RECEIVING2 et RECEIVING2. Par contre, nous ne posons aucune hypothèse d'équité sur l'événement LOSING2, qui représente une action de l'environnement (perte de messages).

Nous définissons aussi la relation NEXT :

$$\begin{aligned} \text{NEXT} \ \widehat{=} \quad & \vee BA(\text{SENDING2})(x_{12}, x_{12}') \\ & \vee BA(\text{LOSING2})(x_{12}, x_{12}') \\ & \vee BA(\text{RESENDING2})(x_{12}, x_{12}') \\ & \vee BA(\text{FORWARD2})(x_{12}, x_{12}') \\ & \vee BA(\text{D\_RECEIVING2})(x_{12}, x_{12}') \\ & \vee BA(\text{RECEIVING2})(x_{12}, x_{12}') \\ & \vee (x_{12} = x_{12}') \end{aligned}$$

Nous nous intéressons maintenant aux propriétés de vivacité satisfaites par la machine M12. Mais auparavant, nous posons :

— $R_{12} \;\widehat{=}\; \left(\begin{array}{l} \wedge m \in ran(sent\_by\_s) \\ \wedge m \notin rcvd \\ \wedge m \notin ran(store) \end{array}\right)$
Cette propriété exprime que le message m n'a pas été reçu globalement par ses destinataires (il a été envoyé par sa source et transite vers ses destinataires) et il ne circule pas dans le réseau : le message m est perdu. Nous posons $dist(x, d, m) \in \mathbb{N}$, la distance entre la position courante x du message m non-perdu ($m \in ran(store)$) et une de ses destinations d : il s'agit du nombre de nœuds entre x et d. Nous posons $dist(x, d, m, i) \;\widehat{=}\; (dist(x, d, m) = i)$, où $i \in \mathbb{N}$.
— Nous posons :

$$D_1(m) \;\widehat{=}\; \left\{ d \;\middle|\; \begin{array}{l} \wedge d \in NODES \\ \wedge d \in m\_dst^{-1}[\{m\}] \\ \wedge d \notin rcvd\_by\_d^{-1}[\{m\}] \\ \wedge dist(x, d, m, 0) \\ \wedge m \notin rcvd \\ \wedge m \in ran(store) \end{array} \right\}$$

$D_1(m)$ est un ensemble contenant les destinataires d du message m (non perdu et non reçu globalement), vers lesquels il a été envoyé, mais qui ne l'ont pas encore reçu individuellement, alors qu'il se trouve à leurs niveaux dans le réseau ($dist(x, d, m, 0)$). Nous posons $ND_1(m, n) \;\widehat{=}\; (card(D_1(m)) = n)$, où $n \in \mathbb{N}$.
— $Q_{12} \;\widehat{=}\; (m \in rcvd \wedge d \mapsto m \notin ran(store))$, exprimant qu'un message m est reçu globalement par tous ses destinataires et qu'il est retiré du réseau.

Nous définissons la liste Φ_{12} des propriétés de vivacité satisfaites par la machine M12 comme suit :

$$\Phi_{12} \;\widehat{=}\; \left\{ \begin{array}{l} P_{11} \leadsto (\exists d, i \cdot i > 0 \wedge d \in dst(m) \wedge dist(x, d, m, i)), \\ R_{12} \leadsto (\exists d, i \cdot i \in \mathbb{N} \wedge d \in dst(m) \wedge dist(x, d, m, i)), \\ dist(x, d, m, i + 1) \leadsto dist(x, d, m, i), \\ (\exists i \cdot i \in \mathbb{N} \wedge dist(x, d, m, i)) \leadsto dist(x, d, m, 0), \\ (\exists d \cdot d \in dst(m) \wedge dist(x, d, m, 0)) \leadsto (\exists n \cdot n \in \mathbb{N} \wedge ND_1(m, n)), \\ R_{12} \leadsto (\exists n \cdot n \in \mathbb{N} \wedge ND_1(m, n)), ND_1(m, n + 1) \leadsto ND_1(m, n), \\ (\exists n \cdot n \in \mathbb{N} \wedge ND_1(m, n)) \leadsto ND_1(m, 0), \\ (ND_1(m, 0) \wedge m \notin rcvd \wedge m \in ran(store)) \leadsto Q_{12} \end{array} \right\}$$

Démonstration. Nous montrons que M12 satisfait les propriétés de Φ_{12}.
⟨1⟩1. Nous montrons que M12 satisfait $P_{11} \leadsto (\exists d, i \cdot i > 0 \wedge d \in dst(m) \wedge dist(x, d, m, i))$. Nous notons $\gamma \;\widehat{=}\; (\exists d, i \cdot i > 0 \wedge d \in dst(m) \wedge dist(x, d, m, i))$. Cette propriété exprime qu'il existe au moins une destination d à une distance i non-nulle de la position courante x du message m. Une hypothèse d'équité faible est posée sur l'événement SENDING2 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de SENDING2 sont observables lorsque P_{11} est vrai, leurs observations ne falsifient pas le prédicat P_{11}, condition d'observation de SENDING2, ou conduisent à γ.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$P_{11} \wedge [\mathsf{NEXT}]_{x_{12}} \Rightarrow (P'_{11} \vee \gamma')$$

Soit :
— $P_{11} \wedge BA(\mathsf{SENDING2})(x_{12}, x_{12}') \Rightarrow (P'_{11} \vee \gamma')$
— $P_{11} \wedge BA(\mathsf{RESENDING2})(x_{12}, x_{12}') \Rightarrow (P'_{11} \vee \gamma')$
— $P_{11} \wedge BA(\mathsf{FORWARD2})(x_{12}, x_{12}') \Rightarrow (P'_{11} \vee \gamma')$
— $P_{11} \wedge BA(\mathsf{D\_RECEIVING2})(x_{12}, x_{12}') \Rightarrow (P'_{11} \vee \gamma')$
— $P_{11} \wedge BA(\mathsf{RECEIVING2})(x_{12}, x_{12}') \Rightarrow (P'_{11} \vee \gamma')$
— $P_{11} \wedge BA(\mathsf{LOSING2})(x_{12}, x_{12}') \Rightarrow (P'_{11} \vee \gamma')$
— $P_{11} \wedge (x_{12} = x_{12}') \Rightarrow (P'_{11} \vee \gamma')$
La propriété suivante **(3)** : $P_{11}(x_{12}) \wedge BA(\mathsf{SENDING2})(x_{12}, x_{12}') \Rightarrow \gamma(x_{12}')$, et la condition de faisabilité **(4)** : $P_{11}(x_{12}) \Rightarrow (\exists x_{12}' \cdot BA(\mathsf{SENDING2})(x_{12}, x_{12}'))$ sont satisfaites par l'événement SENDING2. **(3)** nous permet de déduire $P_{11} \wedge \langle \mathsf{NEXT} \wedge \mathsf{SENDING2} \rangle_{x_{12}} \Rightarrow \gamma'$ et **(4)** permet de déduire $P_{11} \Rightarrow ENABLED\langle \mathsf{SENDING2} \rangle_{x_{12}}$, où $ENABLED\langle \mathsf{SENDING2} \rangle_{x_{12}} \;\widehat{=}\; (\exists y \cdot BA(\mathsf{SENDING2})(x_{12}, y))$.

L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M12}) \vdash P_{11} \rightsquigarrow \gamma$, donc que M12 satisfait $P_{11} \rightsquigarrow \gamma$. □

⟨1⟩2. Nous montrons que M12 satisfait $R_{12} \rightsquigarrow (\exists d, i \cdot i \in \mathbb{N} \wedge d \in dst(m) \wedge dist(x,d,m,i))$. Nous notons $\gamma \triangleq (\exists d, i \cdot i \in \mathbb{N} \wedge d \in dst(m) \wedge dist(x,d,m,i))$. Cette propriété exprime qu'il existe une distance i entre la position courante x du message m et un destinataire d. Une hypothèse d'équité faible est posée sur l'événement RESENDING2 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de RESENDING2 sont observables lorsque R_{12} est vrai, leurs observations ne falsifient pas le prédicat R_{12}, condition d'observation de RESENDING2, ou conduisent à γ.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$R_{12} \wedge [\mathsf{NEXT}]_{x_{12}} \Rightarrow (R'_{12} \vee \gamma')$$

Soit :
— $R_{12} \wedge BA(\mathsf{SENDING2})(x_{12}, x_{12}') \Rightarrow (R'_{12} \vee \gamma')$
— $R_{12} \wedge BA(\mathsf{RESENDING2})(x_{12}, x_{12}') \Rightarrow (R'_{12} \vee \gamma')$
— $R_{12} \wedge BA(\mathsf{FORWARD2})(x_{12}, x_{12}') \Rightarrow (R'_{12} \vee \gamma')$
— $R_{12} \wedge BA(\mathsf{D\_RECEIVING2})(x_{12}, x_{12}') \Rightarrow (R'_{12} \vee \gamma')$
— $R_{12} \wedge BA(\mathsf{RECEIVING2})(x_{12}, x_{12}') \Rightarrow (R'_{12} \vee \gamma')$
— $R_{12} \wedge BA(\mathsf{LOSING2})(x_{12}, x_{12}') \Rightarrow (R'_{12} \vee \gamma')$
— $R_{12} \wedge (x_{12} = x_{12}') \Rightarrow (R'_{12} \vee \gamma')$

La propriété **(3)** : $R_{12}(x_{12}) \wedge BA(\mathsf{RESENDING2})(x_{12}, x_{12}') \Rightarrow \gamma(x_{12}')$, et la condition de faisabilité **(4)** : $R_{12}(x_{12}) \Rightarrow (\exists x_{12}' \cdot BA(\mathsf{RESENDING2})(x_{12}, x_{12}'))$ sont satisfaites par RESENDING2. **(3)** permet de déduire $R_{12} \wedge \langle \mathsf{NEXT} \wedge \mathsf{RESENDING2} \rangle_{x_{12}} \Rightarrow \gamma'$ et **(4)** permet de déduire $R_{12} \Rightarrow ENABLED\langle \mathsf{RESENDING2} \rangle_{x_{12}}$, où $ENABLED\langle \mathsf{RESENDING1} \rangle_{x_{12}} \triangleq (\exists y \cdot BA(\mathsf{RESENDING2})(x_{12}, y))$.
L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M12}) \vdash R_{12} \rightsquigarrow \gamma$, donc que M12 satisfait $R_{12} \rightsquigarrow \gamma$. □

⟨1⟩3. Nous montrons que M12 satisfait $dist(x,d,m,i+1) \rightsquigarrow dist(x,d,m,i)$. Nous notons $\gamma_1 \triangleq dist(x,d,m,i+1)$ et $\gamma_2 \triangleq dist(x,d,m,i)$. Pour montrer que M02 satisfait $\gamma_1 \rightsquigarrow \gamma_2$, nous allons montrer que M12 satisfait $(R_{12} \vee \gamma_1) \rightsquigarrow \gamma_2$. Une hypothèse d'équité forte est posée sur l'événement FORWARD2 et nous nous trouvons dans le cas suivant **(1)** : un événement LOSING2 peut nous ramener, d'un état satisfaisant γ_1, à un état satisfaisant R_{12} (tel que $R_{12} \rightsquigarrow \gamma_1$), sinon les observations des autres événements, différents de LOSING2 et FORWARD2, ne falsifient pas $(R_{12} \vee \gamma_1)$ ou peuvent nous conduire à γ_2.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(R_{12} \vee \gamma_1) \wedge [\mathsf{NEXT}]_{x_{12}} \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$$

Soit :
— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{SENDING2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{RESENDING2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{LOSING2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{FORWARD2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{D\_RECEIVING2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{RECEIVING2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge (x_{12} = x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$

La propriété **(3)** : $\gamma_1(x_{12}) \wedge BA(\mathsf{FORWARD2})(x_{12}, x_{12}') \Rightarrow \gamma_2(x_{12}')$, ainsi que la condition de faisabilité **(4)** : $\gamma_1(x_{12}) \Rightarrow (\exists x_{12}' \cdot BA(\mathsf{FORWARD2})(x_{12}, x_{12}'))$ sont satisfaites par FORWARD2. **(3)** nous permet de déduire $(R_{12} \vee \gamma_1) \wedge \langle \mathsf{NEXT} \wedge \mathsf{FORWARD2} \rangle_{x_{12}} \Rightarrow \gamma'_2$. Les propriétés **(3)**, **(4)** et $R_{12} \rightsquigarrow \gamma_1$ nous permettent de déduire que tant que γ_2 n'est pas satisfait, alors l'événement FORWARD2 sera fatalement activable, ce qui nous donne la propriété : $(\neg\gamma_2) \Rightarrow \Diamond ENABLED\langle \mathsf{FORWARD2} \rangle_{x_{12}}$, où $ENABLED\langle \mathsf{FORWARD2} \rangle_{x_{12}} \triangleq (\exists y \cdot BA(\mathsf{FORWARD2})(x_{12}, y))$. Nous déduisons, en tenant compte de l'hypothèse $R_{12} \rightsquigarrow \gamma_1$: $\Box(R_{12} \vee \gamma_1) \wedge \Box[\mathsf{NEXT}]_{x_{12}} \Rightarrow \Diamond ENABLED\langle \mathsf{FORWARD2} \rangle_{x_{12}}$.
L'application de la règle SF1 donne $(\mathcal{S}pec(\mathsf{M12}), R_{12} \rightsquigarrow \gamma_1) \vdash ((R_{12} \vee \gamma_1) \rightsquigarrow \gamma_2)$. La machine M12 satisfait $(R_{12} \vee \gamma_1) \rightsquigarrow \gamma_2$, donc $\gamma_1 \rightsquigarrow \gamma_2$. □

⟨1⟩4. Nous montrons que M12 satisfait $(\exists i \cdot i \in \mathbb{N} \wedge dist(x,d,m,i)) \rightsquigarrow dist(x,d,m,0)$. Nous avons montré que M12 satisfait $dist(x,d,m,i+1) \rightsquigarrow dist(x,d,m,i)$. En prenant comme relation d'ordre bien fondée $>$ sur l'ensemble des entiers naturels $\mathbb{N}$, soit $(\mathbb{N}, >)$ comme structure bien fondée, nous appliquons la règle d'inférence treillis (LATTICE), pour démontrer la convergence vers $dist(x,d,m,0)$.

Nous montrons ainsi $\mathcal{S}pec(\mathsf{M12}) \vdash (\exists i \cdot i \in \mathbb{N} \wedge dist(x,d,m,i)) \rightsquigarrow dist(x,d,m,0)$. □

⟨1⟩5. Nous montrons que M12 satisfait $(\exists d \cdot d \in dst(m) \wedge dist(x,d,m,0)) \rightsquigarrow (\exists n \cdot n \in \mathbb{N} \wedge ND_1(m,n))$. Nous notons $\gamma_1 \triangleq (\exists d \cdot d \in dst(m) \wedge dist(x,d,m,0))$ et $\gamma_2 \triangleq (\exists n \cdot n \in \mathbb{N} \wedge ND_1(m,n))$. La propriété γ_1 exprime qu'il existe au moins un destinataire d de m, tel que m se trouve au niveau de d dans le réseau (à une distance 0 de d) et que d ne l'a pas encore reçu ; γ_2 exprime qu'il existe n destinataires tels que le message m se trouve à leurs niveaux dans le réseau (à une distance 0 des destinataires) et que les n destinataires ne l'ont pas encore reçu. En supposant que le nombre de destinataires d n'ayant pas encore reçu le message m est n, nous pouvons déduire que $\gamma_1 \Rightarrow \gamma_2$. L'application de la règle de déduction (*dedu*), nous permet de déduire $\mathcal{S}pec(\mathsf{M12}) \vdash \gamma_1 \rightsquigarrow \gamma_2$. □

⟨1⟩6. Nous montrons que M12 satisfait $R_{12} \rightsquigarrow (\exists n \cdot n \in \mathbb{N} \wedge ND_1(m,n))$. Les étapes ⟨1⟩2, ⟨1⟩3, ⟨1⟩4 nous permettent de déduire $R_{12} \rightsquigarrow (\exists d \cdot d \in dst(m) \wedge dist(x,d,m,0))$. L'étape ⟨1⟩5 et l'application de la règle de transitivité (*trans*) nous permettent de déduire $\mathcal{S}pec(\mathsf{M12}) \vdash R_{12} \rightsquigarrow (\exists n \cdot n \in \mathbb{N} \wedge ND_1(m,n))$. □

⟨1⟩7. Nous montrons que M12 satisfait $ND_1(m,n+1) \rightsquigarrow ND_1(m,n)$. Nous notons $\gamma_1 \triangleq ND_1(m,n+1)$ et $\gamma_2 \triangleq ND_1(m,n)$. Pour montrer que M12 satisfait $\gamma_1 \rightsquigarrow \gamma_2$, nous allons montrer que M12 satisfait $(R_{12} \vee \gamma_1) \rightsquigarrow \gamma_2$. Une hypothèse d'équité forte est posée sur l'événement D_RECEIVING2 et nous nous trouvons dans le cas suivant **(1)** : un événement LOSING2 peut nous ramener, d'un état satisfaisant γ_1, à un état satisfaisant R_{12} (tel que $R_{12} \rightsquigarrow \gamma_1$), sinon les observations des autres événements, différents de LOSING2 et D_RECEIVING2, ne falsifient pas $(R_{12} \vee \gamma_1)$ ou peuvent nous conduire à γ_2.

Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(R_{12} \vee \gamma_1) \wedge [\mathsf{NEXT}]_{x_{12}} \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$$

Soit :

— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{SENDING2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{RESENDING2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{LOSING2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{FORWARD2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{D\_RECEIVING2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge BA(\mathsf{RECEIVING2})(x_{12}, x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$
— $(R_{12} \vee \gamma_1) \wedge (x_{12} = x_{12}') \Rightarrow ((R'_{12} \vee \gamma'_1) \vee \gamma'_2)$

La propriété **(3)** : $\gamma_1(x_{12}) \wedge BA(\mathsf{D\_RECEIVING2})(x_{12}, x_{12}') \Rightarrow \gamma_2(x_{12}')$, ainsi que la condition de faisabilité **(4)** : $\gamma_1(x_{12}) \Rightarrow (\exists x_{12}' \cdot BA(\mathsf{D\_RECEIVING2})(x_{12}, x_{12}'))$ sont satisfaites par D_RECEIVING2. **(3)** nous permet de déduire $(R_{12} \vee \gamma_1) \wedge \langle \mathsf{NEXT} \wedge \mathsf{D\_RECEIVING2} \rangle_{x_{12}} \Rightarrow \gamma'_2$. Les propriétés **(3)**, **(4)** et $R_{11} \rightsquigarrow \gamma_1$ nous permettent de déduire que tant que γ_2 n'est pas satisfait, alors l'événement D_RECEIVING1 sera fatalement activable, ce qui nous donne : $(\neg\gamma_2) \Rightarrow \Diamond ENABLED\langle \mathsf{D\_RECEIVING2} \rangle_{x_{12}}$, où $ENABLED\langle \mathsf{D\_RECEIVING2} \rangle_{x_{12}} \triangleq (\exists y \cdot BA(\mathsf{D\_RECEIVING2})(x_{12}, y))$. Nous déduisons, en tenant compte de $R_{12} \rightsquigarrow \gamma_1$: $\Box(R_{12} \vee \gamma_1) \wedge \Box[\mathsf{NEXT}]_{x_{12}} \Rightarrow \Diamond ENABLED\langle \mathsf{D\_RECEIVING2} \rangle_{x_{12}}$. L'application de la règle SF1 donne $(\mathcal{S}pec(\mathsf{M12}), R_{12} \rightsquigarrow \gamma_1) \vdash ((R_{12} \vee \gamma_1) \rightsquigarrow \gamma_2)$. La machine M12 satisfait $(R_{12} \vee \gamma_1) \rightsquigarrow \gamma_2$, donc $\gamma_1 \rightsquigarrow \gamma_2$. □

⟨1⟩8. Nous montrons que M12 satisfait $(\exists n \cdot n \in \mathbb{N} \wedge ND_1(m,n)) \rightsquigarrow ND_1(m,0)$. Nous avons montré que M12 satisfait $ND_1(m,n+1) \rightsquigarrow ND_1(m,n)$. En prenant comme relation d'ordre bien fondée $>$ sur l'ensemble des entiers naturels $\mathbb{N}$, soit $(\mathbb{N}, >)$ comme structure bien fondée, nous appliquons la règle d'inférence treillis (LATTICE), pour démontrer la convergence vers $ND_1(m,0)$. Nous montrons ainsi $\mathcal{S}pec(\mathsf{M12}) \vdash (\exists n \cdot n \in \mathbb{N} \wedge ND_1(m,n)) \rightsquigarrow ND_1(m,0)$. □

⟨1⟩9. Nous montrons que M12 satisfait $(ND_1(m,0) \wedge m \notin rcvd \wedge m \in ran(store)) \rightsquigarrow Q_{12}$. Nous notons $\rho \triangleq (ND_1(m,0) \wedge m \notin rcvd \wedge m \in ran(store))$. Pour montrer que M12 satisfait $\rho \rightsquigarrow Q_{12}$, nous allons montrer $(R_{12} \vee \rho) \rightsquigarrow Q_{12}$.Une hypothèse d'équité forte est posée sur l'événement RECEIVING2 et nous nous trouvons dans le cas suivant **(1)** : un événement LOSING2 peut nous ramener, d'un état satisfaisant ρ, à un état satisfaisant R_{12} (tel que $R_{12} \rightsquigarrow \rho$), sinon les observations des autres événements, différents de LOSING2 et RECEIVING2, ne falsifient pas $(R_{12} \vee \rho)$ ou peuvent nous conduire à Q_{12}.

Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(R_{12} \vee \rho) \wedge [\mathsf{NEXT}]_{x_{12}} \Rightarrow ((R'_{12} \vee \rho') \vee Q'_{12})$$

Soit :

— $(R_{12} \vee \rho) \wedge BA(\mathsf{SENDING2})(x_{12}, x_{12}') \Rightarrow ((R_{12}' \vee \rho') \vee Q_{12}')$
— $(R_{12} \vee \rho) \wedge BA(\mathsf{RESENDING2})(x_{12}, x_{12}') \Rightarrow ((R_{12}' \vee \rho') \vee Q_{12}')$
— $(R_{12} \vee \rho) \wedge BA(\mathsf{LOSING2})(x_{12}, x_{12}') \Rightarrow ((R_{12}' \vee \rho') \vee Q_{12}')$
— $(R_{12} \vee \rho) \wedge BA(\mathsf{FORWARD2})(x_{12}, x_{12}') \Rightarrow ((R_{12}' \vee \rho') \vee Q_{12}')$
— $(R_{12} \vee \rho) \wedge BA(\mathsf{D\_RECEIVING2})(x_{12}, x_{12}') \Rightarrow ((R_{12}' \vee \rho') \vee Q_{12}')$
— $(R_{12} \vee \rho) \wedge BA(\mathsf{RECEIVING2})(x_{12}, x_{12}') \Rightarrow ((R_{12}' \vee \rho') \vee Q_{12}')$
— $(R_{12} \vee \rho) \wedge (x_{12} = x_{12}') \Rightarrow ((R_{12}' \vee \rho') \vee Q_{12}')$

La propriété **(3)** : $\rho(x_{12}) \wedge BA(\mathsf{RECEIVING2})(x_{12}, x_{12}') \Rightarrow Q_{12}(x_{12}')$, ainsi que la condition de faisabilité **(4)** : $\rho(x_{12}) \Rightarrow (\exists x_{12}' \cdot BA(\mathsf{RECEIVING1})(x_{12}, x_{12}'))$ sont satisfaites par RECEIVING2. **(3)** nous permet de déduire $(R_{12} \vee \rho) \wedge \langle \mathsf{NEXT} \wedge \mathsf{RECEIVING2} \rangle_{x_{12}} \Rightarrow Q_{12}'$. Les propriétés **(3)**, **(4)** et $R_{12} \rightsquigarrow \rho$ nous permettent de déduire que tant que Q_{12} n'est pas satisfait, alors l'événement RECEIVING2 sera fatalement activable, ce qui nous donne : $(\neg Q_{12}) \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING2} \rangle_{x_{12}}$, où $ENABLED\langle \mathsf{RECEIVING2} \rangle_{x_{12}} \ \widehat{=} \ (\exists y \cdot BA(\mathsf{RECEIVING2})(x_{12}, y))$. Nous déduisons, en tenant compte de $R_{12} \rightsquigarrow \rho$: $\Box(R_{12} \vee \rho) \wedge \Box[\mathsf{NEXT}]_{x_{12}} \Rightarrow \Diamond ENABLED\langle \mathsf{RECEIVING2} \rangle_{x_{12}}$. L'application de la règle SF1 donne $(\mathcal{S}pec(\mathsf{M12}), R_{12} \rightsquigarrow \rho) \vdash ((R_{12} \vee \rho) \rightsquigarrow Q_{12})$. La machine M12 satisfait $(R_{12} \vee \rho) \rightsquigarrow Q_{12}$, donc $\rho \rightsquigarrow Q_{12}$. □

⟨1⟩10. Les étapes ⟨1⟩1, ⟨1⟩2, ⟨1⟩3, ⟨1⟩4, ⟨1⟩5, ⟨1⟩6, ⟨1⟩7, ⟨1⟩8 et ⟨1⟩9 nous permettent de déduire que M12 satisfait les propriétés de vivacité de Φ_{12}. QED.

□

Soit Φ_{11} l'ensemble des propriétés caractérisant le raffinement M11 précédent :

$$\Phi_{11} \ \widehat{=} \ \left\{ \begin{array}{l} P_{11} \rightsquigarrow (\exists n \cdot n > 0 \wedge ND(m,n)), ND(m,n+1) \rightsquigarrow ND(m,n), \\ R_{11} \rightsquigarrow (\exists n \cdot n \in \mathbb{N} \wedge ND(m,n)), (\exists n \cdot n \in \mathbb{N} \wedge ND(m,n)) \rightsquigarrow ND(m,0) \\ (ND(m,0) \wedge m \notin (rcvd \cup lost)) \rightsquigarrow Q \end{array} \right\}$$

Démonstration. Nous montrons que M12 satisfait les propriétés de Φ_{11}.

⟨1⟩1. Nous montrons que M12 satisfait $P_{11} \rightsquigarrow (\exists n \cdot n > 0 \wedge ND(m,n))$. Nous rappelons la définition de $ND(m,n)$:

$$D(m) \ \widehat{=} \ \left\{ d \ \middle| \ \begin{array}{l} \wedge d \in NODES \\ \wedge d \in m\_dst^{-1}[\{m\}] \\ \wedge d \notin rcvd\_by\_d^{-1}[\{m\}] \\ \wedge m \notin (rcvd \cup lost) \end{array} \right\}$$

Nous notons $ND(m,n) \ \widehat{=} \ (card(D(m)) = n)$. Nous pouvons décomposer $D(m)$ en :

$$D_0(m) \ \widehat{=} \ \left\{ d \ \middle| \ \begin{array}{l} \wedge d \in NODES \\ \wedge d \in m\_dst^{-1}[\{m\}] \\ \wedge d \notin rcvd\_by\_d^{-1}[\{m\}] \\ \wedge dist(x,d,m) > 0 \\ \wedge m \notin rcvd \\ \wedge m \in ran(store) \end{array} \right\} \text{ et } D_1(m) \ \widehat{=} \ \left\{ d \ \middle| \ \begin{array}{l} \wedge d \in NODES \\ \wedge d \in m\_dst^{-1}[\{m\}] \\ \wedge d \notin rcvd\_by\_d^{-1}[\{m\}] \\ \wedge dist(x,d,m,0) \\ \wedge m \notin rcvd \\ \wedge m \in ran(store) \end{array} \right\}$$

$D_0(m)$ exprime qu'il existe un ensemble de destinataires d, auxquels un message m, non-perdu, non encore reçu, a été envoyé. Le message m se trouve des distances non nulles des destinataires d dans le réseau. $D_1(m)$ exprime qu'il existe un ensemble de destinataires d, auxquels un message m, non-perdu, non encore reçu, a été envoyé. Le message m se trouve aux niveaux des destinataires d dans le réseau. Nous notons $ND_0(m,n_0) \ \widehat{=} \ (card(D_0(m)) = n_0)$ et $ND_1(m,n_1) \ \widehat{=} \ (card(D(m)) = n_1)$. En supposant, $n = n_0 + n_1$, nous pouvons établir : $P_{11} \rightsquigarrow ((\exists n_0 \cdot ND_0(m,n_0)) \vee (\exists n_1 \cdot ND_1(m,n_1)))$. Or, nous pouvons aussi établir : $(\exists d, i \cdot i > 0 \wedge d \in dst(m) \wedge dist(x,d,m,i)) \Leftrightarrow (\exists n_0 \cdot n_0 > 0 \wedge ND_0(m,n_0))$, et nous savons aussi que M12 satisfait $P_{11} \rightsquigarrow (\exists d, i \cdot i > 0 \wedge d \in dst(m) \wedge dist(x,d,m,i))$. Nous pouvons alors déduire que M12 satisfait $P_{11} \rightsquigarrow (\exists n \cdot n > 0 \wedge ND(m,n))$. □

⟨1⟩2. Nous montrons que M12 satisfait $ND(m,n+1) \rightsquigarrow ND(m,n)$. Nous avons décomposé à l'étape précédente $D(m)$ en $D_0(m)$ et $D_1(m)$. Nous pouvons alors en déduire que $n+1 = card(D_0(m)) + card(D_1(m))$. Nous supposons $card(D_0(m)) = n_0$ et $card(D_1(m)) = n_1$. Les étapes ⟨1⟩4 et ⟨1⟩5, de la démonstration précédente nous permettent de déduire : $(ND_0(m,n_0) \wedge ND_1(m,n_1)) \rightsquigarrow$

$(ND_0(m, n_0 - 1) \land ND_1(m, n_1 + 1))$. En prenant comme relation d'ordre bien fondée $>$ sur l'ensemble l'ensemble $0..n_0$, nous appliquons la règle d'inférence treillis (LATTICE), pour démontrer la convergence de $ND_0(m, n_0)$ vers $ND_0(m, 0)$. Nous remarquons ainsi que la quantité n_0 décroit proportionnellement à l'augmentation de n_1. Nous savons que M12 satisfait $ND_1(m, n_1 + 1) \rightsquigarrow ND(m, n_1)$. Nous pouvons alors déduire que M12 satisfait $ND(m, n + 1) \rightsquigarrow ND(m, n)$.

⟨1⟩3. Nous montrons que M12 satisfait $R_{11} \rightsquigarrow (\exists n \cdot n \in \mathbb{N} \land ND(m, n))$. En appliquant la décomposition de $D(m)$ en $D_0(m)$ et $D_1(m)$, vue précédemment, et en supposant, $n = n_0 + n_1$, nous pouvons établir : $R_{11} \rightsquigarrow ((\exists n_0 \cdot ND_0(m, n_0)) \lor (\exists n_1 \cdot ND_1(m, n_1)))$. Or, nous avons : $R_{11} \Leftrightarrow R_{12}$ (déduite des invariants $inv4$ de M0 et $inv5$ de M12), et nous savons aussi que M12 satisfait $R_{12} \rightsquigarrow (\exists n_1 \cdot ND_1(m, n_1))$. Nous pouvons alors déduire que M12 satisfait $R_{11} \rightsquigarrow (\exists n \cdot n \in \mathbb{N} \land ND(m, n))$. □

⟨1⟩4. Nous montrons que M12 satisfait $(\exists n \cdot n \in \mathbb{N} \land ND(m, n)) \rightsquigarrow ND(m, 0)$. Nous avons montré que M12 satisfait $ND(m, n + 1) \rightsquigarrow ND(m, n)$. En prenant comme relation d'ordre bien fondée $>$ sur l'ensemble des entiers naturels $\mathbb{N}$, soit $(\mathbb{N}, >)$ comme structure bien fondée, nous appliquons la règle d'inférence treillis (LATTICE), pour démontrer la convergence vers $ND(m, 0)$. Nous montrons ainsi $\mathcal{S}pec(\mathsf{M12}) \vdash (\exists n \cdot n \in \mathbb{N} \land ND(m, n)) \rightsquigarrow ND(m, 0)$. □

⟨1⟩5. Nous montrons que M12 satisfait $(ND(m, 0) \land m \notin (rcvd \cup lost)) \rightsquigarrow Q$. En appliquant la décomposition de $D(m)$ en $D_0(m)$ et $D_1(m)$, vue précédemment, et en supposant, $0 = n_0 + n_1$, pour $n_0 \in \mathbb{N}$ et $n_1 \in \mathbb{N}$, nous pouvons établir : $ND(m, 0) \Leftrightarrow ND_1(m, 0)$. Nous avons alors : $(ND(m, 0) \land m \notin (rcvd \cup lost)) \Leftrightarrow (ND_1(m, 0) \land m \notin rcvd \land m \in ran(store))$. Nous pouvons décomposer Q en $Q_0 \ \widehat{=}\ m \in rcvd \land m \in ran(store)$ et $Q_{12} \ \widehat{=}\ m \in rcvd \land m \notin ran(store)$; or nous savons que Q_0 n'est jamais pris en compte dans M12, seulement Q_{12}. Nous pouvons alors établir $Q \Leftrightarrow Q_{12}$. Nous savons que M12 satisfait $(ND_1(m, 0) \land m \notin rcvd \land m \in ran(store)) \rightsquigarrow Q_{12}$. Nous déduisons donc que M12 satisfait $(ND(m, 0) \land m \notin (rcvd \cup lost)) \rightsquigarrow Q$. □

⟨1⟩6. QED.

□

Nous utilisons les propriétés de Φ_{12}, pour guider le raffinement de M11 en M12, et obtenir ainsi les événements SENDING2, RESENDING2, D_RECEIVING2, RECEIVING2, LOSING2 et FORWARD2. Nous ne présentons ici que les événements ayant changé suite au raffinement et les nouveaux événements :

— Pour les raffinements des événements
$$< \mathsf{X} > \in \{\mathsf{SENDING}, \mathsf{RESENDING}\},$$
nous rajoutons juste le fait qu'à l'envoi, un message m est déposé par sa source s dans le réseau ($actg1$).

```
EVENT < X >2 REFINES < X >1 ≙
  ANY
    ...
  WHERE
    ...
  THEN
    ...
    ⊕ actg1 : store := store ∪ {s ↦ m}
  END
```

— Un nouvel événement FORWARD2 est aussi introduit : un message m circulant dans le réseau n'est plus déplacé du nœud x au nœud y, il est recopié en y. Nous tenons ainsi compte du fait que d'autres destinataires puissent être atteints à partir du nœud x.

EVENT FORWARD2 $\widehat{=}$
ANY
x, y, m, d
WHERE
$grd1 : x \in NODES$
$grd2 : y \in NODES$
$grd3 : d \in NODES$
$grd4 : m \in MSG$
$grd5 : x \mapsto m \in store$
$grd6 : y \mapsto m \notin store$
$grd7 : x \mapsto y \in graph$
$grd8 : d \in dst^{-1}[\{m\}]$
$grd9 : x \neq d$
THEN
$act1 : store := store \cup \{y \mapsto m\}$
END

— L'événement D_RECEIVING1 est raffiné de telle manière qu'un message m n'est reçu par un destinataire d que s'il ne se trouve à son niveau dans le réseau ($grdg2$).

EVENT D_RECEIVING2
REFINES D_RECEIVING1 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd3$
$\oplus$ $grdg1 : d \mapsto m \notin rcvd\_by\_d$
$\oplus$ $grdg2 : d \mapsto m \in store$
THEN
...
END

— L'événement RECEIVING2 est observé si un message m a été reçu individuellement par tous ses destinataires, n'a pas encore été marqué comme ayant été reçu globalement par tous ses destinataires et s'il se trouve encore dans le réseau. Après la réception, le message m est supprimé du réseau.

EVENT RECEIVING2
REFINES RECEIVING1 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd3$
$\oplus$ $grdg1 : m \notin got$
$\oplus$ $grdg2 : m \in ran(store)$
THEN
...
$actg1 : store := store \rhd \{m\}$
END

— L'événement LOSING2 est observé si un message m n'a pas encore été reçu globalement par tous ses destinataires et s'il se trouve encore dans le réseau. Dans ce cas, le message m est supprimé du réseau.

EVENT LOSING2
REFINES LOSING1 $\widehat{=}$
ANY
...
WHERE
...
$\ominus$ $grd3$
$\oplus$ $grdg1 : m \notin got$
$\oplus$ $grdg2 : m \in ran(store)$
THEN
...
$actg1 : store := store \rhd \{m\}$
END

— Comme vu précédemment, deux nouveaux événement ADD_LINK2 et REMOVE_LINK2 sont introduits :

— ADD_LINK2 modélise l'ajout dans le graphe courant $graph$, de liens qui n'existaient pas auparavant entre des nœuds x et y ($grd4$), qui deviennent de ce fait adjacents ($act1$). L'événement ADD_LINK2 est observé si le graphe résultant respecte les conditions définies par GR ($grd5$).
— REMOVE_LINK2 modélise la suppression ($act1$) dans le graphe courant $graph$, des liens existants ($grd3$) entre deux nœuds x et y. L'événement REMOVE_LINK2 est observé si le graphe résultant respecte les conditions définies par GR ($grd5$).

EVENT ADD_LINK2 $\widehat{=}$
ANY
x, y
WHERE
$grd1 : x \in NODES$
$grd2 : y \in NODES$
$grd3 : x \neq y$
$grd4 : x \mapsto y \notin graph$
$grd5 : graph \cup \{x \mapsto y, y \mapsto x\} \in GR$
THEN
$act1 : graph := graph \cup \{x \mapsto y, y \mapsto x\}$
END

EVENT REMOVE_LINK2 $\widehat{=}$
ANY
x, y
WHERE
$grd1 : x \in NODES$
$grd2 : y \in NODES$
$grd3 : x \mapsto y \in graph$
$grd4 : x \neq y$
$grd5 : graph \setminus \{x \mapsto y, y \mapsto x\} \in GR$
THEN
$act1 : graph := graph \setminus \{x \mapsto y, y \mapsto x\}$
END

Nous pouvons résumer ce raffinement M12 par le diagramme 6.15 suivant :

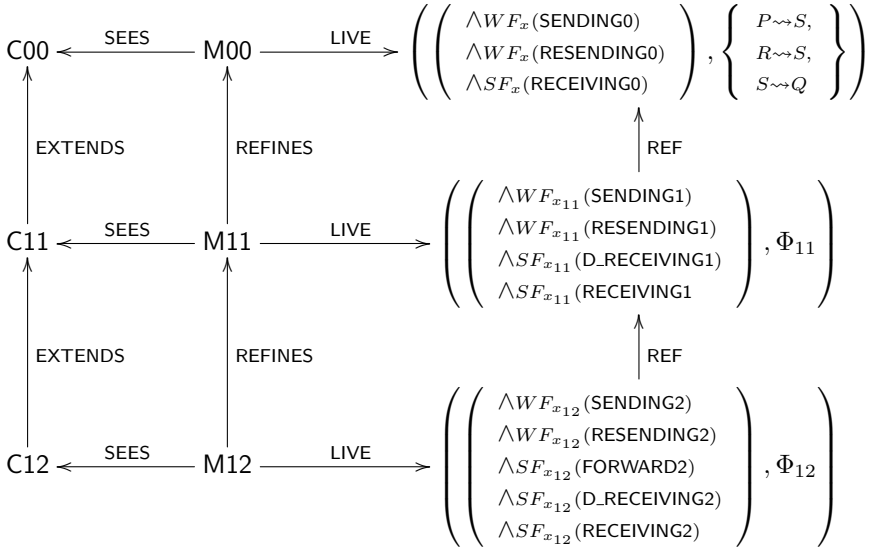


FIGURE 6.15 – Patron (destinataires multiples) : deuxième raffinement

Nous avons présenté dans cette sous-section et les précédentes, un patron de conception pour le routage, s'adressant à deux cas que nous avons identifiés : celui du routage de type 1-1 (communications entre une source et une destination) et celui de type 1-n (communications entre une source et un groupe de destinataires). La sous-section suivante présente nos conclusions et synthèses pour ce patron de routage.

6.2.2 Patrons de conception : synthèse

Nous avons abordé dans cette deuxième section la notion de patron de conception [2, 13, 22, 7, 17], que nous définissons comme suit : il s'agit d'un développement formel en Event-B répondant à un problème commun, fréquent (e.g. les actions/réactions dans les systèmes réactifs [1], les communications entre processus [13], le routage [5], etc). L'objectif est ici la réutilisation d'un modèle et des preuves de sa correction (preuves de propriétés de sûreté, de vivacité, d'équité), pour aider un utilisateur au développement d'un problème plus complexe : la problématique que nous posons est la réduction des efforts de preuves et de développement fournis par un utilisateur dans un développement large, par la résolution d'une partie de ce développement grâce à l'instanciation, l'adaptation et l'incorporation de modèles fournis par des patrons de conception.

Nous avons défini dans cette section un patron de conception pour le routage, inspiré par notre développement formel du protocole Anycast RP : il s'agit d'un patron composé de trois niveaux d'abstraction, c'est-à-dire d'une abstraction de l'idée de routage (envoi - réception de paquets/ messages), d'une identification des noeuds/processus participant au routage et enfin de l'introduction d'un système réparti/réseau support du routage. Nous n'avons pas introduit les tables de routage, leur gestion et les algorithmes de routage car ces aspects relèvent spécifiquement du problème étudié (routage XY/XYZ pour les Réseaux-sur-puce 2D/3D [4], algorithme DSR pour certains types de réseaux mobiles [18], etc). Le patron de conception pour le routage que nous présentons, prend en compte deux types de communication :

— Celles de type $1-1$: une source et un destinataire, c'est-à-dire *unicast* ou *anycast* (choix d'un destinataire parmi plusieurs possibles).
— Celles de type $1-n$: une source et n destinataires, c'est-à-dire *multicast*.

La complexité du développement est donnée par le tableau suivant présentant les statistiques des obligations de preuves prouvées automatiquement ou manuellement. Il est à noter ici que nous ne présentons que les statistiques générées par l'utilisation exclusive des prouveurs de l'Atelier B.

Modèles	Total	Automatiques		Interactives	
C00	0	0	100%	0	0%
C01	1	1	100%	0	0%
C02	0	0	100%	0	0%
C11	1	1	100%	0	0%
C12	0	0	100%	0	0%
M00	9	9	100%	0	0%
M01	34	27	79.41%	7	20.59%
M02	19	15	78.95%	4	21.05%
M11	30	27	90%	3	10%
M12	21	17	80.95%	4	19.05%
Total	115	97	84.35%	18	15.65%

Table 6.1 – Patron de conception : Statistiques des POs

Les obligations de preuves interactives constatées dans ce tableau proviennent essentiellement des aspects suivants :

— Nous identifions, dans les contextes et machines C01, M01, C11 et M11, les sources et destinataires de chaque paquet et nous exprimons des propriétés telles que les différences entre ces sources et ces destinataires, l'unicité des sources, et selon les cas, l'unicité ou la multiplicité des destinataires.
— Les machines M02 et M12 introduisent le réseau, ainsi que des propriétés sur ce réseau et les modifications possibles sur ce dernier.

Nous utilisons ce patron de conception pour le routage pour résoudre notamment le cas des Réseaux-sur-puce 2D et de l'algorithme de routage XY associé [4].

6.3 « Réseaux-sur-puce » et routage XY

Un « Réseau-sur-puce » ou « Network-on-Chip » (NoC) est une technique, un paradigme permettant de résoudre les problèmes de conception liés aux « Systèmes-sur-puce » (System-on-Chip, SoC) [8] : la conception et l'analyse des « Systèmes-sur-puce » se fait en utilisant les techniques dédiées aux réseaux et au routage [29]. En d'autres termes, un SoC est vu comme un micro-réseau [9] : la puce est divisée en

blocs/composants IP (Intellectual Property) et le « Réseau-sur-puce » (NoC) permet de relier ces blocs. Un routeur est inclus dans chaque bloc et lui donne ainsi la possibilité de communiquer avec ses voisins.

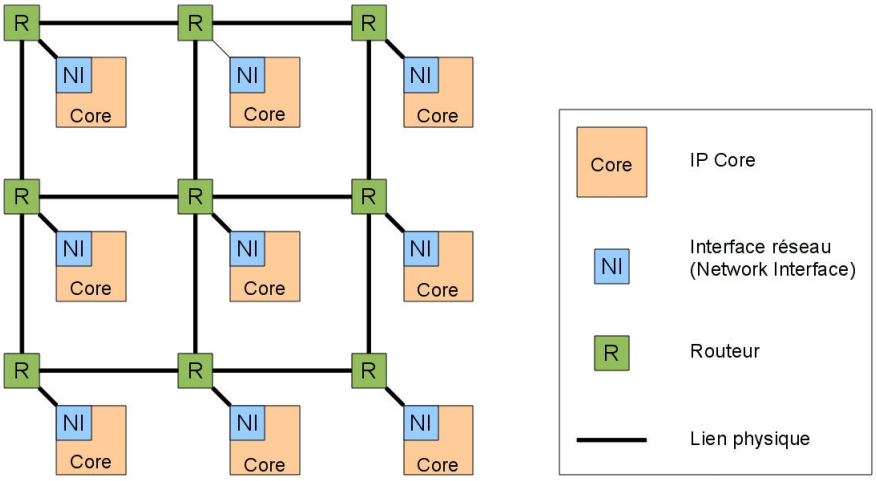


FIGURE 6.16 – Un NoC 2-D de dimension 3×3 [5]

Nous étudions dans ce chapitre les « Réseaux-sur-puce » (Network-on-Chip, NoC) bidimensionnels (2-D), sous forme de réseaux maillés (voir figure 6.16). Nous nous intéressons dans cette section à la communication entre les routeurs dans de tels réseaux, particulièrement à une forme de routage qualifiée de distribuée [29] : le chemin qu'un paquet doit prendre n'est pas décidé à la source, mais chaque routeur qui reçoit le paquet décide dans quelle direction il doit l'envoyer. Nous pouvons voir d'après la figure 6.16, que la structure globale d'un routeur est la suivante : il possède des ports d'entrée et de sorties *externes* permettant de communiquer avec ses voisins, ainsi qu'un port d'entrée et un port de sortie internes, permettant la communication avec le composant IP qui lui est associé.

Dans les « Réseaux-sur-puce » (Network-on-Chip, NoC) bidimensionnels, un routeur est caractérisé par une paire de coordonnées (X : abscisse, Y : ordonnée). Il est aussi caractérisé par quatres directions par lesquelles il peut router un paquet : nous avons la direction **(1)** Nord (N), qui consiste à transmettre le paquet au routeur de coordonnées (X,Y+1), **(2)** Sud (S), qui consiste à transmettre le paquet au routeur de coordonnées (X,Y-1), **(3)** Est (E), qui consiste à transmettre le paquet au routeur de coordonnées (X+1,Y) et **(4)** Ouest (W), qui consiste à transmettre le paquet au routeur de coordonnées (X-1,Y).

Wang Zhang, Ligang Hou [29] proposent un algorithme adapté aux réseaux maillés 2-D, il s'agit de l'algorithme XY, qui peut se présenter comme suit (voir Algorithme 3) : cet algorithme compare pour un paquet les coordonnées (Cx, Cy) du routeur courant dans lequel le paquet se trouve aux coordonnées (Dx, Dy) de sa destination. Si les coordonnées (Cx, Cy) sont les mêmes que (Dx, Dy), le paquet est arrivé à destination. Sinon, les coordonnées en abscisses sont comparées : si Cx < Dx, alors le paquet est routé dans la direction Est (E), sinon si Cx > Dx, alors le paquet est routé dans la direction dans la direction Ouest (W). Dans le cas où Cx = Dx, si Cy < Dy, alors le paquet prend la direction Nord (N), sinon si Cy > Dy, le paquet prend la direction Sud (S).

Nous nous intéressons surtout à une variante de cet algorithme (voir algorithme 4), présentée dans [4] et qui prend en compte le fait que des routeurs, sur le chemin par lequel un paquet est acheminé, peuvent être désactivés, et ne peuvent par conséquent être utilisés pour la transmission du paquet : l'idée proposée dans [4] est de contourner le routeur désactivé, en effectuant un ou plusieurs pas selon les coordonnées en ordonnées (Y) et après de revenir au comportement présenté par l'algorithme XY.

Un « Réseau-sur-puce » (Network-on-Chip, Noc), ainsi que son comportement sont généralement évalués par simulation [19, 20]. Cependant, la simulation à elle seule ne permet pas l'exploration exhaustive de tous les états d'un système tel que le NoC, lorsque celui-ci devient complexe [14]. Nous proposons dans ce chapitre une vérification du NoC en utilisant les méthodes formelles, particulièrement la méthode

5. `http://sips.inesc-id.pt/~nfvr/msc_theses/msc12ac/`

Algorithm 3 Algorithme XY

D : routeur destination. Coordonnées (Dx : abscisse, Dy : ordonnée).
C : routeur courant. Coordonnées (Cx : abscisse, Cy : ordonnée).

```
1: if Cx > Dx then
2:    return W ;
3: else if Cx < Dx then
4:    return E ;
5: else if Cx = Dx then
6:    if Cy < Dy then
7:       return N ;
8:    else if Cy > Dy then
9:       return S ;
10:   end if
11: end if
```

Algorithm 4 Algorithme XY adaptatif

D : routeur destination. Coordonnées (Dx : abscisse, Dy : ordonnée).
C : routeur courant. Coordonnées (Cx : abscisse, Cy : ordonnée).

```
1: if Cx > Dx then
2:    return W ;
3: else if Cx < Dx then
4:    return E ;
5: else if (Cx = Dx) ∨ ((Cx > Dx) ∧ W est bloquée) ∨ ((Cx < Dx) ∧ E est bloquée) then
6:    if Cy < Dy then
7:       return N ;
8:    else if Cy > Dy then
9:       return S ;
10:   end if
11: end if
```

Event-B, et les paradigmes de *correction-par-construction* et *service-as-event*. Un de nos objectifs est de réutiliser le patron de conception pour le routage vu dans le chapitre précédent pour modéliser le NoC.

Nous proposons dans cette section un algorithme adaptatif pour la gestion des communications dans le NoC. Pour modéliser ces communications dans le NoC, nous avons réutilisé le patron de conception pour le routage vu dans le chapitre précédent (page 156 à page 193) et nous l'avons adapté aux caractéristiques du NoC, telles que proposées par les électroniciens des laboratoires LIM de l'Université de Tiaret (Algérie) et LICM de l'Université de Lorraine, avec qui nous avons travaillé. Parmi ces caractéristiques, figurent notamment des hypothèses posées sur le NoC que nous étudions : un mécanisme de reconfiguration, que nous détaillerons dans la section suivante, nous assure qu'un paquet pourra toujours être acheminé vers sa destination et reçu par cette dernière. En effet, la reconfiguration permet que les routeurs désactivés soient réactivés et qu'ainsi ils redeviennent capable d'assurer la transmission des paquets. Un de nos buts est aussi la spécification de ce mécanisme de reconfiguration.

6.3.1 Reconfiguration

Ce chapitre nous a introduit aux puces électroniques et circuits intégrés, principalement aux circuits logiques programmables (Programmable Logical Devices, PLD) [28] : il s'agit de circuits intégrés *logiques* dont la *reprogrammation* est possible après la fabrication de la puce. Nous notons qu'au moment de sa fabrication, un PLD n'a pas encore de fonction définie. Par conséquent, avant de pouvoir être utilisé, un PLD doit être *reprogrammé* ou plus précisément être ***reconfiguré*** : ce que nous entendons ici par *reconfiguration* est la capacité du circuit logique à s'adapter lors des calculs et traitements effectués, en fonction des données fournies en entrée ou de résultats ou situations intermédiaires [24].

Nous rappelons ici que les circuits intégrés que nous étudions ici se composent de cellules et bascules logiques dont les interactions et connexions peuvent être librement définies. Des exemples de reconfiguration sont donc la modification des connexions entre les composants de la puce, des comportements et fonctionnalités de ces derniers, le changement de la programmation logique de la puce (connexion des portes logiques de base entre elles, etc), la redéfinition de l'architecture matérielle de la puce, pour avoir une architecture adaptée à la résolution d'un calcul, d'un problème traité par la puce, etc.

Nous donnons ici quelques exemples de techniques utilisées pour la *reconfiguration* d'une puce électronique :

- Différentes configurations sont stockées dans un cache mémoire et chargées par la puce en fonction des données à traiter passées en entrées à la puce ou encore de traitements intermédiaires à effectuer par la puce. Ces configurations définissent les connectivités entre les cellules et blocs de la puces, les fonctions de ces derniers, etc. La puce est alors reprogrammée avec la configuration choisie [11]. Une extension de cette idée est la *self-configuration* : la puce est capable de créer elle-même les configurations adaptées à des problèmes et autres calculs [23].
- Les composants de la puce sont redondants : des cellules, des lignes ou des colonnes (de cellules) sont disponibles en plusieurs exemplaires, dont certains serviront de rechange en cas de pannes ou de fautes [25]. Ce type de reconfiguration implique une capacité supplémentaire de la puce : cette dernière doit être capable de détecter les pannes et fautes des composants, pour pouvoir demander leur remplacement.
- Un composant spécial de la puce, appelé « Reconfigurable Hardware Unit » dans [21], détecte et isole les composants, blocs fautifs de la puce. Ce composant « Reconfigurable Hardware Unit » se charge ensuite de remplacer les composants fautifs actuellement utilisés à l'aide du chargement d'une configuration adaptée, stockée en mémoire.

D'autres techniques de reconfiguration existent, mais nous avons choisi de citer ces trois techniques, car elles permettent de comprendre rapidement la notion de reconfiguration. Pour être efficace, la reconfiguration doit pouvoir avoir lieu lors de l'exécution (*runtime*) d'une tâche par la puce [21, 23, 25] : un bloc ou un composant fautif doit pouvoir être reconfiguré indépendamment des autres blocs/composants de la puce, qui peuvent ainsi exécuter d'autres instructions, traiter d'autres calculs en parallèle.

Le NoC que nous considérons dans ce chapitre peut être sujet à des fautes : des routeurs et composants peuvent ainsi être désactivés et ne plus ainsi être capable de transmettre des paquets. Nous nous intéressons ici à une reconfiguration capable de réactiver ces routeurs et composants désactivés. Lors de la modélisation du NoC, que nous présentons dans la section suivante, nous ne donnons qu'une spécifi-

cation abstraite de ce mécanisme : un composant/un routeur fautif est réactivé, mais nous ne décrivons pas le *comment*, c'est-à-dire la/les techniques utilisées pour reconfigurer les parties de la puce en faute. Des étapes de raffinement supplémentaires peuvent permettre de définir précisément ce mécanisme. Les pannes et le mécanisme de reconfiguration sont des propositions faites par les électroniciens des laboratoires LIM de l'Université de Tiaret (Algérie) et LICM de l'Université de Lorraine. Nous avons étudié le NoC sous son aspect réseau et algorithme de routage : nous avons réutilisé le développement formel du patron de conception pour le routage, auquel nous avons intégré les spécificités du NoC proposées par les électroniciens.

6.3.2 Plan du développement

Nous décrivons dans cette section le développement formel du paradigme de « Réseau-sur-puce » (NoC). Nous présentons ici les différents niveaux de raffinement composant le modèle du « Réseau-sur-puce » :

- La machine M0 décrit l'abstraction du problème qui nous intéresse ici : il s'agit d'un problème de routage, d'acheminement de paquets entre des sources et des destinations.
- Le premier raffinement M1 introduit les identités des routeurs/nœuds participant au routage, dont les sources de paquets et les destinations.
- Le deuxième raffinement M2 introduit le système réparti (réseau) support du routage, ainsi que le mécanisme de reconfiguration du réseau, en cas de n oe uds défaillants.
- Le troisième raffinement M3 introduit l'existence de canaux entre les nœuds du réseau.
- Les cinq raffinements suivants détaillent graduellement sur la structure d'un routeur :
  - La machine M4 introduit l'existence d'un module IP par routeur.
  - La machine M5 associe à chaque routeur des ports de sortie.
  - La machine M6 associe à chaque routeur des ports d'entrée.
  - La machine M7 détaille chaque port d'entrée et de sortie d'un routeur en y ajoutant un tampon (buffer) caractérisé par un nombre de places limitées. Nous avons ici un modèle d'un NoC générique, sans architecture définie du réseau. Il suffit d'instancier ce modèle en lui définissant une architecture de réseau, pour obtenir des NoCs 2-D (carrés, rectangles, etc), 3-D (cubes, etc).
  - La machine M8 instancie M7 et modélise un NoC 2-D. Des détails tels que l'existence de quatre directions (Nord, Sud, Est, Ouest) sont introduits dans cette machine. L'algorithme XY est aussi modélisé dans cette machine M8.

Une utilisation des patrons de conception. Ce plan de développement nous a permis de remarquer que les trois premiers niveaux du développement : M0, M1, M2 étaient similaires aux étapes M00, M01 et M02 (page 156 à page 193) proposées par le patron de conception pour le routage. Ce développement par raffinement du problème du NoC nous a ainsi permis d'expérimenter sur la réutilisation et la génération de modèles et de preuves, notamment via l'utilisation du greffon « Pattern » [13] qui automatise la démarche d'instanciation d'un patron de conception : il permet de choisir le patron de conception à instancier et le problème à résoudre correspondant.

Les sous-sections suivantes détaillent les différents niveaux de raffinement obtenus lors du développement du problème du NoC.

6.3.3 Développement formel du problème du NoC

6.3.3.1 Abstraction

Nous commençons par abstraire le comportement attendu du « Réseau-sur-puce » (NoC) : nous le présentons comme un simple algorithme de routage dont l'objectif est la diffusion *« unicast »* ***sans perte, ni renvoi*** (caractéristique proposée par nos collaborateurs électroniciens du LIM et du LICM, et justifiée par le mécanisme de ***reconfiguration***) d'un paquet d'un cœur source à un cœur destination de la puce. Pour cela, nous définissons dans un contexte C0 un ensemble de paquets $PACKETS$ fini ($axm2$) et non-vide ($axm1$).

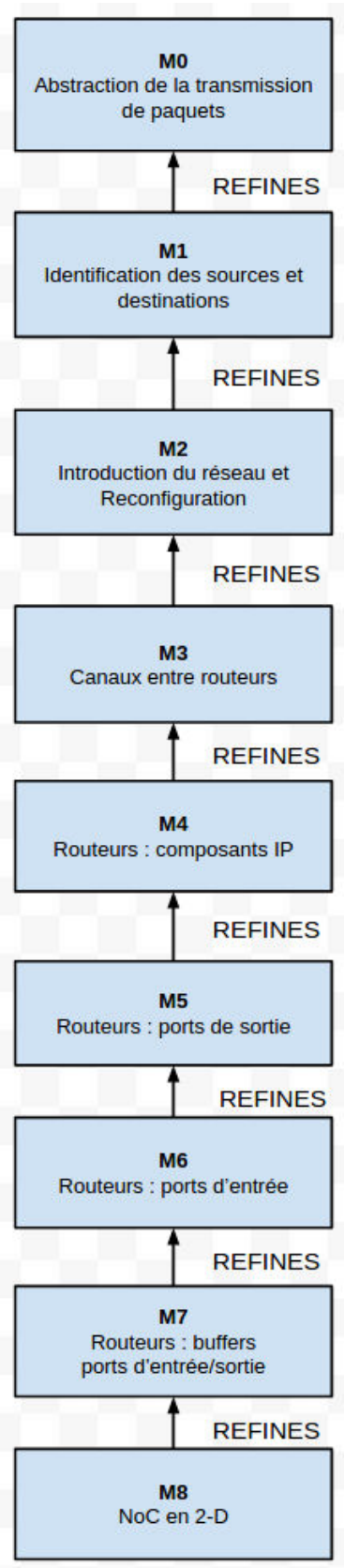


FIGURE 6.17 – NoC : plan du développement

CONTEXT C0
SETS
$PACKETS$
AXIOMS
$axm1$: $PACKETS \neq \varnothing$
$axm2$: $finite(PACKETS)$
END

Ce contexte sera utilisé par la machine abstraite M0.

Nous introduisons dans la machine abstraite M0 deux variables, sous-ensembles de $PACKETS$:

— *sent* contient les paquets envoyés par des sources.

— *rcvd* contient les paquets reçus par des destinataires.

Nous notons x l'ensemble des variables de cette machine M0.

Nous définissons aussi dans la machine abstraite M0 deux événements : SENDING0 et RECEIVING0 modélisant respectivement l'envoi et la réception d'un paquet. Nous détaillerons ces événements un peu plus loin dans cette section. L'hypothèse d'équité L_0 que nous posons pour le modèle M0 est la suivante :

$$L_0 \ \widehat{=}\ WF_x(\mathsf{SENDING0}) \wedge WF_x(\mathsf{RECEIVING0})$$

Nous posons des hypothèses d'équité faibles sur les deux événements.

Nous caractérisons la machine abstraite M0 par le diagramme de preuves 6.18.

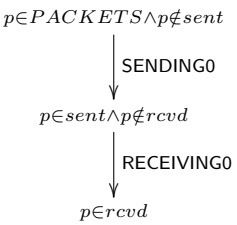


FIGURE 6.18 – Diagramme d'assertions pour le modèle abstrait

et définissons les propriétés suivantes :

— $P \ \widehat{=}\ (p \in PACKETS \wedge p \notin sent)$, qui exprime qu'un paquet p n'a pas encore été envoyé par une source.

— $S \ \widehat{=}\ (p \in sent \wedge p \notin rcvd)$, qui exprime qu'un paquet p a été envoyé par une source et n'est pas reçu par son(ses) destinataire(s) : il transite entre sa source et son(ses) destinataire(s).

— $Q \ \widehat{=}\ (p \in rcvd)$, exprimant qu'un paquet p est reçu par son(ses) destinataire(s).

Le diagramme 6.18 nous permet d'exprimer simplement la diffusion d'un paquet p par une source à sa(ses) destination(s), à l'aide des propriétés de vivacité suivantes :

1. $P \rightsquigarrow S$ qui exprime qu'un paquet p non-encore envoyé par sa source, transitera fatalement entre cette dernière et les destinataires.
2. $S \rightsquigarrow Q$ qui exprime qu'un paquet p transitant vers ses destinataires sera fatalement reçu par ces derniers.

L'ensemble Φ_0 des propriétés caractérisant M0 est donc : $\Phi_0 \ \widehat{=}\ \{P \rightsquigarrow S, S \rightsquigarrow Q\}$.

Démonstration. Nous montrons maintenant que la machine M0 satisfait les propriétés de vivacité de Φ_0.

⟨1⟩1. Nous montrons que M0 satisfait $P \rightsquigarrow S$. Une hypothèse d'équité faible est posée sur l'événement SENDING0 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de SENDING0 sont observables lorsque P est vrai, leurs observations ne falsifient pas le prédicat P, condition d'observation de SENDING0, ou conduisent à S.

Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$P \wedge [\mathsf{NEXT}]_x \Rightarrow (P' \vee S')$$

Soit :

— $P \wedge BA(\mathsf{SENDING0})(x, x') \Rightarrow (P' \vee S')$

— $P \wedge BA(\mathsf{RECEIVING0})(x, x') \Rightarrow (P' \vee S')$
— $P \wedge (x = x') \Rightarrow (P' \vee S')$

La propriété suivante **(3)** : $P(x) \wedge BA(\mathsf{SENDING0})(x, x') \Rightarrow S(x')$, et la condition de faisabilité **(4)** : $P(x) \Rightarrow (\exists x' \cdot BA(\mathsf{SENDING0})(x, x'))$ sont satisfaites par l'événement SENDING0. La propriété **(3)** nous permet de déduire $P \wedge \langle \mathsf{NEXT} \wedge \mathsf{SENDING0} \rangle_x \Rightarrow S'$ et **(4)** nous permet de déduire $P \Rightarrow ENABLED\langle \mathsf{SENDING0} \rangle_x$, où $ENABLED\langle \mathsf{SENDING0} \rangle_x \ \widehat{=} \ (\exists y \cdot BA(\mathsf{SENDING0})(x, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M0}) \vdash P \rightsquigarrow S$, donc que M0 satisfait $P \rightsquigarrow S$. □

⟨1⟩2. Nous montrons que M0 satisfait $S \rightsquigarrow Q$. Une hypothèse d'équité faible est posée sur l'événement RECEIVING0 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de RECEIVING0 sont observables lorsque S est vrai, leurs observations ne falsifient pas le prédicat S, condition d'observation de RECEIVING0, ou conduisent à Q.

Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$S \wedge [\mathsf{NEXT}]_x \Rightarrow (S' \vee Q')$$

Soit :

— $S \wedge BA(\mathsf{SENDING0})(x, x') \Rightarrow (S' \vee Q')$
— $S \wedge BA(\mathsf{RECEIVING0})(x, x') \Rightarrow (S' \vee Q')$
— $S \wedge (x = x') \Rightarrow (S' \vee Q')$

La propriété suivante **(3)** : $S(x) \wedge BA(\mathsf{RECEIVING0})(x, x') \Rightarrow Q(x')$, et la condition de faisabilité **(4)** : $S(x) \Rightarrow (\exists x' \cdot BA(\mathsf{RECEIVING0})(x, x'))$ sont satisfaites par l'événement RECEIVING0. La propriété **(3)** permet de déduire $S \wedge \langle \mathsf{NEXT} \wedge \mathsf{RECEIVING0} \rangle_x \Rightarrow Q'$ et **(4)** permet de déduire $S \Rightarrow ENABLED\langle \mathsf{RECEIVING0} \rangle_x$, où $ENABLED\langle \mathsf{RECEIVING0} \rangle_x \ \widehat{=} \ (\exists y \cdot BA(\mathsf{RECEIVING0})(x, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M0}) \vdash S \rightsquigarrow Q$, donc que M0 satisfait $S \rightsquigarrow Q$. □

⟨1⟩3. Les étapes ⟨1⟩1, ⟨1⟩2 nous permettent de déduire que M00 satisfait les propriétés de vivacité de Φ_0. QED.

□

Nous nous aidons de ces propriétés de vivacité de Φ_0 et du patron de conception pour le routage pour détailler la machine Event-B abstraite M0 du NoC :

— Nous pouvons identifier des éléments de cette abstraction avec certains des éléments présentés par le patron de conception :
  — L'ensemble des paquets $PACKETS$ est un renommage de l'ensemble MSG de messages présenté dans le patron.
  — La variable *sent* du modèle M0 peut être identifiée avec la variable *sent* définie dans l'abstraction M00 du patron.
  — Il en est de même pour la variable *rcvd* de la machine M0 et la variable *rcvd* du patron.
  — Nous avons les événements SENDING0 et RECEIVING0 de la machine M0 qui sont des instanciations des événements SENDING0 et RECEIVING0 du modèle abstrait M00 du patron.

  Dans un premier temps, il s'agit de recopier ces éléments du patrons dans l'abstraction du « Réseau-sur-puce » (NoC) et de les renommer (éventuellement) pour les adapter à notre problème courant.
— Nous remarquons aussi que des notions présentées par le patron de conception ne sont pas présentes dans le problème courant : il s'agit de la perte et du renvoi de paquets supposés perdus. Cela implique que lors de l'instanciation des modèles fournis par le patron de conception, pour la modélisation du « Réseau-sur-puce » (NoC), nous supprimons toutes les éléments (événements, variables, invariants, etc) modélisant les pertes et renvoi de paquets. Dans cette abstraction M0, nous ne réutilisons par conséquent pas la variable *lost*, les invariants qui relatifs à *lost*, ainsi que les événements LOSING0 et RESENDING0 fournis par le patron de conception.
— L'abstraction du patron de conception pour le routage, ainsi que les propriétés de vivacité définies par le diagramme 6.18, nous guident dans l'élaboration de la machine abstraite M0. Les deux variables *sent* et *rcvd* sont initialisées à l'aide de l'ensemble vide ($\varnothing$), car à l'état initial, aucun paquet n'a été envoyé ou reçu par les cœurs (nœuds) du NoC.

INITIALISATION $\widehat{=}$
BEGIN
$sent, rcvd := \varnothing$
END

$inv1 : sent \subseteq PACKETS$
$inv2 : rcvd \subseteq PACKETS$
$inv3 : rcvd \subseteq sent$

Les invariants $inv1$ et $inv2$ permettent de typer les variables $sent$ et $rcvd$: ce sont des sous-ensembles de $PACKETS$. L'invariant $inv3$ exprime que chaque paquet reçu par un destinataire a été au préalable envoyé par une source.

— L'événement SENDING0 est une recopie de l'événement correspondant du patron de conception :

EVENT SENDING0 $\widehat{=}$
ANY
p
WHERE
$grd1 : p \in PACKETS$
$grd2 : p \notin sent$
THEN
$act1 : sent := sent \cup \{p\}$
END

Un paquet p, qui n'a pas encore été envoyé ($grd2$), est envoyé ($act1$) par une source à un destinataire.

— L'événement RECEIVING0 est une adaptation de l'événement correspondant du patron de conception :

EVENT RECEIVING0 $\widehat{=}$
ANY
p
WHERE
$grd1 : p \in PACKETS$
$grd2 : p \in sent$
$grd3 : p \notin rcvd$
THEN
$act1 : rcvd := rcvd \cup \{p\}$
END

Un paquet p, qui a été envoyé ($grd2$) par une source et qui n'a pas encore été reçu ($grd3$) par un destinataire, est reçu ($act1$) par un destinataire. Nous remarquons que la garde ($grd3$) a été simplifié : nous ne tenons pas en compte le fait que le paquet ne soit pas perdu avant la réception, puisque le routage auquel nous nous intéressons est un routage sans perte.

Nous pouvons résumer cette abstraction M0 par le diagramme 6.19 suivant :

$$\mathsf{C0} \xleftarrow{\text{SEES}} \mathsf{M0} \xrightarrow{\text{LIVE}} \left(\left(\begin{array}{l} \wedge WF_x(\text{SENDING0}) \\ \wedge WF_x(\text{RECEIVING0}) \end{array} \right), \left\{ \begin{array}{l} P \rightsquigarrow S, \\ S \rightsquigarrow Q \end{array} \right\} \right)$$

FIGURE 6.19 – Abstraction du NoC

Ce premier modèle abstrait présente le comportement attendu du NoC : il s'agit d'une communications entre des cœurs dans une puce, par diffusion de paquet. Nous précisons un peu plus ce mécanisme dans la section suivante, en identifiant les sources et les destinataires des paquets.

6.3.3.2 Identification des routeurs

L'étape suivante de notre développement formel est l'introduction des identités des sources et destinataires des paquets. Pour cela, nous appliquons le premier raffinement décrit par le patron de conception pour le routage : nous commençons par définir dans un contexte C1 étendant le précédent contexte C0, un ensemble de nœuds (cœurs/routeurs) $ROUTERS$ non-vide ($axm1$). Par rapport au patron de conception, nous ajoutons une propriété supplémentaire à cet ensemble $ROUTERS$: il s'agit d'un ensemble fini ($axm2$). Nous y définissons aussi des constantes src et dst qui associent respectivement chaque paquet à un routeur source ($axm3$) et à un routeur destinataire ($axm4$). Nous exprimons aussi le fait que la source d'un paquet p et son destinataire son différents ($axm5$).

CONTEXT C1 EXTENDS C0
SETS
$ROUTERS$
CONSTANTS
src, dst
AXIOMS
$axm1 : ROUTEURS \neq \varnothing$
$axm2 : finite(ROUTERS)$
$axm3 : src \in PACKETS \rightarrow ROUTERS$
$axm4 : dst \in PACKETS \rightarrow ROUTERS$
$axm5 : \forall p \cdot p \in PACKETS \Rightarrow src(p) \neq dst(p)$
END

Ce contexte C1 sera utilisé par la machine M1. Nous introduisons dans cette machine M1 deux nouvelles variables concrètes, remplaçant les variables abstraites du modèle M0 précédent :

INITIALISATION $\widehat{=}$
BEGIN
$\ominus\ sent, rcvd := \varnothing$
$\oplus\ sent\_by\_core, rcvd\_by\_core := \varnothing$
END

— La variable *sent_by_core* remplace la variable *sent*. Elle indique les paquets envoyés ainsi que les routeurs sources qui les ont envoyés.
— La variable *rcvd_by_core* remplace la variable *rcvd*. Elle associe les paquets reçus aux routeurs destinataires qui les ont reçus.
— Nous remarquons ici que par rapport au patron de conception, nous n'utilisons pas de variable intermédiaire pour indiquer vers quel destinataire transite un paquet envoyé. Nous avons considéré que cette information était déjà encapsulée dans le paquet à envoyer et qu'il n'était donc pas nécessaire de modéliser cette encapsulation à l'envoi.

Nous notons x_1 l'ensemble des variables de cette machine M1.

Un invariant que nous notons $I_1(x_1)$ contraint ces variables et est défini comme suit :
— *inv*1 et *inv*2 permettent de typer les variables introduites dans cette machine M1.

$inv1 : sent\_by\_core \in ROUTERS \leftrightarrow PACKETS$
$inv2 : rcvd\_by\_core \in ROUTERS \leftrightarrow PACKETS$

— Nous modélisons une diffusion de paquets de type « *unicast* » entre un routeur source et un routeur destinataire uniques. Nous choisissons donc d'appliquer le premier raffinement M01 du patron de conception, correspondant à une diffusion vers un destinataire unique, pour construire les invariants de la machine M1 :
— Nous instancions les invariants vus dans le premier raffinement fourni par le patron pour caractériser la variable *sent_by_core* :
— *inv*3 exprime que si un paquet p a été diffusé par un routeur n, cela signifie que n est la source du paquet p.
— *inv*4 exprime qu'un paquet p est diffusé par une seule et unique source.
— *inv*5 exprime que chaque paquet envoyé au niveau concret l'est aussi au niveau abstrait. Cet invariant nous permet de remplacer la variable *sent* par *sent_by_core*.

$$
\begin{array}{l}
inv3 : \forall n, p \cdot n \in ROUTERS \wedge p \in PACKETS \wedge n \mapsto p \in sent\_by\_core \Rightarrow n = src(p) \\
inv4 : \forall n1, n2, p \cdot \left(\begin{array}{l} \wedge n1 \in ROUTERS \\ \wedge n2 \in ROUTERS \\ \wedge p \in PACKETS \\ \wedge n1 \mapsto p \in sent\_by\_core \\ \wedge n2 \mapsto p \in sent\_by\_core \end{array} \right) \Rightarrow n1 = n2 \\
inv5 : ran(sent\_by\_core) = sent
\end{array}
$$

— De la même manière que précédemment, nous instancions les invariants vus dans le premier raffinement fourni par le patron pour caractériser la variable *rcvd_by_core* :
— *inv*6 exprime que si un paquet p a été reçu par un routeur n, cela signifie que n est le destinataire du paquet p.

— *inv*7 exprime qu'un paquet p est reçu par une seule et unique destination.
— *inv*8 exprime que chaque paquet reçu au niveau concret l'est aussi au niveau abstrait. Cet invariant nous permet de remplacer la variable *rcvd* par *rcvd_by_core*.

$$
\begin{array}{|ll}
inv6 & : \forall n, p \cdot n \in ROUTERS \land p \in PACKETS \land n \mapsto p \in rcv\_by\_core \Rightarrow n = dst(p) \\
inv7 & : \forall n1, n2, p \cdot \left(\begin{array}{l} \land n1 \in ROUTERS \\ \land n2 \in ROUTERS \\ \land p \in PACKETS \\ \land n1 \mapsto p \in rcvd\_by\_core \\ \land n2 \mapsto p \in rcvd\_by\_core \end{array} \right) \Rightarrow n1 = n2 \\
inv8 & : ran(rcvd\_by\_core) = rcvd
\end{array}
$$

— L'invariant *inv*9 exprime que chaque paquet reçu par un routeur destinataire a été au préalable envoyé par un routeur source.

$$
\boxed{inv9 \ : \ ran(rcvd\_by\_core) \subseteq ran(sent\_by\_core)}
$$

Nous raffinons aussi les événements SENDING0 et RECEIVING0 du modèle précédent en deux événements SENDING1 et RECEIVING1, que nous détaillerons un peu plus tard. L'hypothèse d'équité L_1 que nous posons pour cette machine M1 est la suivante :

$$L_1 \ \widehat{=} \ WF_{x_1}(\mathsf{SENDING1}) \land WF_{x_1}(\mathsf{RECEIVING1})$$

Nous posons des contraintes d'équité faibles sur les événements SENDING1 et RECEIVING1.

Le diagramme de preuves 6.20 caractérise le raffinement M1, et décrit la liste Φ_1 des propriétés de vivacité satisfaites par M1, en tenant compte des hypothèses d'équité définies ci-dessus. Nous posons :

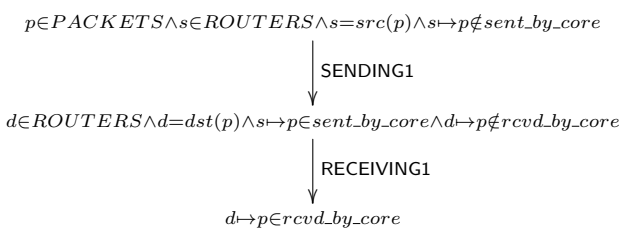


FIGURE 6.20 – Diagramme d'assertions pour le premier raffinement

— $P_1 \ \widehat{=} \ \left(\begin{array}{l} \land p \in PACKETS \\ \land s \in ROUTERS \\ \land s = src(p) \\ \land s \mapsto p \notin sent\_by\_core \end{array} \right)$
Cette propriété exprime qu'un paquet p n'a pas encore été envoyé par sa source s.
— $S_1 \ \widehat{=} \ \left(\begin{array}{l} \land d \in ROUTERS \\ \land d = dst(p) \\ \land s \mapsto p \in sent\_by\_core \\ \land d \mapsto p \notin rcvd\_by\_core \end{array} \right)$
Cette propriété exprime qu'un paquet p a été envoyé par sa source s et n'est pas reçu par son destinataire d : il transite entre la source s et le destinataire d.
— $Q_1 \ \widehat{=} \ d \mapsto p \in rcvd\_by\_core$
Cette propriété exprime qu'un paquet p a été reçu par son destinataire d.

L'ensemble Φ_1 des propriétés caractérisant M1 est défini comme suit :

$$\Phi_1 \ \widehat{=} \ \{P_1 \rightsquigarrow S_1, S_1 \rightsquigarrow Q_1\}$$

Démonstration. Nous montrons maintenant que la machine M1 satisfait les propriétés de vivacité de Φ_1.

⟨1⟩1. Nous montrons que M1 satisfait $P_1 \rightsquigarrow S_1$. Une hypothèse d'équité faible est posée sur l'événement SENDING1 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de SENDING1 sont observables lorsque P_1 est vrai, leurs observations ne falsifient pas le prédicat P_1, condition d'observation de SENDING1, ou conduisent à S_1.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$P_1 \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow (P_1' \vee S_1')$$

Soit :
— $P_1 \wedge BA(\mathsf{SENDING1})(x_1, x_1') \Rightarrow (P_1' \vee S_1')$
— $P_1 \wedge BA(\mathsf{RECEIVING1})(x_1, x_1') \Rightarrow (P_1' \vee S_1')$
— $P_1 \wedge (x_1 = x_1') \Rightarrow (P_1' \vee S_1')$
La propriété suivante **(3)** : $P_1(x_1) \wedge BA(\mathsf{SENDING1})(x_1, x_1') \Rightarrow S_1(x_1')$, et la condition de faisabilité **(4)** : $P_1(x_1) \Rightarrow (\exists x_1' \cdot BA(\mathsf{SENDING1})(x_1, x_1'))$ sont satisfaites par l'événement SENDING1. La propriété **(3)** nous permet de déduire $P_1 \wedge \langle \mathsf{NEXT} \wedge \mathsf{SENDING1} \rangle_{x_1} \Rightarrow S_1'$ et **(4)** permet de déduire $P_1 \Rightarrow ENABLED\langle \mathsf{SENDING1} \rangle_{x_1}$, où $ENABLED\langle \mathsf{SENDING1} \rangle_{x_1} \ \widehat{=}\ (\exists y \cdot BA(\mathsf{SENDING1})(x_1, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M1}) \vdash P_1 \rightsquigarrow S_1$, donc que M1 satisfait $P_1 \rightsquigarrow S_1$. □

⟨1⟩2. Nous montrons que M1 satisfait $S_1 \rightsquigarrow Q_1$. Une hypothèse d'équité faible est posée sur l'événement RECEIVING1 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de RECEIVING1 sont observables lorsque S_1 est vrai, leurs observations ne falsifient pas le prédicat S_1, condition d'observation de RECEIVING1, ou conduisent à Q_1.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$S_1 \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow (S_1' \vee Q_1')$$

Soit :
— $S_1 \wedge BA(\mathsf{SENDING1})(x_1, x_1') \Rightarrow (S_1' \vee Q_1')$
— $S_1 \wedge BA(\mathsf{RECEIVING1})(x_1, x_1') \Rightarrow (S_1' \vee Q_1')$
— $S_1 \wedge (x_1 = x_1') \Rightarrow (S_1' \vee Q_1')$
La propriété suivante **(3)** : $S_1(x_1) \wedge BA(\mathsf{RECEIVING1})(x_1, x_1') \Rightarrow Q_1(x_1')$, et la condition de faisabilité **(4)** : $S_1(x) \Rightarrow (\exists x_1' \cdot BA(\mathsf{RECEIVING0})(x_1, x_1'))$ sont satisfaites par l'événement RECEIVING0. La propriété **(3)** permet de déduire $S_1 \wedge \langle \mathsf{NEXT} \wedge \mathsf{RECEIVING1} \rangle_{x_1} \Rightarrow Q_1'$ et **(4)** permet de déduire $S_1 \Rightarrow ENABLED\langle \mathsf{RECEIVING1} \rangle_{x_1}$, où $ENABLED\langle \mathsf{RECEIVING0} \rangle_{x_1} \ \widehat{=}\ (\exists y \cdot BA(\mathsf{RECEIVING1})(x_1, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M1}) \vdash S_1 \rightsquigarrow Q_1$, donc que M0 satisfait $S_1 \rightsquigarrow Q_1$. □

⟨1⟩3. Les étapes ⟨1⟩1, ⟨1⟩2 nous permettent de déduire que M00 satisfait les propriétés de vivacité de Φ_1. QED.

□

Soit Φ_0 l'ensemble des propriétés caractérisant M0 :

$$\Phi_0 \ \widehat{=}\ \{P \rightsquigarrow S, S \rightsquigarrow Q\}$$

Démonstration. Nous montrons que M1 satisfait les propriétés de Φ_0. Auparavant, nous établissons quelques équivalences :
— Les invariant $inv3$, $inv4$, $inv5$ permettent d'établir l'équivalence **(1)** $P_1 \Leftrightarrow P$.
— Les invariant $inv3$, $inv4$, $inv5$, $inv6$, $inv7$ et $inv8$ permettent d'établir l'équivalence **(2)** $S_1 \Leftrightarrow S$.
— Les invariant $inv6$, $inv7$, $inv8$ permettent d'établir l'équivalence **(3)** $Q_1 \Leftrightarrow Q$.

⟨1⟩1. Nous montrons que M1 satisfait $P \rightsquigarrow S$. Nous savons que M1 satisfait $P_1 \rightsquigarrow S_1$. Les équivalences **(1)** et **(2)** permettent de déduire que M1 satisfait $P \rightsquigarrow S$. □

⟨1⟩2. Nous montrons que M1 satisfait $S \rightsquigarrow Q$. Nous savons que M1 satisfait $S_1 \rightsquigarrow Q_1$. Les équivalences **(2)** et **(3)** permettent de déduire que M1 satisfait $S \rightsquigarrow Q$. □

□

Nous nous aidons ensuite des propriétés de vivacité de Φ_1 et du patron de conception pour le routage pour détailler la machine M1 du NoC :

— L'événement SENDING1, raffinement de SENDING0 prend maintenant en compte l'identité d'une source s ($grd3$) diffusant un paquet p : si le paquet p n'a pas encore été diffusé par sa source s ($grd4$), alors cette dernière diffuse le paquet p ($act1$).

EVENT SENDING1 REFINES SENDING0 $\widehat{=}$
ANY
s, p
WHERE
$grd1$: $s \in ROUTERS$
$grd2$: $p \in PACKETS$
$grd3$: $s = src(p)$
$grd4$: $s \mapsto p \notin sent\_by\_core$
THEN
$act1$: $sent\_by\_core := sent\_by\_core \cup \{s \mapsto p\}$
END

— Il en est de même pour l'événement RECEIVING1, concernant le destinataire d ($grd3$) d'un paquet p : si le destinataire d n'a pas encore reçu le paquet p ($grd5$), alors il le reçoit ($act1$).

EVENT RECEIVING1 REFINES RECEIVING0 $\widehat{=}$
ANY
d, p
WHERE
$grd1$: $d \in ROUTERS$
$grd2$: $p \in PACKETS$
$grd3$: $d = dst(p)$
$grd4$: $p \in ran(sent\_by\_core)$
$grd5$: $d \mapsto p \notin rcvd\_by\_core$
THEN
$act1$: $rcvd\_by\_core := rcvd\_by\_core \cup \{d \mapsto p\}$
END

Nous pouvons résumer ce premier raffinement M1 par le diagramme 6.21 suivant :

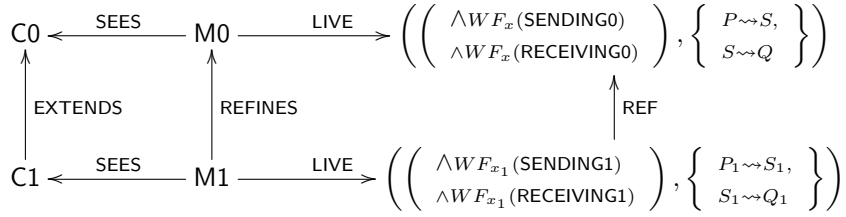


FIGURE 6.21 – Premier raffinement du NoC

Ce premier raffinement identifie les acteurs du routage dans un NoC : il s'agit de routeurs permettant des communications entre des cœurs dans une puce, par diffusion de paquet. Nous précisons un peu plus ce mécanisme dans la section suivante, en rajoutant le système réparti permettant à ces routeurs de communiquer. Le détail des preuves et des développements formels des raffinements suivants sont donnés en annexe [6].

6.3.3.3 Réseau et reconfiguration

Le raffinement que nous introduisons consiste à définir dans les modèles du « réseau-sur-puce » (NoC) le réseau (graphe) reliant les cœurs/routeurs de la puce et par lequel ces derniers communiquent. Nous commençons par définir un contexte C2 étendant le contexte C1 vu dans la section précédente. Nous y définissons une fonction *closure* qui associe à un graphe sa fermeture transitive reflexive ($axm1$, $axm2$), ainsi que le réseau (graphe) original $ntwrk$ ayant les propriétés suivantes ($axm3$) : $ntwrk$ est non-vide, symétrique, non-réflexif, tous les routeurs appartiennent à $ntwrk$ et $ntwrk$ est fortement connexe.

6. `http://andriami.bruno.free.fr/andriami/main/short/annexe.pdf`

CONTEXT C2 EXTENDS C1
CONSTANTS
$closure, ntwrk$
AXIOMS
$axm1 : closure \in (ROUTERS \leftrightarrow ROUTERS) \rightarrow (ROUTERS \leftrightarrow ROUTERS)$
$axm2 : \forall r, s \cdot \left(\begin{array}{l} \wedge r \subseteq closure(r) \\ \wedge closure(r); r \subseteq closure(r) \\ \wedge r \subseteq s \\ \wedge s; r \subseteq s \Rightarrow closure(r) \subseteq s \end{array} \right)$
$axm3 : ntwrk \in ROUTERS \leftrightarrow ROUTERS$
$axm4 : \left(\begin{array}{l} \wedge ntwrk \neq \varnothing \\ \wedge ntwrk = ntwrk^{-1} \\ \wedge ROUTERS \lhd id \cap ntwrk = \varnothing \\ \wedge ran(src) \cup ran(dst) \subseteq dom(ntwrk) \\ \wedge ROUTERS = dom(ntwrk) \\ \wedge ROUTERS = ran(ntwrk) \\ \wedge (ROUTERS \times ROUTERS) \setminus id \subseteq closure(ntwrk) \end{array} \right)$
END

Nous remarquons ici que par rapport au second niveau de raffinement fourni par le patron de conception, nous ne définissons pas l'ensemble des graphes/réseaux possibles : cela n'est pas nécessaire, car les réseaux possibles ne sont finalement que des variations du réseau original, c'est-à-dire des sous-graphes de *ntwrk*.

Nous définissons ensuite une machine M2 raffinant la machine M1 précédente et utilisant le contexte C2. Nous introduisons dans cette machine deux nouvelles variables *network* et *str* : *network* représente le réseau et *str* permet de savoir dans quels routeurs, dans le réseau, se situe les paquets diffusés par leurs sources.

INITIALISATION $\widehat{=}$
BEGIN
$\ldots$
$\oplus$ $str := \varnothing$
$\oplus$ $network := ntwrk$
END

Nous notons x_2 l'ensemble des variables de cette machine M2.

Un invariant que nous notons $I_2(x_2)$ contraignent ces variables et est défini comme suit :

$inv1 : str \in ROUTERS \leftrightarrow PACKETS$
$inv2 : network \subseteq ROUTERS \times ROUTERS$
$inv3 : \forall p, n1, n2 \cdot \left(\begin{array}{l} \wedge n1 \in ROUTERS \\ \wedge n2 \in ROUTERS \\ \wedge p \in PACKETS \\ \wedge n1 \mapsto p \in str \\ \wedge n2 \mapsto p \in str \end{array} \right) \Rightarrow n1 = n2$
$inv4 : ran(str) \subseteq ran(sent\_by\_core)$
$inv5 : network \subseteq ntwrk$
$inv6 : network \neq \varnothing$
$inv7 : network = network^{-1}$
$inv8 : ROUTERS \lhd id \cap network = \varnothing$
$inv9 : \forall p, d \cdot p \in PACKETS \wedge d \in ROUTERS \wedge d = dst(p) \wedge p \in rcvd[\{d\}] \Rightarrow p \notin ran(str)$

— $inv1$ et $inv2$ sont des propriétés de typage.
— $inv3$ exprime qu'un paquet p ne peut se trouver que dans un seul routeur du réseau.
— $inv4$ exprime que chaque paquet circulant dans le réseau a été envoyé par une source.
— $inv5$ exprime que le réseau/graphe courant est un sous-graphe du réseau original *ntwrk*.
— Les invariants $inv6$, $inv7$, $inv8$ expriment respectivement que le réseau courant *network* n'est pas vide, est symétrique et n'est pas réflexif.
— $inv9$ exprime qu'un paquet p reçu par un destinataire d et stocké par ce dernier, ne circule plus dans le réseau.

Cette machine contient des événements SENDING2 et RECEIVING2, raffinements de SENDING1 et RECEIVING1 ; un événement FORWARDING2, modélisant l'acheminement d'un paquet dans le réseau, de routeur en routeur ; un événement DISABLING2, modélisant la désactivation d'un routeur et le retrait de ses liens avec tous ses voisins ; et un événement RECONFIGURING2, modélisation la réactivation d'un routeur désactivé et des liens entre ce routeur et ses voisins.

Nous nous intéressons maintenant aux propriétés de vivacités satisfaites par le second raffinement

M2. Une liste Φ_2 de propriétés de vivacité (cf. annexe [7]) nous permet de raffiner la machine M1 en une machine M2 :

— Nous modifions par raffinement l'événement SENDING1 : le raffinement SENDING2 prend maintenant en compte le fait que la source s d'un paquet p à envoyer doit être active, faire partie du réseau ($grd5$) et que lors de l'envoi, le paquet p est déposé par sa source dans le réseau ($act2$). L'événement SENDING2 est une instanciation de l'événement SENDING2 correspondant du patron de conception. Nous avons adapté l'événement du pattern en en supprimant des paramètres (le destinataire vers qui le paquet p est envoyé) et en rajoutant une garde supplémentaire relative à l'activation du routeur source s.

EVENT SENDING2
REFINES SENDING1 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\oplus$ $grd5$: $s \in dom(network)$
  THEN
    ...
    $\oplus$ $act2$: $str := str \cup \{s \mapsto p\}$
  END

— Un nouvel événement FORWARDING2 modélise le transfert ($act1$) d'un paquet p d'un nœud x ($grd4$) à un nœud y ($grd5$) voisin ($grd6$), jusqu'à ce que sa destination $dst(p)$ soit atteinte ($grd7$). Nous avons ici une instanciation de l'événement FORWARD2 correspondant du patron de conception.

EVENT FORWARDING2 $\widehat{=}$
  ANY
    x, y, p
  WHERE
    $grd1$: $x \in ROUTERS$
    $grd2$: $y \in ROUTERS$
    $grd3$: $p \in PACKETS$
    $grd4$: $x \mapsto p \in str$
    $grd5$: $y \mapsto p \notin str$
    $grd6$: $x \mapsto y \in network$
    $grd7$: $x \neq dst(p)$
  THEN
    $act1$: $str := (str \setminus \{x \mapsto p\}) \cup \{y \mapsto p\}$
  END

— L'événement RECEIVING2 est observé si le paquet p se trouve au niveau de sa destination d dans le réseau ($grd4$) et si cette destination est active ($grd6$). Dans ce cas, le paquet p est reçu par la destination d et est retiré du réseau ($act2$). De la même manière que l'événement SENDING2, nous avons ici une instanciation de l'événement RECEIVING2 du patron de conception que nous avons adapté.

EVENT RECEIVING2
REFINES RECEIVING1 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd4$: $p \in ran(sent\_by\_core)$
    $\oplus$ $grd4$: $d \mapsto p \in str$
    ...
    $\oplus$ $grd6$: $d \in dom(network)$
  THEN
    ...
    $\oplus$ $act2$: $str := str \cup \{d \mapsto p\}$
  END

— Nous introduisons aussi dans ce modèle M2 deux événements DISABLING2 et RECONFIGURING2 de modification du graphe.

7. `http://andriami.bruno.free.fr/andriami/main/short/annexe.pdf`

— L'événement DISABLING2 du modèle M2 est une instanciation de l'événement REMOVE_LINK2 du second niveau de raffinement du modèle fourni par le patron. Nous avons adapté ici l'événement REMOVE_LINK2 au problème de la modélisation du NoC : il ne s'agit plus de retirer du réseau simplement les liens existant entre deux routeurs voisins, mais de retirer ($grd4$, $act1$) plutôt tous les liens existants entre un routeur r et ses voisins et de désactiver ainsi ce routeur r. Cet événement est observé si le réseau obtenu après la désactivation du routeur r est non-vide ($grd5$)

EVENT DISABLING2 $\widehat{=}$
ANY
$r, suppr$
WHERE
$grd1 : r \in ROUTERS$
$grd2 : suppr \subseteq ROUTERS \times ROUTERS$
$grd3 : r \in dom(network)$
$grd4 : suppr = (\{r\} \times network[\{r\}])$
$grd5 : network \setminus (suppr \cup suppr^{-1}) \neq \varnothing$
THEN
$act1 : network := network \setminus (suppr \cup suppr^{-1})$
END

— L'événement RECONFIGURING2 est une instanciation de l'événement ADD_LINK2 du second niveau de raffinement du modèle fourni par le patron. Il ne s'agit pas ici d'une copie exacte de cet événement ADD_LINK2, mais plutôt d'une adaptation. L'événement ADD_LINK2 modélise l'ajout dans un réseau de liens qui n'existaient pas au préalable entre deux routeurs, alors que ce que l'événement RECONFIGURING2 modélise est la réactivation dans le réseau d'un routeur désactivé : le routeur désactivé ($grd3$) est réactivé à la même place où il se trouvait dans le réseau et les liens (qui ont disparu du réseau) entre le routeur et ses voisins sont rétablis ($grd4$, $act1$).

EVENT RECONFIGURING2 $\widehat{=}$
ANY
$r, reconf$
WHERE
$grd1 : r \in ROUTERS$
$grd2 : reconf \subseteq ROUTERS \times ROUTERS$
$grd3 : r \notin dom(network)$
$grd4 : reconf = \{r\} \times ntwrk[\{r\}]$
THEN
$act1 : network := network \cup (reconf \cup reconf^{-1})$
END

Nous pouvons résumer ce deuxième raffinement M2 par le diagramme 6.22. Ce diagramme nous montre aussi que les propriétés de vivacité Φ_1 du niveau de raffinement précédent sont satisfaites par M2 (voir l'annexe pour la preuve). Lors de ces trois premiers raffinements, nous avons détaillé un mecanisme de routage généraliste, qui consiste à faire transiter un paquet envoyé par un routeur source dans un réseau, de routeur en routeur, jusqu'à ce que le paquet atteigne sa destination. Nous avons aussi montré lors du développement de ces trois premiers raffinement l'application du patron de conception pour le routage vu dans le chapitre précédent : nous avons identifié les parties des modèles correspondant à ceux définis dans le patron de conception et nous avons soit réutilisé directement les éléments définis dans ces modèles, soit nous les avons adaptés au problème du NoC. Les prochains raffinements se concentrent plus sur la structure du NoC et des cœurs/routeurs qui le composent.

6.3.3.4 Canaux et routeurs

Nous commençons par détailler dans ce troisième raffinement la structure du « Réseau-sur-puce » (NoC), en introduisant des canaux entre les différents routeurs du réseau. Pour cela, nous définissons dans un contexte C3 étendant le contexte C2. Nous y définissons un ensemble de canaux $CHANNELS$

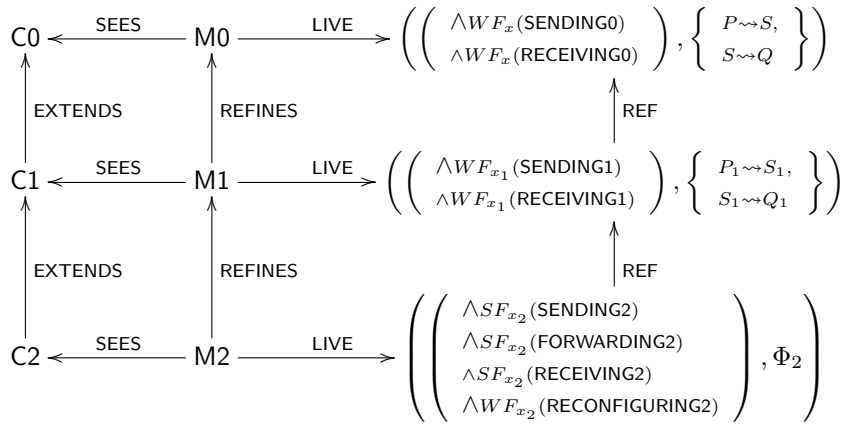


FIGURE 6.22 – Deuxième raffinement du NoC

non-vide ($axm1$) et nous associons à chaque lien du réseau (reliant deux routeurs) le canal correspondant ($axm2$), à l'aide d'une bijection *chan*. Nous notons ici que nous associons un canal à deux routeurs : les deux routeurs se trouvent aux extrémités du canal.

CONTEXT C3 EXTENDS C2
SETS
$CHANNELS$
CONSTANTS
$chan$
AXIOMS
$axm1$: $CHANNELS \neq \varnothing$
$axm2$: $chan \in ntwrk \rightarrowtail\!\!\!\!\!\rightarrow CHANNELS$
END

Ce contexte sera utilisé par la machine M3 raffinant la machine M2 vue dans la section précédente.

Nous introduisons dans ce raffinement M3, deux nouvelles variables : *rtr_cnt* et *chn_cnt*.

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus$ $str := \varnothing$
$\oplus$ $rtr\_cnt := \varnothing$
$\oplus$ $chn\_cnt := \varnothing$
END

Ces variables modélisent respectivement le contenu des routeurs dans le réseau ($inv1$) (quels paquets se trouvent au niveau d'un routeur dans le réseau *network*) et le contenu des canaux du réseau ($inv2$) (quels paquets transitent dans un canal du réseau *network*) et remplacent la variable *str* que nous supprimons de nos modèles. Ces variables sont initialisées à l'aide de l'ensemble vide car à l'état initial, aucun paquet ne circule dans les routeurs et canaux. Nous notons x_3 les variables de cette machine M3.

Un invariant noté $I_3(x_3)$ contraignant ces variables est défini comme suit :

— $inv1$ et $inv2$ permettent de typer les nouvelles variables introduites dans M3 :

$inv1$: $rtr\_cnt \in ROUTERS \leftrightarrow PACKETS$
$inv2$: $chn\_cnt \in CHANNELS \leftrightarrow PACKETS$

— Les propriétés suivantes, qui sont des invariants de collage permettent le remplacement de *str* par *rtr_cnt* et *chn_cnt* :

$$
\begin{array}{l}
inv3 \ : \ \forall r, p \cdot \left(\begin{array}{l} \wedge r \in ROUTERS \\ \wedge p \in PACKETS \\ \wedge r \mapsto p \in rtr\_cnt \end{array} \right) \Rightarrow r \mapsto p \in str \\
inv4 \ : \ \forall r, c, p \cdot \left(\begin{array}{l} \wedge r \in ROUTERS \\ \wedge c \in CHANNELS \\ \wedge p \in PACKETS \\ \wedge prj2(chan^{-1}(c)) = r \\ \wedge c \mapsto p \in chn\_cnt \end{array} \right) \Rightarrow r \mapsto p \in str \\
inv5 \ : \ \forall r, p \cdot \left(\begin{array}{l} \wedge r \in ROUTERS \\ \wedge p \in PACKETS \\ \wedge r \mapsto p \in str \end{array} \right) \Rightarrow \left(\begin{array}{l} \vee r \mapsto p \in rtr\_cnt \\ \vee \exists c \cdot \left(\begin{array}{l} \wedge c \in CHANNELS \\ \wedge prj2(chan^{-1}(c)) = r \\ \wedge c \mapsto p \in chn\_cnt \end{array} \right) \end{array} \right)
\end{array}
$$

— $inv3$ exprime que si un paquet p se trouve dans un routeur r dans le réseau, alors ce paquet p circule dans le réseau au niveau du routeur r.
— $inv4$ exprime que si un routeur r se trouve au bout d'un canal c qui le relie à un autre routeur, et qu'un paquet p circule dans ce canal c, alors cela signifie qu'au niveau abstrait, le paquet p se trouve dans le réseau au niveau du routeur r.
— $inv5$ exprime que si un paquet p se trouve au niveau du routeur r dans le réseau, au niveau abstrait, cela signifie soit que le routeur r contient le paquet p, soit que le routeur r est relié par un canal c à un autre routeur et le paquet p circule dans ce canal c.

— D'autres propriétés permettent d'exprimer la sûreté :

$$
\begin{array}{l}
inv6 \ : \ ran(chn\_cnt) \cap ran(rtr\_cnt) = \varnothing \\
inv7 \ : \ ran(chn\_cnt) \cup ran(rtr\_cnt) \subseteq ran(sent\_by\_core) \\
inv8 \ : \ \forall p, n1, n2 \cdot \left(\begin{array}{l} \wedge n1 \in ROUTERS \\ \wedge n2 \in ROUTERS \\ \wedge p \in PACKETS \\ \wedge n1 \mapsto p \in rtr\_cnt \\ \wedge n2 \mapsto p \in rtr\_cnt \end{array} \right) \Rightarrow n1 = n2 \\
inv9 \ : \ \forall p, c1, c2 \cdot \left(\begin{array}{l} \wedge c1 \in CHANNELS \\ \wedge c2 \in CHANNELS \\ \wedge p \in PACKETS \\ \wedge c1 \mapsto p \in chn\_cnt \\ \wedge c2 \mapsto p \in chn\_cnt \end{array} \right) \Rightarrow c1 = c2
\end{array}
$$

— $inv6$ exprime que l'intersection des contenus des canaux et routeurs est vide : un paquet circulant dans le réseau se trouve soit dans un routeur, soit dans un canal.
— $inv7$ exprime que chaque paquet circulant dans les canaux ou les routeurs a été envoyé par une source.
— $inv8$ exprime qu'un paquet p ne peut circuler dans plusieurs routeurs différents en même temps, il ne peut circuler à un moment donné que dans un routeur unique.
— $inv9$ exprime qu'un paquet p ne peut circuler dans plusieurs canaux différents en même temps, il ne peut circuler à un moment donné que dans un canal unique.

Des raffinements des événements précédents et de nouveaux événements sont introduits :

— SENDING3, RECEIVING3 pour les envois et réceptions de paquets.
— RECONFIGURING3, DISABLING3 pour la reconfiguration et la désactivation de routeurs.
— FORWARDING_ROUTER_CHANNEL3, FORWARDING_CHANNEL_ROUTER3, décompositions de FORWARDING2, pour l'acheminement de paquets : respectivement **(1)** d'un routeur à un canal adjacent et **(2)** d'un canal à un routeur adjacent.

Une liste Φ_3 de propriétés de vivacité, détaillée en annexe, nous permet raffiner M2 en M3 :

— Lors de l'envoi d'un paquet p par sa source s, modélisé par l'événement SENDING3, le paquet p à envoyer est mis sur le réseau, dans le routeur s ($act2$). Cette action est possible, si le paquet p n'a pas encore été mis dans le routeur s ($grd5$) sur le réseau et si le routeur s est actif et possède des voisins ($grd6$).

EVENT SENDING3
REFINES SENDING2 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd5$: $s \in dom(network)$
    $\oplus$ $grd5$: $p \notin rtr\_cnt[\{s\}]$
    $\oplus$ $grd6$: $network[\{s\}] \neq \varnothing$
  THEN
    ...
    $\ominus$ $act2$: $str := str \cup \{s \mapsto p\}$
    $\oplus$ $act2$: $rtr\_cnt := rtr\_cnt \cup \{s \mapsto p\}$
  END

— Nous détaillons l'événement FORWARDING2 en le décomposant en deux parties : un événement FORWARDING_ROUTER_CHANNEL3 modélisant le transfert d'un paquet d'un routeur à un canal connecté au routeur et un événement FORWARDING_CHANNEL_ROUTER3 modélisant le passage d'un paquet d'un canal à un routeur adjacent.
  — FORWARDING_ROUTER_CHANNEL3 : le paquet p se trouve dans un routeur, que nous notons x et qui est le routeur « entrant » connecté à un canal c ($grd3$) ($x = prj1(chan^{-1}(c))$) ; le paquet p ne se trouve pas dans le canal c ($grd5$) et x n'est pas la destination de p : p peut donc passer du routeur x au canal c ($act1$, $act2$).

EVENT FORWARDING_ROUTER_CHANNEL3
REFINES FORWARDING2 $\widehat{=}$
  ANY
    c, p
  WHERE
    $grd1$: $c \in CHANNELS$
    $grd2$: $p \in PACKETS$
    $grd3$: $prj1(chan^{-1}(c)) \mapsto p \in rtr\_cnt$
    $grd4$: $chan^{-1}(c) \in network$
    $grd5$: $c \mapsto p \notin chn\_cnt$
    $grd6$: $prj1(chan^{-1}(c)) \neq dst(p)$
  WITH
    x : $x = prj1(chan^{-1}(c))$
    y : $y = prj2(chan^{-1}(c))$
  THEN
    $act1$: $rtr\_cnt := rtr\_cnt \setminus \{prj1(chan^{-1}(c)) \mapsto p\}$
    $act2$: $chn\_cnt := chn\_cnt \cup \{c \mapsto p\}$
  END

  — FORWARDING_CHANNEL_ROUTER3 : le paquet p se trouve dans un canal c ($grd3$), connecté à routeur « sortant » que nous notons y ($y = prj2(chan^{-1}(c))$) ; le paquet p ne se trouve pas dans le routeur y ($grd4$) : p peut donc passer du canal c au routeur y ($act1$, $act2$).

EVENT FORWARDING_CHANNEL_ROUTER3 $\widehat{=}$
  ANY
    c, p
  WHERE
    $grd1$: $c \in CHANNELS$
    $grd2$: $p \in PACKETS$
    $grd3$: $c \mapsto p \in chn\_cnt$
    $grd4$: $prj2(chan^{-1}(c)) \mapsto p \notin rtr\_cnt$
    $grd5$: $chan^{-1}(c) \in network$
  THEN
    $act1$: $chn\_cnt := chn\_cnt \setminus \{c \mapsto p\}$
    $act2$: $rtr\_cnt := rtr\_cnt \cup \{prj2(chan^{-1}(c)) \mapsto p\}$
  END

— L'événement RECEIVING3, modélisant la réception d'un paquet p par un routeur destinataire d, est observé si le paquet p se trouve dans le routeur d sur le réseau ($grd5$), si le routeur d n'est pas désactivé et est relié à des voisins ($grd6$). Le paquet p est alors retiré du réseau ($act2$).

EVENT RECEIVING3
REFINES RECEIVING2 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd5$: $d \mapsto p \in str$
    $\ominus$ $grd6$: $d \in dom(network)$
    $\oplus$ $grd5$: $p \in rtr\_cnt[\{d\}]$
    $\oplus$ $grd6$: $network[\{d\}] \neq \varnothing$
  THEN
    ...
    $\ominus$ $act2$: $str := str \setminus \{d \mapsto p\}$
    $\oplus$ $act2$: $rtr\_cnt := rtr\_cnt \setminus \{d \mapsto p\}$
  END

Nous pouvons résumer ce troisième raffinement M3 par le diagramme 6.23 suivant :

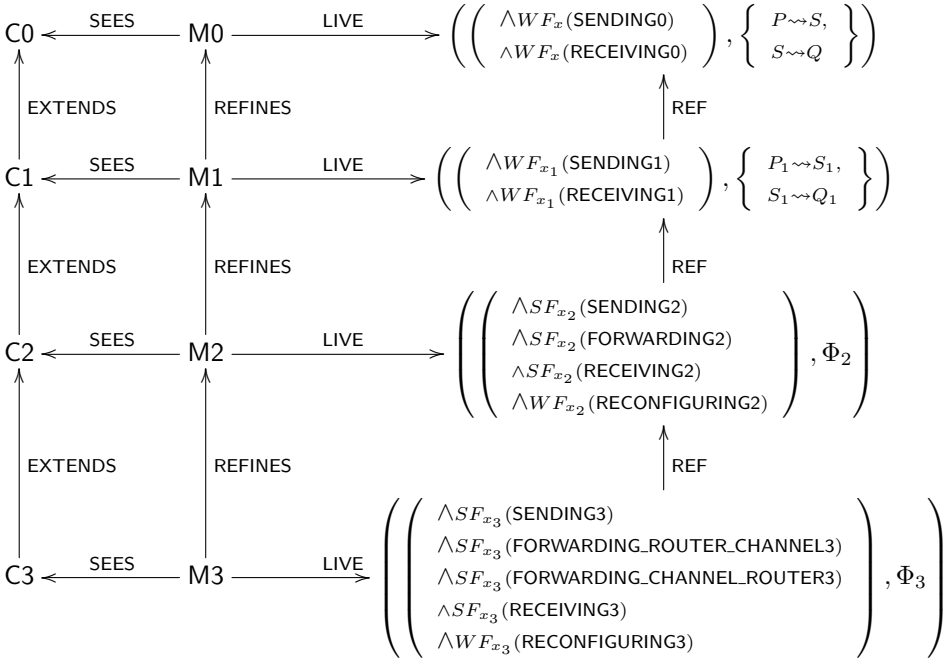


FIGURE 6.23 – Troisième raffinement du NoC

Ce diagramme nous montre aussi que les propriétés de vivacité Φ_2 du niveau de raffinement précédent sont satisfaites par M3 (voir l'annexe pour la preuve). Le raffinement suivant détaille encore plus la structure d'un routeur : nous introduisons les composants « Intellectual Property » (IP) au sein des routeurs.

6.3.3.5 Composants IP et routeurs

Ce quatrième raffinement nous permet de détailler l'architecture d'un routeur du NoC : nous décomposons ce dernier en une composante « Intellectual Property » (IP) et en une autre composante que nous appelons « routeur ». Ces deux composantes communiquent entre elles.

Dans un premier temps, nous définissons dans un contexte C4, un ensemble IP fini ($axm2$) et non-vide ($axm1$) représentant les composants IP. Nous associons ensuite chaque routeur à son composant IP, à l'aide d'une bijection *ips* ($axm3$). Les paquets à envoyer par des routeurs sources sont stockés dans les modules IP de ces derniers et il en est de même pour les paquets reçus : nous définissons dans le contexte, une fonction *ip_src*, qui associe à chaque module IP les paquets que le routeur propriétaire du module IP doit envoyer ($axm4$, $axm5$, $axm6$) ; nous en faisons de même pour les destinataires des paquets :

un paquet a maintenant pour destinataire la composante IP d'un routeur, ce que nous définissons à l'aide d'une fonction *ip_dst* ($axm7$, $axm8$, $axm9$). L'axiome $axm10$ est un théorème qui exprime que le composant IP source d'un paquet p est différent du composant IP destinataire de ce même paquet p.

CONTEXT C4 EXTENDS C3
SETS
IP
CONSTANTS
ips, ip_src, ip_dst
AXIOMS
$axm1 : IP \neq \varnothing$
$axm2 : finite(IP)$
$axm3 : ips \in ROUTERS \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow IP$
$axm4 : ip\_src \in PACKETS \rightarrow IP$
$axm5 : \forall p, ip \cdot p \in PACKETS \wedge ip \in IP \wedge p \mapsto ip \in ip\_src \Rightarrow ips^{-1}(ip) = src(p)$
$axm6 : \forall p, ip \cdot p \in PACKETS \wedge ip \in IP \wedge ips^{-1}(ip) = src(p) \Rightarrow p \mapsto ip \in ip\_src$
$axm7 : ip\_dst \in PACKETS \rightarrow IP$
$axm8 : \forall p, ip \cdot p \in PACKETS \wedge ip \in IP \wedge p \mapsto ip \in ip\_dst \Rightarrow ips^{-1}(ip) = dst(p)$
$axm9 : \forall p, ip \cdot p \in PACKETS \wedge ip \in IP \wedge ips^{-1}(ip) = dst(p) \Rightarrow p \mapsto ip \in ip\_dst$
$axm10 : \forall p \cdot p \in PACKETS \Rightarrow ip\_dst(p) \neq ip\_src(p)$
END

Ce contexte C4 est utilisé par une machine M4 raffinant le modèle M3 vu précédemment.

Nous introduisons dans ce modèle M4 une nouvelle variable *ip_cnt*, qui modélise le contenu (en termes de paquets) de chaque composant IP de chaque routeur du NoC ($inv1$). Le contenu initial d'un module IP d'un routeur est composé des paquets que le routeur doit envoyer : nous utilisons l'inverse de la fonction *ip_src* pour initialiser la variable *ip_cnt*.

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus\ sent\_by\_core, rcvd\_by\_core := \varnothing$
$\oplus\ ip\_cnt := ip\_src^{-1}$
END

Nous notons x_4 l'ensemble des variables de la machine M4.

Un invariant noté $I_4(x_4)$ caractérise cette machine M4 et est défini comme suit :

— La propriété $inv1$ permet de typer la variable *ip_cnt* :

$$inv1 : ip\_cnt \in IP \leftrightarrow PACKETS$$

— Nous supprimons dans cette machine M4 les variables *sent_by_core* et *rcvd_by_core* : en effet, la présence d'un paquet p dans un composant IP d'un routeur, nous permet de dire si celui-ci a été envoyé ou reçu. Les propriétés suivantes permettent de remplacer *sent_by_core* et *rcvd_by_core* par la variable *ip_cnt*.

$$inv2 : \forall p, i \cdot \left(\begin{array}{l} \wedge i \in IP \\ \wedge p \in PACKETS \\ \wedge i = ip\_src(p) \\ \wedge i \mapsto p \in ip\_cnt \end{array} \right) \Rightarrow p \notin ran(sent\_by\_core)$$

$$inv3 : \forall p, i \cdot \left(\begin{array}{l} \wedge i \in IP \\ \wedge p \in PACKETS \\ \wedge i = ip\_dst(p) \\ \wedge i \mapsto p \notin ip\_cnt \end{array} \right) \Rightarrow p \notin ran(rcvd\_by\_core)$$

$$inv4 : \forall p \cdot \left(\begin{array}{l} \wedge p \in PACKETS \\ \wedge ips(dst(p)) \mapsto p \in ip\_cnt \end{array} \right) \Rightarrow dst(p) \mapsto p \in rcvd\_by\_core$$

$$inv5 : \forall p \cdot \left(\begin{array}{l} \wedge p \in PACKETS \\ \wedge dst(p) \mapsto p \in rcvd\_by\_core \end{array} \right) \Rightarrow ips(dst(p)) \mapsto p \in ip\_cnt$$

— $inv2$ exprime que si un paquet p n'a pas encore quitté le composant IP de son routeur source, cela signifie qu'il n'a pas encore été envoyé par le routeur source.

— $inv3$ exprime que si un paquet p ne se trouve pas dans le composant IP de son routeur destinataire, cela signifie qu'il n'a pas encore été reçu par ce dernier.

— $inv4$ exprime que si un paquet p se trouve dans le composant IP de son routeur destinataire, cela signifie qu'il a été reçu par ce dernier.

— $inv5$ exprime que si un paquet p a été reçu par son routeur destinataire, alors il se trouve dans le composant IP de ce dernier.

— Les propriétés de $inv6$ à $inv9$ expriment la sûreté :

$$
\begin{array}{l}
inv6 \ : \ \forall p, i1, i2 \cdot \left(\begin{array}{l} \wedge i1 \in IP \\ \wedge i2 \in IP \\ \wedge p \in PACKETS \\ \wedge i1 \mapsto p \in ip\_cnt \\ \wedge i2 \mapsto p \in ip\_cnt \end{array} \right) \Rightarrow i1 = i2 \\
inv7 \ : \ ran(ip\_cnt) \cap ran(rtr\_cnt) = \varnothing \\
inv8 \ : \ ran(ip\_cnt) \cap ran(chn\_cnt) = \varnothing \\
inv9 \ : \ \forall ip, p \cdot \left(\begin{array}{l} \wedge ip \in IP \\ \wedge p \in PACKETS \\ \wedge ip \mapsto p \in ip\_cnt \end{array} \right) \Rightarrow ips^{-1}(ip) \in \{src(p), dst(p)\}
\end{array}
$$

— $inv6$ exprime qu'un paquet p ne peut se trouver que dans un composant IP unique.
— $inv7$ exprime que chaque paquet dans un routeur est soit dans le composant IP du routeur, soit dans le composant « routeur » qui y est connecté.
— $inv8$ exprime que l'intersection des contenus des composants IP des routeurs et des canaux du réseau est vide.
— $inv9$ exprime que si un paquet p se trouve dans un composant IP ip d'un routeur, alors ce composant IP ip appartient soit au routeur source, soit au routeur destination du paquet p.

Nous retrouvons dans cette machine les raffinements des événements de la machine précédente, c'est-à-dire :

— SENDING4, RECEIVING4 pour les envois et réceptions de paquets.
— RECONFIGURING4, DISABLING4 pour la reconfiguration et la désactivation de routeurs.
— FORWARDING_ROUTER_CHANNEL4, FORWARDING_CHANNEL_ROUTER4 pour l'acheminement de paquets : respectivement **(1)** d'un routeur à un canal adjacent et **(2)** d'un canal à un routeur adjacent.

Les remplacements de variables et constantes obtenus par raffinement de données sont résumés par le tableau suivant :

Variable/Constante abstraite	**Variable/Constante concrète**
src	*ip_src*
dst	*ip_dst*
sent_by_core	*ip_cnt*
rcvd_by_core	*ip_cnt*

Les axiomes du contexte C4, ainsi que l'invariant de collage I_4 nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M3 en M4, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M4, ainsi que pour justifier la préservation des propriétés Φ_3 lors du raffinement.

Une liste Φ_4 de propriétés de vivacité, détaillée en annexe, nous permet de raffiner M3 en M4 :

— L'événement SENDING4 prend en compte le composant IP ip du routeur source d'un paquet p lors de l'envoi de ce dernier : si un paquet p se trouve dans le composant IP ip de cette source ($grd3$), s'il ne se trouve pas dans le composant « routeur » de la source ($grd6$) et si la source n'est pas désactivée ($grd6$), alors le paquet p se déplace du composant IP ip de sa source ($act2$) au composant « routeur » de ce dernier ($act1$).

EVENT SENDING4 REFINES SENDING3 $\widehat{=}$
ANY
ip, p
WHERE
$grd1 \ : \ ip \in IP$
$grd2 \ : \ p \in PACKETS$
$grd3 \ : \ p \in ip\_cnt[\{ip\}]$
$grd4 \ : \ ip = ip\_src(p)$
$grd5 \ : \ network[\{src(p)\}] \neq \varnothing$
$grd6 \ : \ p \notin rtr\_cnt[\{src(p)\}]$
WITH
$s \ : \ s = src(p) \wedge s = ips^{-1}(ip)$
THEN
$act1 \ : \ rtr\_cnt := rtr\_cnt \cup \{src(p) \mapsto p\}$
$act2 \ : \ ip\_cnt := ip\_cnt \setminus \{ip \mapsto p\}$
END

— L'événement RECEIVING4 tient compte du composant IP ip du routeur destinataire d'un paquet p lors de la réception de ce dernier : si un paquet p se trouve dans le composant « routeur » de son destinataire ($grd5$), s'il n'est pas encore stocké dans le composant IP ip du destinataire ($grd4$) et que ce dernier n'est pas désactivé ($grd6$), alors le paquet p passe du composant « routeur » ($act2$) au composant IP ($act1$).

EVENT RECEIVING4 REFINES RECEIVING3 $\widehat{=}$
ANY
ip, p
WHERE
$grd1 \ : \ ip \in IP$
$grd2 \ : \ p \in PACKETS$
$grd3 \ : \ ip = ip\_dst(p)$
$grd4 \ : \ p \notin ip\_cnt[\{ip\}]$
$grd5 \ : \ p \in rtr\_cnt[\{dst(p)\}]$
$grd6 \ : \ network[\{dst(p)\}] \neq \varnothing$
WITH
$d \ : \ d = dst(p) \wedge d = ips^{-1}(ip)$
THEN
$act1 \ : \ ip\_cnt := ip\_cnt \cup \{ip \mapsto p\}$
$act2 \ : \ rtr\_cnt := rtr\_cnt \setminus \{dst(p) \mapsto p\}$
END

Nous pouvons résumer ce quatrième raffinement M4 par le diagramme 6.24 suivant :

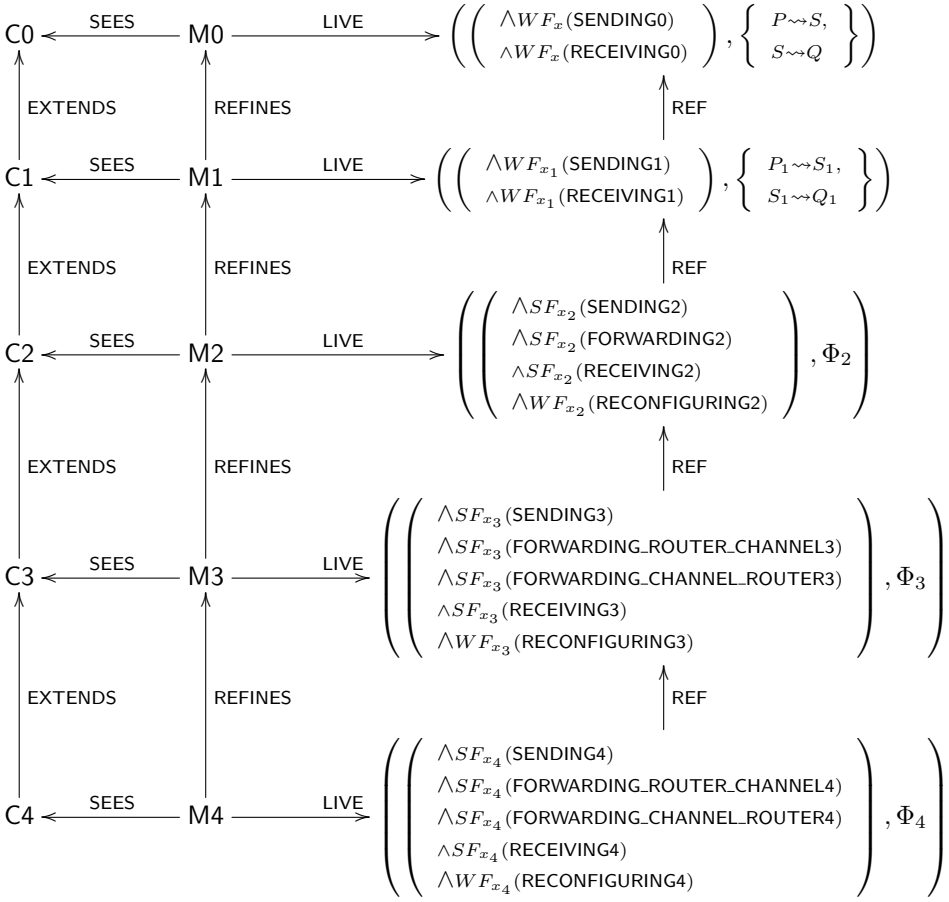


FIGURE 6.24 – Quatrième raffinement du NoC

Le prochain raffinement détaille encore plus la structure d'un routeur en y ajoutant les ports de sortie.

6.3.3.6 Routeurs : ports de sortie

Pour ce cinquième raffinement, nous commençons par définir dans un contexte C5 les ports de sortie des routeurs, à l'aide d'un ensemble $OUTPUTPORTS$. Nous associons ensuite à chaque routeur, à l'aide d'une bijection *out_p*, des ports de sortie menant vers les routeurs voisins et un port de sortie interne menant vers le composant IP du routeur ($axm1$).

CONTEXT C5 EXTENDS C4
SETS
$OUTPUTPORTS$
CONSTANTS
out_p
AXIOMS
$axm1$: $out\_p \in (ntwrk \cup (dom(ntwrk) \lhd id)) \rightarrowtail OUTPUTPORTS$
END

Ce contexte C5 est utilisé par une machine M5 raffinant la machine M4 vu précédemment.

La machine M5 décompose maintenant un routeur en trois composants : un composant IP, un composant « routeur » et un composant port de sortie.

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus$ $rtr\_cnt := \varnothing$
$\oplus$ $out\_cnt, routr\_cnt := \varnothing$
END

— La variable *out_cnt* modélise le contenu en termes de paquets de chaque port de sortie de chaque routeur du réseau ($inv1$).
— La variable *routr_cnt* modélise les paquets contenu dans les routeurs ($inv2$).

Nous notons x_5 l'ensemble des variables la machine M5.

Un invariant que nous notons $I_5(x_5)$ contraint ces variables et est défini comme suit :
— $inv1$ et $inv2$ permettent le typage des variables *out_cnt* et *routr_cnt* :

$$\begin{array}{l} inv1 \ : \ out\_cnt \in OUTPUTPORTS \leftrightarrow PACKETS \\ inv2 \ : \ routr\_cnt \in ROUTERS \leftrightarrow PACKETS \end{array}$$

— Les propriétés $inv3$ et $inv4$ sont des invariants de collage nous permettant de remplacer la variable *rtr_cnt* par les variables *routr_cnt* et *out_cnt* :

$$\begin{array}{l} inv3 \ : \ \forall p, r \cdot \left(\begin{array}{l} \wedge p \in PACKETS \\ \wedge r \in ROUTERS \\ \wedge \left(\begin{array}{l} \vee r \mapsto p \in routr\_cnt \\ \vee \exists o \cdot \left(\begin{array}{l} \wedge o \in OUTPUTPORTS \\ \wedge prj1(out\_p^{-1}(o)) = r \\ \wedge o \mapsto p \in out\_cnt \end{array} \right) \end{array} \right) \end{array} \right) \Rightarrow r \mapsto p \in rtr\_cnt \\ inv4 \ : \ \forall p, r \cdot \left(\begin{array}{l} \wedge p \in PACKETS \\ \wedge r \in ROUTERS \\ \wedge r \mapsto p \in rtr\_cnt \end{array} \right) \Rightarrow \left(\begin{array}{l} \vee r \mapsto p \in routr\_cnt \\ \vee \exists o \cdot \left(\begin{array}{l} \wedge o \in OUTPUTPORTS \\ \wedge prj1(out\_p^{-1}(o)) = r \\ \wedge o \mapsto p \in out\_cnt \end{array} \right) \end{array} \right) \end{array}$$

— $inv3$ exprime que si un paquet p se trouve dans un routeur r ou dans un des ports de sortie o connecté à ce routeur r au niveau concret, alors le paquet p se trouve au niveau du routeur r au niveau abstrait.
— $inv4$ exprime que si un paquet p se trouve au niveau d'un routeur r au niveau abstrait, alors il se trouve soit dans ce même routeur r au niveau concret, soit dans un port de sortie connecté à ce routeur r.
— Les propriétés suivantes expriment la sûreté :

$$\begin{array}{l} inv5 \ : \ ran(out\_cnt) \cap ran(routr\_cnt) = \varnothing \\ inv6 \ : \ ran(out\_cnt) \cap ran(chn\_cnt) = \varnothing \\ inv7 \ : \ ran(routr\_cnt) \cap ran(chn\_cnt) = \varnothing \\ inv8 \ : \ ran(out\_cnt) \cap ran(ip\_cnt) = \varnothing \\ inv9 \ : \ ran(routr\_cnt) \cap ran(ip\_cnt) = \varnothing \\ inv10 \ : \ \forall p, r1, r2 \cdot \left(\begin{array}{l} \wedge r1 \in ROUTERS \\ \wedge r2 \in ROUTERS \\ \wedge p \in PACKETS \\ \wedge r1 \mapsto p \in routr\_cnt \\ \wedge r2 \mapsto p \in routr\_cnt \end{array} \right) \Rightarrow r1 = r2 \\ inv11 \ : \ \forall p, o1, o2 \cdot \left(\begin{array}{l} \wedge o1 \in OUTPUTPORTS \\ \wedge o2 \in OUTPUTPORTS \\ \wedge p \in PACKETS \\ \wedge o1 \mapsto p \in out\_cnt \\ \wedge o2 \mapsto p \in out\_cnt \end{array} \right) \Rightarrow o1 = o2 \end{array}$$

— Les invariants de $inv5$ à $inv9$ expriment que chaque paquet p dans le réseau ne se trouve que dans un seul endroit à la fois : soit dans un routeur, soit dans un port de sortie, soit dans un canal, soit dans un composant IP.
— $inv10$ exprime qu'un paquet p ne peut se trouver dans deux routeurs différents en même temps : il se trouve dans un routeur unique à un moment donné.
— $inv11$ exprime qu'un paquet p ne peut se trouver dans deux ports de sorties différents en même temps : un paquet p ne peut se trouver à un moment donné que dans un seul port de sortie.

Nous retrouvons dans cette machine les raffinements des événements de la machine précédente, c'est-à-dire :

— SENDING5, RECEIVING5 pour les envois et réceptions de paquets.
— RECONFIGURING5, DISABLING5 pour la reconfiguration et la désactivation de routeurs.
— FORWARDING_ROUTER_OUTPUTPORT5, FORWARDING_OUTPUTPORT_CHANNEL5, FORWARDING_CHANNEL_ROUTER5 pour l'acheminement de paquets : respectivement **(1)** d'un routeur à un port de sortie, **(2)** d'un port de sortie à un canal adjacent, **(3)** d'un canal à un routeur adjacent.

Une liste Φ_5 de propriétés de vivacité, détaillée en annexe, nous permet de raffiner M4 en M5 :

— L'événement SENDING5 consiste toujours à faire passer un paquet p non-encore envoyé, du composant IP d'une source ($act2$) au composant « routeur » de la source ($act1$).

EVENT SENDING5 REFINES SENDING4 $\widehat{=}$
  ANY
    ip, p
  WHERE
    $grd1$: $ip \in IP$
    $grd2$: $p \in PACKETS$
    $grd3$: $p \in ip\_cnt[\{ip\}]$
    $grd4$: $ip = ip\_src(p)$
    $grd5$: $network[\{src(p)\}] \neq \varnothing$
    $grd6$: $p \notin routr\_cnt[\{src(p)\}]$
  THEN
    $act1$: $routr\_cnt := routr\_cnt \cup \{src(p) \mapsto p\}$
    $act2$: $ip\_cnt := ip\_cnt \setminus \{ip \mapsto p\}$
END

— Les opérations de FORWARDING sont maintenant décomposés en trois événements :
  — FORWARDING_ROUTER_OUTPUTPORT5 modélise le transit d'un paquet p d'un routeur r à un port de sortie o : si un paquet p se trouve dans un routeur r ($grd5$), et que ce routeur r est la destination du paquet p ($grd8$), alors le paquet p est déplacé du routeur r au port de sortie o interne menant au composant IP du routeur r ($grd8$, $act1$, $act2$), sinon le paquet p est déplacé du routeur r à un port de sortie o menant à un voisin de r non-désactivé ($grd9$, $act1$, $act2$).

EVENT FORWARDING_ROUTER_OUTPUTPORT5 $\widehat{=}$
  ANY
    r, o, p
  WHERE
    $grd1$: $r \in ROUTERS$
    $grd2$: $o \in OUTPUTPORTS$
    $grd3$: $p \in PACKETS$
    $grd4$: $network[\{r\}] \neq \varnothing$
    $grd5$: $p \in routr\_cnt[\{r\}]$
    $grd6$: $r = prj1(out\_p^{-1}(o))$
    $grd7$: $p \notin out\_cnt[\{o\}]$
    $grd8$: $r = dst(p) \Rightarrow o = out\_p(r \mapsto r)$
    $grd9$: $r \neq dst(p) \Rightarrow (o \neq out\_p(r \mapsto r) \wedge prj2(out\_p^{-1}(o)) \in network[\{r\}])$
  THEN
    $act1$: $routr\_cnt := routr\_cnt \setminus \{r \mapsto p\}$
    $act2$: $out\_cnt := out\_cnt \cup \{o \mapsto p\}$
  END

  — FORWARDING_OUTPUTPORT_CHANNEL5 modélise le transfert d'un paquet p d'un port de sortie o à un canal c : un paquet p passe d'un port de sortie o à un canal c ($act1$, $act2$), si le port de sortie o est connecté au canal c, si les routeurs connectés par le canal c ne sont pas désactivés (appartiennent au réseau, $grd6$), si le paquet p est dans le port de sortie o ($grd4$), mais pas encore dans le canal c ($grd7$).

EVENT FORWARDING_OUTPUTPORT_CHANNEL5 REFINES FORWARDING_ROUTER_CHANNEL4 $\widehat{=}$
ANY
c, p, o
WHERE
$grd1$: $c \in CHANNELS$
$grd2$: $p \in PACKETS$
$grd3$: $o \in OUTPUTPORTS$
$grd4$: $p \in out\_cnt[\{o\}]$
$grd5$: $out\_p^{-1}(o) = chan^{-1}(c)$
$grd6$: $chan^{-1}(c) \in network$
$grd7$: $p \notin chn\_cnt[\{c\}]$
$grd8$: $prj1(chan^{-1}(c)) \neq dst(p)$
THEN
$act1$: $out\_cnt := out\_cnt \setminus \{o \mapsto p\}$
$act2$: $chn\_cnt := chn\_cnt \cup \{c \mapsto p\}$
END

— FORWARDING_CHANNEL_ROUTER5 modélise le passage d'un paquet p d'un canal c ($act1$) au routeur « sortant » qui y est connecté ($act2$) : l'événement est observé si le paquet p se trouve dans le canal c ($grd3$), s'il ne se trouve pas dans le routeur ($grd4$) et si les routeurs connectés par le canal c ne sont pas désactivés ($grd5$).

EVENT FORWARDING_CHANNEL_ROUTER5 REFINES FORWARDING_CHANNEL_ROUTER4 $\widehat{=}$
ANY
c, p
WHERE
$grd1$: $c \in CHANNELS$
$grd2$: $p \in PACKETS$
$grd3$: $p \in chn\_cnt[\{c\}]$
$grd4$: $p \notin routr\_cnt[\{prj2(chan^{-1}(c))\}]$
$grd5$: $chan^{-1}(c) \in network$
THEN
$act1$: $chn\_cnt := chn\_cnt \setminus \{c \mapsto p\}$
$act2$: $routr\_cnt := routr\_cnt \cup \{prj2(chan^{-1}(c)) \mapsto p\}$
END

— L'événement RECEIVING5 prend maintenant en compte le port de sortie interne menant au composant IP du routeur destinataire : un paquet p passe d'un port de sortie interne ($act2$) d'un routeur au composant IP de ce même routeur ($act1$), si le routeur est le routeur destinataire du paquet ($grd3$), si le paquet p n'est pas stocké dans le composant IP ($grd4$), s'il est dans le port de sortie interne ($grd5$) et si le destinataire du paquet p n'est pas désactivé ($grd6$).

EVENT RECEIVING5 REFINES RECEIVING4 $\widehat{=}$
ANY
ip, p
WHERE
$grd1$: $ip \in IP$
$grd2$: $p \in PACKETS$
$grd3$: $ip = ip\_dst(p)$
$grd4$: $p \notin ip\_cnt[\{ip\}]$
$grd5$: $p \in out\_cnt[\{out\_p(dst(p) \mapsto dst(p))\}]$
$grd6$: $network[\{dst(p)\}] \neq \varnothing$
THEN
$act1$: $ip\_cnt := ip\_cnt \cup \{ip \mapsto p\}$
$act2$: $out\_cnt := out\_cnt \setminus \{out\_p(dst(p) \mapsto dst(p)) \mapsto p\}$
END

Nous pouvons résumer ce cinquième raffinement M5 par le diagramme 6.25 suivant :

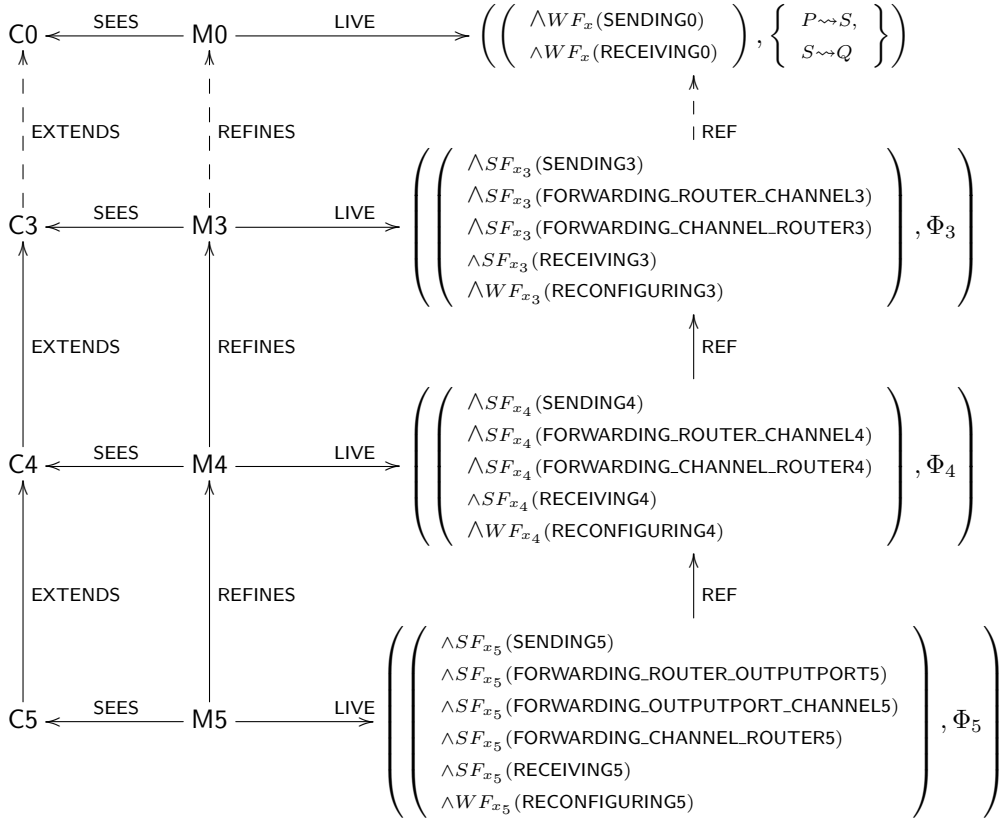


FIGURE 6.25 – Cinquième raffinement du NoC

Ce diagramme nous montre aussi que les propriétés de vivacité Φ_4 du niveau de raffinement précédent sont satisfaites par M5 (voir l'annexe pour la preuve). De la même manière que ce raffinement introduit les ports de sortie de chaque routeur, le raffinement suivant introduit les ports d'entrée de chaque routeur.

6.3.3.7 Routeurs : ports d'entrée

Ce sixième raffinement ajoute plus de détails à la structure d'un routeur : nous introduisons, en plus du composant IP, des ports de sortie, les ports d'entrée d'un routeur. Pour ce faire, nous définissons dans un contexte C6 étendant le contexte C5 vue précédemment, l'ensemble des ports d'entrée $INPUTPORTS$. Une bijection in_p nous permet d'associer à chaque routeur, des ports d'entrée externes, reliant le routeur à ses voisins et un port d'entrée interne reliant le composant IP du routeur à ce dernier ($axm1$).

CONTEXT C6 EXTENDS C5
SETS
$INPUTPORTS$
CONSTANTS
in_p
AXIOMS
$axm1$: $in\_p \in (ntwrk \cup (dom(ntwrk) \lhd id)) \rightarrowtail INPUTPORTS$
END

Ce contexte C6 est utilisé par une machine M6.

Nous introduisons dans la machine M6 une nouvelle variable in_cnt qui modélise le contenu (en termes de paquets) de chaque port d'entrée de chaque routeur du réseau ($inv1$).

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus$ $routr\_cnt := \varnothing$
$\oplus$ $in\_cnt := \varnothing$
END

Nous notons x_6 l'ensemble des variables de cette machine M6.

Un invariant que nous notons $I_6(x_6)$ caractérise ces variables x_6 :

— La propriété $inv1$ permet de typer la nouvelle variable in_cnt.

$$inv1 \ : \ in\_cnt \in INPUTPORTS \leftrightarrow PACKETS$$

— Nous supprimons dans ce modèle la variable $routr\_cnt$ et nous la remplaçons pas in_cnt. Les propriétés suivantes, qui sont appelés invariants de collage, permettent ce remplacement :

$$inv2 \ : \ \forall p, i \cdot \left(\begin{array}{l} \wedge p \in PACKETS \\ \wedge i \in INPUTPORTS \\ \wedge i \mapsto p \in in\_cnt \end{array} \right) \Rightarrow prj2(in\_p^{-1}(i)) \mapsto p \in routr\_cnt$$
$$inv3 \ : \ \forall p, r \cdot \left(\begin{array}{l} \wedge p \in PACKETS \\ \wedge r \in ROUTERS \\ \wedge r \mapsto p \in routr\_cnt \end{array} \right) \Rightarrow \exists i \cdot \left(\begin{array}{l} \wedge i \in INPUTPORTS \\ \wedge prj2(in\_p^{-1}(i)) = r \\ \wedge i \mapsto p \in in\_cnt \end{array} \right)$$

— $inv2$ exprime que si un paquet p se trouve dans un port d'entrée i au niveau concret, alors ce paquet p se trouve dans le routeur au bout de ce port d'entrée i au niveau abstrait.

— $inv3$ exprime que si un paquet p se trouve dans un routeur r au niveau abstrait, alors le paquet p se trouve au niveau concret dans un port d'entrée i menant au routeur r.

— Les autres propriétés expriment la sûreté :

$$inv4 \ : \ \forall p, i1, i2 \cdot \left(\begin{array}{l} \wedge i1 \in INPUTPORTS \\ \wedge i2 \in INPUTPORTS \\ \wedge p \in PACKETS \\ \wedge i1 \mapsto p \in in\_cnt \\ \wedge i2 \mapsto p \in in\_cnt \end{array} \right) \Rightarrow i1 = i2$$
$$inv5 \ : \ ran(out\_cnt) \cap ran(in\_cnt) = \varnothing$$
$$inv6 \ : \ ran(ip\_cnt) \cap ran(in\_cnt) = \varnothing$$
$$inv7 \ : \ ran(chn\_cnt) \cap ran(in\_cnt) = \varnothing$$

— $inv4$ exprime qu'un paquet p ne peut se trouver que dans un seul port d'entrée à un moment donné.

— Les invariants $inv5$, $inv6$, $inv7$, couplés aux invariants $inv5$ à $inv9$ de la machine M5 précédente, expriment que chaque paquet p dans le réseau ne se trouve que dans un seul endroit à la fois : soit dans un port d'entrée, soit dans un port de sortie, soit dans un canal, soit dans un composant IP.

Nous retrouvons dans cette machine les raffinements des événements de la machine précédente, c'est-à-dire :

— Les événements SENDING6, RECEIVING6 pour les envois et réceptions de paquets.

— Les événements RECONFIGURING6, DISABLING6 pour la reconfiguration et la désactivation de routeurs.

— Les événements FORWARDING_I_O6, FORWARDING_I_DEST_O6, FORWARDING_O_C6, FORWARDING_C_I6 pour l'acheminement de paquets : respectivement **(1)** d'un port d'entrée (interne ou externe) d'un routeur à un port de sortie (externe), **(2)** d'un port d'entrée (externe) du destinataire d'un paquet vers un port de sortie (interne) menant au module IP du destinataire, **(3)** d'un port de sortie (externe) à un canal adjacent, **(4)** d'un canal à un port d'entrée (externe).

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
$routr\_cnt$	in_cnt

Les axiomes du contexte C6, ainsi que l'invariant de collage I_6 nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M5 en M6, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés de vivacité Φ_6 satisfaites par M6, ainsi que pour justifier la préservation des propriétés Φ_5 lors du raffinement.

Une liste Φ_6 de propriétés de vivacité, détaillée en annexe, nous permet de raffiner M5 en M6 :

— L'événement SENDING6 modélise l'envoi par une source d'un paquet p : la source fait passer le paquet p de son composant IP ($act1$) à son port d'entrée interne ($act2$), si le paquet p se trouve dans le composant IP ($grd4$), mais pas dans le port d'entrée interne ($grd5$, $grd8$) et si le routeur source n'est pas désactivé.

EVENT SENDING6 REFINES SENDING5 $\widehat{=}$
ANY
ip, p, i
WHERE
$grd1 : ip \in IP$
$grd2 : p \in PACKETS$
$grd3 : i \in INPUTPORTS$
$grd4 : p \in ip\_cnt[\{ip\}]$
$grd5 : i = in\_p(src(p) \mapsto src(p))$
$grd6 : ip = ip\_src(p)$
$grd7 : network[\{src(p)\}] \neq \varnothing$
$grd8 : p \notin in\_cnt[\{i\}]$
THEN
$act1 : in\_cnt := in\_cnt \cup \{i \mapsto p\}$
$act2 : ip\_cnt := ip\_cnt \setminus \{ip \mapsto p\}$
END

— L'événement FORWARDING_I_O6 modélise le passage d'un paquet p d'un port d'entrée i ($act1$) à un port de sortie o ($act2$) menant à un routeur voisin ($grd9$), qui n'est pas le destinataire de p ($grd8$) : l'événement est observé si le paquet p se trouve dans le port d'entrée i ($grd5$), mais pas dans le port de sortie o ($grd7$), si les ports i et o appartiennent au même routeur ($grd6$), si le routeur et le voisin vers lequel p est acheminé ne sont pas désactivés ($grd10$).

EVENT FORWARDING_I_O6 REFINES FORWARD_ROUTER_OUTPUTPORT5 $\widehat{=}$
ANY
i, o, p
WHERE
$grd1 : i \in INPUTPORTS$
$grd2 : o \in OUTPUTPORTS$
$grd3 : p \in PACKETS$
$grd4 : network[\{prj2(in\_p^{-1}(i))\}] \neq \varnothing$
$grd5 : p \in in\_cnt[\{i\}]$
$grd6 : prj2(in\_p^{-1}(i)) = prj1(out\_p^{-1}(o))$
$grd7 : p \notin out\_cnt[\{o\}]$
$grd8 : prj2(in\_p^{-1}(i)) \neq dst(p)$
$grd9 : o \neq out\_p(prj2(in\_p^{-1}(i)) \mapsto prj2(in\_p^{-1}(i)))$
$grd10 : prj2(out\_p^{-1}(o)) \in network[\{prj1(out\_p^{-1}(o))\}]$
WITH
$r : r = prj2(in\_p^{-1}(i))$
THEN
$act1 : in\_cnt := in\_cnt \setminus \{i \mapsto p\}$
$act2 : out\_cnt := out\_cnt \cup \{o \mapsto p\}$
END

— L'événement FORWARDING_I_DEST_O6 est similaire à l'événement précédent. La différence réside dans le fait que le paquet p se trouve dans un des ports d'entrée du routeur destinataire. Il s'agit donc de le rediriger vers le port de sortie interne menant au module IP du routeur destinataire.

EVENT FORWARDING_I_DEST_O6 REFINES FORWARDING_ROUTER_OUTPUTPORT5 $\widehat{=}$
ANY
i, o, p
WHERE
$grd1 : i \in INPUTPORTS$
$grd2 : o \in OUTPUTPORTS$
$grd3 : p \in PACKETS$
$grd4 : network[\{prj2(in\_p^{-1}(i))\}] \neq \varnothing$
$grd5 : p \in in\_cnt[\{i\}]$
$grd6 : prj2(in\_p^{-1}(i)) = prj1(out\_p^{-1}(o))$
$grd7 : p \notin out\_cnt[\{o\}]$
$grd8 : prj2(in\_p^{-1}(i)) = dst(p)$
$grd9 : o = out\_p(prj2(in\_p^{-1}(i)) \mapsto prj2(in\_p^{-1}(i)))$
WITH
$r : r = prj2(in\_p^{-1}(i))$
THEN
$act1 : in\_cnt := in\_cnt \setminus \{i \mapsto p\}$
$act2 : out\_cnt := out\_cnt \cup \{o \mapsto p\}$
END

— L'événement FORWARDING_C_I6 modélise le passage d'un paquet p d'un canal c ($act1$) à un port d'entrée i ($act2$) : l'événement est observé si le paquet p se trouve dans le canal c ($grd4$) connecté au port d'entrée i ($grd5$), si le paquet p ne se trouve pas dans le port d'entrée i et que les routeurs reliés par le canal c ne sont pas désactivé ($grd7$).

EVENT FORWARDING_C_I6 REFINES FORWARDING_CHANNEL_ROUTER5 $\widehat{=}$
ANY
c, p, i
WHERE
$grd1 : c \in CHANNELS$
$grd2 : p \in PACKETS$
$grd3 : i \in INPUTPORTS$
$grd4 : p \in chn\_cnt[\{c\}]$
$grd5 : chan^{-1}(c) = in\_p^{-1}(i)$
$grd6 : p \notin in\_cnt[\{i\}]$
$grd7 : chan^{-1}(c) \in network$
THEN
$act1 : chn\_cnt := chn\_cnt \setminus \{c \mapsto p\}$
$act2 : in\_cnt := in\_cnt \cup \{i \mapsto p\}$
END

— FORWARDING_O_C6, raffinement de FORWARDING_OUTPUTPORT_CHANNEL5 n'est pas présenté car il n'est pas modifié.

Nous pouvons résumer ce sixième raffinement M6 par le diagramme 6.26. Le prochain raffinement introduits des caractéristiques des ports d'entrée, de sortie, ainsi que des canaux, telles que les nombres de places dans les ports, les états des canaux (occupés, libre), etc.

6.3.3.8 Routeurs : buffers d'entrées/sorties

Pour ce septième raffinement, nous commençons par définir dans un contexte C7 un ensemble $CHAN - STATE$ modélisant les états des canaux : libre (ch_free) ou occupé (ch_busy) ($axm1$, $axm2$). Nous y définissons aussi deux constantes $buffer\_size\_in$ et $buffer\_size\_out$ représentant respectivement les tailles des tampons des ports d'entrée ($axm3$) et de sortie ($axm4$). Les axiomes de $axm5$ à $axm12$ expriment les possibles variations des quantités de paquets se trouvant dans des ports d'entrée $i0$ et de sortie $o0$. Ils décrivent les situations pour lesquelles ces quantités augmentent, restent stable, ou diminuent. Ces axiomes sont notamment utilisés pour démontrer que ces quantités sont finies et bornées par $buffer\_size\_in$ (pour les ports d'entrée) et $buffer\_size\_out$ (pour les ports de sortie). Ce contexte C7 est utilisé par une machine M7.

Nous définissons une nouvelle variable chn_state dans la machine M7 : cette variable modélise l'état de chaque canal du réseau ($inv1$) et nous initialisons l'état de chaque canal à « libre » (ch_free).

INITIALISATION $\widehat{=}$
BEGIN
...
$\oplus\ chn\_state := CHANNELS \times \{ch\_free\}$
END

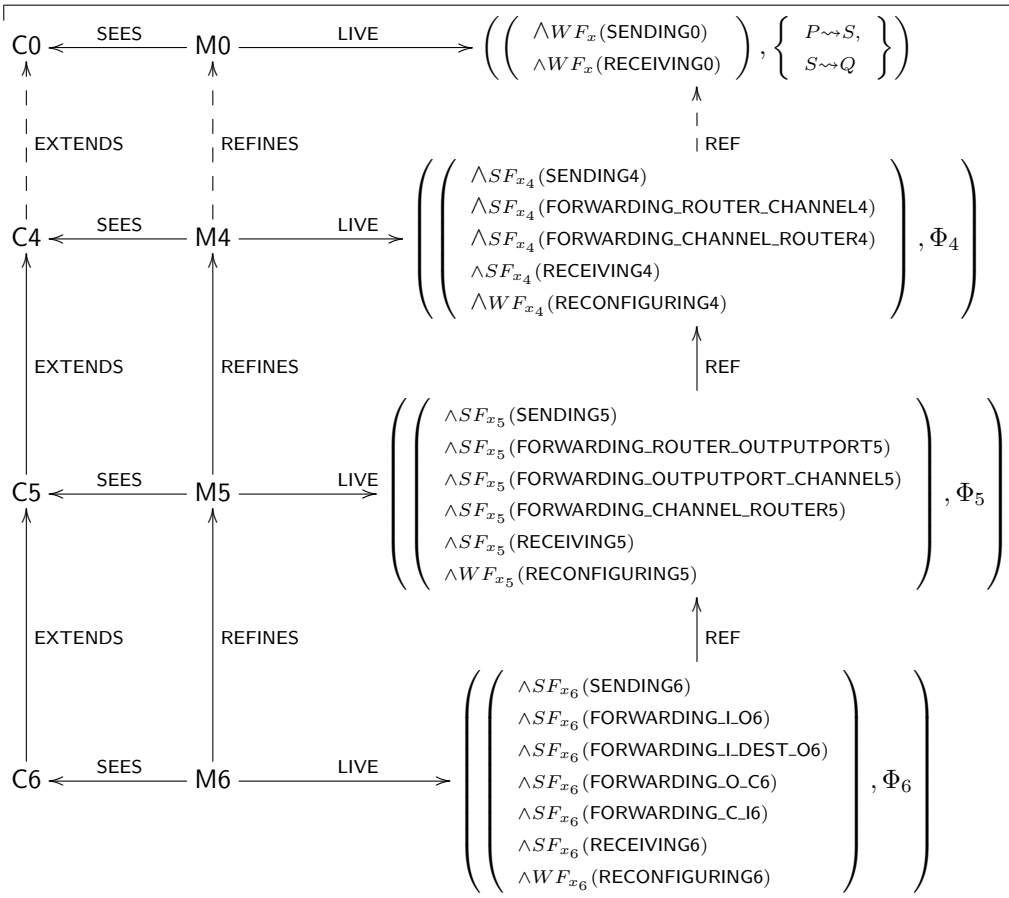


FIGURE 6.26 – Sixième raffinement du NoC

CONTEXT C7 EXTENDS C6

SETS

$CHANSTATE$

CONSTANTS

$ch\_free, ch\_busy, buffer\_size\_in, buffer\_size\_out$

AXIOMS

$axm1 : CHANSTATE = \{ch\_free, ch\_busy\}$

$axm2 : ch\_free \neq ch\_busy$

$axm3 : buffer\_size\_in \in \mathbb{N}_1$

$axm4 : buffer\_size\_out \in \mathbb{N}_1$

$$axm5 : \forall a, b, i, i0 \cdot \left(\begin{array}{l} \wedge a \in INPUTPORTS \leftrightarrow PACKETS \\ \wedge b \in PACKETS \\ \wedge i0 \in INPUTPORTS \\ \wedge i \in INPUTPORTS \\ \wedge i \mapsto b \notin a \\ \wedge finite(a) \\ \wedge i0 = i \end{array} \right) \Rightarrow \left(\begin{array}{c} card((a \cup \{i \mapsto b\})[\{i0\}]) \\ = \\ card(a[\{i0\}]) + 1 \end{array} \right)$$

$$axm6 : \forall a, b, i, i0 \cdot \left(\begin{array}{l} \wedge a \in INPUTPORTS \leftrightarrow PACKETS \\ \wedge b \in PACKETS \\ \wedge i0 \in INPUTPORTS \\ \wedge i \in INPUTPORTS \\ \wedge i \mapsto b \notin a \\ \wedge finite(a) \\ \wedge i0 \neq i \end{array} \right) \Rightarrow \left(\begin{array}{c} card((a \cup \{i \mapsto b\})[\{i0\}]) \\ = \\ card(a[\{i0\}]) \end{array} \right)$$

$$axm7 : \forall a, b, i, i0 \cdot \left(\begin{array}{l} \wedge a \in INPUTPORTS \leftrightarrow PACKETS \\ \wedge b \in PACKETS \\ \wedge i0 \in INPUTPORTS \\ \wedge i \in INPUTPORTS \\ \wedge i \mapsto b \in a \\ \wedge finite(a) \\ \wedge i0 = i \end{array} \right) \Rightarrow \left(\begin{array}{c} card((a \setminus \{i \mapsto b\})[\{i0\}]) \\ = \\ card(a[\{i0\}]) - 1 \end{array} \right)$$

$$axm8 : \forall a, b, i, i0 \cdot \left(\begin{array}{l} \wedge a \in INPUTPORTS \leftrightarrow PACKETS \\ \wedge b \in PACKETS \\ \wedge i0 \in INPUTPORTS \\ \wedge i \in INPUTPORTS \\ \wedge i \mapsto b \in a \\ \wedge finite(a) \\ \wedge i0 \neq i \end{array} \right) \Rightarrow \left(\begin{array}{c} card((a \setminus \{i \mapsto b\})[\{i0\}]) \\ = \\ card(a[\{i0\}]) \end{array} \right)$$

$$axm9 : \forall a, b, o, o0 \cdot \left(\begin{array}{l} \wedge a \in OUTPUTPORTS \leftrightarrow PACKETS \\ \wedge b \in PACKETS \\ \wedge o0 \in OUTPUTPORTS \\ \wedge o \in OUTPUTPORTS \\ \wedge o \mapsto b \notin a \\ \wedge finite(a) \\ \wedge o0 = o \end{array} \right) \Rightarrow \left(\begin{array}{c} card((a \cup \{o \mapsto b\})[\{o0\}]) \\ = \\ card(a[\{o0\}]) + 1 \end{array} \right)$$

$$axm10 : \forall a, b, o, o0 \cdot \left(\begin{array}{l} \wedge a \in OUTPUTPORTS \leftrightarrow PACKETS \\ \wedge b \in PACKETS \\ \wedge o0 \in OUTPUTPORTS \\ \wedge o \in OUTPUTPORTS \\ \wedge o \mapsto b \notin a \\ \wedge finite(a) \\ \wedge o0 \neq o \end{array} \right) \Rightarrow \left(\begin{array}{c} card((a \cup \{o \mapsto b\})[\{o0\}]) \\ = \\ card(a[\{o0\}]) \end{array} \right)$$

$$axm11 : \forall a, b, o, o0 \cdot \left(\begin{array}{l} \wedge a \in OUTPUTPORTS \leftrightarrow PACKETS \\ \wedge b \in PACKETS \\ \wedge o0 \in OUTPUTPORTS \\ \wedge o \in OUTPUTPORTS \\ \wedge o \mapsto b \in a \\ \wedge finite(a) \\ \wedge o0 = o \end{array} \right) \Rightarrow \left(\begin{array}{c} card((a \setminus \{o \mapsto b\})[\{o0\}]) \\ = \\ card(a[\{o0\}]) - 1 \end{array} \right)$$

$$axm12 : \forall a, b, o, o0 \cdot \left(\begin{array}{l} \wedge a \in OUTPUTPORTS \leftrightarrow PACKETS \\ \wedge b \in PACKETS \\ \wedge o0 \in OUTPUTPORTS \\ \wedge o \in OUTPUTPORTS \\ \wedge o \mapsto b \in a \\ \wedge finite(a) \\ \wedge o0 \neq o \end{array} \right) \Rightarrow \left(\begin{array}{c} card((a \setminus \{o \mapsto b\})[\{o0\}]) \\ = \\ card(a[\{o0\}]) \end{array} \right)$$

END

Nous notons x_7 l'ensemble des variables de cette machine M7.

Un invariant $I_7(x_7)$ contraint ces variables :

— La propriété $inv1$ permet le typage de la variable *chn_state* :

$$inv1 \ : \ chn\_state \in CHANNELS \rightarrow CHANSTATE$$

— Les propriétés de $inv2$ à $inv7$ expriment des propriétés de sûreté sur les tampons des ports d'entrée et de sorties :

$$\begin{array}{l} inv2 \ : \ \forall i \cdot i \in INPUTPORTS \Rightarrow finite(in\_cnt[\{i\}]) \\ inv3 \ : \ \forall i \cdot i \in INPUTPORTS \Rightarrow card(in\_cnt[\{i\}]) \leq buffer\_size\_in \\ inv4 \ : \ finite(in\_cnt) \\ inv5 \ : \ \forall o \cdot o \in OUTPUTPORTS \Rightarrow finite(out\_cnt[\{o\}]) \\ inv6 \ : \ \forall o \cdot o \in OUTPUTPORTS \Rightarrow card(out\_cnt[\{o\}]) \leq buffer\_size\_out \\ inv7 \ : \ finite(out\_cnt) \end{array}$$

  — $inv2$ exprime que le nombre de places dans le tampon d'un port d'entrée i est fini.
  — $inv3$ exprime que le nombre de places occupés dans le tampon d'un port d'entrée i est inférieur ou égal à $buffer\_size\_in$.
  — $inv4$ exprime que le nombre de (tampons de) ports d'entrée est fini.
  — $inv5$ exprime que le nombre de places dans le tampon d'un port de sortie o est fini.
  — $inv6$ exprime que le nombre de places occupés dans le tampon d'un port de sortie o est inférieur ou égal à $buffer\_size\_out$.
  — $inv7$ exprime que le nombre de (tampons de) ports de sortie est fini.

— La propriété $inv8$ exprime une propriété de sûreté concernant les canaux :

$$inv8 \ : \ \forall c \cdot c \in CHANNELS \wedge chn\_state(c) = ch\_busy \Rightarrow chn\_cnt[\{c\}] \neq \varnothing$$

  — $inv8$ exprime que si un canal c est dans l'état occupé (*ch_busy*), alors il contient des paquets p (son contenu est non-vide).

Nous retrouvons dans cette machine les raffinements des événements de la machine précédente, c'est-à-dire :

— Les événements SENDING7, RECEIVING7 pour les envois et réceptions de paquets.
— Les événements RECONFIGURING7, DISABLING7 pour la reconfiguration et la désactivation de routeurs.
— Les événements FORWARDING_I_O7, FORWARDING_I_DEST_O7, FORWARDING_O_C7, FORWARDING_C_I7 pour l'acheminement de paquets : respectivement **(1)** d'un port d'entrée (interne ou externe) d'un routeur à un port de sortie (externe), **(2)** d'un port d'entrée (externe) du destinataire d'un paquet vers un port de sortie (interne) menant au module IP du destinataire, **(3)** d'un port de sortie (externe) à un canal adjacent, **(4)** d'un canal à un port d'entrée (externe).

Une liste Φ_7 de propriétés de vivacité, détaillée en annexe, et utilisée pour guider le raffinement de M6 en M7 :

— L'événement SENDING7 modifie le contenu du port d'entrée interne d'un routeur source, en y rajoutant un paquet p à envoyer. Dans ce raffinement, nous ajoutons une contrainte supplémentaire : l'ajout du paquet p dans le port d'entrée interne du routeur source n'est possible que si le tampon de ce dernier dispose de places (le nombre de paquets dans le tampon est inférieur à $buffer\_size\_in$, $grd9$).

EVENT SENDING7 REFINES SENDING6 $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\oplus grd9 \ : \ card(in\_cnt[\{i\}]) < buffer\_size\_in$
  THEN
    ...
  END

— Pour les événements FORWARDING_I_O7 et FORWARDING_I_DEST_O7, qui rajoutent des paquets dans un port de sortie o d'un routeur, nous nous assurons avant de rajouter un paquet p à transmettre dans le port de sortie, que le tampon de celui-ci dispose de places libres (le nombre de places occupées est inférieur à $buffer\_size\_out$, $grd11$ du premier événement et $grd10$ du second événement).

EVENT FORWARDING_I_O7 REFINES FORWARDING_I_O6 $\widehat{=}$
ANY
...
WHERE
...
$\oplus grd11$: $card(out\_cnt[\{o\}]) < buffer\_size\_out$
THEN
...
END

EVENT FORWARDING_I_DEST_O7 REFINES FORWARDING_I_DEST_O6 $\widehat{=}$
ANY
...
WHERE
...
$\oplus grd10$: $card(out\_cnt[\{o\}]) < buffer\_size\_out$
THEN
...
END

— Les événements modifiant les contenus des canaux les font aussi changer d'état :
— FORWARDING_O_C7 modélise l'occupation d'un canal c par un paquet p : l'état du canal c passe de libre (ch_free, $grd9$) à occupé (ch_busy, $act3$).

EVENT FORWARDING_O_C7 REFINES FORWARDING_O_C6 $\widehat{=}$
...
WHERE
...
$\oplus grd9$: $chn\_state(c) = ch\_free$
THEN
...
$\oplus act3$: $chn\_state(c) := ch\_busy$
END

— FORWARDING_C_I7 modélise la libération d'un canal c : un paquet p occupant le canal c le quitte et l'état du canal c passe d'occupé (ch_busy, $grd10$) à libre (ch_free, $act3$). Nous remarquons que cet événement rajoute aussi des paquets à un port d'entrée i : nous nous assurons au préalable que le tampon du port d'entrée i dispose de suffisamment de places libres (le nombre de places occupées est inférieur à $buffer\_size\_in$, $grd9$).

EVENT FORWARDING_C_I7 REFINES FORWARDING_C_I6 $\widehat{=}$
...
WHERE
...
$\oplus grd9$: $card(in\_cnt[\{i\}]) < buffer\_size\_in$
$\oplus grd10$: $chn\_state(c) = ch\_busy$
THEN
...
$\oplus act3$: $chn\_state(c) := ch\_free$
END

Nous pouvons résumer ce septième raffinement M7 par le diagramme 6.27 suivant :

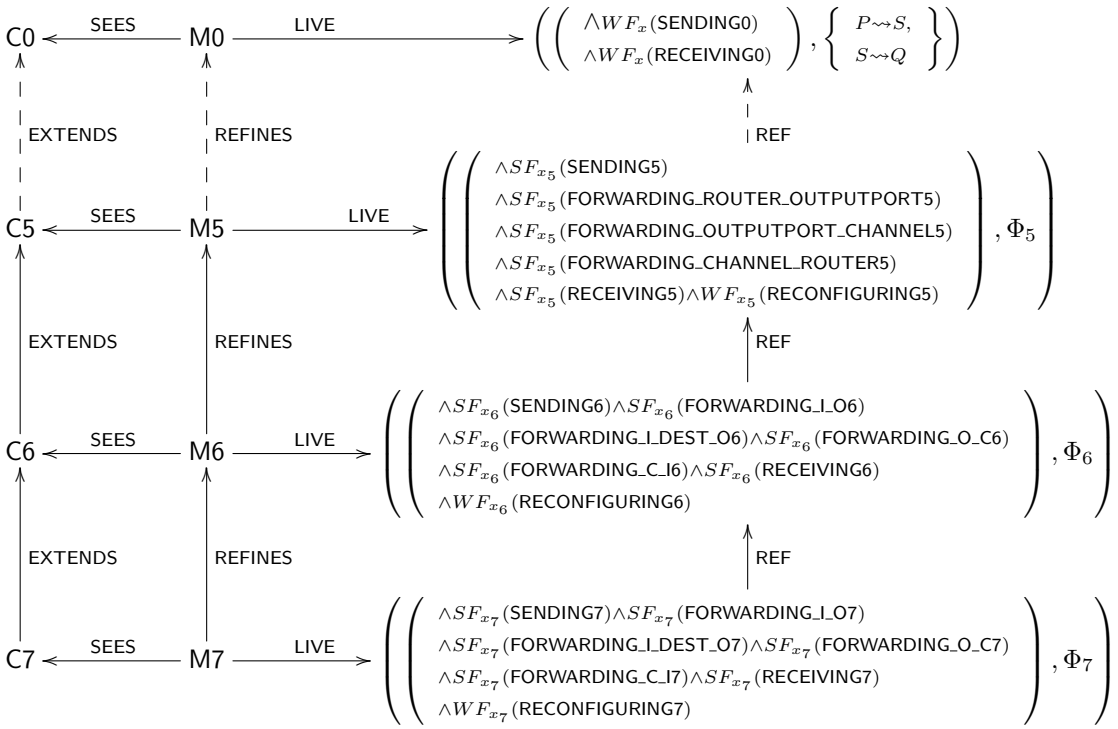


FIGURE 6.27 – Septième raffinement du NoC

Ce diagramme nous montre aussi que les propriétés de vivacité Φ_6 du niveau de raffinement précédent sont satisfaites par M7 (voir l'annexe pour la preuve).

Nous avons présenté à l'aide du modèle abstrait et de ses sept premiers raffinements, un développement générique d'un « Réseau-sur-puce » (Network-on-Chip, NoC). Nous y avons détaillé petit à petit l'architecture du réseau (existence de canaux entre les routeurs du réseau), ainsi que la structure internes des routeurs : ceux-ci disposent de ports d'entrée et de sortie, externes, pour la communication avec les routeurs voisins et internes, pour la communication avec les composants « Intellectual Properties » (IP). Nous avons défini aussi des contraintes concernant ces composants, notamment les ports d'entrée et de sorties, pour lesquels nous avons défini une limite du nombre de paquets qu'ils peuvent contenir et les canaux, pour lesquels nous avons identifié deux états, occupé (un paquet transite sur un canal) et libre (aucun paquet ne transite sur un canal).

Nous avons identifié ici un patron de conception répondant au problème de modélisation des « Réseaux-sur-puce » (Network-on-Chip, NoC) et que nous pouvons instancier pour plusieurs topologies possibles de réseaux : il suffit de raffiner le dernier modèle M7 et d'introduire la topologie désirée, par exemple, bidimensionnelle, tridimensionnelle, etc, et de modifier par raffinement les événements de type FORWARDING, pour prendre en compte la topologie du réseau, ainsi que les algorithmes de routage utilisés (e.g XY, etc). Il est aussi possible de raffiner les événements de modification du réseau RECONFIGURING7 et DISABLING7, pour prendre en charge des cas spécifiques (e.g établir des priorités sur les réactivations de routeurs : réactiver ceux qui contiennent des paquets avant ceux qui sont vides, etc).

Nous illustrons, dans la prochaine section, l'instanciation de ce patron en modélisant un NoC bidimensionnel.

6.3.3.9 NoC 2-D et au-delà

Cette section présente le raffinement instanciant un « Réseau-sur-puce » (Network-on-Chip, NoC) bidimensionnel. Pour modéliser ce NoC bidimensionnel, nous commençons tout d'abord par étendre à l'aide d'un contexte C8, le contexte C7 vu précédemment. Nous utilisons ce contexte C8 pour définir des

éléments de la topologie du réseau. Nous nous focaliserons ici, plus précisément, sur un réseau maillé de forme carrée.

Nous commençons par définir une constante $nsize$ naturelle non-nulle, supérieure à 1 ($axm2$). Celle-ci nous permet de définir le nombre de routeurs soit $nsize^2$ ($axm2$). Elle nous permet aussi de définir un système de coordonnées $coord$ ($axm3$) bidimensionnel, allant des coordonnées $(0,0)$ à $(nsize-1, nsize-1)$. Une fonction bijective pos nous permet d'associer à chaque routeur des coordonnées en X (abscisse) et en Y (ordonnée) ($axm4$). Deux fonctions $coordX$ ($axm5$) et $coordY$ ($axm6$) permettent de récupérer respectivement pour chaque routeur, sa coordonnée en abscisse ($axm11$) et sa coordonnée en ordonnée ($axm12$). Les constantes Ln, Rn, Dn, Un, permettent de récupérer respectivement, pour chaque routeur soit le routeur voisin à gauche (coordonnées (X-1,Y)) ($axm19$), soit le routeur voisin à droite (coordonnées (X+1,Y)) ($axm22$), soit le routeur voisin au-dessous (coordonnées (X,Y-1)) ($axm27$), soit le routeur voisin au-dessus (coordonnées (X,Y+1)) ($axm30$), lorsque ce routeur voisin existe ($axm18$, $axm21$, $axm26$, $axm29$). Les axiomes $inv15$ et $inv16$ définissent les conditions qui nous permettent de considérer deux routeurs $r1$ et $r2$ comme étant voisins : les routeurs $r1$ et $r2$ sont voisins s'il partagent la même coordonnée en abscisse et que la distance entre leurs coordonnées en ordonnée est de 1, ou s'ils partagent la même coordonnée en ordonnée et que la distance entre leurs coordonnées en abscisse est de 1. Les autres axiomes permettent d'exprimer des propriétés telles que : si un routeur r ne se trouve pas sur le bord gauche du carré, il possède un voisin à gauche ($axm17$) ; si un routeur r ne se trouve pas sur le bord droit du carré, il possède un voisin à droite ($axm20$) ; si un routeur r ne se trouve pas sur le bord haut du carré, il possède un voisin au-dessus ($axm28$) ; si un routeur r ne se trouve pas sur le bord bas du carré, il possède un voisin au-dessous ($axm25$). Les autres axiomes ($axm13$, $axm14$, $axm23$, $axm24$, $axm31$, $axm32$) permettent d'exprimer des propriétés similaires pour un routeur x du réseau, mais cette fois-ci, par rapport aux coordonnées d'un autre routeur d. Ce contexte C8 est utilisé par une machine M8 dans laquelle nous modélisons le comportement dynamique du NoC bidimensionnel.

Contrairement aux autres raffinements vus précédemment, nous n'introduisons pas de nouvelles variables dans ce raffinement. L'ensemble des variables de cette machine M8 est donc x_7.

Un invariant noté $I_8(x_7)$ caractérise ce raffinement et est défini comme suit :

— Une propriété de sûreté $inv1$ caractérise la bidimensionnalité du réseau : deux routeurs $n1$ et $n2$ sont voisins s'il partagent la même coordonnée en abscisse et que la distance entre leurs coordonnées en ordonnée est de 1, ou s'ils partagent la même coordonnée en ordonnée et que la distance entre leurs coordonnées en abscisse est de 1.

$$inv1 \ : \ \forall n1, n2 \cdot \left(\begin{array}{l} \wedge n1 \in ROUTERS \\ \wedge n2 \in ROUTERS \\ \wedge n1 \mapsto n2 \in network \end{array} \right) \Rightarrow \left(\begin{array}{l} \vee \left(\begin{array}{l} \wedge coordX(n1) = coordX(n2) \\ \wedge (coordY(n1) - coordY(n2)) \in \{-1, 1\} \end{array} \right) \\ \vee \left(\begin{array}{l} \wedge coordY(n1) = coordY(n2) \\ \wedge (coordX(n1) - coordX(n2)) \in \{-1, 1\} \end{array} \right) \end{array} \right)$$

Nous nous contentons dans cette machine M8 de spécialiser les événements d'acheminement de paquets (FORWARDING*) pour que ces derniers prennent en compte l'algorithme de routage XY (acheminement des paquets selon l'axe des abscisses, puis acheminement selon l'axe des ordonnées) et s'adaptent aux possibles difficultés rencontrées (désactivation d'un routeur, etc). Les événements de cette machine M8 sont donc les suivants :

— SENDING8, RECEIVING8 pour les envois et réceptions de paquets.

— RECONFIGURING8, DISABLING8 pour la reconfiguration et la désactivation de routeurs.

— FORWARDING_I_O_N8, FORWARDING_I_O_S8, FORWARDING_I_O_E8, FORWARDING_I_O_W8, FORWARDING_I_DEST_O8, FORWARDING_O_C8, FORWARDING_C_I8 pour l'acheminement de paquets : respectivement **(1)** d'un port d'entrée (interne ou externe) d'un routeur à un port de sortie (externe) dans la direction Nord, **(2)** d'un port d'entrée (interne ou externe) d'un routeur à un port de sortie (externe) dans la direction Sud, **(3)** d'un port d'entrée (interne ou externe) d'un routeur à un port de sortie (externe) dans la direction Est, **(4)** d'un port d'entrée (interne ou externe) d'un routeur à un port de sortie (externe) dans la direction Ouest, **(5)** d'un port d'entrée (externe) du destinataire d'un paquet vers un port de sortie (interne) menant au module IP du destinataire, **(6)** d'un port de sortie (externe) à un canal adjacent, **(7)** d'un canal à un port d'entrée (externe).

Nous présentons graphiquement la liste Φ_8 les propriétés de vivacité satisfaites par M8, à l'aide des diagrammes 6.28, 6.29, 6.30 et 6.31. Nous remarquons que nous spécialisons surtout dans cette machine

CONTEXT C8 EXTENDS C7

CONSTANTS

$nsize, coord, pos, coordX, coordY, Ln, Rn, Dn, Un$

AXIOMS

$axm1 : nsize \in \mathbb{N}_1 \land nsize > 1$

$axm2 : card(ROUTERS) = nsize * nsize$

$axm3 : coord = (0 \mathinner{..} nsize - 1) \times (0 \mathinner{..} nsize - 1)$

$axm4 : pos \in ROUTERS \rightarrowtail\!\!\!\!\!\!\rightarrow coord$

$axm5 : coordX \in ROUTERS \rightarrow \mathbb{N}$

$axm6 : coordY \in ROUTERS \rightarrow \mathbb{N}$

$axm7 : Ln \in ROUTERS \nrightarrow ROUTERS$

$axm8 : Rn \in ROUTERS \nrightarrow ROUTERS$

$axm9 : Dn \in ROUTERS \nrightarrow ROUTERS$

$axm10 : Un \in ROUTERS \nrightarrow ROUTERS$

$axm11 : \forall r \cdot r \in ROUTERS \Rightarrow coordX(r) = prj1(pos(r))$

$axm12 : \forall r \cdot r \in ROUTERS \Rightarrow coordY(r) = prj2(pos(r))$

$$axm13 : \forall x \cdot \left(\begin{array}{l} \land x \in ROUTERS \\ \land (\exists d \cdot \left(\begin{array}{l} \land d \in ROUTERS \\ \land coordX(x) < coordX(d) \end{array} \right)) \end{array} \right) \Rightarrow \left(\begin{array}{c} coordX(x) + 1 \mapsto coordY(x)) \\ \in \\ ran(pos) \end{array} \right)$$

$$axm14 : \forall x \cdot \left(\begin{array}{l} \land x \in ROUTERS \\ \land (\exists d \cdot \left(\begin{array}{l} \land d \in ROUTERS \\ \land coordX(x) > coordX(d) \end{array} \right)) \end{array} \right) \Rightarrow \left(\begin{array}{c} (coordX(x) - 1 \mapsto coordY(x)) \\ \in \\ ran(pos) \end{array} \right)$$

$$axm15 : \forall r1, r2 \cdot \left(\begin{array}{l} \land r1 \in ROUTERS \\ \land r2 \in ROUTERS \\ \land \left(\begin{array}{l} \lor \left(\begin{array}{l} \land coordX(r1) = coordX(r2) \\ \land (coordY(r1) - coordY(r2)) \in \{-1, 1\} \end{array} \right) \\ \lor \left(\begin{array}{l} \land (coordX(r1) - coordX(r2)) \in \{-1, 1\} \\ \land coordY(r1) = coordY(r2) \end{array} \right) \end{array} \right) \end{array} \right) \Rightarrow r1 \mapsto r2 \in ntwrk$$

$$axm16 : \forall r1, r2 \cdot \left(\begin{array}{l} \land r1 \in ROUTERS \\ \land r2 \in ROUTERS \\ \land r1 \mapsto r2 \in ntwrk \end{array} \right) \Rightarrow \left(\begin{array}{l} \lor \left(\begin{array}{l} \land coordX(r1) = coordX(r2) \\ \land (coordY(r1) - coordY(r2)) \in \{-1, 1\} \end{array} \right) \\ \lor \left(\begin{array}{l} \land (coordX(r1) - coordX(r2)) \in \{-1, 1\} \\ \land coordY(r1) = coordY(r2) \end{array} \right) \end{array} \right)$$

$axm17 : \forall n \cdot n \in ROUTERS \land coordX(n) > 0 \Rightarrow (coordX(n) - 1 \mapsto coordY(n)) \in ran(pos)$

$axm18 : dom(Ln) = \{r | r \in ROUTERS \land coordX(r) > 0\}$

$axm19 : \forall n \cdot n \in dom(Ln) \Rightarrow Ln(n) = pos^{-1}(coordX(n) - 1 \mapsto coordY(n))$

$axm20 : \forall n \cdot n \in ROUTERS \land coordX(n) < nsize - 1 \Rightarrow (coordX(n) + 1 \mapsto coordY(n)) \in ran(pos)$

$axm21 : dom(Rn) = \{r | r \in ROUTERS \land coordX(r) < nsize - 1\}$

$axm22 : \forall n \cdot n \in dom(Rn) \Rightarrow Rn(n) = pos^{-1}(coordX(n) + 1 \mapsto coordY(n))$

$axm23 : \forall x \cdot x \in ROUTERS \land (\exists d \cdot d \in ROUTERS \land coordX(x) < coordX(d)) \Rightarrow coordX(x) < nsize - 1$

$axm24 : \forall x \cdot x \in ROUTERS \land (\exists d \cdot d \in ROUTERS \land coordX(x) > coordX(d)) \Rightarrow coordX(x) > 0$

$axm25 : \forall n \cdot n \in ROUTERS \land coordY(n) > 0 \Rightarrow (coordX(n) \mapsto coordY(n) - 1) \in ran(pos)$

$axm26 : dom(Dn) = \{r | r \in ROUTERS \land coordY(r) > 0\}$

$axm27 : \forall n \cdot n \in dom(Dn) \Rightarrow Dn(n) = pos^{-1}(coordX(n) \mapsto coordY(n) - 1)$

$axm28 : \forall n \cdot n \in ROUTERS \land coordY(n) < nsize - 1 \Rightarrow (coordX(n) \mapsto coordY(n) + 1) \in ran(pos)$

$axm29 : dom(Un) = \{r | r \in ROUTERS \land coordY(r) < nsize - 1\}$

$axm30 : \forall n \cdot n \in dom(Un) \Rightarrow Un(n) = pos^{-1}(coordX(n) \mapsto coordY(n) + 1)$

$axm31 : \forall x \cdot x \in ROUTERS \land (\exists d \cdot d \in ROUTERS \land coordY(x) < coordY(d)) \Rightarrow coordY(x) < nsize - 1$

$axm32 : \forall x \cdot x \in ROUTERS \land (\exists d \cdot d \in ROUTERS \land coordY(x) > coordY(d)) \Rightarrow coordY(x) > 0$

END

les événements de type FORWARDING_I_O, c'est-à-dire les événements d'acheminement de paquets d'un port d'entrée à un port de sortie d'un nœud. Nous notons :

— Nous posons x comme étant la position courante d'un paquet p acheminé dans le réseau : il s'agit du routeur, présent et non- désactivé au sein du réseau ($x \in dom(network)$) dans lequel le paquet p se trouve. Soit d le routeur destinataire du paquet p ($d = dst(p)$). Nous posons $dist(x, d, p)$ comme étant le nombre de nœuds entre la position courante x du paquet p et sa destination d, sur un chemin donné, avec $dist(x, d, p) \in \mathbb{N}$, et nous définissons $D(x, d, p, n) \mathrel{\widehat{=}} (dist(x, d, p) = n)$, où $n \in \mathbb{N}$. Nous notons que cette propriété implique que p n'est pas encore reçu par son destinataire d.

— $\sigma_7 \mathrel{\widehat{=}} \left(\begin{array}{l} \wedge x \in ROUTERS \\ \wedge i \in INPUTPORTS \\ \wedge o \in OUTPUTPORTS \\ \wedge network[\{x\}] \neq \varnothing \wedge x = prj_2(in\_p^{-1}(i)) \\ \wedge p \in in\_cnt[\{i\}] \\ \wedge x = prj_1(out\_p^{-1}(o)) \\ \wedge x \neq dst(p) \\ \wedge o \neq out\_p(x \mapsto x) \\ \wedge p \notin out\_cnt[\{o\}] \\ \wedge card(out\_cnt[\{o\}]) < buffer\_size\_out \\ \wedge prj_2(out\_p^{-1}(o)) \in network[\{x\}] \end{array} \right)$

Cette propriété exprime la condition de passage d'un paquet p d'un port d'entrée i d'un routeur x, à un port de sortie (externe) o de ce même routeur : le nœud x est activé, i est effectivement un port entrant dans x, o un port sortant de x avec suffisamment de places libres, p est dans i, mais pas dans o et le routeur auquel o conduit n'est pas désactivé.

— $A_8 \mathrel{\widehat{=}} \left(\begin{array}{l} \wedge coordX(prj2(in\_p^{-1}(i))) < coordX(dst(p)) \\ \wedge prj2(out\_p^{-1}(o)) = Rn(prj2(in\_p^{-1}(i))) \end{array} \right)$

Cette propriété exprime que la position de la destination se trouve à l'est de la courante du paquet p ; et que le voisin choisi pour continuer l'acheminement du paquet p est celui de droite (à l'est).

— $B_8 \mathrel{\widehat{=}} \left(\begin{array}{l} \wedge coordX(prj2(in\_p^{-1}(i))) > coordX(dst(p)) \\ \wedge prj2(out\_p^{-1}(o)) = Ln(prj2(in\_p^{-1}(i))) \end{array} \right)$

Cette propriété exprime que la position de la destination se trouve à l'ouest de la courante du paquet p ; et que le voisin choisi pour continuer l'acheminement du paquet p est celui de gauche (à l'ouest).

— $C_8 \mathrel{\widehat{=}} \left(\begin{array}{l} \wedge coordY(prj2(in\_p^{-1}(i))) < coordY(dst(p)) \\ \quad \wedge \left(\begin{array}{l} \vee coordX(prj2(in\_p^{-1}(i))) = coordX(dst(p)) \\ \quad \vee \left(\begin{array}{l} \wedge (coordX(prj2(in\_p^{-1}(i))) > coordX(dst(p))) \\ \wedge (Ln(prj2(in\_p^{-1}(i))) \notin network[\{prj2(in\_p^{-1}(i))\}]) \end{array} \right) \\ \quad \vee \left(\begin{array}{l} \wedge (coordX(prj2(in\_p^{-1}(i))) < coordX(dst(p))) \\ \wedge (Rn(prj2(in\_p^{-1}(i))) \notin network[\{prj2(in\_p^{-1}(i))\}]) \end{array} \right) \end{array} \right) \\ \wedge prj2(out\_p^{-1}(o)) = Un(prj2(in\_p^{-1}(i))) \end{array} \right)$

Cette propriété exprime que la position de la destination se trouve au nord de la courante du paquet p ; et soit que la position courante du paquet p est sur la même abscisse que la destination, soit que le paquet p ne peut se déplacer ni à l'est, ni à l'ouest ; et que le voisin choisi pour continuer l'acheminement du paquet p est celui du haut (au nord).

— $D_8 \mathrel{\widehat{=}} \left(\begin{array}{l} \wedge coordY(prj2(in\_p^{-1}(i))) < coordY(dst(p)) \\ \quad \wedge \left(\begin{array}{l} \vee coordX(prj2(in\_p^{-1}(i))) = coordX(dst(p)) \\ \quad \vee \left(\begin{array}{l} \wedge (coordX(prj2(in\_p^{-1}(i))) > coordX(dst(p))) \\ \wedge (Ln(prj2(in\_p^{-1}(i))) \notin network[\{prj2(in\_p^{-1}(i))\}]) \end{array} \right) \\ \quad \vee \left(\begin{array}{l} \wedge (coordX(prj2(in\_p^{-1}(i))) < coordX(dst(p))) \\ \wedge (Rn(prj2(in\_p^{-1}(i))) \notin network[\{prj2(in\_p^{-1}(i))\}]) \end{array} \right) \end{array} \right) \\ \wedge prj2(out\_p^{-1}(o)) = Un(prj2(in\_p^{-1}(i))) \end{array} \right)$

Cette propriété exprime que la position de la destination se trouve au sud de la courante du paquet p ; et soit que la position courante du paquet p est sur la même abscisse que la destination, soit que le paquet p ne peut se déplacer ni à l'est, ni à l'ouest ; et que le voisin choisi pour continuer

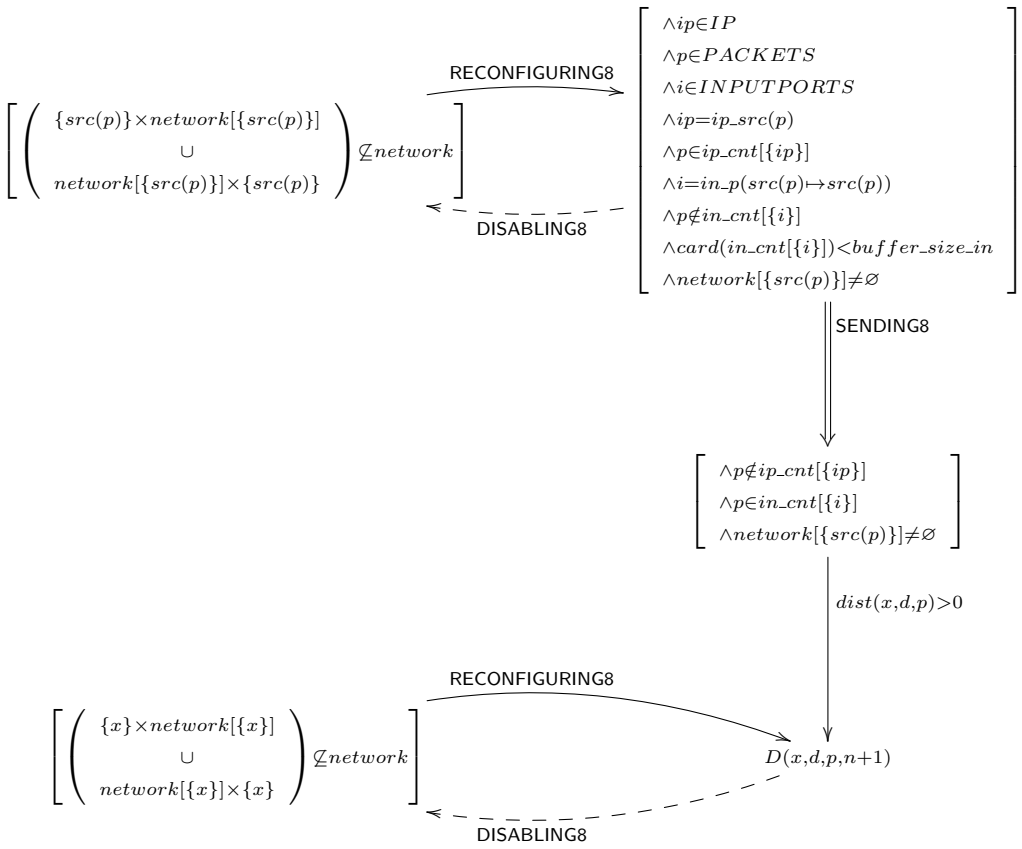


FIGURE 6.28 – Diagramme d'assertions pour le huitième raffinement (Partie 1)

l'acheminement du paquet p est celui du bas (au sud).

Nous nous servons des propriétés de vivacité de Φ_8 pour raffiner M7 en M8. Nous raffinons ensuite l'événement FORWARDING_I_O7 en quatre versions, qui nous permettent de déplacer un paquet p par rapport à la position de la destination $dst(p)$ de ce dernier :

— FORWARDING_I_O_E8 (déplacement d'un paquet à l'Est/à droite) est observé lorsque la coordonnée en abscisse de la position courante du paquet est inférieure à la coordonnée en abscisse de sa destination ($grd12$). Le port de sortie (de la position courante du paquet) menant au voisin droit est alors choisi ($grd13$) pour le déplacement du paquet.

EVENT FORWARDING_I_O_E8 REFINES FORWARDING_I_O7 $\widehat{=}$
...
WHERE
...
$\oplus grd12$: $coordX(prj2(in\_p^{-1}(i))) < coordX(dst(p))$
$\oplus grd13$: $prj2(out\_p^{-1}(o)) = Rn(prj2(in\_p^{-1}(i)))$
THEN
...
END

— FORWARDING_I_O_W8 (déplacement d'un paquet à l'Ouest/à gauche) est observé lorsque la coordonnée en abscisse de la position courante du paquet est supérieure à la coordonnée en abscisse de sa destination ($grd12$). Le port de sortie (de la position courante du paquet) menant au voisin gauche est alors choisi ($grd13$) pour le déplacement du paquet.

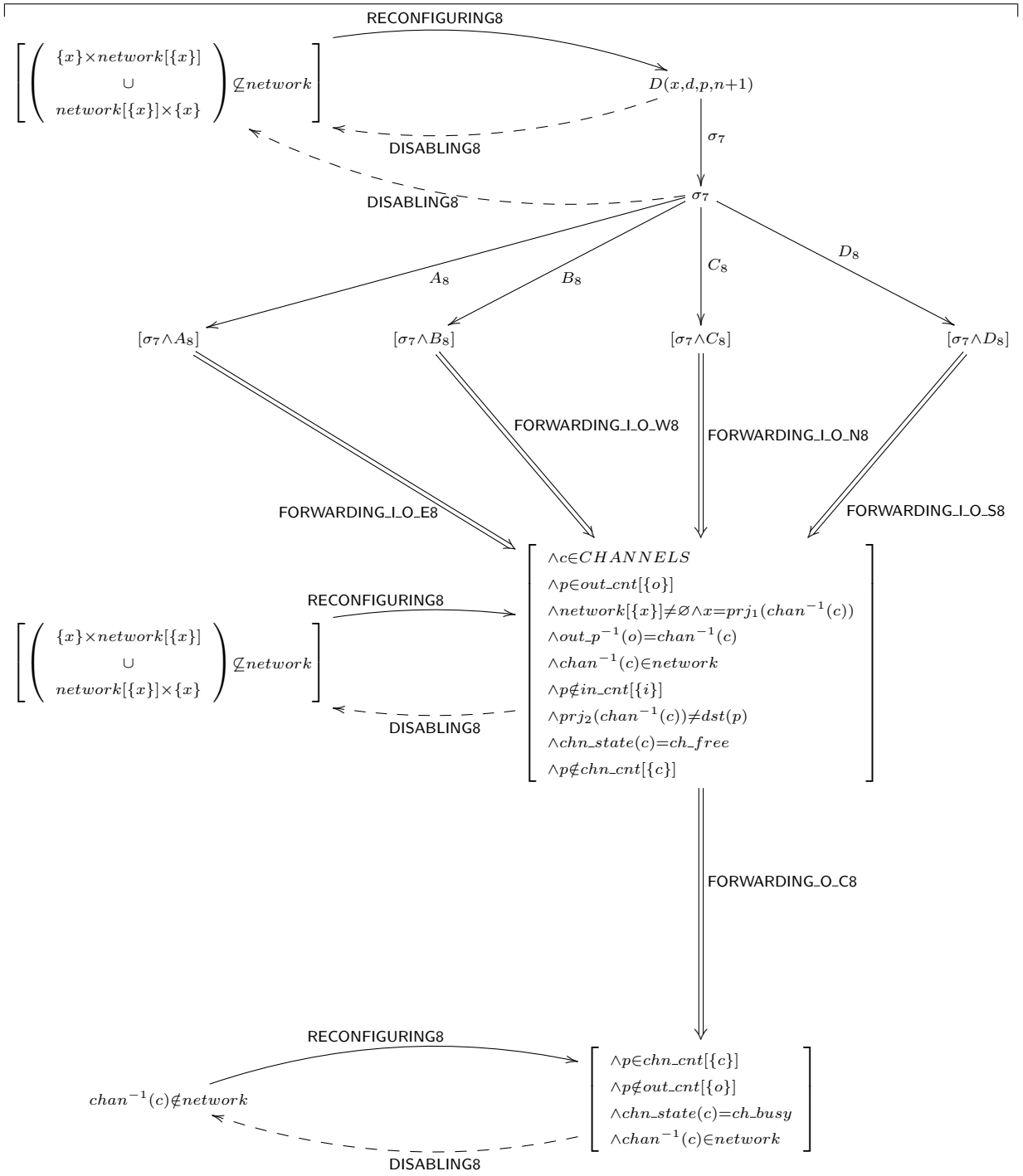


FIGURE 6.29 – Diagramme d'assertions pour le huitième raffinement (Partie 2)

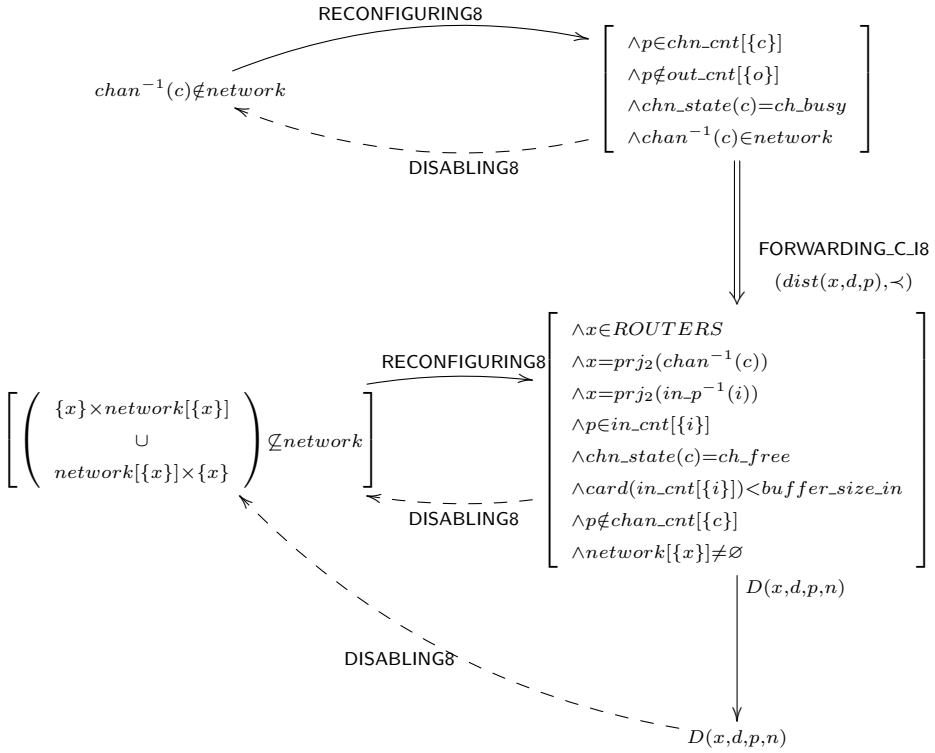


FIGURE 6.30 – Diagramme d'assertions pour le huitième raffinement (Partie 3)

EVENT FORWARDING_I_O_W8 REFINES FORWARDING_I_O7 $\widehat{=}$
...
WHERE
...
$\oplus grd12$: $coordX(prj2(in\_p^{-1}(i))) > coordX(dst(p))$
$\oplus grd13$: $prj2(out\_p^{-1}(o)) = Ln(prj2(in\_p^{-1}(i)))$
THEN
...
END

— FORWARDING_I_O_N8 (déplacement d'un paquet au Nord/en haut) est observé lorsque la coordonnées en ordonnée de la position courante du paquet est inférieure à la coordonné en ordonnée de sa destination ($grd12$) et lorsque l'une des conditions suivantes ($grd13$) est observée : les coordonnées en abscisse de la position courante et de la destination sont les mêmes, ou les coordonnées en abscisse sont différentes, mais la désactivation des voisins de droite ou de gauche ne permet pas de se déplacer sur l'axe des abscisses. Dans ce cas, le port de sortie (de la position courante du paquet) menant au voisin au-dessus est alors choisi ($grd14$) pour le déplacement du paquet.

EVENT FORWARDING_I_O_N8 REFINES FORWARDING_I_O7 $\widehat{=}$
...
WHERE
...
$\oplus grd12$: $coordY(prj2(in\_p^{-1}(i))) < coordY(dst(p))$
$\oplus grd13$: $\left(\begin{array}{l} \vee coordX(prj2(in\_p^{-1}(i))) = coordX(dst(p)) \\ \vee \left(\begin{array}{l} \wedge (coordX(prj2(in\_p^{-1}(i))) > coordX(dst(p))) \\ \wedge (Ln(prj2(in\_p^{-1}(i))) \notin network[\{prj2(in\_p^{-1}(i))\}]) \end{array} \right) \\ \vee \left(\begin{array}{l} \wedge (coordX(prj2(in\_p^{-1}(i))) < coordX(dst(p))) \\ \wedge (Rn(prj2(in\_p^{-1}(i))) \notin network[\{prj2(in\_p^{-1}(i))\}]) \end{array} \right) \end{array} \right)$
$\oplus grd14$: $prj2(out\_p^{-1}(o)) = Un(prj2(in\_p^{-1}(i)))$
THEN
...
END

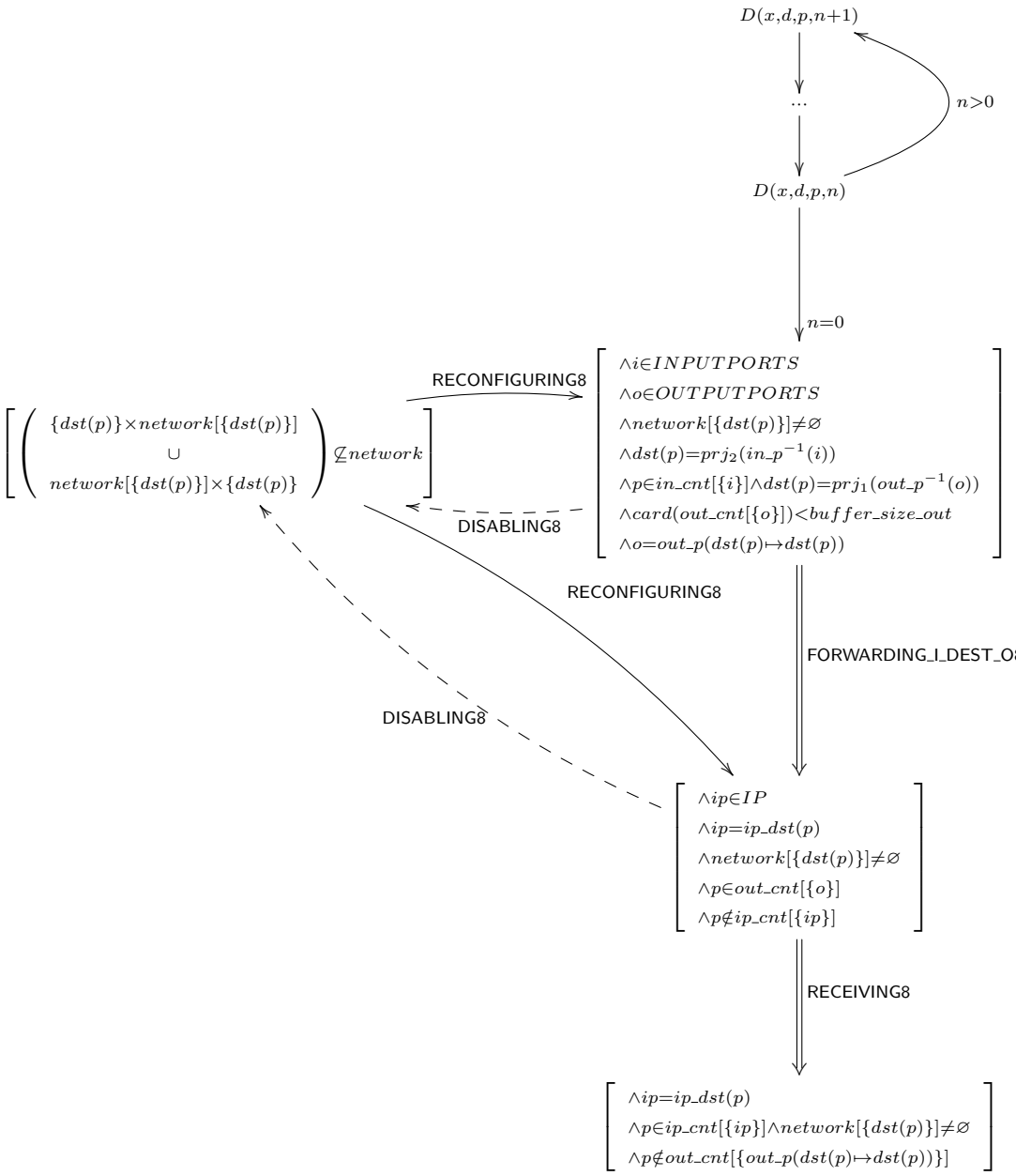


FIGURE 6.31 – Diagramme d'assertions pour le huitième raffinement (Partie 4)

— FORWARDING_I_O_S8 (déplacement d'un paquet au Sud/en bas) est observé lorsque la coordonnées en ordonnée de la position courante du paquet est supérieure à la coordonné en ordonnée de sa destination ($grd12$) et lorsque l'une des conditions suivantes ($grd13$) est observée : les coordonnées en abscisse de la position courante et de la destination sont les mêmes, ou les coordonnées en abscisse sont différentes, mais la désactivation des voisins de droite ou de gauche ne permet pas de se déplacer sur l'axe des abscisses. Dans ce cas, le port de sortie (de la position courante du paquet) menant au voisin au-dessous est alors choisi ($grd14$) pour le déplacement du paquet.

$$
\begin{array}{l}
\text{EVENT FORWARDING\_I\_O\_S8 REFINES FORWARDING\_I\_O7} \ \widehat{=} \\
\quad \ldots \\
\quad \text{WHERE} \\
\quad\quad \ldots \\
\quad\quad \oplus grd12 \ : \ coordY(prj2(in\_p^{-1}(i))) > coordY(dst(p)) \\
\quad\quad \oplus grd13 \ : \left(\begin{array}{l} \vee coordX(prj2(in\_p^{-1}(i))) = coordX(dst(p)) \\ \vee \left(\begin{array}{l} \wedge (coordX(prj2(in\_p^{-1}(i))) > coordX(dst(p))) \\ \wedge (Ln(prj2(in\_p^{-1}(i))) \notin network[\{prj2(in\_p^{-1}(i))\}]) \end{array} \right) \\ \vee \left(\begin{array}{l} \wedge (coordX(prj2(in\_p^{-1}(i))) < coordX(dst(p))) \\ \wedge (Rn(prj2(in\_p^{-1}(i))) \notin network[\{prj2(in\_p^{-1}(i))\}]) \end{array} \right) \end{array} \right) \\
\quad\quad \oplus grd14 \ : \ prj2(out\_p^{-1}(o)) = Dn(prj2(in\_p^{-1}(i))) \\
\quad \text{THEN} \\
\quad\quad \ldots \\
\quad \text{END}
\end{array}
$$

Ces quatres événements FORWARDING_I_O_W8, FORWARDING_I_O_E8, FORWARDING_I_O_N8 et FORWARDING_I_O_S8 modélisent la variante de l'algorithme XY [4] (voir algorithme 4), présentée dans les sections précédentes et que nous reprenons ici :

Algorithm 5 Algorithme XY adaptatif

D : routeur destination. Coordonnées (Dx : abscisse, Dy : ordonnée).
C : routeur courant. Coordonnées (Cx : abscisse, Cy : ordonnée).

```
1: if Cx > Dx then
2:    return W ;
3: else if Cx < Dx then
4:    return E ;
5: else if (Cx = Dx) ∨ ((Cx > Dx) ∧ W est bloquée) ∨ ((Cx < Dx) ∧ E est bloquée) then
6:    if Cy < Dy then
7:       return N ;
8:    else if Cy > Dy then
9:       return S ;
10:   end if
11: end if
```

— les lignes 1 et 2 de l'algorithme ci-dessus correspondent à l'acheminement d'un paquet p vers la direction Ouest (W) : la condition (IF) en ligne 1 est exprimée par la garde $grd12$ de l'événement FORWARDING_I_O_W8 ; et l'instruction « return W ; » en ligne 2 est donnée conjointement par la garde $grd13$ du même événement et l'action $act1 \ : \ out\_cnt := out\_cnt \cup \{o \mapsto p\}$, il s'agit du choix du port de sortie o pointant vers la direction Ouest (W) pour l'acheminement du paquet p.
— les lignes 3 et 4 de l'algorithme ci-dessus correspondent à l'acheminement d'un paquet p vers la direction Est (E) : la condition (ELSE IF) en ligne 3 est exprimée par la garde $grd12$ de l'événement FORWARDING_I_O_E8 ; et l'instruction « return E ; » en ligne 4 est donnée conjointement par la garde $grd13$ du même événement et l'action $act1 \ : \ out\_cnt := out\_cnt \cup \{o \mapsto p\}$, il s'agit du choix du port de sortie o pointant vers la direction Est (E) pour l'acheminement du paquet p.
— la condition (ELSE IF) en ligne 5 est donnée par la garde $grd13$ commune aux événements FORWARDING_I_O_N8 et FORWARDING_I_O_S8.
— les lignes 6 et 7 de l'algorithme ci-dessus correspondent à l'acheminement d'un paquet p vers la direction Nord (N) : la condition (IF) en ligne 6 est exprimée par la garde $grd12$ de l'événement

FORWARDING_I_O_N8 ; et l'instruction « return N ; » en ligne 2 est donnée conjointement par la garde $grd14$ du même événement et l'action $act1 \ : \ out\_cnt := out\_cnt \cup \{o \mapsto p\}$, il s'agit du choix du port de sortie o pointant vers la direction Nord (N) pour l'acheminement du paquet p.

— les lignes 8 et 9 de l'algorithme ci-dessus correspondent à l'acheminement d'un paquet p vers la direction Sud (S) : la condition (ELSE IF) en ligne 8 est exprimée par la garde $grd12$ de l'événement FORWARDING_I_O_S8 ; et l'instruction « return S ; » en ligne 9 est donnée conjointement par la garde $grd14$ du même événement et l'action $act1 \ : \ out\_cnt := out\_cnt \cup \{o \mapsto p\}$, il s'agit du choix du port de sortie o pointant vers la direction Sud (S) pour l'acheminement du paquet p.

Nous pouvons résumer ce huitième raffinement M8 par le diagramme 6.32 suivant :

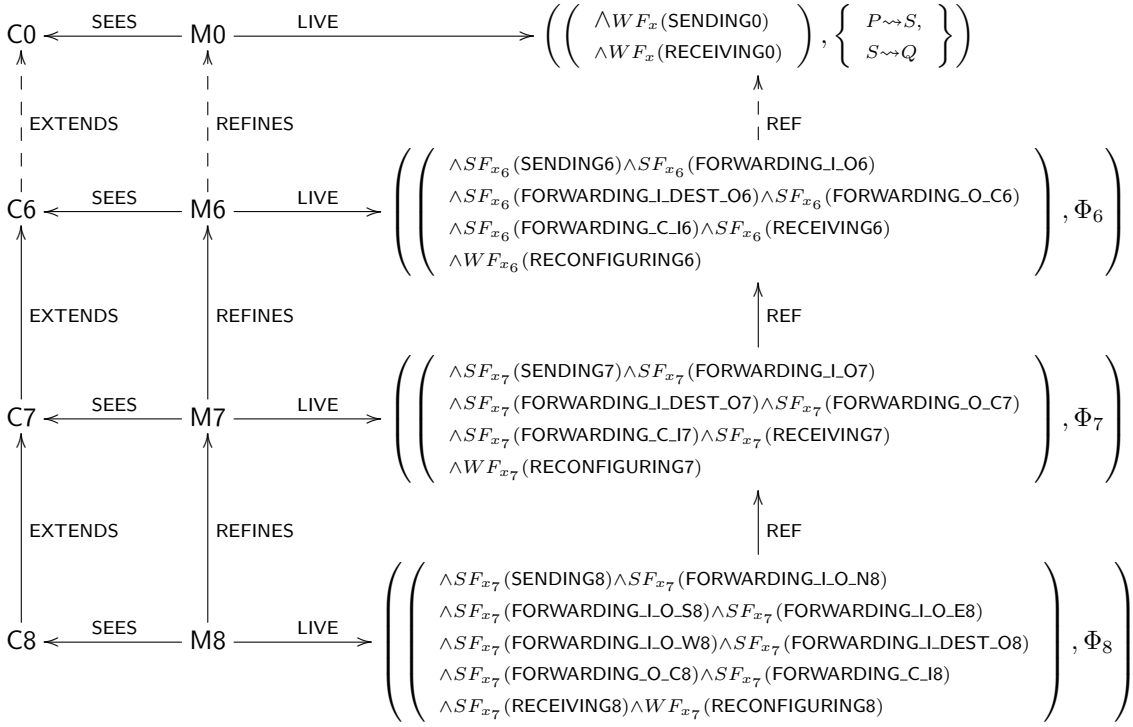


FIGURE 6.32 – Huitième raffinement du NoC

Ce diagramme nous montre aussi que les propriétés de vivacité Φ_7 du niveau de raffinement précédent sont satisfaites par M8 (voir l'annexe pour la preuve).

Nous notons que tous les événements modélisant les phases de l'algorithme de routage XY modifié sont sous hypothèse d'équité forte (RECONFIGURING8 est un événement relatif à l'environnement, mais non à l'algorithme) : cela est dû aux hypothèses posées par nos collaborateurs électroniciens sur le NoC, notamment sur le fait que les routeurs acheminant les paquets peuvent être désactivés, perturbant les phases de l'algorithme. Les hypothèses d'équité forte sont justifiées par la nécessité de progression de l'algorithme, pour éviter des blocages dans des cycles infinis de type « désactivation - reconfiguration ».

Nous avons utilisé le raffinement pour aboutir à un modèle présentant un « Réseau-sur-puce » (Network-on-Chip, NoC) : cette dernière machine M8 modélise l'algorithme XY pour l'acheminement des paquets dans le réseau, grâce aux raffinements de l'événement FORWARDING_I_O7 ; ainsi que l'architecture du réseau, sa topologie, et la structure interne des routeurs composant le réseau. Nous présentons dans la section suivante nos conclusions pour ce chapitre.

6.4 Conclusion

Nous avons présenté dans ce chapitre une modélisation et une vérification formelle d'un composant électronique, plus précisément, d'un circuit intégré, d'une puce électronique. Nous nous sommes particulièrement intéressés aux communications et aux transmissions de paquets entre les composants de la puce : nous nous sommes focalisés sur le paradigme *« Réseau-sur-puce »* (Network-on-Chip, NoC) qui permet de réaliser un réseau au niveau d'une (micro)puce et permet ainsi aux composants électronique de la puce de communiquer entre eux.

Le paradigme NoC, qui permet d'établir un réseau sur une puce et permet aux composants de cette dernière de communiquer, emprunte des techniques des réseaux de communication, par exemple la prise en compte du routage des paquets, qui se fait de façon distribuée : chaque composant routeur qui reçoit un paquet choisit où ce dernier doit être acheminée. La puce est ainsi analysée du point de vue axé sur les études/analyse de *réseaux*. Le circuit électronique que nous avons choisi d'étudier présente aussi des caractéristiques intéressantes : des erreurs, des fautes peuvent faire en sorte que des composants soient désactivés ; un mécanisme de *reconfiguration* est alors déclenché pour réactiver les composants fautifs. Il s'agit d'un système distribué analogue à un système distribué mobile. Nous nous sommes ainsi servis de ce cas d'étude sur les circuits intégrés et les NoCs pour expérimenter les techniques vues dans le chapitre précédent sur le routage et les systèmes distribués mobiles, principalement la réutilisation des modèles Event-B du patron de conception sur le routage, vu dans le chapitre précédent. Une partie de ces travaux a été réalisée en collaboration avec les laboratoires LIM de l'Université de Tiaret (Algérie) et LICM de l'Université de Lorraine, dans le cadre du projet STIC-Algérie 2011-2013 et a donné lieu à une publication [4].

Traditionnellement, la vérification de tels systèmes se fait par simulation [19, 20]. Cependant, plus ces systèmes deviennent complexes, plus il devient difficile d'explorer tous les espaces d'états possibles. Nous proposons par conséquent ici une vérification à l'aide des méthodes formelles : il s'agit d'un développement formel incrémental en Event-B, utilisant le raffinement de modèles. Ce développement formel est composé de neufs modèles et peut être divisé en trois étapes :

1. Une étape d'abstraction formée par les modèles M0, M1, M2.
2. Une étape de description des détails des structures des routeurs et du réseau : modèles M3 à M8.
3. Une étape d'instanciation de la topologie du NoC et de l'algorithme de routage : modèle M8.

La complexité du développement est donnée par le tableau 6.2 qui présente les statistiques des obligations de preuves (PO) déchargées automatiquement ou interactivement. Nous ne présentons ici que les statistiques relatives à l'utilisation exclusive des prouveurs de l'Atelier B. La première étape d'abstraction nous a permis de réutiliser des éléments du patron de conception pour le routage, défini dans le chapitre 5 (page 156 à page 193), notamment la partie du patron de conception définissant le routage 1-1 (une seule source, un seul destinataire) ; et d'en supprimer certains et/ou d'en adapter d'autres : les obligations de preuves des machines M0 et M1 sont moins nombreuses que celles présentées pour le patron de routage (voir page 193), car nous avons supprimé lors de la modélisation du NoC toute notion relative à la perte et renvoi de messages ; par contre nous avons plus d'obligations de preuves pour la machine M2, parce que nous avons du adapter et donc modifier les événements REMOVE_LINK et ADD_LINK du patron de conception pour modéliser respectivement la désactivation d'un routeur (DISABLING) et sa réactivation par reconfiguration (RECONFIGURING). Nous remarquons aussi que nous avons un nombre assez important d'obligations de preuves interactives pour les modèles M6, M7, M8 : cela est dû au fait que nous introduisons dans ces modèles les contraintes particulières du NoC, comme par exemple le nombre de places disponibles dans les ports d'entrée et de sortie des routeurs, prises en compte à l'aide des cardinalités, ou encore les différentes directions que peut prendre un paquet en sortant d'un routeur (Nord, Sud, Est, Ouest), etc.

Des travaux similaires aux nôtres existent : Kamali et al présentent dans [14] une modélisation des communications *multicast* et *unicast* dans un NoC tridimensionnel (un cube). Nous présentons ici dans ce chapitre, un développement plus modulaire : nos modèles de M0 à M7 modélisent un NoC générique, dont la topologie et l'algorithme de routage ne sont pas définis. Des raffinements du dernier modèle M7, en spécifiant la topologie du NoC (2-D, 3-D, etc) et l'algorithme de routage permet d'obtenir différents modèles concrets de NoCs (e.g M8, qui présente un NoC bidimensionnel carré). Nous tenons aussi compte

Modèles	Total	Automatiques		Interactives	
C0	0	0	100%	0	0%
C1	1	1	100%	0	0%
C2	0	0	100%	0	0%
C3	3	2	66.67%	1	33.33%
C4	8	7	87.5%	1	12.5%
C5	0	0	100%	0	0%
C6	0	0	100%	0	0%
C7	8	0	0%	8	100%
C8	23	21	91.3%	2	8.7%
M0	3	3	100%	0	0%
M1	15	12	80%	3	20%
M2	28	19	67.86%	9	32.14%
M3	53	40	75.47%	13	24.53%
M4	57	41	71.93%	16	28.07%
M5	72	36	50%	36	50%
M6	65	28	43.08%	37	56.92%
M7	45	17	37.78%	28	62.22%
M8	43	19	44.19%	24	55.81%
Total	424	245	57.78%	179	42.22%

TABLE 6.2 – NoC : Statistiques des POs

des erreurs et fautes pouvant affecter la puce : nous avons ici des événements modélisant la désactivation d'un routeur (DISABLING) et sa réactivation par reconfiguration (RECONFIGURING).

La modélisation du NoC et du protocole de routage XY nous a permis ici d'expérimenter l'application des patrons de conception et de réutiliser des concepts et modèles vus dans le cadre d'autres algorithmes (ANYCAST RP). Ces travaux nous ont aussi permis déjà de voir à l'aide du mécanisme de reconfiguration, un aperçu d'un système semblable aux systèmes auto-stabilisants [10, 16, 6] : le NoC a la capacité, grâce à la reconfiguration, de revenir à un état stable, permettant l'acheminement des paquets dans le réseau, de leurs sources à leurs destinataires. En d'autres termes, un paquet parviendra fatalement toujours à sa destination.

Bibliographie

[1] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*, chapter A Mechanical Press Controller. Cambridge University Press, 2009.

[2] J.-R. Abrial and T. S. Hoang. Using design patterns in formal methods : An event-b approach. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigün, editors, *ICTAC*, volume 5160 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2008.

[3] C. Alexander, S. Ishikawa, and M. Silverstein. *A pattern language : towns, buildings, construction.* Oxford University Press, 1977.

[4] M. B. Andriamiarina, H. Daoud, M. Belarbi, D. Méry, and C. Tanougast. Formal Verification of Fault Tolerant NoC-based Architecture. In *First International Workshop on Mathematics and Computer Science (IWMCS2012)*, Tiaret, Algérie, Dec. 2012.

[5] M. B. Andriamiarina, D. Méry, and N. K. Singh. Integrating proved state-based models for constructing correct distributed algorithms. In E. B. Johnsen and L. Petre, editors, *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 2013.

[6] M. B. Andriamiarina, D. Méry, and N. K. Singh. Analysis of self-* and p2p systems using refinement. In Y. Aït-Ameur and K.-D. Schewe, editors, *ABZ*, volume 8477 of *Lecture Notes in Computer Science*, pages 117–123. Springer, 2014.

[7] D. Cansell, D. Méry, and J. Rehm. Time Constraint Patterns for Event B Development. In O. K. Jacques Julliand, editor, *The 7th International B Conference - B 2007*, volume 4355 of *Lecture Notes in Computer Science*, pages 140–154, Besançon, France, 2007. Springer.

[8] S. D. Chawade, M. A. Gaikwad, and R. M. Patrikar. Article : Review of xy routing algorithm for network-on-chip architecture. *International Journal of Computer Applications*, 43(21/973-93-80867-69-8) :20–23, April 2012. Published by Foundation of Computer Science, New York, USA.

[9] W. J. Dally and B. Towles. Route packets, not wires : On-chip interconnection networks. In *DAC*, pages 684–689. ACM, 2001.

[10] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11) :643–644, Nov. 1974.

[11] P. C. French and R. W. Taylor. A self-reconfiguring processor. In D. A. Buell and K. L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 50–59, Los Alamitos, CA, April 1993. IEEE Computer Society Press.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] T. S. Hoang, A. Furst, and J.-R. Abrial. Event-b patterns and their tool support. *Software Engineering and Formal Methods, International Conference on*, 0 :210–219, 2009.

[14] M. Kamali, L. Petre, K. Sere, and M. Daneshtalab. Formal modeling of multicast communication in 3d nocs. In *DSD*, pages 634–642. IEEE, 2011.

[15] J. Kang, J. Sucec, V. Kaul, S. Samtani, and M. A. Fecko. Robust pim-sm multicasting using anycast rp in wireless ad hoc networks. In *Proceedings of the 2009 IEEE international conference on Communications*, ICC'09, pages 5139–5144, Piscataway, NJ, USA, 2009. IEEE Press.

[16] C. C. Marquezan and L. Z. Granville. *Self-* and P2P for Network Management - Design Principles and Case Studies*. Springer Briefs in Computer Science. Springer, 2012.

[17] D. Méry. Refinement-based guidelines for algorithmic systems. *International Journal of Software and Informatics*, 3(2-3) :197–239, June/September 2009.

[18] D. Méry and N. K. Singh. Analysis of DSR protocol in event-B. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, SSS'11, pages 401–415, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] A. Nayebi, S. Meraji, A. Shamaei, and H. Sarbazi-Azad. Xmulator : A listener-based integrated simulation platform for interconnection networks. In *Modelling Simulation, 2007. AMS '07. First Asia International Conference on*, pages 128–132, March 2007.

[20] M. Palesi, R. Holsmark, S. Kumar, and V. Catania. Application specific routing algorithms for networks on chip. *IEEE Trans. Parallel Distrib. Syst.*, 20(3) :316–330, 2009.

[21] S. Rajesh, V. K. C., S. Srivatsan, R.and Harini, and A. Shanthi. Fault tolerance in multicore processors with reconfigurable hardware unit. In *15th international conference on high performance computing*, pages 166–171, 2008.

[22] J. Rehm. *Gestion du temps par le raffinement*. Thèse, Université Henri Poincaré - Nancy I, Dec. 2009.

[23] R. P. S. Sidhu, A. Mei, and V. K. Prasanna. String matching on multicontext fpgas using self-reconfiguration. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, pages 217–226, New York, NY, USA, 1999. ACM.

[24] R. P. S. Sidhu, S. Wadhwa, A. Mei, and V. K. Prasanna. A self-reconfigurable gate array architecture. In R. W. Hartenstein and H. Grünbacher, editors, *FPL*, volume 1896 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2000.

[25] I. Suneetha and T. Venkateswarlu. Self-repairable reconfigurable circuit using embedded autonomously restructuring cores. *International Journal of Computer Science and Network Security*, 10(7) :142–153, January 2010.

[26] C. Systems. Anycast RP. `http://www.cisco.com/en/US/docs/ios/solutions_docs/ip_multicast/White_papers`.

[27] C. Systems. Anycast RP using PIM. `http://tools.ietf.org/html/draft-ietf-pim-anycast-rp-07`.

[28] C. Tavernier. *Circuits logiques programmables*. Microcontrôleurs et environnement. Dunod, 1996.

[29] W. Zhang, L. Hou, J. Wang, S. Geng, and W. Wu. Comparison research between xy and odd-even routing algorithm of a 2-dimension 3x3 mesh topology network-on-chip. In *Proceedings of the 2009 WRI Global Congress on Intelligent Systems - Volume 03*, GCIS '09, pages 329–333, Washington, DC, USA, 2009. IEEE Computer Society.

7

Les algorithmes de snapshot

Sommaire

7.1 Introduction

Un algorithme de *snapshot* [3] consiste en l'enregistrement d'un état global d'un système réparti. Nous commençons par définir cette notion d'*état global* d'un système réparti. Un système réparti est constitué d'un ensemble de processus [7] distants, ne partageant pas de mémoire commune et communiquant par messages circulant dans les canaux de communications reliant les processus. L'état d'un processus est caractérisé par son état local (état de sa mémoire locale), ainsi que l'historique de ses activité (envois, réceptions et traitements de messages). L'état d'un canal est constitué par les messages transitant dans ce canal, mais qui ne sont pas encore reçus par leurs destinataires. L'*état global* d'un système réparti est l'ensemble des états locaux de ses processus et canaux à un moment donné. L'enregistrement d'un tel état global définit un *snapshot*. Un algorithme de *snapshot* de l'état global d'un système réparti fonctionne généralement de manière parallèle/concurrente, sans interférence [3], par rapport à d'autres algorithmes répartis ayant le même système réparti pour support.

Le calcul d'un *snapshot* d'un système réparti a plusieurs usages et applications dans le domaine de l'algorithmique répartie [3, 7, 4], dont nous citons quelques exemples ici :

- analyse des propriétés d'un système réparti de manière « hors-ligne » : l'analyse se fait sur le snapshot (fixe) plutôt que sur les états courants des processus qui évoluent et changent. Un *snapshot* peut ainsi servir à la détection de propriétés *stables*, telles que la terminaison, les interblocages (deadlocks), etc.
- l'état global capturé par un *snapshot* peut servir à la restauration d'un système réparti, en cas de panne : un *snapshot* du système réparti est pris régulièrement et en cas de panne, c'est l'état global le plus récent capturé qui est restauré. Un « reset » du système est ainsi évité.
- un snapshot peut aussi aider au débogage d'un système réparti : une analyse « hors-ligne » d'un snapshot d'un état global défectueux peut révéler les causes (fautes, pannes, etc) à l'origine de l'état défectueux.

Nous présentons dans ce chapitre une modélisation et une analyse par raffinement du problème du *snapshot*. Notre objectif principal est de montrer que nous pouvons appliquer le paradigme *service-as-event* pour la modélisation et l'étude de problèmes existants et connus, tels que le *snapshot* [3, 7, 4, 5, 6, 8], qui opère en concurrence avec un autre algorithme dans un même système réparti, et dont le but est de prendre une photographie des états du système réparti et de ses composants (processus, canaux) à un moment donné. Nous proposons dans ce chapitre, un cadre, un patron de conception pour le développement formel par raffinement d'algorithmes de *snapshot*, tels que les algorithmes de Chandy et Lamport [3], Lai et Yang [5] et Morgan [6]. Ce cadre nous permet aussi d'établir des relations sémantiques entre les algorithmes de snapshot étudiés (e.g. les algorithmes de Chandy et Lamport [3], Lai et Yang [5]) à l'aide du raffinement de modèles. Un autre de nos objectifs est aussi qu'à l'aide de notre développement formel, les lecteurs comprennent le comportement théorique d'un algorithme de *snapshot*.

Nous avons introduit de manière générale nos objectifs et le problème du *snapshot* dans cette section. La prochaine section nous permettra de détailler les idées théoriques à la base du *snapshot*.

7.2 Présentation du snapshot

Nous considérons ici un système réparti composé d'un ensemble P de n processus distants, interconnectés par des canaux de communication. Nous posons les hypothèses suivantes : les processus ne partagent pas de mémoire globale et peuvent seulement communiquer par messages ; les envois et réceptions de messages par les processus sont asynchrones ; les communications sont fiables, c'est-à-dire que les messages envoyés sont fatalement reçus par leur destinataires après un temps fini. Ce système peut être représenté par un graphe dirigé dont les nœuds sont les processus et les arcs, les canaux de communications entre les processus.

Pour chaque processus p ($p \in P$), nous définissons un ensemble d'événements (locaux) E_p, tels que :

- *send* : envoi de messages par le processus p concerné. Nous notons $sent_{pq}$ l'ensemble des messages envoyés par un processus p à un processus q adjacent.
- *receive* : réception de messages par le processus p. Nous notons $rcvd_{pq}$ l'ensemble des messages reçus par un processus q d'un processus p adjacent.

— *internal* : un événement interne modifiant l'état du processus p.

Un *ordre partiel* appelé *ordre local causal* [7] et noté $\preceq_p$ pour un processus p, est défini sur les événements (locaux) du processus p : la relation $e_p^i \preceq_p e_p^j$, entre deux événements $e_p^i \in E_p$ et $e_p^j \in E_p$, signifie que e_p^i se produit avant l'événement e_p^j. Une coupure C de l'ensemble des événements locaux d'un processus p est un sous-ensemble de E_p satisfaisant la relation suivante :

$$\forall p, e, f \cdot p \in P \wedge e \in E_p \wedge f \in C \wedge e \preceq_p f \Rightarrow e \in C$$

Un événement e antérieur à la coupure C (un événement *e pre-shot*) fait également partie de la coupure C. Les événements ultérieurs à la coupure C (événements *post-shot*) ne font pas partie de cette dernière.

Un ordre plus global et général, appelé *ordre causal* [7] et noté $\preceq$ est aussi défini. Il s'agit de la plus petite relation contenant l'*ordre local causal* ($\preceq_p$) et satisfaisant l'ordre *send/receive* (envoi/réception de messages) existant entre les processus. Nous notons E_v l'ensemble des événements d'un système réparti, incluant les événement locaux d'un processus p (E_p) du système et les événements globaux de ce dernier (communication entre les processus, etc). La relation $e_m \preceq e_n$, entre deux événements e_m et e_n du système réparti ($e_m \in E_v$ et $e_n \in E_v$), signifie que e_m se produit avant e_n et que ces deux événements vérifient l'un des cas suivants :

1. Si e_m et e_n sont des événements locaux d'un processus p, alors $e_m \preceq_p e_n$.
2. Si e_m est l'envoi d'un message par un processus source que nous notons p_1, alors e_n est la réception de ce message par un processus destinataire que nous notons p_2.
3. Il existe un événement e_k, tel que $e_m \preceq e_k$ et $e_k \preceq e_n$.

Une coupure consistante C d'un ensemble d'événements E_v d'un système réparti est un sous-ensemble de E_v, satisfaisant la relation suivante :

$$\forall e, f \cdot e \in E_v \wedge f \in C \wedge e \preceq f \Rightarrow e \in C$$

Un état global est consistant, si chaque message reçu par un processus dans cet état global lui a été envoyé par un autre processus dans le même état global. Un tel état global préserve la causalité : un message ne peut être reçu s'il n'a pas été envoyé. Un état global consistant est un état global ayant *un sens* (c'est-à-dire ne contenant pas de contradiction, telle que $rcvd_{pq} \not\subseteq sent_{pq}$, pour deux processus p et q).

Nous rappelons ici la définition d'un snapshot S d'un système réparti : il s'agit d'un état global du système réparti, qui est en fait une collection des états locaux des processus du système et des états locaux des canaux de communication du système (l'état local d'un canal entre deux processus p et q est $sent_{pq} \setminus rcvd_{pq}$), produits par des événements internes ou des événements de communications entre processus. Un snapshot S est faisable, si pour chaque paire de processus voisins p et q, nous avons $rcvd_{pq} \subseteq sent_{pq}$. Un snapshot S a un sens, si et seulement si, le snapshot S est faisable et la coupure C induite par S est consistante.

Le théorème suivant [7] relie les notions de coupure et de snapshot :

Théorème 7.2.1. **[7]** *Soit S un snapshot et C la coupure induite par S, alors les trois affirmations suivantes sont équivalentes.*
— *S est faisable.*
— *C est une coupure consistante.*
— *S a un sens.*

Le but d'un snapshot est le calcul d'un état global consistant ou d'une coupure consistante d'un système réparti à partir des états locaux des composants de ce dernier (processus, canaux).

Nous présentons maintenant dans les sections suivante une modélisation par raffinement du calcul du *snapshot* d'un système réparti. Nous partons d'une abstraction et aboutissons à trois algorithmes concrets de snapshot : il s'agit des algorithmes de Chandy et Lamport [3], Lai et Yang [5] et Morgan [6].

7.3 Développement

7.3.1 Plan du développement

Nous commençons par présenter le plan que nous avons suivi lors du développement formel du problème du snapshot :

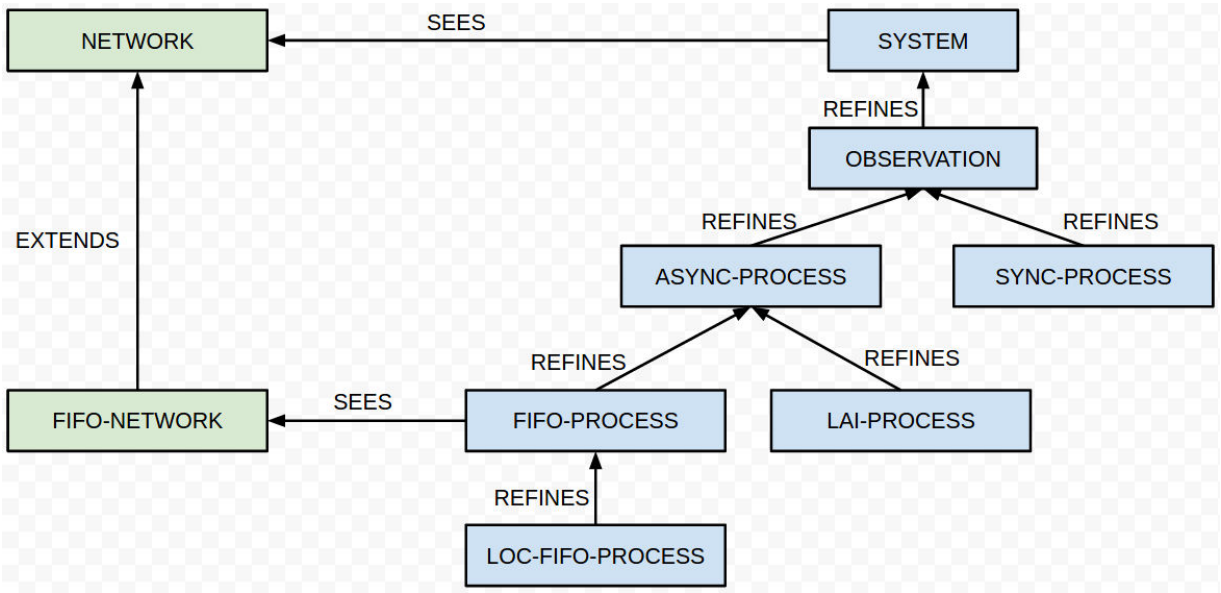


FIGURE 7.1 – Plan de développement du problème du *snapshot*

- Une machine SYSTEM utilise un contexte NETWORK : ce premier modèle décrit un système réparti composé de processus communiquant par messages.
- La machine SYSTEM est ensuite raffinée par une machine OBSERVATION : ce premier raffinement introduit le service requis par le problème, qui est le calcul du snapshot du système réparti. Ce calcul du snapshot se fait en parallèle et sans interférences avec les autres activités du système réparti. Le modèle ne détaille pas les étapes du calcul du snapshot, mais modélise comment obtenir un snapshot consistant en un coup.
- Les modèles SYNC-PROCESS et ASYNC-PROCESS sont dérivés du modèle OBSERVATION et modélisent deux manières différentes de calculer un snapshot :
  - SYNC-PROCESS présente un calcul synchrone en une seule phase du snapshot par tous les nœuds du système réparti. Le modèle SYNC-PROCESS modélise l'algorithme de Morgan [6].
  - ASYNC-PROCESS présente un calcul asynchrone du snapshot, avec une première phase d'initialisation et une phase de construction. Le modèle ASYNC-PROCESS modélise une version abstraite de l'algorithme de Lai et Yang [5].
- Le modèle LAI-PROCESS présente une version plus concrète du modèle ASYNC-PROCESS, par conséquent de l'algorithme de Lai et Yang.
- Un contexte FIFO-NETWORK étend le contexte NETWORK, en introduisant les canaux de communication FIFO. Ce contexte est utilisé par une machine FIFO-PROCESS, raffinement de ASYNC-PROCESS, et modélisant l'algorithme de Chandy et Lamport [3].
- Le modèle LOC-FIFO-PROCESS raffine FIFO-PROCESS et présente une version plus concrète de l'algorithme de Chandy et Lamport.

7.3.2 Premier modèle : le système réparti

Le premier modèle, composé d'un contexte NETWORK et d'une machine SYSTEM, modélise un système réparti, composé de processus qui s'échangent des messages et qui sont capables d'actions internes

CONTEXT NETWORK
SETS
$P, M, PStates$
CONSTANTS
$C, o0, E, h0, state, l0, store0, c0, cut0$
AXIOMS
$axm1 : P \neq \varnothing$
$axm2 : M \neq \varnothing$
$axm3 : PStates \neq \varnothing$
$axm4 : C = (P \times P) \setminus id$
$axm5 : \forall p \cdot p \in P \Rightarrow p \in dom(C)$
$axm6 : C \neq \varnothing$
$axm7 : o0 \in P \to \mathbb{N}$
$axm8 : \forall p \cdot p \in P \Rightarrow o0(p) = 0$
$axm9 : E = P \times P$
$axm10 : h0 \in P \to (\mathbb{N} \nrightarrow E)$
$axm11 : \forall p \cdot p \in P \Rightarrow h0(p) = \{0 \mapsto (p \mapsto p)\}$
$axm12 : state \in PStates$
$axm13 : l0 \in P \to PStates$
$axm14 : \forall p \cdot p \in P \Rightarrow l0(p) = state$
$axm15 : store0 \in P \to \mathbb{P}(M)$
$axm16 : \forall p \cdot p \in P \Rightarrow store0(p) = \varnothing$
$axm17 : c0 \in C \to \mathbb{P}(M)$
$axm18 : \forall c \cdot c \in C \Rightarrow c0(c) = \varnothing$
$axm19 : cut0 \in P \to \mathbb{N}$
$axm20 : \forall p \cdot p \in P \Rightarrow cut0(p) = 0$

$$axm21 : \exists \begin{pmatrix} ct, \\ hi, \\ se, \\ rc \end{pmatrix} \cdot \begin{pmatrix} \wedge ct \in P \to \mathbb{N} \\ \wedge hi \in P \to (\mathbb{N} \nrightarrow E) \\ \wedge se \in C \times \mathbb{N} \rightarrowtail M \\ \wedge rc \in C \times \mathbb{N} \rightarrowtail M \\ \wedge \begin{pmatrix} \forall p, q, i, j, m \cdot \begin{pmatrix} \wedge p \in P \\ \wedge q \in P \\ \wedge m \in M \\ \wedge p \neq q \\ \wedge i \in dom(hi(p)) \\ \wedge j \in dom(hi(q)) \\ \wedge (p \mapsto q \mapsto i) \mapsto m \in se \\ \wedge (p \mapsto q \mapsto j) \mapsto m \in rc \\ \wedge j \leq ct(q) \end{pmatrix} \Rightarrow i \leq ct(p) \end{pmatrix} \\ \wedge (\forall p \cdot p \in P \Rightarrow ct(p) \in dom(hi(p))) \end{pmatrix}$$

END

(faire évoluer son état local, gérer son pool de messages, etc.).

Nous commençons par présenter le contexte NETWORK : nous définissons dans ce contexte un ensemble P non-vide ($axm1$) de processus, un ensemble M non-vide ($axm2$) de messages, ainsi qu'un ensemble $PState$ non-vide ($axm3$) d'états. Une constante C non-vide ($axm6$) modélise les canaux de communication entre les processus ($axm4$, $axm5$). Ces canaux ne relient que des processus différents entre eux ($axm4$). Une estampille est associée à chaque processus. Une constante $o0$ ($axm7$) définit l'état initial de ces estampilles : elles sont initialisées à 0 ($axm8$). Les processus maintiennent aussi chacun un historique de leurs actions, qu'elles soient internes ou externes. Une constante E ($axm9$) permet de définir les portées des actions des processus : elle peuvent être locales (décrites par un couple (p, p), pour un processus p) ou impliquer des processus différents (décrites par un couple (p, q), pour deux processus p et q différents). Une constante $h0$ associe à chaque processus p son historique initial ($axm10$) : il s'agit d'une action interne au temps 0 ($axm11$). Une constante $l0$ associe chaque processus à un état initial ($axm13$, $axm14$) défini par la constante $state$ ($axm12$). Le contenu initial du pool de messages de chaque processus est décrit par une constante $store0$ ($axm16$) : à l'état initial, le pool de message est vide ($axm15$). Le contenu de chaque canal de communication est décrit similairement par une constante $c0$ ($axm17$) : à l'état initial, les canaux sont vides ($axm18$). Ce contexte définit aussi une constante $cut0$ ($axm19$) qui ne sera utilisée qu'au prochain raffinement : la coupure locale initiale pour chaque processus p est effectué au temps 0 ($axm20$). Le dernier axiome $axm21$ exprime qu'un snapshot ct consistant du système décrit dans ce premier modèle est possible : un envoi d'un message m, par un processus p à un autre processus q, à un instant i, fait partie du snapshot ct, ssi la réception du même message m par le processus q, à un instant j, fait partie de ce même snapshot ct. Le snapshot ct ne doit contenir que des actions enregistrés dans un historique hi des actions des processus. Cet axiome $axm21$ sera utilisé dans le raffinement suivant, notamment pour décharger l'obligation de preuve de *faisabilité* (FIS) liée à l'événement modélisant le snapshot de manière abstraite.

Nous détaillons maintenant la machine SYSTEM qui utilise ce contexte NETWORK. La machine SYSTEM décrit le comportement de chaque composant (processus, canaux) du système réparti définit par le contexte NETWORK. Nous introduisons les variables suivantes dans SYSTEM :

INITIALISATION $\widehat{=}$
BEGIN
$act1 \ : \ o := o0$
$act2 \ : \ l := l0$
$act3 \ : \ h := h0$
$act4 \ : \ store := store0$
$act5 \ : \ chan := c0$
$act6 \ : \ send := \varnothing$
END

$inv1 \ : \ o \in P \to \mathbb{N}$
$inv2 \ : \ l \in P \to PStates$
$inv3 \ : \ h \in P \to (\mathbb{N} \nrightarrow E)$
$inv4 \ : \ store \in P \to \mathbb{P}(M)$
$inv5 \ : \ chan \in C \to \mathbb{P}(M)$
$inv6 \ : \ send \subseteq M$

— Une variable o associe à chaque processus son estampille courante ($inv1$). Sa valeur initiale est définie par la constante $o0$.
— Une variable l associe à chaque processus son état courant ($inv2$). Sa valeur initiale est définie par la constante $l0$.
— Une variable h associe à chaque processus son historique courant ($inv3$). Sa valeur initiale est définie par la constante $h0$.
— Une variable *store* associe à chaque processus les messages stockés par ce dernier dans son pool de messages ($inv4$). A l'état initial, ce pool est vide, tel que défini par la constante $store0$.
— Une variable *chan* modélise le contenu de chaque canal de communication ($inv5$) : à l'état initial, chaque canal est vide, tel que défini par la constante $c0$.
— Une variable *send* modélise les messages envoyés par les processus ($inv6$). Elle est initialisée à l'aide de l'ensemble vide, car à l'état initial, aucun processus n'a encore envoyé de messages.

Les invariants suivants expriment des propriétés de sûreté contraignant ces variables :

$$
\begin{array}{l}
inv7 \ : \ \forall p \cdot p \in P \Rightarrow dom(h(p)) = 0 \mathinner{..} o(p) \\
inv8 \ : \ \forall c, m \cdot c \in C \wedge m \in M \wedge m \in chan(c) \Rightarrow m \in send \\
inv9 \ : \ \forall m, p \cdot m \in store(p) \Rightarrow m \in send \\
inv10 \ : \ \forall m, p1, p2, q1, q2 \cdot \left(\begin{array}{l} \wedge p1 \mapsto q1 \in C \\ \wedge p2 \mapsto q2 \in C \\ \wedge m \in chan(p1 \mapsto q1) \\ \wedge m \in chan(p2 \mapsto q2) \end{array} \right) \Rightarrow p1 = p2 \wedge q1 = q2
\end{array}
$$

— $inv7$ exprime que l'historique d'un processus p est indexé par des estampilles allant de 0 à la valeur courante $o(p)$ de l'estampille du processus p.
— $inv8$ exprime que si un message m circule dans un canal c, alors ce message m a été envoyé par un processus.
— $inv9$ exprime que que si un message m est stocké par un processus p, alors cela signifie que le message m a été envoyé au processus p par un autre processus.
— $inv10$ exprime le fait qu'un message m ne peut circuler que dans un canal unique à un moment donné.

Des événements décrivent les actions qu'un processus du système réparti peut effectuer. Il peut s'agir :
— d'événements modélisant des actions internes/locales à un processus.
  — L'événement InternalLocal modélise le fait qu'un processus p fasse évoluer son état local vers un nouvel état ns ($act1$) : la valeur de l'estampille du processus p est incrémentée ($act2$) et le fait que le processus p a effectué une action locale au temps $o(p) + 1$ est sauvegardé dans son historique ($act3$).

EVENT InternalLocal $\widehat{=}$
ANY
p, ns
WHERE
$grd1 \ : \ p \in P$
$grd2 \ : \ ns \in PStates$
THEN
$act1 \ : \ l(p) := ns$
$act2 \ : \ o(p) := o(p) + 1$
$act3 \ : \ h(p) := h(p) \cup \{o(p) + 1 \mapsto (p \mapsto p)\}$
END

— l'événement InternalMessage modélise la manipulation par un processus p de son pool de messages $store(p)$. Nous modélisons ici une suppression d'un message m de ce pool ($grd3$) par le processus p ($act3$) : la valeur de l'estampille du processus p est incrémentée ($act1$) et le fait que le processus p a effectué une action locale au temps $o(p)+1$ est sauvegardé dans son historique ($act2$).

EVENT InternalMessage $\widehat{=}$
ANY
p, m
WHERE
$grd1 \ : \ p \in P$
$grd2 \ : \ m \in M$
$grd3 \ : \ m \in store(p)$
THEN
$act1 \ : \ o(p) := o(p) + 1$
$act2 \ : \ h(p) := h(p) \cup \{o(p) + 1 \mapsto (p \mapsto p)\}$
$act3 \ : \ store(p) := store(p) \setminus \{m\}$
END

— d'événements modélisant des actions externes impliquant un processus et un de ces voisins.
— l'événement Sending modélise l'envoi d'un message m ($act1$), par un processus p, via un canal sortant c ($act2$) connecté au processus p ($grd4$) menant à un autre processus voisin. Le message m est envoyé, s'il ne l'a pas encore été ($grd5$) et s'il ne circule pas dans le canal c ($grd6$). La valeur de l'estampille courante du processus p est alors incrémentée ($act3$) et le fait que le processus p a effectué une action externe (impliquant un autre processus) au temps $o(p) + 1$ est sauvegardé dans son historique ($act4$).

EVENT Sending $\widehat{=}$
ANY
p, m, c
WHERE
$grd1 \ : \ p \in P$
$grd2 \ : \ m \in M$
$grd3 \ : \ c \in C$
$grd4 \ : \ prj1(c) = p$
$grd5 \ : \ m \notin send$
$grd6 \ : \ m \notin chan(c)$
THEN
$act1 \ : \ send := send \cup \{m\}$
$act2 \ : \ chan(c) := chan(c) \cup \{m\}$
$act3 \ : \ o(p) := o(p) + 1$
$act4 \ : \ h(p) := h(p) \cup \{o(p) + 1 \mapsto c\}$
END

— l'événement Receiving modélise la réception d'un message m, par un processus q ($act4$), via un canal entrant c ($act1$) connecté au processus p ($grd4$, $grd5$). Le message m est reçu par q, s'il a au préalable circulé dans le canal c ($grd5$). Le message m passe alors du canal c au pool de messages de q ($act4$). La valeur de l'estampille courante du processus p est alors incrémentée ($act3$) et le fait que le processus q a effectué une action externe (impliquant un autre processus) au temps $o(q) + 1$ est sauvegardé dans son historique ($act4$).

EVENT Receiving $\widehat{=}$
ANY
q, m, c
WHERE
$grd1 : q \in P$
$grd2 : m \in M$
$grd3 : c \in C$
$grd4 : prj2(c) = q$
$grd5 : m \in chan(c)$
THEN
$act1 : chan(c) := chan(c) \setminus \{m\}$
$act2 : o(q) := o(q) + 1$
$act3 : h(q) := h(q) \cup \{o(q) + 1 \mapsto c\}$
$act4 : store(q) := store(q) \cup \{m\}$
END

Le raffinement suivant introduit l'idée du *snapshot*, qui est la capture d'un état global consistant du système réparti défini dans ce premier modèle.

7.3.3 Le modèle OBSERVATION

Ce premier raffinement OBSERVATION de la machine SYSTEM modélise l'idée du *snapshot* : il s'agit d'une photographie à un moment donné d'un état global du système distribué. Pour cela, la machine OBSERVATION introduit trois variables :

INITIALISATION $\widehat{=}$
BEGIN
...
$\oplus : r := \varnothing$
$\oplus : s := \varnothing$
$\oplus : cut := cut0$
END

$inv1 : s \in C \times \mathbb{N} \nrightarrow M$
$inv2 : r \in C \times \mathbb{N} \nrightarrow M$
$inv3 : cut \in P \rightarrow \mathbb{N}$

— s associe à chaque canal entre deux processus, les messages envoyés par un processus via le canal couplés aux instants (estampilles) auxquels ils ont été envoyés ($inv1$). Elle est initialisée à l'aide de l'ensemble vide ($\varnothing$).
— r associe à chaque canal entre deux processus, les messages reçus par un processus via le canal couplés aux instants (estampilles) auxquels ils ont été reçus ($inv2$). Elle est initialisée à l'aide de l'ensemble vide ($\varnothing$).
— *cut* contient le snapshot : il associe à chaque processus l'instant (l'estampille) auquel le snapshot local du processus a été calculé ($inv3$). Elle est initialisé à l'aide de la constante $cut0$ définie dans le contexte NETWORK.

Nous notons x_{obs} l'ensemble des variables de la machine OBSERVATION.

Un invariant noté $I_{obs}(x_{obs})$ caractérise ces variables et est défini comme suit :

— $inv4$ à $inv13$ expriment des propriétés de sûreté satisfaites par le modèle OBSERVATION :

$$
\begin{array}{l}
inv4 \ : \ \forall p,q,m \cdot \left(\begin{array}{l} \wedge p \in P \\ \wedge q \in P \\ \wedge m \in M \\ \wedge p \mapsto q \in C \\ \wedge m \in chan(p \mapsto q) \end{array} \right) \Rightarrow m \notin ran(r) \\
inv5 \ : \ \forall p,q,m \cdot \left(\begin{array}{l} \wedge p \in P \\ \wedge q \in P \\ \wedge m \in M \\ \wedge p \mapsto q \in C \\ \wedge m \notin (send \cup chan(p \mapsto q)) \end{array} \right) \Rightarrow m \notin ran(s) \\
inv6 \ : \ \forall p,q,n \cdot p \mapsto q \mapsto n \in dom(s) \Rightarrow n \leq o(p) \\
inv7 \ : \ \forall p,q,n \cdot p \mapsto q \mapsto n \in dom(r) \Rightarrow n \leq o(q) \\
inv8 \ : \ \forall p,q,m \cdot \left(\begin{array}{l} \wedge p \in P \\ \wedge q \in P \\ \wedge m \in M \\ \wedge p \mapsto q \in C \\ \wedge m \notin send \end{array} \right) \Rightarrow m \notin ran(r) \\
inv9 \ : \ ran(s) = send \\
inv10 \ : \ ran(r) \subseteq send \\
inv11 \ : \ \forall p,q,m,j \cdot (p \mapsto q \mapsto j \mapsto m) \in r \Rightarrow (\exists i \cdot (p \mapsto q \mapsto i \mapsto m) \in s) \\
inv12 \ : \ \forall p,q,m,j \cdot (p \mapsto q \mapsto j \mapsto m) \in r \Rightarrow j \in dom(h(q)) \\
inv13 \ : \ \forall p,q,m,i \cdot (p \mapsto q \mapsto i \mapsto m) \in s \Rightarrow i \in dom(h(p))
\end{array}
$$

— $inv4$ exprime que si un message m circule dans un canal entre deux processus p et q, alors le message m n'a pas encore été reçu par le processus q.
— $inv5$ exprime que si un message m n'a été ni envoyé, ni ne circule dans aucun canal entre deux processus quelconques p et q, alors cela signifie que le message m n'a pas encore été associé dans la variable s à un canal et à un instant (estampille) d'envoi.
— $inv6$ exprime que l'estampille (l'instant) d'envoi n d'un message par un processus p à un processus q est inférieure ou égale à l'estampille courante du processus p.
— $inv7$ exprime que l'estampille (l'instant) de réception n d'un message par un processus q est inférieure ou égale à l'estampille courante du processus q.
— $inv8$ exprime qu'un message m non-envoyé ne peut être reçu.
— $inv9$ exprime que chaque message associé dans la variable s à un canal et à un instant (estampille) d'envoi, a été effectivement envoyé par un processus.
— $inv10$ exprime que chaque message associé dans la variable r à un canal et à un instant (estampille) de réception, a été envoyé par un processus.
— $inv11$ exprime que si un message m envoyé par un processus p à un processus q a été reçu par ce dernier à un instant j, alors il existe un instant i, auquel le processus p a envoyé le message m.
— $inv12$ exprime que si un processus q a reçu un message m d'un processus p à un instant j, alors q a enregistré ce moment j dans son historique.
— $inv13$ exprime que si un processus p a envoyé un message m à un processus q à un instant i, alors p a enregistré ce moment i dans son historique.

Ce raffinement contient les raffinement des événements Sending, Receiving, InternalLocal ou InternalMessage. Un nouvel événement Snapshot modélisant le calcul d'un snapshot consistant du système étudié y est aussi introduit. Nous définissons maintenant la condition d'équité L_0 associée à la machine OBSERVATION :

$$L_0 \ \widehat{=} \ WF_{x_{obs}}(\mathsf{Snapshot})$$

Nous posons une hypothèse d'équité faible sur l'événement Snapshot et aucune hypothèse d'équité sur les autres événements, car Snapshot est le seul événement participant à la réalisation d'un *snapshot* du système réparti.

Nous définissons aussi la relation NEXT pour la machine OBSERVATION :

$$
\begin{array}{rcl}
\mathsf{NEXT} & \widehat{=} & \vee BA(\mathsf{InternalLocal})(x_{obs}, {x_{obs}}') \\
& & \vee BA(\mathsf{InternalMessage})(x_{obs}, {x_{obs}}') \\
& & \vee BA(\mathsf{Sending})(x_{obs}, {x_{obs}}') \\
& & \vee BA(\mathsf{Receiving})(x_{obs}, {x_{obs}}') \\
& & \vee BA(\mathsf{Snapshot})(x_{obs}, {x_{obs}}')
\end{array}
$$

Nous nous intéressons maintenant à la liste Φ_0 des propriétés de vivacité caractérisant la machine OBSERVATION. Pour cela nous posons :

— $Q_{obs}(x_{obs}) \;\widehat{=}\; \left(\begin{array}{l} \wedge\ cut \in P \to \mathbb{N} \\ \wedge\ \forall p, q, i, j, m \left(\begin{array}{l} \wedge q \in P \\ \wedge m \in M \\ \wedge p \neq q \\ \wedge i \in dom(h(p)) \\ \wedge j \in dom(h(q)) \\ \wedge (p \mapsto q \mapsto i) \mapsto m \in s \\ \wedge (p \mapsto q \mapsto j) \mapsto m \in r \\ \wedge j \leq cut(q) \end{array} \right) \Rightarrow i \leq cut(p) \\ \wedge\ \forall p \cdot p \in P \Rightarrow cut(p) \in dom(h(p)) \end{array} \right)$

Cette propriété Q_{obs} exprime que la valeur de la variable *cut* après le calcul d'un snapshot contient un snapshot consistant : si le moment de réception j par un processus d'un message m fait partie du snapshot, alors le moment d'envoi i de ce même message fait aussi partie du snapshot.

Nous définissons la liste Φ_0 comme suit : $\Phi_0 \;\widehat{=}\; \{I_{obs} \rightsquigarrow Q_{obs}\}$. La propriété $I_{obs} \rightsquigarrow Q_{obs}$ exprime qu'un état satisfaisant l'invariant I_{obs} conduit à un état où un snapshot consistant du système réparti a été calculé.

Le diagramme 7.2 suivant décrit la procédure abstraite du snapshot, ainsi que les propriétés de vivacité qui caractérise cette procédure :

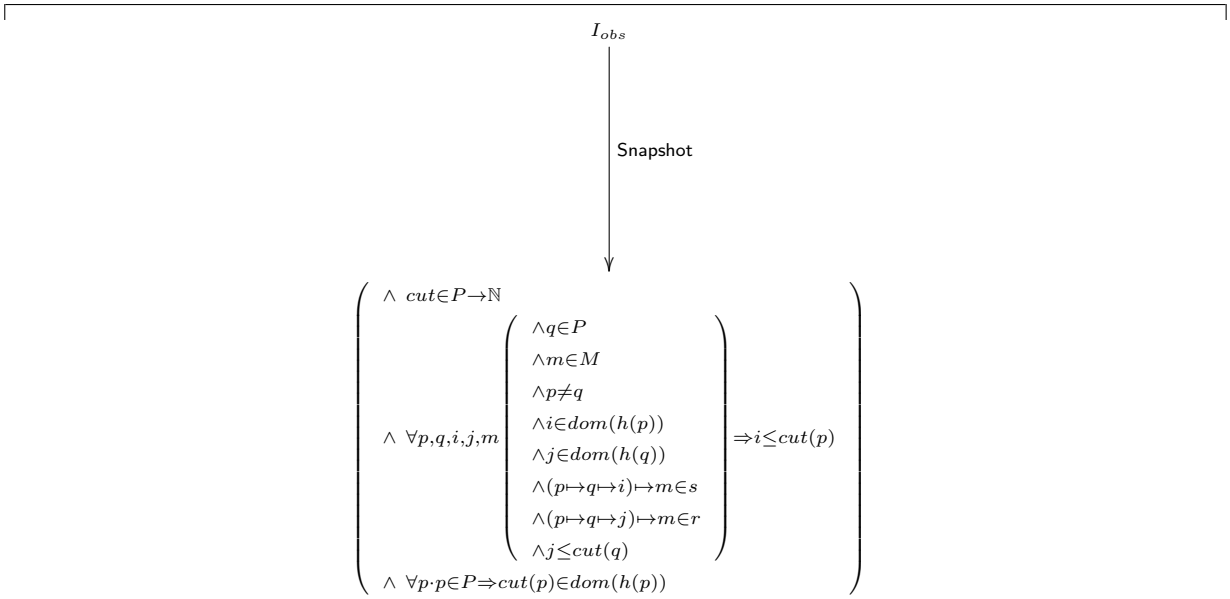


FIGURE 7.2 – Diagramme d'assertions pour OBSERVATION

Démonstration. Nous montrons maintenant que la machine OBSERVATION satisfait les propriétés de vivacité de Φ_0.

⟨1⟩1. Nous montrons que OBSERVATION satisfait $P \rightsquigarrow S$. Une hypothèse d'équité faible est posée sur l'événement Snapshot et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de Snapshot sont observables lorsque P est vrai, leurs observations ne falsifient pas le prédicat P, condition d'observation de Snapshot, ou conduisent à S.

Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$I_{obs} \wedge [\mathsf{NEXT}]_{x_{obs}} \Rightarrow (I_{obs}{}' \vee Q_{obs}{}')$$

Soit :

— $I_{obs} \wedge BA(\mathsf{InternalLocal})(x_{obs}, x_{obs}{}') \Rightarrow (I_{obs}{}' \vee Q_{obs}{}')$

— $I_{obs} \wedge BA(\mathsf{InternalMessage})(x_{obs}, x_{obs}') \Rightarrow (I_{obs}' \vee Q_{obs}')$
— $I_{obs} \wedge BA(\mathsf{Sending})(x_{obs}, x_{obs}') \Rightarrow (I_{obs}' \vee Q_{obs}')$
— $I_{obs} \wedge BA(\mathsf{Receiving})(x_{obs}, x_{obs}') \Rightarrow (I_{obs}' \vee Q_{obs}')$
— $I_{obs} \wedge BA(\mathsf{Snapshot})(x_{obs}, x_{obs}') \Rightarrow (I_{obs}' \vee Q_{obs}')$
— $I_{obs} \wedge (x_{obs} = x_{obs}') \Rightarrow (I_{obs}' \vee Q_{obs}')$

La propriété suivante **(3)** : $I_{obs}(x_{obs}) \wedge BA(\mathsf{Snapshot})(x_{obs}, x_{obs}') \Rightarrow Q_{obs}(x_{obs}')$, et la condition de faisabilité **(4)** : $I_{obs}(x_{obs}) \Rightarrow (\exists x_{obs}' \cdot BA(\mathsf{Snapshot})(x_{obs}, x_{obs}'))$ sont satisfaites par l'événement Snapshot. La propriété **(3)** nous permet de déduire $I_{obs} \wedge \langle \mathsf{NEXT} \wedge \mathsf{Snapshot} \rangle_{x_{obs}} \Rightarrow Q_{obs}'$ et **(4)** nous permet de déduire $I_{obs} \Rightarrow ENABLED\langle \mathsf{Snapshot} \rangle_{x_{obs}}$, où $ENABLED\langle \mathsf{Snapshot} \rangle_{x_{obs}} \ \widehat{=}\ (\exists y \cdot BA(\mathsf{Snapshot})(x_{obs}, y))$. L'application de WF1 permet de montrer que $\mathcal{S}pec(\mathsf{OBSERVATION}) \vdash I_{obs} \rightsquigarrow Q_{obs}$, donc que OBSERVATION satisfait $I_{obs} \rightsquigarrow Q_{obs}$. □

⟨1⟩2. L'étape ⟨1⟩1 nous permet de déduire que OBSERVATION satisfait les propriétés de vivacité de Φ_0. QED.

□

Nous utilisons maintenant les propriétés de vivacité de Φ_0 pour définir les événements de la machine OBSERVATION :

— L'événement Snapshot modélise une abstraction des étapes nécessaires à la réalisation d'un *snapshot* et exprime qu'une coupure consistante est obtenue en un seul coup : l'action $act1$ exprime qu'un envoi d'un message m, par un processus p à un autre processus q, à un instant i, fait partie de la coupure cut, ssi la réception du même message m par le processus q, à un instant j, fait partie de la coupure cut. En d'autres termes, nous interdisons les inconsistances ; par exemple, lorsque la réception d'un message se trouve dans la coupure, mais pas son envoi.

$$
\begin{array}{l}
\text{EVENT Snapshot } \widehat{=} \\
\quad \text{BEGIN} \\
\qquad act1 \ : \ cut : | \left(\begin{array}{l}
\wedge\ cut' \in P \to \mathbb{N} \\
\wedge\ \forall p, q, i, j, m \left(\begin{array}{l}
\wedge q \in P \\
\wedge m \in M \\
\wedge p \neq q \\
\wedge i \in dom(h(p)) \\
\wedge j \in dom(h(q)) \\
\wedge (p \mapsto q \mapsto i) \mapsto m \in s \\
\wedge (p \mapsto q \mapsto j) \mapsto m \in r \\
\wedge j \leq cut'(q)
\end{array} \right) \Rightarrow i \leq cut'(p) \\
\wedge\ \forall p \cdot p \in P \Rightarrow cut'(p) \in dom(h(p))
\end{array} \right) \\
\quad \text{END}
\end{array}
$$

— Nous modifions par raffinement les événements Sending et Receiving pour prendre en compte les variables s et r :
  — L'événement Sending utilise la variable s pour indiquer dans quel canal c et à quel moment (estampille) un message m a été envoyé ($act5$).

```
EVENT Sending REFINES Sending ≙
  ANY
    ...
  WHERE
    ...
  THEN
    ...
    ⊕ act5 : s := s ∪ {c ↦ o(p) + 1 ↦ m}
  END
```

  — L'événement Receiving enregistre dans la variable r, le canal c par lequel a transité un message m reçu par un processus q et l'instant (estampille) auquel ce message a été reçu ($act5$). Pour que cet événement soit observé, le message m doit avoir au préalable été envoyé au processus destinataire q, via le canal c, à un instant (estampille) i ($grd6$, $grd7$).

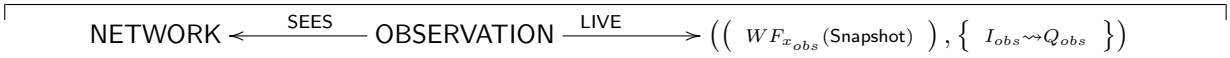


FIGURE 7.3 – Abstraction du snapshot

```
EVENT Receiving REFINES Receiving ≙
  ANY
    ...
    ⊕ i
  WHERE
    ...
    ⊕ grd6 : i ∈ ℕ
    ⊕ grd7 : (c ↦ i ↦ m) ∈ s
  THEN
    ...
    ⊕ act5 : r := r ∪ {c ↦ o(q) + 1 ↦ m}
  END
```

— Les autres événements du modèle SYSTEM, tels que InternalLocal ou InternalMessage ne sont pas modifiés durant ce raffinement. Nous ne les présentons donc plus dans cette sous-section.

Nous pouvons résumer cette abstraction par le diagramme 7.3 suivant :

Nous avons présenté ici une modélisation abstraite du snapshot d'un système distribué en introduisant un événement qui calcule le snapshot en un coup. Des raffinements sont nécessaires pour introduire les étapes de la procédure conduisant au calcul d'une coupure consistante. Le but est la définition d'une procédure inductive réduisant, à chaque appel, le nombre de processus qui n'ont pas encore effectué leur snapshots locaux, jusqu'à ce que ce nombre soit égal à 0 : tous les processus ont effectué leurs snapshot locaux. Nous présentons ces raffinements dans les sous-sections suivantes. Ces raffinements sont détaillés en annexe [8].

7.3.4 Calcul synchrone d'un snapshot (SYNC-PROCESS)

La machine SYNC-PROCESS est un raffinement de la machine OBSERVATION. La machine SYNC-PROCESS définit le calcul synchrone par les processus d'un système réparti d'un snapshot consistant : un élément/paramètre global partagé par tous les processus (par exemple une horloge) déclenche la procédure de snapshot exécutée par tous les processus, au même moment. La machine SYNC-PROCESS modélise les étapes de l'algorithme de snapshot de Morgan [6], qui est basé sur la disponibilité d'une horloge globale partagée et accessible par chaque processus du système réparti étudié.

Nous introduisons dans la machine SYNC-PROCESS de nouvelles variables :

```
INITIALISATION ≙
  BEGIN
    ...
    ⊕ : cstate := ∅
    ⊕ : pstate := ∅
    ⊕ : t := FALSE
    ⊕ : mark_m := ∅
    ⊕ : pcut := ∅
  END
```

— *cstate* enregistre, pour chaque processus, les messages pre-shot entrants, reçus via un canal connecté au processus ($inv2$). Elle est initialisée à l'aide de l'ensemble vide ($\varnothing$).

— *pstate* contient, pour chaque processus, l'état du processus lors du snapshot ($inv3$). Elle est initialisée à l'aide de l'ensemble vide ($\varnothing$).

— t est un booléen représentant une horloge globale ($inv5$). Sa valeur est « vrai » quand l'instant de déclenchement de la procédure de snapshot est atteint, « faux » sinon. Elle a pour valeur initiale « faux ».

— *mark_m* contient les messages envoyés après le snapshot ($inv4$). Elle est initialisée à l'aide de l'ensemble vide ($\varnothing$).

8. http://andriami.bruno.free.fr/andriami/main/short/annexe.pdf

— *pcut* est une variable intermédiaire contenant le calcul pas-à-pas du snapshot ($inv1$) par chaque processus. Elle est initialisée à l'aide de l'ensemble vide ($\varnothing$).

Nous notons x_s l'ensemble des variables de cette machine.

Un invariant que nous notons $I_s(x_s)$ définit des contraintes sur ces variables :

— Les propriétés de $inv1$ à $inv5$ permettent de typer les nouvelles variables introduites dans SYNC-PROCESS :

$$\begin{array}{l} inv1 \ : \ pcut \in P \pfun \mathbb{N} \\ inv2 \ : \ cstate \in C \pfun \mathbb{P}(M) \\ inv3 \ : \ pstate \in P \pfun PStates \\ inv4 \ : \ mark\_m \subseteq M \\ inv5 \ : \ t \in BOOL \end{array}$$

— Les propriétés de $inv6$ à $inv10$ expriment des propriétés de sûreté :

$$\begin{array}{l} inv6 \ : \ \forall q \cdot q \in P \land q \in dom(pcut) \Rightarrow (\forall p \cdot p \mapsto q \in C \Rightarrow p \mapsto q \in dom(cstate)) \\ inv7 \ : \ t = FALSE \Rightarrow pcut = \varnothing \\ inv8 \ : \ dom(pcut) = P \Rightarrow t = TRUE \\ inv9 \ : \ \forall p \cdot p \in P \land p \in dom(pcut) \Rightarrow pcut(p) \leq o(p) \\ inv10 \ : \ \forall p, q, i, j, m \cdot p \in P \left(\begin{array}{l} \land q \in P \\ \land m \in M \\ \land p \neq q \\ \land p \in dom(pcut) \\ \land q \in dom(pcut) \\ \land (p \mapsto q \mapsto i) \mapsto m \in s \\ \land (p \mapsto q \mapsto j) \mapsto m \in r \\ \land j \leq pcut(q) \end{array} \right) \Rightarrow i \leq pcut(p) \end{array}$$

— $inv6$ exprime que si un processus q a effectué son snapshot local, alors il est prêt à enregistrer tous les messages pre-shots qu'il reçoit par ses canaux entrants.

— $inv7$ exprime que si l'instant de déclenchement de la procédure de snapshot n'est pas atteint, alors aucun processus n'a encore effectué de snapshot local.

— $inv8$ exprime que si tous les processus ont effectué leurs snapshots locaux, cela signifie l'instant l'instant de déclenchement de la procédure de snapshot a été atteint, ou même dépassé.

— $inv9$ exprime que si un processus p a effectué son snapshot local à un instant (estampille) donné, alors cet instant donné est inférieur ou égal à la valeur indiquée par son estampille courante.

— $inv10$ exprime la consistance d'un snapshot : si un message m est envoyé par un processus p à un processus q à un instant i et si l'instant j de la réception de ce même message m par le processus q se produit avant le snapshot, alors l'envoi de m à l'instant i a aussi eu lieu avant le snapshot.

Cette machine SYNC-PROCESS introduit les raffinements des événements Sending, Receiving, InternalMessage, InternalLocal, Snapshot des événements précédents :

— SendingBeforeCut, SendingAfterCut, raffinements de Sending.

— ReceivingPreCutMessages, ReceivingPostCutMessages, ReceivingBeforeCut, raffinements de Receiving.

— InternalMessage et InternalLocal.

— Snapshot.

Un nouvel événement ProcessingSnapshot, modélisant les étapes intermédiaires de calcul du snapshot est introduit et un événement Tick modélise le passage du temps. Nous avons pu voir précédemment que l'objectif dans cette machine SYNC-PROCESS est de réduire à l'aide d'une ou de plusieurs itérations d'un événement clé, tel que ProcessingSnapshot, le nombre de processus n'ayant pas encore effectué leurs snapshots locaux. Nous posons $F(n+1) \ \widehat{=} \ (card(P \setminus dom(pcut)) = n+1)$, indiquant que le nombre de processus n'ayant pas effectué leurs snapshots locaux est $n+1$. Le diagramme 7.4 suivant nous permet de résumer le service de snapshot modélisé par la machine SYNC-PROCESS, ainsi que la liste Φ_s des propriétés de vivacité qui le caractérisent. La propriété $(\forall c \cdot c \in C \Rightarrow chan(c) \subseteq mark\_m)$ signifie que tous les messages transitant par chaque canal c du système sont tous des messages *post-shot*.

Conformément aux propriétés de vivacité vues dans Φ_s, nous ajoutons dans SYNC-PROCESS des événements du modèle modélisent les étapes de calcul du snapshot :

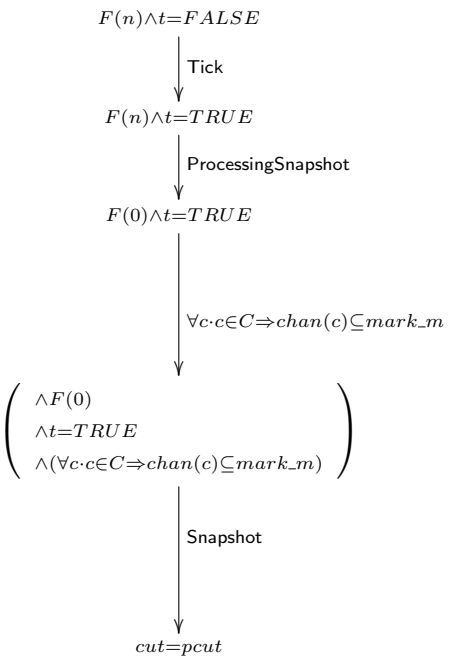


FIGURE 7.4 – Diagramme d'assertions pour SYNC-PROCESS

— L'événement Tick modélise le flot du temps : nous donnons à la variable t la valeur $TRUE$ (vrai) pour modéliser le fait que le temps global prédéfini pour le déclenchement de la procédure du snapshot est atteint ($act1$).

EVENT Tick $\widehat{=}$
  WHEN
    $grd1 \ : \ t = FALSE$
  THEN
    $act1 \ : \ t := TRUE$
  END

— L'événement ProcessingSnapshot modélise les calculs simultanés par tous les processus de leurs snapshots locaux. Quand le temps global prédéfini pour le calcul du snapshot est atteint ($grd2$) et que le calcul du snapshot du système réparti n'a pas encore eu lieu ($grd3$), chaque processus enregistre son état local ($act3$), commence à enregistrer les messages pre-shot arrivant par ses canaux entrants ($grd1$, $grd4$, $act1$) et enregistre l'instant auquel le calcul du snapshot local a eu lieu ($act2$).

EVENT ProcessingSnapshot $\widehat{=}$
  ANY
    $ncstate$
  WHERE
    $grd1 \ : \ ncstate \in C \rightarrow \mathbb{P}(M)$
    $grd2 \ : \ t = TRUE$
    $grd3 \ : \ pcut = \varnothing$
    $grd4 \ : \ ncstate = \{k \mapsto \varnothing | k \in C\}$
  THEN
    $act1 \ : \ cstate := ncstate$
    $act2 \ : \ pcut := o$
    $act3 \ : \ pstate := l$
  END

— L'événement Snapshot présente le snapshot global. La variable $pcut$ contient une coupure globale consistante des états du système réparti, et sa valeur est calculé lors de l'observation de l'événement

ProcessingSnapshot. Quand tous les processus ont enregistré leurs états locaux ($grd1$), ainsi que les messages pré-shot arrivant par leurs canaux entrants ($grd3$), alors l'état global contenu dans $pcut$ est sauvegardé dans la variable cut ($act1$).

```
EVENT Snapshot REFINES Snapshot ≙
  WHEN
    grd1 : dom(pcut) = P
    grd2 : t = TRUE
    grd3 : ∀p·p ∈ C ⇒ chan(p) ⊆ mark_m
  THEN
    act1 : cut := pcut
  END
```

Les événements Sending et Receiving sont modifiés par raffinement. Leurs raffinements modélisent les activités du système *avant* et *après* le snapshot :

— L'événement SendingBeforeCut modélise l'envoi de messages par un processus avant que le temps prédéfini pour le calcul du snapshot ne soit atteint ($grd7$). Les actions de cet événement sont similaires aux actions de l'événement abstrait Sending.

```
EVENT SendingBeforeCut REFINES Sending ≙
  ANY
    ...
  WHERE
    ...
    ⊕ grd7 : t = FALSE
  THEN
    ...
  END
```

— L'événement SendingAfterCut modélise l'envoi de messages par un processus p, après que ce dernier ait effectué son snapshot local ($grd7$). Les messages envoyés sont marqués comme étant envoyés par le processus p après le snapshot local ($act6$).

```
EVENT SendingAfterCut REFINES Sending ≙
  ANY
    ...
  WHERE
    ...
    ⊕ grd7 : p ∈ dom(pcut)
  THEN
    ...
    ⊕ act6 : mark_m := mark_m ∪ {m}
  END
```

— ReceivingPreCutMessages modélise la réception d'un message m *pre-shot* ($grd9$), par un processus q ayant déjà calculé son snapshot local ($grd8$) : le message m, qui a transité par un canal entrant c, est alors enregistré par le processus q ($act6$).

```
EVENT ReceivingPreCutMessages REFINES Receiving ≙
  ANY
    ...
  WHERE
    ...
    ⊕ grd8 : q ∈ dom(pcut)
    ⊕ grd9 : m ∉ mark_m
  THEN
    ...
    ⊕ act6 : cstate(c) := cstate(c) ∪ {m}
  END
```

— ReceivingPostCutMessages modélise la réception d'un message m *post-shot* ($grd9$), par un processus q ayant déjà calculé son snapshot local ($grd8$) : le message m est alors retiré de la liste des messages *post-shot* ($act6$).

EVENT ReceivingPostCutMessages REFINES Receiving $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd8$: $q \in dom(pcut)$
$\oplus$ $grd9$: $m \in mark\_m$
THEN
...
$\oplus$ $act6$: $mark\_m := mark\_m \setminus \{m\}$
END

— ReceivingBeforeCut modélise la réception d'un message m ($grd9$), par un processus q avant que le temps prédéfini pour le calcul du snapshot ne soit atteint ($grd8$). Les actions de cet événement sont les mêmes que celles de l'événement abstrait Receiving.

EVENT ReceivingBeforeCut REFINES Receiving $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd8$: $t = FALSE$
$\oplus$ $grd9$: $m \notin mark\_m$
THEN
...
END

L'algorithme suivant décrit ce snapshot selon Morgan :

Algorithm 6 Algorithme de Morgan

Hypothèse : Partage d'une horloge globale par les processus

Init-M : {A un moment prédéfini t}

1: **for** $p \in P$ **do**
2: marquer tous les messages sortants par leur moment d'envoi ;
3: **record**(state(p)) ;
4: enregistrer les messages reçus par p à t ou après t, qui lui ont été envoyés avant le moment t ;
5: **end for**

A un instant prédéfini t, tous les processus effectuent les étapes suivantes :

1. Marquer les messages sortants par les moments auxquels ils sont envoyés.
2. Enregistrer leurs états locaux avant de continuer leurs calculs.
3. Enregistrer les messages reçus à l'instant t ou après et qui leur ont été envoyés (strictement) avant l'instant t : il s'agit des messages circulant dans les canaux de communication à l'instant t.

Deux événements modélisent principalement cet algorithme de Morgan ; il s'agit des événements ProcessingSnapshot et SendingAfterCut :

— la condition d'initialisation **Init-M** du snapshot est exprimée par $t = TRUE$ et le fait qu'un processus p ait calculé son snapshot est exprimé par $p \in pcut$.
  — l'événement SendingAfterCut exprime que chaque processus p marque tous les messages qu'il envoie, aussitôt que le moment t est atteint. Il s'agit de l'étape 1 (ligne 2) de l'algorithme de Morgan.
  — les actions $act2$ et $act3$ de l'événement ProcessingSnapshot exprime l'étape 2 (ligne 3) de l'algorithme de Morgan : l'enregistrement par chaque processus p de son état local.
  — l'action $act1$ de l'événement ProcessingSnapshot exprime l'étape 3 (ligne 4) de l'algorithme de Morgan : chaque processus p se prépare à enregistrer les messages entrants lui ayant été envoyés (strictement) avant l'instant t.

Nous pouvons résumer ce raffinement par le diagramme 7.5 suivant :

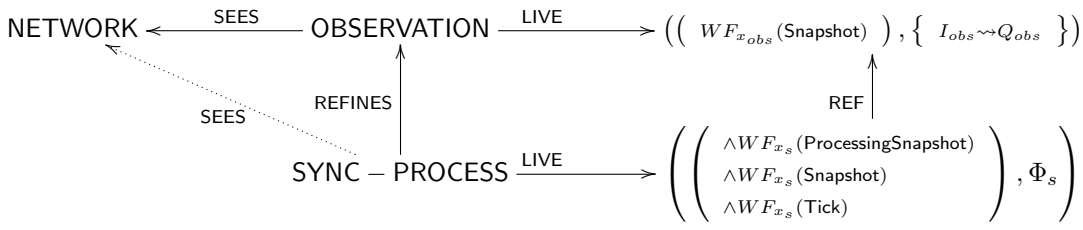


FIGURE 7.5 – Snapshot synchrone

Nous avons présenté dans cette sous-section un modèle de l'algorithme de snapshot de Morgan [6] : il s'agit d'un algorithme de calcul synchrone d'un snapshot par tous les processus d'un système réparti, suite à un stimulus global, partagé et observable par tous les processus. Les sous-sections suivantes décrivent la dérivation d'autres algorithmes de snapshot [3, 5], qui diffèrent de ce premier algorithme par le fait qu'aucune ressource, mémoire, aucun élément, stimulus ne sont partagés par les processus composant le système.

7.3.5 Modèles d'algorithmes asynchrones

Nous commençons par introduire dans cette sous-section des modèles d'algorithmes de *snapshot* asynchrones. Le premier modèle est un modèle abstrait ASYNC-PROCESS modélisant l'algorithme de Lai et Yang. Les autres modèles d'algorithmes sont dérivés par raffinement de ce modèle ASYNC-PROCESS. Le deuxième modèle est un modèle plus concret de l'algorithme de Lai et Yang : il s'agit d'une localisation du modèle ASYNC-PROCESS. Les autres modèles sont les suivants : un premier modèle transforme les canaux de communication non-ordonnées en file FIFO et modélise ainsi de manière abstraite l'algorithme de Chandy et Lamport ; un second modèle raffine ce dernier et est un modèle plus concret de l'algorithme de Chandy et Lamport.

7.3.5.1 Algorithme de Lai et Yang abstrait (ASYNC-PROCESS)

La machine ASYNC-PROCESS raffine la machine OBSERVATION et présente une procédure de construction asynchrone d'un snapshot global par les processus composant un système réparti étudié. Par rapport à la machine SYNC-PROCESS, pour faire la séparation entre les étapes pré et post-snapshot, nous ajoutons ici un contrôle sur les messages : les messages *post-shot* seront marqués, ce qui permettra de les distinguer clairement de leurs homologues *pre-shot*, qui ne sont pas marqués. La machine ASYNC-PROCESS modélise l'algorithme de snapshot de Lai et Yang[5], qui se comporte comme suit :

1. Nous supposons que chaque message $mess$ envoyé par chaque processus p du système est couplé à un booléen, qui est la valeur de la variable locale $taken_p$ (**Send-LY**$_p$) : si la valeur du booléen est $false$, cela signifie que le message $mess$ a été envoyé avant le snapshot, il s'agit d'un message *pre-shot*, sinon, il s'agit d'un message *post-shot*.
2. Nous supposons qu'au moins un processus p n'ayant pas encore enregistré son état local initie le *snapshot* (**Init-LY**$_p$) :
   — Le processus p initiateur enregistre son état local.
   — Il indique qu'il a enregistré son état local en mettant sa variable locale $taken_p$ à $true$.
   — Le processus p commence ensuite à enregistrer tous les messages *pre-shot* entrants.
3. A la réception d'un message $\langle mess, c\rangle$ (**Receive-LY**$_p$), un processus p vérifie si la valeur de c est true. Dans ce cas, si p n'a pas encore enregistré son état local ($\neg taken_p$), les mêmes étapes algorithmiques que celles décrites dans le point **2** sont exécutées par le processus p.

La machine ASYNC-PROCESS introduit de nouvelles variables :

```
INITIALISATION ≙
BEGIN
    ...
    ⊕ : pcut := ∅
    ⊕ : send_mark := ∅
    ⊕ : cstate := ∅
    ⊕ : pstate := ∅
    ⊕ : initiator :∈ P
    ⊕ : mark_m := ∅
END
```

— *pcut* est une variable intermédiaire contenant la construction graduelle du snapshot global ($inv1$). Elle est initialisée à l'aide de l'ensemble vide ($\varnothing$).
— *send_mark* représente le contrôle sur les messages. Cette variable permet aux processus de savoir si des messages marqués (post-shot) ont déjà circulé dans les canaux qui leurs sont adjacents et de se préparer à marquer les messages qu'ils expédieront ($inv2$). Cette variable est initialisée à l'aide de l'ensemble vide ($\varnothing$).
— *cstate* enregistre, pour chaque processus, les messages pre-shot entrants, reçus via un canal connecté au processus ($inv3$). Elle est initialisée à l'aide de l'ensemble vide ($\varnothing$).
— *pstate* contient, pour chaque processus, l'état du processus lors du snapshot ($inv4$). Elle est initialisée à l'aide de l'ensemble vide ($\varnothing$).
— *initiator* contient le processus qui initialise le calcul du snapshot ($inv5$). Sa valeur initiale est un processus quelconque du système réparti étudié.
— *mark_m* contient les messages *post-shot*. Sa valeur initiale est l'ensemble vide ($\varnothing$).

L'ensemble des variables de cette machine ASYNC-PROCESS est notées x_a.

Un invariant noté $I_a(x_a)$ contraint ces variables :

— Les propriétés de $inv1$ à $inv6$ permettent de typer les nouvelles variables introduites dans ASYNC-PROCESS :

$$
\begin{array}{l}
inv1 \ : \ pcut \in P \pfun \mathbb{N} \\
inv2 \ : \ cstate \in C \pfun \mathbb{P}(M) \\
inv3 \ : \ send\_mark \subseteq C \\
inv4 \ : \ pstate \in P \pfun PStates \\
inv5 \ : \ initiator \in P \\
inv6 \ : \ mark\_m \subseteq M
\end{array}
$$

— Les propriétés de $inv7$ à $inv14$ sont des propriétés de sûreté :

$$
\begin{array}{l}
inv6 \ : \ initiator \notin dom(pcut) \Rightarrow pcut = \varnothing \\
inv7 \ : \ dom(send\_mark) \neq \varnothing \Rightarrow initiator \in dom(pcut) \\
inv8 \ : \ initiator \notin dom(pcut) \Rightarrow send\_mark = \varnothing \\
inv9 \ : \ initiator \notin dom(pcut) \Rightarrow pstate = \varnothing \\
inv10 \ : \ \forall p, q \cdot p \mapsto q \in send\_mark \Rightarrow q \mapsto p \in dom(cstate) \\
inv11 \ : \ \forall p \cdot p \in P \land p \in dom(pcut) \Rightarrow pcut(p) \leq o(p) \\
inv12 \ : \ \forall p \cdot p \in dom(pcut) \Rightarrow (\forall i \cdot p \mapsto i \in C \Rightarrow p \mapsto i \in send\_mark) \\
inv13 \ : \ \forall p \cdot p \in dom(pcut) \Rightarrow (\forall i \cdot p \mapsto i \in C \Rightarrow i \mapsto p \in dom(cstate)) \\
inv14 \ : \ \forall m \cdot m \in mark\_m \Rightarrow (\exists i, j \cdot i \mapsto j \in C \land m \in chan(i \mapsto j))
\end{array}
$$

— $inv7$ exprime que si aucun processus initiateur n'a calculé son snapshot local, alors aucun processus n'a effectué le calcul de son snapshot local.
— $inv8$ exprime que si des canaux ont déjà acheminé des messages marqués, alors cela signifie qu'au moins un processus (l'initiateur) a calculé son snapshot local.
— $inv9$ exprime que si aucun processus initiateur n'a calculé son snapshot local, alors aucun état local n'a été enregistré.
— $inv10$ exprime que si un message marqué a été envoyé par un processus p à un processus q, alors q se prépare déjà à enregistré les messages *pre-shot* entrants qu'il recevra de p.
— $inv11$ exprime que si un processus p a calculé son snapshot local à un instant (estampille) donné, alors cet instant donné est soit inférieur ou égal à son estampille courante.
— $inv12$ exprime que si un processus p a calculé son snapshot local, alors tous les messages qu'il va envoyer par les canaux qui lui sont connectés seront marqués.

— $inv13$ exprime que si un processus p a calculé son snapshot local, alors il va commencer à sauvegarder les messages pre-shot reçus par les canaux qui lui sont connectés.
— $inv14$ exprime que si un message m est marqué, cela veut dire qu'il circule dans un canal du système réparti.

— Nous classons les invariants de $inv16$ à $inv18$ à part, car ces derniers caractérisent la consistance d'un snapshot :

$$inv15 \ : \ \forall p, q, i, j, m \cdot \left(\begin{array}{l} \land p \in P \\ \land q \in P \\ \land m \in M \\ \land p \neq q \\ \land p \in dom(pcut) \\ \land q \in dom(pcut) \\ \land (p \mapsto q \mapsto i) \mapsto m \in s \\ \land (p \mapsto q \mapsto j) \mapsto m \in r \\ \land j \leq pcut(q) \end{array} \right) \Rightarrow i \leq pcut(p)$$

$$inv16 \ : \ \forall p, q, i, j, m \cdot \left(\begin{array}{l} \land p \in P \\ \land q \in P \\ \land m \in M \\ \land p \neq q \\ \land p \in dom(pcut) \\ \land q \notin dom(pcut) \\ \land (p \mapsto q \mapsto i) \mapsto m \in s \\ \land (p \mapsto q \mapsto j) \mapsto m \in r \end{array} \right) \Rightarrow i \leq pcut(p)$$

$$inv17 \ : \ \forall p, q, i, m \cdot \left(\begin{array}{l} \land p \in P \\ \land q \in P \\ \land m \in M \\ \land p \neq q \\ \land p \in dom(pcut) \\ \land q \notin dom(pcut) \\ \land (p \mapsto q \mapsto i) \mapsto m \in s \\ \land m \notin mark\_m \end{array} \right) \Rightarrow i \leq pcut(p)$$

— $inv15$: si un message m est envoyé par un processus p à un instant i et s'il est reçu par un processus q à un instant j avant le snapshot, alors l'instant i se situe aussi avant le snapshot.
— $inv16$: si un message m est envoyé à un instant i par un processus p, qui a déjà effectué son snapshot local, est reçu à un instant j par un processus q qui n'a pas encore calculé son snapshot local, alors cela signifie que l'instant i se situe avant le snapshot.
— $inv17$: si un message m non-marqué est envoyé à un instant i par un processus p, qui a déjà effectué son snapshot local et est reçu à un instant j par un processus q qui n'a pas encore calculé son snapshot local, alors cela signifie que l'instant i se situe avant le snapshot.

Cette machine ASYNC-PROCESS introduit les raffinements des événements Sending, Receiving, InternalMessage, InternalLocal, Snapshot des événements précédents :

— SendingBeforeCut, SendingAfterCut, raffinements de Sending.
— ReceivingPreCutMessages, ReceivingPostCutMessages, ReceivingBeforeCut, raffinements de Receiving.
— InternalMessage et InternalLocal.
— Snapshot.

De nouveaux événements StartingSnapshot, ProgressingSnapshot, modélisant les étapes intermédiaires de calcul du snapshot sont aussi introduits.

Nous avons pu voir précédemment que l'objectif dans cette machine ASYNC-PROCESS est de réduire à l'aide d'une ou de plusieurs itérations d'un ou de plusieurs événements clés, tels que StartingSnapshot et ProgressingSnapshot, le nombre de processus n'ayant pas encore effectué leurs snapshots locaux. Nous posons $F(n+1) \ \widehat{=} \ (card(P \setminus dom(pcut)) = n+1)$, indiquant que le nombre de processus n'ayant pas effectué leurs snapshots locaux est $n+1$. Nous posons aussi k, avec $k > 2$, le nombre de nœuds dans le système réparti : à l'état initial, k nœuds n'ont pas encore calculé leurs snapshots locaux, soit $F(k)$. Le diagramme 7.6 suivant nous permet de résumer le service de snapshot modélisé par la machine ASYNC-PROCESS, ainsi que la liste Φ_a des propriétés de vivacité qui le caractérisent :

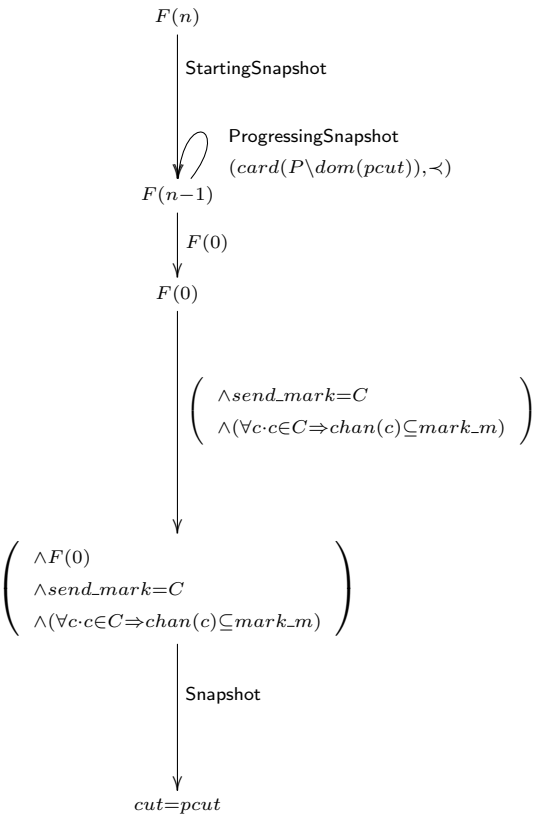


FIGURE 7.6 – Diagramme d'assertions pour ASYNC-PROCESS

La propriété $send\_mark = C$ exprime que tous les canaux reliant les nœuds du système ont vu transiter un message m couplé à un booléen marqueur ; $(\forall c \cdot c \in C \Rightarrow chan(c) \subseteq mark\_m)$ signifie que tous les messages transitant par chaque canal c du système sont tous des messages *post-shot*.

Conformément aux propriétés de vivacité vues dans Φ_a, nous ajoutons dans ASYNC-PROCESS des événements du modèle modélisent les étapes de calcul du snapshot :

— StartingSnapshot : un processus initiateur (*initiator*) commence le calcul du snapshot. Il sauvegarde son état local ($act4$), l'instant auquel il a calculé son snapshot local ($act3$). Il se prépare aussi à envoyer par ses canaux sortants des messages marqués ($grd4$, $act1$) et commence à enregistrer les messages *pre-shot* arrivant par ses canaux entrants ($grd5$, $act2$).

EVENT StartingSnapshot $\widehat{=}$
  ANY
    $nsm, ncstate$
  WHERE
    $grd1$: $nsm \subseteq C$
    $grd2$: $ncstate \in C \nrightarrow \mathbb{P}(M)$
    $grd3$: $initiator \notin dom(pcut)$
    $grd4$: $nsm = send\_mark \cup \{d | d \in C \wedge prj1(d) = initiator\}$
    $grd5$: $ncstate = cstate \cup \{d \mapsto \varnothing | d \in C \wedge prj2(d) = initiator\}$
  THEN
    $act1$: $send\_mark := nsm$
    $act2$: $cstate := ncstate$
    $act3$: $pcut(initiator) := o(initiator)$
    $act4$: $pstate(initiator) := l(initiator)$
  END

— ProgressingSnapshot : un processus i a reçu via un canal entrant c des messages marqués (*post-shot*) ($grd6$, $grd7$). Il effectue alors le calcul de son snapshot local. Il sauvegarde son état local ($act2$), l'instant auquel il a calculé son snapshot local ($act1$). Il se prépare aussi à envoyer par

ses canaux sortants des messages marqués ($grd9$, $act4$) et commence à enregistrer les messages *pre-shot* arrivant par ses canaux entrants ($grd10$, $act3$).

EVENT ProgressingSnapshot $\widehat{=}$
ANY
$i, c, nsm, ncstate$
WHERE
$grd1$: $i \in P$
$grd2$: $c \in C$
$grd3$: $nsm \subseteq C$
$grd4$: $ncstate \in C \pfun \mathbb{P}(M)$
$grd5$: $i = prj2(c)$
$grd6$: $c \in send\_mark$
$grd7$: $chan(c) \subseteq mark\_m$
$grd8$: $i \notin dom(pcut)$
$grd9$: $nsm = send\_mark \cup \{d | d \in C \land prj1(d) = i\}$
$grd10$: $ncstate = cstate \cup \{d \mapsto \varnothing | d \in C \land prj2(d) = i\}$
THEN
$act1$: $pcut(i) := o(i)$
$act2$: $pstate(i) := l(i)$
$act3$: $cstate := ncstate$
$act4$: $send\_mark := nsm$
END

— Snapshot : tous les processus ont reçus des messages marqués et en ont envoyés ($grd2$) ; ils ont aussi calculé leurs snapshots locaux ($grd1$) et les messages circulant dans le système ne sont plus que des messages *post-shot* ($grd3$). La valeur de la coupure *pcut* calculée durant les événements StartingSnapshot et ProgressingSnapshot devient alors la valeur de la variable *cut* modélisant le snapshot ($act1$).

EVENT Snapshot REFINES Snapshot $\widehat{=}$
WHEN
$grd1$: $dom(pcut) = P$
$grd2$: $send\_mark = C$
$grd3$: $\forall p \cdot p \in C \land p \in send\_mark \Rightarrow chan(p) \subseteq mark\_m$
THEN
$act1$: $cut := pcut$
END

Les événements d'envoi et de réception de messages sont raffinés en des versions *pré* et *post-shot* :
— L'événement SendingBeforeCut modélise l'envoi de messages *pré-shot* par un processus p : un processus p qui n'a pas encore effectué son snapshot local ($grd7$) envoie un message à un autre processus via un canal c adjacent.

EVENT SendingBeforeCut REFINES Sending $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd7$: $p \notin dom(pcut)$
THEN
...
END

— L'événement SendingAfterCut modélise l'envoi de messages *post-shot* par un processus p : un processus p ayant effectué son snapshot local ($grd7$) envoie un message à un autre processus via un canal c adjacent.

EVENT SendingAfterCut REFINES Sending $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd7$: $p \in dom(pcut)$
THEN
...
$\oplus$ $act6$: $mark\_m := mark\_m \cup \{m\}$
END

— L'événement ReceivingPreCutMessages modélise la réception d'un message *pre-shot* m ($grd9$), via un canal c adjacent, par un processus q ayant effectué son snapshot local ($grd8$). Le processus q sauvegarde l'état du canal c en enregistrant le message*pre-shot* m ($act6$).

EVENT ReceivingPreCutMessages REFINES Receiving $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd8$: $q \in dom(pcut)$
$\oplus$ $grd9$: $m \notin mark\_m$
THEN
...
$\oplus$ $act6$: $cstate(c) := cstate(c) \cup \{m\}$
END

— L'événement ReceivingPostCutMessages modélise la réception d'un message *post-shot* m ($grd9$), via un canal c adjacent, par un processus q ayant effectué son snapshot local ($grd8$). Le message *pre-shot* m est retiré de la liste des messages marqués/post-shot ($act6$).

EVENT ReceivingPostCutMessages REFINES Receiving $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd8$: $q \in dom(pcut)$
$\oplus$ $grd9$: $m \in mark\_m$
THEN
...
$\oplus$ $act6$: $mark\_m := mark\_m \setminus \{m\}$
END

— L'événement ReceivingBeforeCut modélise la réception d'un message *pre-shot* m ($grd9$), via un canal c adjacent, par un processus q n'ayant pas encore effectué son snapshot local ($grd8$). Les actions de cet événement sont les mêmes que celles de l'événement abstrait Receiving.

EVENT ReceivingBeforeCut REFINES Receiving $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd8$: $q \notin dom(pcut)$
$\oplus$ $grd9$: $m \notin mark\_m$
THEN
...
END

Ce modèle ASYNC-PROCESS modélise les phases de l'algorithme de Lai et Yang [5] :

— L'événement StartingSnapshot modélise l'initialisation du snapshot d'un système réparti par un processus *initiator* : ce dernier enregistre son état local, met à jour une variable indiquant qu'il a effectué son snapshot local (*pcut*) et commence à enregistrer les messages *pré-shot* arrivant par ses canaux entrants. Lors d'un envoi de messages, le processus *initiator* marquera chaque message envoyé comme étant un message *post-shot*.

— L'événement ProgressingSnapshot modélise la construction progressive du snapshot : tout processus i n'ayant pas initialisé le snapshot, mais ayant reçu un message *post-shot* (marqué) d'un autre processus, enregistre son état local, met à jour une variable indiquant qu'il a effectué son snapshot local (*pcut*) et commence à enregistrer les messages *pré-shot* arrivant par ses canaux entrants. Lors d'un envoi de messages, le processus i marquera chaque message envoyé comme étant un message *post-shot*.

— L'événement Snapshot est observé lorsque tous les processus ont enregistré leurs états locaux et que les états des canaux au moment du snapshot ont été sauvegardés. Il indique la fin du snapshot.

Nous pouvons résumer ce raffinement par le diagramme 7.7 suivant :

Nous avons présenté dans cette sous-section un modèle de l'algorithme de snapshot de Lai et Yang [5] : il s'agit d'un algorithme de calcul asynchrone d'un snapshot par tous les processus d'un système réparti : le

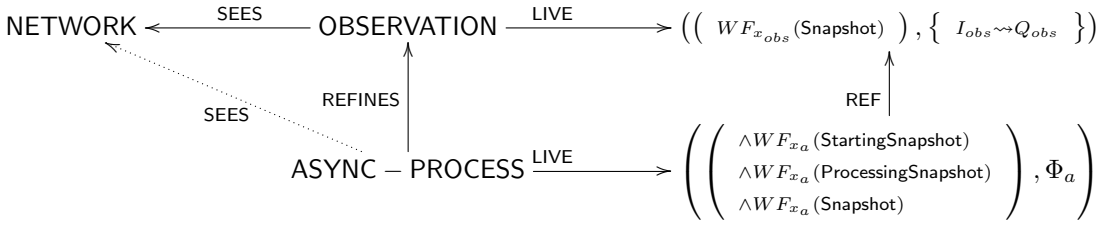


FIGURE 7.7 – Snapshot asynchrone

snapshot est construit progressivement à l'aide d'envois de messages. Les sous-sections suivantes décrivent la dérivation d'algorithmes de snapshot à partir du modèle ASYNC-PROCESS : une version locale de l'algorithme de Lai et Yang et l'algorithme de Chandy et Lamport [3, 5], qui diffèrent de l'algorithme de Lai et Yang par l'utilisation de canaux de communication ordonnés de type FIFO.

7.3.5.2 Algorithme de Lai et Yang local (LAI-PROCESS)

Cette machine LAI-PROCESS est un raffinement de la machine ASYNC-PROCESS précédente. Elle localise cette dernier : dans la machine ASYNC-PROCESS, nous marquions les messages *post-shot* de manière globale, à l'aide d'une variable $mark\_m$; dans la machine LAI-PROCESS, nous remplaçons la variable $mark\_m$ par une variable $mark$, par l'intermédiaire de laquelle chaque processus marque les messages *post-shot* qu'il envoie dans un canal adjacent ($inv1$). Cette nouvelle variable $mark$ est initialisée à l'aide de l'ensemble vide.

INITIALISATION $\widehat{=}$
BEGIN
$\ldots$
$\ominus$: $mark\_m := \varnothing$
$\oplus$: $mark := \varnothing$
END

Nous notons x_l l'ensemble des variables de cette machine LAI-PROCESS.

Un invariant que nous notons $I_l(x_l)$ caractérisent les variables x_l et est défini comme suit :

— $inv1$ permet de typer la variable $mark$, qui associe à un canal son contenu (un sous-ensemble de l'ensemble de message M) :

$$inv1 \ : \ mark \in C \nrightarrow \mathbb{P}(M)$$

— $inv2$, $inv3$, $inv4$, $inv5$ et $inv6$ sont des invariants de collage permettent d'établir les relations entre cette nouvelle variable $mark$ et la variable abstraite $mark\_m$:

$$\begin{array}{ll} inv2 & : \ \forall m, d \cdot d \in C \wedge m \in mark\_m \wedge m \in chan(d) \Rightarrow d \in dom(mark) \\ inv3 & : \ \forall d, m \cdot d \in C \wedge d \in dom(mark) \wedge m \in mark(d) \Rightarrow m \in mark\_m \\ inv4 & : \ \forall m, d \cdot d \in C \wedge m \in mark\_m \wedge m \in chan(d) \Rightarrow m \in mark(d) \\ inv5 & : \ \forall c, m \cdot c \in dom(mark) \wedge m \in chan(c) \wedge m \notin mark(c) \Rightarrow m \notin mark\_m \\ inv6 & : \ \forall c, m \cdot c \in C \wedge c \notin dom(mark) \wedge m \in chan(c) \Rightarrow m \notin mark\_m \end{array}$$

— $inv2$ et $inv4$ expriment que si un message m circule dans un canal d et que ce message m est marqué au niveau abstrait, alors il est marqué au niveau concret localement, au niveau du canal d.

— $inv3$ exprime que si un canal d contient un message m marqué au niveau concret, alors ce dernier est aussi marqué au niveau abstrait.

— $inv5$ exprime que si un message m non-marqué au niveau concret circule dans un canal d par lequel ont déjà circulé des messages marqués, alors le message m n'est pas marqué au niveau abstrait.

— $inv6$ exprime que si un message m circule dans un canal c par lequel des messages non-marqué n'ont pas encore transité, alors le message m n'est pas un message marqué au niveau abstrait.

— les autres invariants expriment des propriétés de sûreté :

$$\begin{array}{l} inv7 \ : \ dom(mark) = send\_mark \\ inv8 \ : \ \forall d \cdot d \in C \wedge d \in dom(mark) \Rightarrow mark(d) \subseteq chan(d) \\ inv9 \ : \ \forall m1, m2, a, b \cdot a \in C \wedge b \in C \wedge m1 \in chan(a) \wedge m2 \in chan(b) \wedge a \neq b \Rightarrow m1 \neq m2 \end{array}$$

— $inv7$ exprime que chaque canal qui contient des messages marqués contient au moins un message marqué (*post-shot*) envoyé par un processus adjacent au canal à un de ses voisins.

— $inv8$ exprime que l'ensemble des messages marqués transitant par un canal d est un sous-ensemble de l'ensemble des messages contenus dans le canal d.

— $inv9$ exprime que si un message $m1$ circule dans un canal a et qu'un message $m2$ circule dans un canal b différent, alors les messages $m1$ et $m2$ sont différents.

Cette machine LAI-PROCESS introduit les raffinements des événements de la machine ASYNC-PROCESS précédente :

— SendingBeforeCut, SendingAfterCut.

— ReceivingPreCutMessages, ReceivingPostCutMessages, ReceivingBeforeCut.

— InternalMessage et InternalLocal.

— StartingSnapshot, ProgressingSnapshot, Snapshot.

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
$send\_mark$	$mark$

L'invariant de collage I_1 nous permet de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de ASYNC-PROCESS en LAI-PROCESS, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par LAI-PROCESS, ainsi que pour justifier la préservation des propriétés Φ_a lors du raffinement.

Nous avons pu voir précédemment que l'objectif dans cette machine LAI-PROCESS est de réduire à l'aide d'une ou de plusieurs itérations d'un ou de plusieurs événements clés, tels que StartingSnapshot et ProgressingSnapshot, le nombre de processus n'ayant pas encore effectué leurs snapshots locaux. Une liste Φ_l des propriétés de vivacité, détaillée en annexe[9], nous permet de construire la machine LAI-PROCESS. Nous modifions ainsi les événements StartingSnapshot, ProgressingSnapshot et Snapshot pour prendre en compte la variable $mark$:

— Événement StartingSnapshot : lors de l'initialisation du snapshot, le processus *initiator* se prépare à marquer les messages *post-shot* qu'il va envoyer via ses canaux sortants d ($grd6$, $grd7$, $act5$).

EVENT StartingSnapshot
REFINES StartingSnapshot $\widehat{=}$
  ANY
    ...
    $\oplus$ nm
  WHERE
    ...
    $\oplus$ $grd6$: $nm \in C \nrightarrow \mathbb{P}(M)$
    $\oplus$ $grd7$: $nm = \{d \mapsto \varnothing | d \in nsm\}$
  THEN
    ...
    $\oplus$ $act5$: $mark := nm$
  END

— Événement ProgressingSnapshot : un processus i ayant reçu pour la première fois un message *post-shot* d'un de ses voisins, se prépare à marquer les messages *post-shot* qu'il va envoyer via ses canaux sortants d ($grd11$, $grd12$, $act5$).

9. `http://andriami.bruno.free.fr/andriami/main/short/annexe.pdf`

EVENT ProgressingSnapshot
REFINES ProgressingSnapshot $\widehat{=}$
  ANY
    ...
    $\oplus$ nm
  WHERE
    ...
    $grd11$: $nm \in C \nrightarrow \mathbb{P}(M)$
    $grd12$: $nm = mark \cup \{d \mapsto \varnothing | d \in C \wedge prj1(d) = i\}$
  THEN
    ...
    $\oplus$ $act5$: $mark := nm$
  END

— Événement Snapshot : tous les processus ont reçus des messages marqués et en ont envoyés ($grd2$) ; ils ont aussi calculé leurs snapshots locaux ; et les messages circulant dans chaque canal p ne sont plus que des messages *post-shot* ($grd3$). La valeur de la coupure *pcut* calculée durant les événements StartingSnapshot et ProgressingSnapshot devient alors la valeur de la variable *cut* modélisant le snapshot.

EVENT Snapshot
REFINES Snapshot $\widehat{=}$
  WHEN
    ...
    $\ominus$ $grd3$: $\forall p \cdot p \in C \wedge p \in send\_mark \Rightarrow chan(p) \subseteq mark\_m$
    $\oplus$ $grd3$: $\forall p \cdot p \in C \wedge p \in send\_mark \Rightarrow chan(p) \subseteq mark(p)$
  THEN
    ...
  END

Les occurences de la variable abstraite $mark\_m$ sont remplacés par la variable concrète $mark$ dans les événements d'envoi et de réception de messages :

— Événement SendingAfterCut : chaque message m envoyé par un processus (ayant effectué son snapshot) via un canal c est marqué comme étant un message *post-shot* transitant par ce canal c ($act6$).

EVENT SendingAfterCut
REFINES Sending $\widehat{=}$
  ANY
    ...
  WHERE
    ...
  THEN
    ...
    $\ominus$ $act6$: $mark\_m := mark\_m \cup \{m\}$
    $\oplus$ $act6$: $mark(c) := mark(c) \cup \{m\}$
  END

— Événements ReceivingPreCutMessages et ReceivingBeforeCut : un message m ayant transité par un canal c n'est pas considéré comme un message *post-shot* s'il ne fait pas partie des messages marqués du canal c ($grd9$).

EVENT ReceivingPreCutMessages
REFINES Receiving $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd9$: $m \notin mark\_m$
    $\oplus$ $grd9$: $c \notin send\_mark \wedge m \notin mark(c)$
  THEN
    ...
  END

EVENT ReceivingBeforeCut
REFINES Receiving $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd9$: $m \notin mark\_m$
    $\oplus$ $grd9$: $c \notin send\_mark \wedge m \notin mark(c)$
  THEN
    ...
  END

— Événement ReceivingPostCutMessages : un message m ayant transité par un canal c est un message *post-shot* s'il fait partie des messages marqués du canal c ($grd9$). Il est retiré, par le processus qui le reçoit, de la liste des messages marqués circulant dans le canal c ($act6$).

EVENT ReceivingPostCutMessages
REFINES Receiving $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus$ $grd9$: $m \in mark\_m$
    $\oplus$ $grd9$: $m \in mark(c)$
  THEN
    ...
    $\ominus$ $act6$: $mark\_m := mark\_m \setminus \{m\}$
    $\oplus$ $act6$: $mark(c) := mark(c) \setminus \{m\}$
  END

Cette machine LAI-PROCESS décrit une version locale de l'algorithme de snapshot de Lai et Yang, qui se présente comme suit :

Algorithm 7 Algorithme de Lai et Yang

var $taken_p$ boolean **init** false ;

Init-LY$_p$: {Initialisation de l'algorithme par au moins un processus p tel que $taken_p = false$}
1: **record**(state(p)) ;
2: $taken_p := true$;

Send-LY$_p$: {Envoi d'un message $mess$ par un processus p}
1: envoyer $\langle mess, taken_p \rangle$

Receive-LY$_p$: {Réception d'un message $\langle mess, c \rangle$ par un processus p}
1: **if** $c \wedge (\neg taken_p)$ **then**
2: **record**(state(p)) ;
3: $taken_p := true$;
4: **end if**

1. Nous supposons que chaque message $mess$ envoyé par chaque processus p du système est couplé à un booléen, qui est la valeur de la variable locale $taken_p$ (**Send-LY$_p$**) : si la valeur du booléen est $false$, cela signifie que le message $mess$ a été envoyé avant le snapshot, il s'agit d'un message *pre-shot*, sinon, il s'agit d'un message *post-shot*.
2. Nous supposons qu'au moins un processus p n'ayant pas encore enregistré son état local initie le *snapshot* (**Init-LY$_p$**) :
   — Le processus p initiateur enregistre son état local.
   — Il indique qu'il a enregistré son état local en mettant sa variable locale $taken_p$ à $true$.
   — Le processus p commence ensuite à enregistrer tous les messages *pre-shot* entrants.
3. A la réception d'un message $\langle mess, c \rangle$ (**Receive-LY$_p$**), un processus p vérifie si la valeur de c est true. Dans ce cas, si p n'a pas encore enregistré son état local ($\neg taken_p$), les mêmes étapes algorithmiques que celles décrites dans le point **2** sont exécutées par le processus p.

Deux événements modélisent principalement ces étapes de l'algorithme de Lai et Yang ; il s'agit de StartingSnapshot et ProgressingSnapshot :
— La phase **Init-LY$_p$** est modélisée par l'événement StartingSnapshot :
  — Nous avons un processus initiateur p appelé *initiator*.
  — Le fait que p n'a pas encore effectué de snapshot local est exprimé par la garde $grd2$.
  — L'action $act4$ exprime l'enregistrement par p de son état local **record**($state(p)$).
  — L'action $act3$ exprime le snapshot local de p ($taken_p = true$).

— L'action $act2$ exprime le début de l'enregistrement par p des messages m pre-shot ($\langle m, 0\rangle$) arrivant par ses canaux entrants.

— La phase **Receive-LY**$_p$ est modélisée par l'événement ProgressingSnapshot :
  — Nous avons un processus p qui a reçu pour la première fois un message m couplé à un marqueur 1 ($\langle m, 1\rangle$) d'un de ses voisins et qui n'a pas encore effectué de snapshot local ($\neg taken_p$) ($grd8$, $grd7$, $grd6$).
  — L'action $act2$ exprime l'enregistrement par p de son état local **record**($state(p)$).
  — L'action $act1$ exprime le snapshot local de p ($taken_p = true$).
  — L'action $act3$ exprime le début de l'enregistrement par p des messages m pre-shot ($\langle m, 0\rangle$) arrivant par ses canaux entrants.

— La phase **Send-LY**$_p$ est à la fois modélisée par StartingSnapshot et ProgressingSnapshot :
  — Les actions $act1$, $act5$ de StartingSnapshot et $act4$, $act5$ de ProgressingSnapshot modélisent que tous les messages m envoyés après le snapshot local de p seront couplés à un marqueur 1 ($\langle m, 1\rangle$).

Nous pouvons résumer ce raffinement par le diagramme 7.8 suivant :

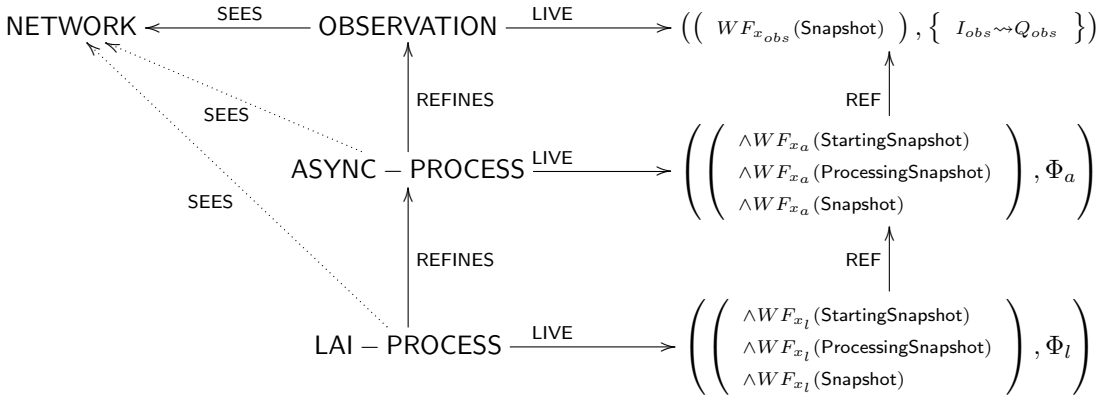


FIGURE 7.8 – Algorithme de Lai et Yang : raffinement concret

Nous avons présenté dans cette sous-section un modèle concret de l'algorithme de snapshot de Lai et Yang [5]. Les sous-sections suivantes décrivent la dérivation d'autres algorithmes de snapshot à partir du modèle ASYNC-PROCESS : nous dérivons, à partir de l'algorithme de Lai et Yang, l'algorithme de Chandy et Lamport [3].

7.3.5.3 Algorithme de Chandy et Lamport abstrait (FIFO-PROCESS)

Nous introduisons ici un modèle raffinant le modèle ASYNC-PROCESS vu précédemment. Notre objectif est de montrer qu'il est possible de dériver à partir d'un modèle de l'algorithme de snapshot de Lai et Yang [5], un modèle de l'algorithme de Chandy et Lamport [3], qui est basé sur des canaux de communication FIFO préservant l'ordre des messages envoyés et reçus par les processus, ainsi qu'un message additionnel appelé *marqueur*.

Pour modéliser cet algorithme, nous commençons par définir un contexte FIFO-NETWORK étendant le contexte NETWORK vu au niveau abstrait. Ce contexte FIFO-NETWORK contient des constantes $ch0$ et $ind0$, ainsi que des propriétés permettant d'attribuer aux canaux de communication des propriétés FIFO :

CONTEXT FIFO − NETWORK
EXTENDS NETWORK
CONSTANTS
$ch0, ind0$
AXIOMS
$axm1$: $ch0 \in C \to (\mathbb{N} \nrightarrow M)$
$axm2$: $\forall d \cdot d \in C \Rightarrow ch0(d) = \varnothing$
$axm3$: $ind0 \in C \to \mathbb{N}$
$axm4$: $\forall d \cdot d \in C \Rightarrow ind0(d) = 0$
$axm5$: $\forall i1, i2 \cdot i1 \in \mathbb{N} \wedge i2 \in \mathbb{N} \wedge i1 \leq i2 \wedge i1 \geq i2 \Rightarrow i1 = i2$
$axm6$: $\forall e, f \cdot f \in \mathbb{N} \nrightarrow M \wedge finite(f) \wedge e \in f \wedge f \neq \varnothing \Rightarrow card(f \setminus \{e\}) = card(f) - 1$
$axm7$: $\forall e, f \cdot f \in \mathbb{N} \nrightarrow M \wedge finite(f) \wedge e \in f \wedge f \setminus \{e\} \neq \varnothing \Rightarrow card(f) > 1$
END

— $ch0$ associe à chaque canal un ensemble de messages numérotés de 0 à n ($axm1$). Nous donnons à cette constante la valeur vide ($\varnothing$), parce que nous l'utiliserons pour initialiser les contenus des canaux.
— $ind0$ associe à chaque canal d l'indice (numéro) courant du premier message circulant dans ce canal ($axm3$). Nous donnons à cet indice la valeur 0 ($axm4$).
— $axm5$ est un théorème exprimant que si un entier naturel $i1$ est plus grand ou égal à un entier naturel $i2$ et $i2$ est plus grand que ou égal à $i1$, alors $i1$ est égal à $i2$.
— $axm6$ est un théorème exprimant que si un ensemble f de messages numérotés est fini, alors le cardinal de l'ensemble constitué par f duquel on a retiré un élément e, est égal au cardinal de f auquel on a soustrait 1.
— $axm7$ exprime que si un ensemble f de messages numérotés est fini, et qu'un élément e est retiré de f et que l'ensemble constitué par f duquel on a retiré un élément e est non-vide, alors cela signifie que le cardinal de f est supérieur à 1.

Ce contexte FIFO-NETWORK est utilisé par une machine FIFO-PROCESS qui modélise l'algorithme de snapshot de Chandy et Lamport. Cette machine introduit de nouvelles variables :

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus$: $chan := c0$
$\ominus$: $mark\_m := \varnothing$
$\oplus$: $ch := ch0$
$\oplus$: $snd\_index := ind0$
$\oplus$: $m\_index := \varnothing$
$\oplus$: $rcv\_index := ind0$
END

— ch associe chaque canal à un contenu composé de messages numérotés ($inv1$). Les messages sont numérotés car nous modélisons ici des files FIFO : les numéros nous permettront d'ordonner les messages circulant dans un canal. La variable ch est initialisée à l'aide de la constante $ch0$.
— rcv_index associe chaque canal à un indice (numéro) courant de réception ($inv2$) : il s'agit du numéro du dernier message reçu par le processus dont le canal constitue un canal entrant. Elle est initialisée à l'aide de la constante $ind0$.
— snd_index associe chaque canal à un indice (numéro) courant d'envoi ($inv3$) : il s'agit du numéro du dernier message envoyé par le processus dont le canal constitue un canal sortant. Elle est aussi initialisée à l'aide de la constante $ind0$.
— m_index est une fonction partielle associant chaque canal à un entier ($inv4$) : il s'agit de l'indice (numéro ou rang) auquel un message *marqueur* est envoyé par le processus dont le canal constitue un canal sortant. A l'état initial, aucun message *marqueur* n'a été envoyé : la variable m_index est donc initialisée à l'aide de l'ensemble vide ($\varnothing$).

Nous notons x_f l'ensemble des variables de cette machine FIFO-NETWORK. Un invariant noté $I_f(x_f)$ contraint ces variables :
— Les propriétés de $inv1$ à $inv4$ permettent de typer les nouvelles variables introduites dans cette machine FIFO-NETWORK :

$inv1$: $ch \in C \to (\mathbb{N} \nrightarrow M)$
$inv2$: $rcv\_index \in C \to \mathbb{N}$
$inv3$: $snd\_index \in C \to \mathbb{N}$
$inv4$: $m\_index \in C \nrightarrow \mathbb{N}$

— Nous supprimons aussi dans cette machine des variables abstraites *mark_m*, *chan*. Les propriétés suivantes sont des invariants de collage et nous permettent de supprimer la variable *mark_m* et de la remplacer par d'autres, notamment par une combinaison des variables *m_index*, *rcv_index*, *snd_index* et *send_mark* ; ces invariants nous permettent aussi de remplacer la variable *chan* par *ch*.

$$
\begin{array}{l}
inv5 \ : \ mark\_m \subseteq send \\
inv6 \ : \ \forall d \cdot \left(\ d \in C \land d \in dom(m\_index) \land rcv\_index(d) \geq m\_index(d) \ \right) \Rightarrow (\forall m \cdot m \in ran(ch(d)) \Rightarrow m \in mark\_m) \\
inv7 \ : \ \forall m, d, i \cdot \left(\ d \in send\_mark \land i < m\_index(d) \land i \mapsto m \in ch(d) \ \right) \Rightarrow m \notin mark\_m \\
inv8 \ : \ \forall m, d \cdot \left(\ d \in C \land d \notin send\_mark \land m \in ran(ch(d)) \ \right) \Rightarrow m \notin mark\_m \\
inv9 \ : \ \forall d \cdot d \in C \Rightarrow ran(ch(d)) = chan(d)
\end{array}
$$

— *inv*5 exprime que chaque message marqué (post-shot) a été envoyé par un processus.
— *inv*6 exprime que pour tout canal d, si le processus, que nous notons p (pour plus de clarté), qui a d en tant que canal sortant a envoyé un message marqueur et que le processus, que nous notons q (pour plus de clarté), qui a d en tant que canal entrant a reçu de p des messages avec des indices (numéro) supérieurs au numéro du message *marqueur* envoyé par p, cela signifie que tous les messages circulant dans le canal d (menant de p à q) sont tous des messages marqués (*post-shot*).
— *inv*7 exprime pour tout canal d, ayant un sens que nous noterons d'un processus p à un processus q (pour plus de clarté), que si un message m ayant un numéro inférieur à l'indice d'un message *marqueur* envoyé par le processus p circule dans le canal d, alors le message m est un message *pré-shot*.
— *inv*8 exprime pour tout canal d, ayant un sens que nous noterons d'un processus p à un processus q (pour plus de clarté), que si ce canal d n'a pas encore contenu de message *pré-shot* et qu'un message m circule dans ce canal, alors le message m est un message *pré-shot*.
— *inv*9 exprime que les messages contenus dans un canal d au niveau abstrait se retrouvent au niveau concret, mais numérotés et ordonnés.
— Les propriétés suivantes expriment des propriétés de sûreté :

$$
\begin{array}{l}
inv10 \ : \ send\_mark = dom(m\_index) \\
inv11 \ : \ \forall c \cdot c \in C \Rightarrow snd\_index(c) \geq rcv\_index(c) \\
inv12 \ : \ \forall c \cdot \left(\ c \in C \land snd\_index(c) = rcv\_index(c) \ \right) \Rightarrow chan(c) = \varnothing \\
inv13 \ : \ \forall c \cdot c \in send\_mark \Rightarrow snd\_index(c) \geq m\_index(c) \\
inv14 \ : \ dom(pcut) = dom(dom(m\_index)) \\
inv15 \ : \ \forall c \cdot \left(\ c \in C \land ch(c) \neq \varnothing \ \right) \Rightarrow dom(ch(c)) = rcv\_index(c) \mathinner{..} snd\_index(c) - 1 \\
inv16 \ : \ \forall c \cdot \left(\ c \in C \land ch(c) = \varnothing \ \right) \Rightarrow snd\_index(c) = rcv\_index(c) \\
inv17 \ : \ \forall d \cdot d \in C \Rightarrow ran(ch(d)) \subseteq send \\
inv18 \ : \ \forall d \cdot d \in C \Rightarrow snd\_index(d) \notin dom(ch(d)) \\
inv19 \ : \ \forall d, i1, i2, m \cdot \left(\ d \in C \land i1 \mapsto m \in ch(d) \land i2 \mapsto m \in ch(d) \ \right) \Rightarrow i1 = i2 \\
inv20 \ : \ \forall d \cdot \left(\ d \in C \land prj1(d) \in dom(pcut) \ \right) \Rightarrow snd\_index(d) \geq m\_index(d) \\
inv21 \ : \ \forall m, d, i \cdot \left(\ d \in send\_mark \land i \geq m\_index(d) \land i \mapsto m \in ch(d) \ \right) \Rightarrow m \in mark\_m \\
inv22 \ : \ \forall m1, m2, a, b \cdot \left(\ a \in C \land b \in C \land m1 \in chan(a) \land m2 \in chan(b) \land a \neq b \ \right) \Rightarrow m1 \neq m2 \\
inv23 \ : \ \forall d, i1, m \cdot \left(\ d \in C \land i1 \mapsto m \in ch(d) \ \right) \Rightarrow i1 \geq rcv\_index(d) \\
inv24 \ : \ \forall d, i1, m \cdot \left(\begin{array}{l} d \in C \land i1 \mapsto m \in ch(d) \\ \land i1 \neq rcv\_index(d) \end{array} \right) \Rightarrow i1 > rcv\_index(d) \\
inv25 \ : \ \forall d \cdot \left(\ d \in C \land ch(d) \neq \varnothing \ \right) \Rightarrow snd\_index(d) > rcv\_index(d) \\
inv26 \ : \ \forall d, i1 \cdot \left(\ d \in C \land i1 \in dom(ch(d)) \ \right) \Rightarrow i1 < snd\_index(d) \\
inv27 \ : \ \forall d \cdot d \in C \Rightarrow finite(ch(d)) \\
inv28 \ : \ \forall d \cdot d \in C \Rightarrow snd\_index(d) = rcv\_index(d) + card(ch(d)) \\
inv29 \ : \ \forall p \cdot p \notin dom(pcut) \Rightarrow (\forall k \cdot p \mapsto k \in C \Rightarrow p \mapsto k \notin dom(m\_index)) \\
inv30 \ : \ \forall d, i1, i2, m1, m2 \cdot \left(\begin{array}{l} \land d \in C \\ \land m1 \neq m2 \\ \land i1 \mapsto m1 \in ch(d) \\ \land i2 \mapsto m2 \in ch(d) \end{array} \right) \Rightarrow i1 \neq i2
\end{array}
$$

— *inv*10 exprime que chaque canal par lequel un processus adjacent a envoyé un message marqué est maintenant associé à l'indice (numéro) de ce message *marqueur*
— *inv*11 exprime que pour un canal c, l'indice courant du dernier message envoyé est supérieur ou égal à l'indice courant du dernier message reçu via ce canal c. Cela veut signifie que la numérotation des messages contenus dans le canal c commence par l'indice courant du dernier message via le canal c et se termine par l'indice courant du dernier message envoyé moins 1

($inv15$, $inv18$).
- $inv12$ exprime que pour un canal c, si l'indice courant du dernier message envoyé est égal à l'indice courant du dernier message reçu via ce canal c, alors le canal c est vide.
- $inv13$ exprime que si un canal c a contenu un message marqueur, l'indice courant du dernier message envoyé ayant transité par le canal c est supérieur ou égal à l'indice du message marqueur ayant circulé dans le canal c.
- $inv14$ exprime que les processus ayant effectué leurs snapshots locaux sont ceux qui se trouvent à l'entrée des canaux par lesquels des messages marqueurs ont transité.
- $inv16$ exprime que si un canal c est vide, alors cela signifie que l'indice courant du dernier message envoyé est égal à l'indice courant du dernier message reçu via ce canal c.
- $inv17$ exprime que le contenu d'un canal d est composé de message ayant été envoyés par des processus.
- $inv19$ exprime exprime qu'un message m circulant dans un canal d ne peut avoir qu'un seul indice (numéro).
- $inv20$ exprime que si un processus dont un canal d est un canal sortant a effectué son snapshot local, alors cela signifie que les numéros des messages que le processus envoie via le canal d sont supérieurs ou égaux à l'indice (numéro) du message marqueur envoyé par ce même processus.
- $inv21$ exprime que exprime que si un message m circulant dans un canal d, contenant un message marqueur, a un indice supérieur à l'indice du message marqueur, alors le message m est un message post-shot (marqué).
- $inv22$ exprime que si des messages $m1$ et $m2$ circulent respectivement dans des canaux a et b différents, alors $m1$ est différent de $m2$.
- $inv23$ et $inv24$ expriment que chaque message circulant dans un canal d a un indice qui est au minimum l'indice du dernier message reçu par un processus via le canal d.
- $inv25$ exprime que pour un canal d non-vide, l'indice courant du dernier message envoyé ayant transité par le canal c est supérieur à l'indice du dernier message reçu ayant circulé dans le canal d.
- $inv26$ exprime que pour un canal d, l'indice $i1$ d'un message circulant dans le canal d est à l'indice du dernier message envoyé ayant circulé dans le canal d.
- $inv27$ exprime que le contenu de tout canal d du système distribué est fini.
- $inv28$ exprime que que pour tout canal d l'indice courant du dernier message envoyé via le canal d est égal à la somme de l'indice courant du dernier message via le canal d et du nombre de messages contenu dans le canal d.
- $inv29$ exprime que si un processus p n'a pas encore effectué de snapshot local, alors cela signifie qu'aucun des canaux sortants de p n'a acheminé de message marqueur.
- $inv30$ exprime que si deux messages $m1$ et $m2$ différents circulent dans un même canal d, alors ces deux messages ont des indices $i1$ et $i2$ différents.

Cette machine FIFO-PROCESS introduit les raffinements des événements de la machine ASYNC-PROCESS précédente :
- SendingBeforeCut, SendingAfterCut.
- ReceivingPreCutMessages, ReceivingPostCutMessages, ReceivingBeforeCut.
- InternalMessage et InternalLocal.
- StartingSnapshot, ProgressingSnapshot, Snapshot.

Les événements StartingSnapshot, ProgressingSnapshot permettent en plusieurs itérations de faire diminuer le nombre de processus n'ayant pas encore calculé leurs snapshots locaux.

Une liste Φ_c des propriétés de vivacité, détaillée en annexe [10], nous permet de construire la machine FIFO-PROCESS. Les événements modélisant le calcul du snapshot sont modifiés par raffinement pour prendre en compte les canaux devenus des files FIFO :
- Événement StartingSnapshot : le processus *initiator* envoie maintenant par tous ses canaux sortants des messages marqueurs étiquetés par les indices (numéros) d'envoi courants pour chaque canal d ($grd6$, $grd7$, $act5$).

10. `http://andriami.bruno.free.fr/andriami/main/short/annexe.pdf`

EVENT StartingSnapshot
REFINES StartingSnapshot $\widehat{=}$
  ANY
    ...
    $\oplus$ nmi
  WHERE
    ...
    $\oplus$ $grd6$: $nmi \in C \pfun \mathbb{N}$
    $\oplus$ $grd7$: $nmi = (\{d | d \in C \land prj1(d) = initiator\}) \rhd snd\_index$
  THEN
    ...
    $\oplus$ $act5$: $m\_index := nmi$
  END

— Événement ProgressingSnapshot : chaque processus i ayant reçu un message marqueur d'un autre processus ($grd7$) et n'ayant pas encore effectué de snapshot local envoie aussi maintenant ses canaux sortants des messages marqueurs étiquetés par les indices (numéros) d'envoi courants pour chaque canal d ($grd11$, $grd12$, $act5$).

EVENT ProgressingSnapshot
REFINES ProgressingSnapshot $\widehat{=}$
  ANY
    ...
    $\oplus$ nmi
  WHERE
    ...
    $\ominus$ $grd7$: $chan(c) \subseteq mark\_m$
    $\oplus$ $grd7$: $m\_index(c) = rcv\_index(c)$
    ...
    $\oplus$ $grd11$: $nmi \in C \pfun \mathbb{N}$
    $\oplus$ $grd12$: $nmi = m\_index \ovl (\{d | d \in C \land prj1(d) = i\}) \rhd snd\_index$
  THEN
    ...
    $\oplus$ $act5$: $m\_index := nmi$
  END

— L'événement Snapshot est observé lorsque tous les processus ont effectué leurs snapshots locaux ($grd1$), tous les canaux ont fait transité des messages marqueurs ($grd2$) et tous les états des canaux ont été enregistrés : les canaux ne contiennent plus de messages *pré-shot*, seulement des messages *post-shot* ($grd3$). La valeur du calcul intermédiaire *pcut* est alors affectée à la variable modélisant le snapshot global *cut* ($act1$).

EVENT Snapshot
REFINES Snapshot $\widehat{=}$
  WHEN
    $grd1$: $dom(pcut) = P$
    $grd2$: $send\_mark = C$
    $grd3$: $\forall c \cdot c \in C \land c \in send\_mark \Rightarrow rcv\_index(c) \geq m\_index(c)$
  THEN
    $act1$: $cut := pcut$
  END

Les événements d'envoi et de réception de messages par les processus sont aussi modifiés par raffinement :

— Événement SendingBeforeCut : un message m étiquetté par un indice i ($grd8$), qui est l'indice courant des messages envoyés via un canal c connecté à un processus p ($grd6$) n'ayant pas encore effectué son snapshot ($grd9$), est envoyé par ce processus p via le canal c ($act1$, $act2$). L'indice courant des messages envoyés par le processus p est ensuite incrémenté ($act6$).

EVENT SendingBeforeCut
REFINES SendingBeforeCut
ANY
p, c, m, i
WHERE
$grd1$: $p \in P$
$grd2$: $c \in C$
$grd3$: $m \in M$
$grd4$: $i \in \mathbb{N}$
$grd5$: $prj1(c) = p$
$grd6$: $i = snd\_index(c)$
$grd7$: $m \notin send$
$grd8$: $i \mapsto m \notin ch(c)$
$grd9$: $p \notin dom(pcut)$
THEN
$act1$: $send := send \cup \{m\}$
$act2$: $ch(c) := ch(c) \cup \{i \mapsto m\}$
$act3$: $o(p) := o(p) + 1$
$act4$: $h(p) := h(p) \cup \{o(p) + 1 \mapsto c\}$
$act5$: $s := s \cup \{c \mapsto o(p) + 1 \mapsto m\}$
$act6$: $snd\_index(c) := i + 1$
END

— Événement SendingAfterCut : un message m étiquetté par un indice i ($grd8$), qui est l'indice courant des messages envoyés via un canal c connecté à un processus p ($grd6$) ayant effectué son snapshot ($grd9$), est envoyé par ce processus p via le canal c ($act1$, $act2$). L'indice courant des messages envoyés par le processus p est ensuite incrémenté ($act6$).

EVENT SendingAfterCut
REFINES SendingAfterCut
ANY
p, c, m, i
WHERE
$grd1$: $p \in P$
$grd2$: $c \in C$
$grd3$: $m \in M$
$grd4$: $i \in \mathbb{N}$
$grd5$: $prj1(c) = p$
$grd6$: $i = snd\_index(c)$
$grd7$: $m \notin send$
$grd8$: $i \mapsto m \notin ch(c)$
$grd9$: $p \in dom(pcut)$
THEN
$act1$: $send := send \cup \{m\}$
$act2$: $ch(c) := ch(c) \cup \{i \mapsto m\}$
$act3$: $o(p) := o(p) + 1$
$act4$: $h(p) := h(p) \cup \{o(p) + 1 \mapsto c\}$
$act5$: $s := s \cup \{c \mapsto o(p) + 1 \mapsto m\}$
$act6$: $snd\_index(c) := i + 1$
END

— L'événement ReceivingPreCutMessages modélise la réception d'un message m *pre-shot* ($grd6$) par un processus q qui a effectué son snapshot ($grd11$) : un message m étiqueté par un indice j ($grd8$), qui est l'indice courant de réception d'un canal c connecté à q ($grd7$), transite dans ce canal c ($grd8$). Le message passe alors du canal c ($act1$) au processus q ($act4$), l'état du canal c est sauvegardé par le processus q ($act6$: enregistrement du message m), et l'indice courant de réception d'un message circulant dans le canal c est incrémenté ($act7$).

EVENT ReceivingPreCutMessages
REFINES ReceivingPreCutMessages
ANY
q, c, m, i, j
WHERE
$grd1$: $q \in P$
$grd2$: $c \in C$
$grd3$: $m \in M$
$grd4$: $j \in \mathbb{N}$
$grd5$: $prj2(c) = q$
$grd6$: $c \notin send\_mark \vee j < m\_index(c)$
$grd7$: $j = rcv\_index(c)$
$grd8$: $j \mapsto m \in ch(c)$
$grd9$: $i \in \mathbb{N}$
$grd10$: $(c \mapsto i \mapsto m) \in s$
$grd11$: $q \in dom(pcut)$
THEN
$act1$: $ch(c) := ch(c) \setminus \{j \mapsto m\}$
$act2$: $o(q) := o(q) + 1$
$act3$: $h(q) := h(q) \cup \{o(q) + 1 \mapsto c\}$
$act4$: $store(q) := store(q) \cup \{m\}$
$act5$: $r := r \cup \{c \mapsto o(q) + 1 \mapsto m\}$
$act6$: $cstate(c) := cstate(c) \cup \{m\}$
$act7$: $rcv\_index(c) := j + 1$
END

— L'événement ReceivingPostCutMessages modélise la réception d'un message m *post-shot* ($grd6$) par un processus q qui a effectué son snapshot ($grd11$) : un message m étiqueté par un indice j ($grd8$), qui est l'indice courant de réception d'un canal c connecté à q ($grd7$), transite dans ce canal c ($grd8$). Le message passe alors du canal c ($act1$) au processus q ($act4$) et l'indice courant de réception d'un message circulant dans le canal c est incrémenté ($act6$).

EVENT ReceivingPostCutMessages
REFINES ReceivingPostCutMessages
ANY
q, c, m, i, j
WHERE
$grd1$: $q \in P$
$grd2$: $c \in C$
$grd3$: $m \in M$
$grd4$: $j \in \mathbb{N}$
$grd5$: $prj2(c) = q$
$grd6$: $c \in send\_mark \wedge j \geq m\_index(c)$
$grd7$: $j = rcv\_index(c)$
$grd8$: $j \mapsto m \in ch(c)$
$grd9$: $i \in \mathbb{N}$
$grd10$: $(c \mapsto i \mapsto m) \in s$
$grd11$: $q \in dom(pcut)$
THEN
$act1$: $ch(c) := ch(c) \setminus \{j \mapsto m\}$
$act2$: $o(q) := o(q) + 1$
$act3$: $h(q) := h(q) \cup \{o(q) + 1 \mapsto c\}$
$act4$: $store(q) := store(q) \cup \{m\}$
$act5$: $r := r \cup \{c \mapsto o(q) + 1 \mapsto m\}$
$act6$: $rcv\_index(c) := j + 1$
END

— L'événement ReceivingBeforeCut modélise la réception d'un message m *pré-shot* ($grd6$) par un processus q qui n'a pas encore effectué son snapshot ($grd11$) : un message m étiqueté par un indice j ($grd8$), qui est l'indice courant de réception d'un canal c connecté à q ($grd7$), transite dans ce canal c ($grd8$). Le message passe alors du canal c ($act1$) au processus q ($act4$) et l'indice courant de réception d'un message circulant dans le canal c est incrémenté ($act6$).

EVENT ReceivingBeforeCut
REFINES ReceivingBeforeCut
ANY
q, c, m, i, j
WHERE
$grd1 : q \in P$
$grd2 : c \in C$
$grd3 : m \in M$
$grd4 : j \in \mathbb{N}$
$grd5 : prj2(c) = q$
$grd6 : c \notin send\_mark \vee j < m\_index(c)$
$grd7 : j = rcv\_index(c)$
$grd8 : j \mapsto m \in ch(c)$
$grd9 : i \in \mathbb{N}$
$grd10 : (c \mapsto i \mapsto m) \in s$
$grd11 : q \notin dom(pcut)$
THEN
$act1 : ch(c) := ch(c) \setminus \{j \mapsto m\}$
$act2 : o(q) := o(q) + 1$
$act3 : h(q) := h(q) \cup \{o(q) + 1 \mapsto c\}$
$act4 : store(q) := store(q) \cup \{m\}$
$act5 : r := r \cup \{c \mapsto o(q) + 1 \mapsto m\}$
$act6 : rcv\_index(c) := j + 1$
END

L'algorithme de Chandy et Lamport, modélisé par la machine FIFO-PROCESS, est défini comme suit :

Algorithm 8 Algorithme de Chandy et Lamport

var $taken_p$ boolean **init** false ;

Init-CL$_p$: {Initialisation de l'algorithme par au moins un processus p tel que $taken_p = false$}
1: **record**(state(p)) ;
2: $taken_p := true$;
3: **for** $q \in voisins(p)$ **do**
4: envoyer $\langle mkr \rangle$ à q
5: **end for**

Receive-CL$_p$: {Arrivée d'un message $\langle mkr \rangle$ à un processus p}
1: recevoir $\langle mkr \rangle$
2: **if** $\neg taken_p$ **then**
3: **record**(state(p)) ;
4: $taken_p := true$;
5: **for** $q \in voisins(p)$ **do**
6: envoyer $\langle mkr \rangle$ à q
7: **end for**
8: **end if**

Nous remarquons une certaine similarité avec l'algorithme de Lai et Yang vu précédemment. Les différences résident dans le fait qu'un processus signale qu'il a enregistré son état local en envoyant un message de contrôle appelé *marqueur* ($\langle mkr \rangle$) aux autres processus et dans la structure des messages envoyés par les processus : ceux-ci ne sont pas couplés à des booléens, car il est possible de distinguer clairement les messages *pre* et *post-shot*. En effet, le fait que les canaux de communication sont des files FIFO permet aux processus de savoir que les messages *pre-shot* sont les messages entrants reçus avant le message de contrôle ($\langle mkr \rangle$) et les messages *post-shot* sont ceux reçus après le message ($\langle mkr \rangle$).

Deux événements modélisent principalement les étapes de l'algorithme de Chandy et Lamport ; il s'agit de StartingSnapshot et ProgressingSnapshot :

— La phase **Init-CL**$_p$ est modélisée par l'événement StartingSnapshot :
  — Nous avons un processus initiateur p appelé *initiator*.
  — Le fait que p n'a pas encore effectué de snapshot local est exprimé par la garde $grd2$.
  — L'action $act4$ exprime l'enregistrement par p de son état local **record**$(state(p))$.
  — L'action $act3$ exprime le snapshot local de p ($taken_p = true$).
  — L'action $act2$ exprime le début de l'enregistrement par p des messages pre-shot arrivant par ses canaux entrants.
  — Les actions $act1$ et $act5$ expriment que le processus p va envoyer des messages marqueurs par tous ses canaux sortants.
— La phase **Receive-CL**$_p$ est modélisée par l'événement ProgressingSnapshot :
  — Nous avons un processus p qui a reçu pour la première fois un marqueur d'un de ses voisins et qui n'a pas encore effectué de snapshot local ($\neg taken_p$) ($grd8$, $grd7$, $grd6$).
  — L'action $act2$ exprime l'enregistrement par p de son état local **record**$(state(p))$.
  — L'action $act1$ exprime le snapshot local de p ($taken_p = true$).
  — L'action $act3$ exprime le début de l'enregistrement par p des messages pre-shot arrivant par ses canaux entrants.
  — Les actions $act4$ et $act5$ expriment que le processus p va envoyer des messages marqueurs par tous ses canaux sortants.

Nous pouvons résumer ce raffinement par le diagramme 7.9 suivant :

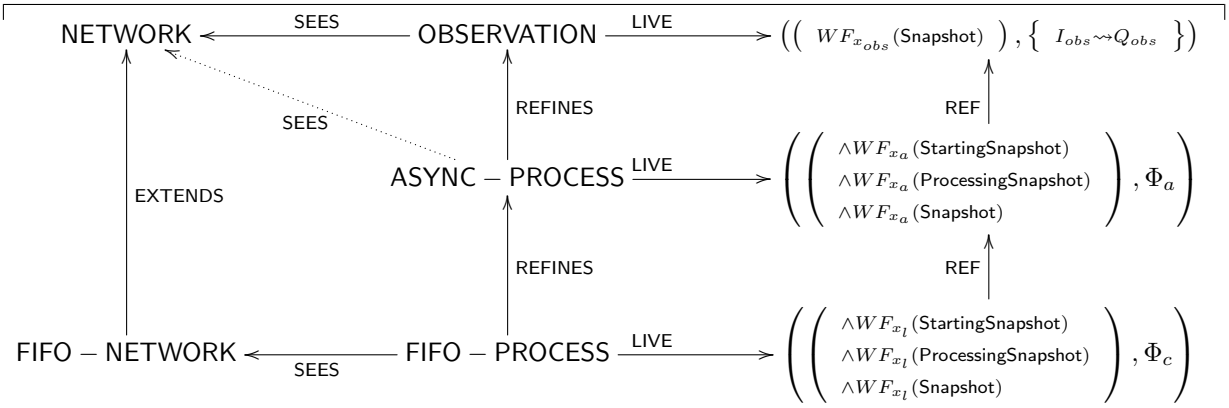


FIGURE 7.9 – Algorithme de Chandy et Lamport

Nous présentons dans la sous-section suivante la machine LOC-FIFO-PROCESS : il s'agit de la localisation de la machine FIFO-PROCESS.

7.3.5.4 Algorithme de Chandy et Lamport local (LOC-FIFO-PROCESS)

La machine LOC-FIFO-PROCESS raffine FIFO-PROCESS : il s'agit d'une simplification de cette dernière. En effet, l'analyse des événements de FIFO-PROCESS nous a permis de constater que des événements *pré* et *post-shot* ayant des actions communes pouvaient être regroupés et partager un même raffinement :

— L'événement Sending est un raffinement des événements SendingAfterCut et SendingBeforeCut. Dans les gardes de ces deux événements, nous avons considéré les cas suivants : $p \in dom(cut)$ et $p \notin dom(cut)$, exprimant respectivement qu'un processus p a effectué ou non son snapshot local. Dans ce raffinement Sending, nous supprimons les gardes $p \in dom(cut)$ ou $p \notin dom(cut)$, puisque le résultat des actions des SendingAfterCut et SendingBeforeCut est le même : il s'agit de faire passer un message d'un processus p ($act1$) à un canal c adjacent ($act2$) et d'incrémenter l'indice du message envoyé via le canal c ($act6$).

EVENT Sending REFINES SendingAfterCut, SendingBeforeCut
ANY
p, c, m, i
WHERE
$grd1 : p \in P$
$grd2 : c \in C$
$grd3 : m \in M$
$grd4 : i \in \mathbb{N}$
$grd5 : prj1(c) = p$
$grd6 : i = snd\_index(c)$
$grd7 : m \notin send$
$grd8 : i \mapsto m \notin ch(c)$
THEN
$act1 : send := send \cup \{m\}$
$act2 : ch(c) := ch(c) \cup \{i \mapsto m\}$
$act3 : o(p) := o(p) + 1$
$act4 : h(p) := h(p) \cup \{o(p) + 1 \mapsto c\}$
$act5 : s := s \cup \{c \mapsto o(p) + 1 \mapsto m\}$
$act6 : snd\_index(c) := i + 1$
END

— Le même raisonnement que précédemment s'applique pour l'événement ReceivingBeforeCutOrAfterCut, raffinement de ReceivingBeforeCut et ReceivingAfterCut. Nous supprimons dans ce raffinement les gardes $q \in dom(cut)$ et $q \notin dom(cut)$, exprimant respectivement qu'un processus q a effectué ou non son snapshot local, lors de la réception d'un message m. Nous pouvons effectuer cette modification parce que le résultat des actions des événements ReceivingBeforeCut et ReceivingAfterCut est le même : il s'agit de faire passer un message d'un canal c ($act1$) à processus q ($act4$) adjacent et d'incrémenter l'indice du message reçu via le canal c ($act6$).

EVENT ReceivingBeforeCutOrAfterCut REFINES ReceivingBeforeCut, ReceivingAfterCut
ANY
q, c, m, i, j
WHERE
$grd1 : q \in P$
$grd2 : c \in C$
$grd3 : m \in M$
$grd4 : j \in \mathbb{N}$
$grd5 : prj2(c) = q$
$grd6 : c \notin send\_mark \vee j < m\_index(c)$
$grd7 : j = rcv\_index(c)$
$grd8 : j \mapsto m \in ch(c)$
$grd9 : i \in \mathbb{N}$
$grd10 : (c \mapsto i \mapsto m) \in s$
THEN
$act1 : ch(c) := ch(c) \setminus \{j \mapsto m\}$
$act2 : o(q) := o(q) + 1$
$act3 : h(q) := h(q) \cup \{o(q) + 1 \mapsto c\}$
$act4 : store(q) := store(q) \cup \{m\}$
$act5 : r := r \cup \{c \mapsto o(q) + 1 \mapsto m\}$
$act6 : rcv\_index(c) := j + 1$
END

Nous avons obtenu un modèle de l'algorithme de snapshot de Chandy et Lamport en raffinant un modèle de l'algorithme de snapshot de Lai et Yang. Les messages échangés par les processus sont simplifiés : les messages ne sont plus étiquettés par des booléens indiquant s'ils sont *pré* ou *post-shot*. Cependant une complexité supplémentaire est rajoutée : les canaux de communications entre les processus préservent l'ordre des messages envoyés et reçus, permettant ainsi une séparation claire entre les messages *pré* et *post-shot*. Lors de ce raffinement, aucun des événements modélisant l'algorithme de snapshot de Chandy et Lamport n'a été modifié. Les propriétés de vivacité caractérisant l'algorithme de snapshot modélisé par le modèle LOC-FIF0-PROCESS sont donc les mêmes que celles satisfaites par le modèle FIFO-PROCESS vues précédemment.

7.4 Conclusion

Nos travaux [1, 2] sur le problème du snapshot nous ont permis la découverte d'une achitecture générique donnant la possibilité de dériver différentes familles d'algorithmes de snapshot. Un modèle

SYSTEM modélise un système réparti de manière abstraite, avec ses processus et leurs activités (calculs, communications, etc.). Ce modèle SYSTEM est ensuite raffiné par un modèle OBSERVATION qui introduit la notion du snapshot se déclenchant en parallèle par rapports aux autres activités du système réparti : un événement abstrait modélise en un coup le snapshot global du système réparti. Ce modèle est le point central de notre développement formel : c'est celui que nous raffinons pour dériver différentes familles d'algorithmes. Nous avons dégagé ensuite deux familles d'algorithmes de snapshot, une première famille d'algorithmes de snapshot synchrones, abstraite par le modèle SYNC-PROCESS et une seconde famille d'algorithmes de snapshot asynchrones, abstraite par le modèle ASYNC-PROCESS. Ces deux modèles décrivent les étapes nécessaires au calcul du snapshot et peuvent être raffinés pour obtenir différents algorithmes : SYNC-PROCESS est un modèle de l'algorithme de snapshot de Morgan [6] ; ASYNC-PROCESS est un modèle de l'algorithme de snapshot de Lai et Yang [5], et nous en dérivons une version locale de cet algorithme (LAI-PROCESS) ainsi que des modèles de l'algorithme de Chandy et Lamport (FIFO-PROCESS et LOC-FIFO-PROCESS). Le raffinement, ainsi que les propriétés de vivacité utilisées pour guider et exprimer ce raffinement, nous permettent ainsi d'établir des relations sémantiques entre les différents algorithmes de snapshot (e.g. les algorithmes de Lai et Yang et Chandy et Lamport). Les distinctions principales entre ces algorithmes consistent en : **(1)** le nombre d'étapes nécessaires au calcul du snapshot (une seule étape de calcul pour les algorithmes synchrones, deux (initialisation et calcul progressif) pour les algorithmes asynchrones) ; **(2)** le traitement des événements et messages pré et post-shots (étiquettage par un booléen/le temps, utilisation de messages marqueurs).

Le tableau suivant 7.1 décrit les statistiques des obligations de preuve et nous donne ainsi une mesure de la complexité du développement, à travers le nombre d'obligations de preuve (PO) déchargées automatiquement ou interactivement (manuellement). Nous ne présentons ici que les statistiques liées à l'utilisation exclusive des prouveurs de l'Atelier B. Les difficultés principales ont été rencontrées notamment lors de l'expression d'un snapshot consistant dans les modèles SYNC-PROCESS et ASYNC-PROCESS : les invariants exprimant cette consistance du snapshot ($inv10$ pour SYNC-PROCESS et $inv16$, $inv17$, $inv18$ pour ASYNC-PROCESS) ont été les points clés du développement et les obligations de preuves générées relatives à ces invariants ont été assez difficiles à décharger. Nous pouvons aussi remarquer que le nombre d'obligations de preuves déchargées manuellement augmente de manière drastique pour le modèle FIFO-PROCESS : cette augmentation est dûe à la transformation des canaux en files FIFO préservant l'ordre des messages, en effet, nous avons eu à démontrer un certain nombre de nouvelles propriétés relatives aux comportements des files FIFO, ainsi qu'aux communications entre les processus qui se retrouvent ainsi ordonnées.

Modèles	Total	Automatiques		Interactives	
NETWORK	6	6	100%	0	0%
FIFO-NETWORK	5	4	80%	1	20%
SYSTEM	55	50	90.91%	5	9.09%
OBSERVATION	41	37	90.24%	4	9.76%
SYNC-PROCESS	55	51	92.73%	4	7.27%
ASYNC-PROCESS	96	66	68.75%	30	31.25%
LAI-PROCESS	85	46	54.12%	39	45.88%
FIFO-PROCESS	229	12	5.24%	217	94.76%
LOC-FIFO-PROCESS	5	4	80%	1	20%
Total	577	276	47.83%	301	51.17%

TABLE 7.1 – Snapshot : Statistiques des POs

Notre développement formel, ainsi que les paradigmes de *correction-par-construction* et *service-as-event*, le raffinement et les propriétés de vivacité guidant le raffinement permettent une compréhension des mécanismes des différents algorithmes de snapshot étudiés, car les étapes de chaque algorithme sont ajoutées et détaillées progressivement.

Bibliographie

[1] M. B. Andriamiarina, D. Méry, and N. K. Singh. Revisiting snapshot algorithms by refinement-based techniques. In H. Shen, Y. Sang, Y. Li, D. Qian, and A. Y. Zomaya, editors, *13th International*

Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2012, Beijing, China, December 14-16, 2012, pages 343–349. IEEE, 2012.

[2] M. B. Andriamiarina, D. Méry, and N. K. Singh. Revisiting Snapshot Algorithms by Refinement-based Techniques (Extended Version). *Computer Science and Information Systems*, 11(1) :251–270, Jan. 2014.

[3] K. M. Chandy and L. Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1) :63–75, 1985.

[4] A. D. Kshemkalyani, M. Raynal, and M. Singhal. An introduction to snapshot algorithms in distributed computing. *Distributed Systems Engineering*, 2(4) :224, 1995.

[5] T.-H. Lai and T. H. Yang. On distributed snapshots. *Inf. Process. Lett.*, 25(3) :153–158, 1987.

[6] C. Morgan. Global and logical time in distributed algorithms. *Inf. Process. Lett.*, 20(4) :189–194, 1985.

[7] G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, New York, NY, USA, 1994.

[8] Z. Yang and T. A. Marsland. Global snapshots for distributed debugging : An overview. Technical Report TR-92-03, Computing Science Department, University of Alberta, 1992.

8

Les systèmes auto-stabilisants

Sommaire

8.1 Introduction

Les systèmes distribués auto-stabilisants ont été introduits par Dijkstra, dans son article fondateur [2]. Nous en tirons une définition du concept d'auto-stabilisation :

Définition 8.1.1. Un système distribué est auto-stabilisant si, à partir d'un état arbitraire, il est garanti que le système converge vers un état légitime en un temps fini. Si le système est dans un état légitime, il demeure dans cet état légitime, sous réserve qu'aucune erreur ne se produise. Un état légitime est un état satisfaisant les spécifications du système distribué.

Dans [2], Dijkstra illustre l'auto-stabilisation à l'aide d'une problématique d'exclusion mutuelle reposant sur un système distribué organisé en anneau : tous les processus composant le système sont reliés entre eux selon une boucle fermée. Un jeton (i.e. un paquet de données) circulant en boucle dans le système, d'un processus à un autre, détermine quel processus se trouve en section critique. Le système est dans un état légitime (correct) s'il y exactement un seul jeton circulant dans le réseau, c'est-à-dire si un seul et unique processus peut entrer dans la section critique. Dijkstra propose dans [2] trois solutions algorithmiques permettant à un système distribué en anneau de converger vers cet état légitime.

En résumé, un système auto-stabilisant est un système capable de récupérer automatiquement d'erreurs (transitoires) [3]. Cette propriété permet par conséquent à un système auto-stabilisant d'être initialisé dans un état arbitraire : fatalement, le système reviendra à un comportement correct [3]. Nous présentons dans ce chapitre l'analyse formelle d'un système distribué auto-stabilisant proposé par Marquezan et al [4] : il s'agit de l'étude d'un système pair à pair (P2P) possédant une propriété d'auto-stabilisation, appelée ici spécifiquement procédure d'« auto-guérison » (*self-healing*).

8.2 Présentation des systèmes étudiés

Nous considérons qu'un système auto-stabilisant $\mathcal{S}$ (voir figure 8.1) est caractérisé par un ensemble d'événements [1], modélisant soit des phases abstraites, soit des actions concrètes, en fonction du niveau d'abstraction. Un système $\mathcal{S}$ est caractérisé par un ensemble d'événements $\mathcal{M} = \mathcal{CL} \cup \mathcal{ST} \cup \mathcal{F}$, où $\mathcal{CL}$, $\mathcal{ST}$ et $\mathcal{F}$ sont des ensembles distincts les uns des autres :

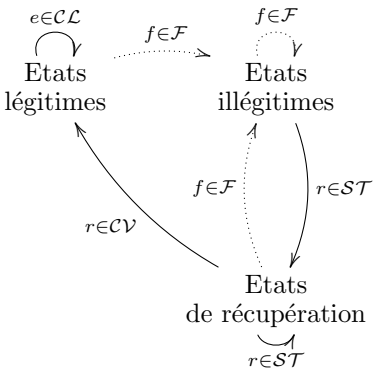


FIGURE 8.1 – Représentation diagrammatique d'un système auto-stabilisant $\mathcal{S}$

— Les événements faisant partie de l'ensemble $\mathcal{CL}$ modélisent les actions pouvant se produire lorsque le système $\mathcal{S}$ se trouve dans un état légitime. Le diagramme 8.1 nous montre que ces événements ne conduisent pas le système $\mathcal{S}$ dans un état d'erreur : il s'agit de la propriété de « fermeture » (closure).
— Les événements faisant partie de l'ensemble $\mathcal{F}$ modélisent les fautes et erreurs (transitoires) possibles pouvant survenir. Nous caractérisons les fautes de manière abstraite : les événement membres

de l'ensemble $\mathcal{F}$ ne font généralement que conduire le système $\mathcal{S}$ dans un état illégitime. Il est possible de développer ces événements de manière plus détaillée, mais la diversité des fautes pouvant se produire nous ont poussés à faire le choix de l'abstraction.

— Les événements faisant partie de l'ensemble $\mathcal{ST}$: la détection de l'entrée du système $\mathcal{S}$ dans un état illégitime permet l'observation de ces événements, qui modélisent les phases de récupération/stabilisation du système. Le système $\mathcal{S}$ effectue un nombre fini d'étapes de stabilisation/récupération et un sous-ensemble $\mathcal{CV}$ de $\mathcal{ST}$ modélise la convergence du système vers un état légitime. Nous nous assurons que cette convergence est toujours possible, à l'aide des événements de ce sous-ensemble $\mathcal{CV}$ conduisent *fatalement* le système $\mathcal{S}$ vers un état légitime. Nous considérons que lorsque le système $\mathcal{S}$ se trouve dans un état légitime, soit que les événements de $\mathcal{ST}$ ne sont pas observables, soit ils préservent la spécification du système $\mathcal{S}$.

Nous utilisons cette représentation, couplée aux paradigmes de *correction-par-construction* et *service-as-event* pour modéliser et analyser formellement les systèmes auto-stabilisants, notamment le système P2P caractérisé par une propriété d'« auto-guérison » (self-healing) et proposé par Marquezan et al [4]. Nous présentons maintenant ce système P2P, ainsi que les procédures mises en places pour garantir l'« auto-guérison ».

Le système P2P que nous étudions ici est une plateforme dans laquelle les pairs (processus) fournissent des services. Nous introduisons maintenant des notions vues dans [4] à propos des pairs et services :

— Services de gestion (Management Services) : services, fonctionnalités fournis par les pairs.
— Instances d'un service de gestion (Instances of a management service) : pairs fournissant un certain type de service.
— Groupe de pairs de gestion (Management Peer group - MPG) : ensemble des pairs instanciant un même service.

Nous nous servirons de ces notions notamment pour expliquer le fonctionnement de la procédure de self-healing dont l'objectif est d'assurer la maintenance et le fonctionnement correct des services proposés par le système. Cette procédure peut se résumer par les étapes suivantes : **(1)** l'identification des pairs entrés dans un état illégitime (entraînant les services qu'ils fournissent dans un état illégitime) et **(2)** la restauration des services et fonctionnalités fournis par ces pairs défaillants. Ces deux étapes (principalement la seconde) sont décomposées selon les phases suivantes :

1. Phase de **self-detection** : cette phase permet de détecter l'entrée d'un service dans un état illégitime et de déterminer les instances défaillantes, dans un état illégitime.
2. Phase de **self-activation** : cette phase est déclenchée après la détection par la précédente phase d'un état illégitime réel d'un service. L'état illégitime du service est évalué afin de déterminer s'il est critique ou non : si le nombre d'instances du service est jugé suffisant, l'état illégitime n'est pas critique et le service retourne dans un état légitime, sinon la phase de **self-configuration** est déclenchée pour guérir/réparer le service.
3. Phase de **self-configuration** : cette phase est activée lorsque l'état illégitime d'un service est critique. Le rôle de cette phase est de déployer de nouvelles instances du service afin de faire revenir ce dernier dans un état légitime.

Les algorithmes suivants détaillent ces phases, ainsi que leur coordination et leur agencement les unes par rapport aux autres :

— L'algorithme 9 introduit le fait que chaque instance d'un service, à chaque instant t envoie un signal (un battement de cœur) à toutes les autres instances du même service, pour montrer qu'elle est encore fonctionnelle. L'absence de réception d'un battement de cœur de la part d'une instance rend cette dernière suspecte (lignes 10 et 11). Pour confirmer la défaillance de l'instance, une des instances fonctionnelles essaie de recontacter celle défaillante (ligne 12). Si l'opération de recontact échoue (ligne 13), le service est déclaré étant dans un état illégitime (lignes 14 et 15) et la phase qui est ensuite activée est celle de **self-activation** (ligne 16). Dans notre développement formel de cette première phase de **self-détection**, nous abstrayons l'envoi des signaux pour détecter les instances suspectes : nous disons juste qu'à un certain moment, un mécanisme que nous ne détaillons pas permet de déterminer les instances devenues suspectes.
— L'algorithme 10 introduit le fait qu'une politique d'évaluation d'un état illégitime et de récupération existe pour chaque service. Cet algorithme 10 décrit le comportement général de la phase de **self-**

Algorithm 9 Self-detection

Require: $PG = \{p_1, p_2, \ldots, p_i\}$ ensemble de pairs formant un groupe
Require: $MS = \{ms_1, ms_2, \ldots, ms_j\}$ ensemble de services de gestion
Require: $SPG = \{PG_1, PG_2, \ldots, PG_k\}$ ensemble de groupe de pairs
Require: $MPG = \{mpg_1, mpg_2, \ldots, mpg_i\}$ où $mpg_l = \{ms_j, PG_k\}$, $ms_j \in MS$ et $PG_k \in SPG$ ensemble de groupe de pairs
Ensure: $i > 0$ où i est l'indice de PG
Ensure: $j > 0$ où j est l'indice de MS
Ensure: $k > 0$ où k est l'indice de SPG
Ensure: $l > 0$ où l est l'indice de MPG
Ensure: $t > 0$ où t est l'interval d'envoi d'un battement de cœur
Ensure: $s > 0$ où s est l'interval d'écoute des battements de cœur des autres instances
Ensure: $r > 0$ où r est l'interval d'attente de la réponse d'un pair suspecté défaillant

```
1: loop
2:    WaitHeartbeatCycle(t)
3:    for all mpg_l ∈ MPG do
4:       num_heartbeats ← 0
5:       retry_contact ← false
6:       num_mpg_instances ← GetNumberPeersInsideMPG(mpg_l)
7:       SendHeartbeatToMPG(mpg_l)
8:       ListenHeartbeats(num_heartbeats, mpg_l, s)
9:       if num_heartbeats < (num_mpg_instances − 1) then
10:         suspicious_peer ← p_i ∈ PG_k from mpg_l without answer
11:         DeclareSuspiciousFailure(suspicious_peer)
12:         RetryContactWithSuspiciousFailedPeer(retry_contact, r, suspicious_peer)
13:         if retry_contact = false then
14:            failed_management_service ← ms_j from mpg_l
15:            DeclareFailedInstance(suspicious_peer, mpg_l)
16:            InformSelfActivationService(failed_management_service,mpg_l)
17:         end if
18:      end if
19:   end for
20: end loop
```

Algorithm 10 Self-activation - boucle de contrôle principale

Ensure: $failed\_mngt\_service$ contient l'identité du service de gestion défaillant

```
1: loop
2:    failed_mngt_service ← ∅
3:    WaitForFailureNotification(failed_mngt_service)
4:    PolicyEvaluationActivation(failed_mngt_service)
5: end loop
```

activation : après avoir reçu de la phase précédente l'identité d'un service dans un état illégitime (ligne 3), une évaluation de cet état illégitime est déclenchée (ligne 4), en accord avec la politique d'évaluation et de récupération liée au service.

— L'algorithme 11 détaille la politique d'évaluation d'un service dans un état illégitime en introduisant le fait que cette politique définit le nombre d'instances minimal requis pour fournir le service, ainsi que le nombre de nouvelles instances du service activables en un coup, tant que le nombre minimal d'instances requis n'a pas été dépassé ou atteint. Si le nombre courant d'instances fonctionnelles d'un service dans un état illégitime est inférieur au nombre minimal requis pour fournir le service, l'état illégitime est critique et un appel à la phase de **self-configuration** est fait, pour déployer de nouvelles instances du service s (ligne 4 à 8).

— L'algorithme 12 détaille la phase de **self-configuration**, principalement le déploiement (activation) de nouvelles instances d'un service dans un état illégitime. Un pair pouvant instancier le service est recherché (ligne 4) : il s'agit d'un pair non défaillant, non suspecté de l'être et n'instanciant pas encore le service en question. Si un tel pair est découvert, une instance du service dans un état illégitime est déployée sur ce pair (lignes 5 et 6).

Algorithm 11 Self-activation - Politique d'évaluation et d'activation

Require: $POLICY = (min\_instances, num\_activations)$
Require: $POLICYSET = \{POLICY_1, POLICY_2, \ldots, POLICY_i\}$ ensemble de politiques
Require: $MS = \{ms_1, ms_2, \ldots, ms_j\}$ ensemble de services de gestion
Require: $MSPOLICY = \{msp_1, msp_2, \ldots, msp_k\}$ où $msp_k = \{ms_j, POLICY_i\}$, $ms_j \in MS$ et $POLICY_i \in POLICYSET$
Ensure: $i > 0$ où i est l'indice de $POLICYSET$
Ensure: $j > 0$ où j est l'indice de MS
Ensure: $k > 0$ où k est l'indice de $MSPOLICY$
Ensure: $failed\_mngt\_service$ contient l'identité du service de gestion défaillant

```
1: loop
2:    num_instance ← GetNumberOfInstances(failed_mngt_service)
3:    (min_instances, num_activations) ← (min_instances, num_activations) = POLICY_i avec POLICY_i ∈
      msp_k et ms_j = failed_mngt_service
4:    if num_instances < min_instances then
5:       for i = 0 to num_activations do
6:          CallSelfConfiguration(failed_mngt_service)
7:       end for
8:       WaitSelfConfigurationAnswer()
9:    end if
10: end loop
```

En résumé, nous remarquons que la procédure de self-healing est décomposée en trois phases séquentielles, qui sont les phases de **self-detection**, **self-activation** et **self-configuration**. Chaque phase est dirigée par une instance fonctionnelle du service dans état illégitime, appelée « token owner » : cette instance coordonne les entrées et sorties du service de chacune des phases. Nous présentons dans la section suivante le développement formel de la procédure de self-healing.

8.3 Développement

8.3.1 Plan du développement

Nous commençons par présenter le plan (voir figure 8.2) que nous avons suivi lors du développement formel du problème d'auto-guérison (« self-healing ») :

— Une abstraction M0 présente la procédure de self-healing : un service passe d'un état légal (correct) à un état illégal (incorrect/fautif) suite à une faute, et une procédure abstraite modélisée par un événement ramène le service dans un état légal.

— Le premier raffinement M1 introduit abstraitement la phase de **self-detection** : l'état illégal d'un service est détecté, permettant l'activation de la procédure de self-healing.

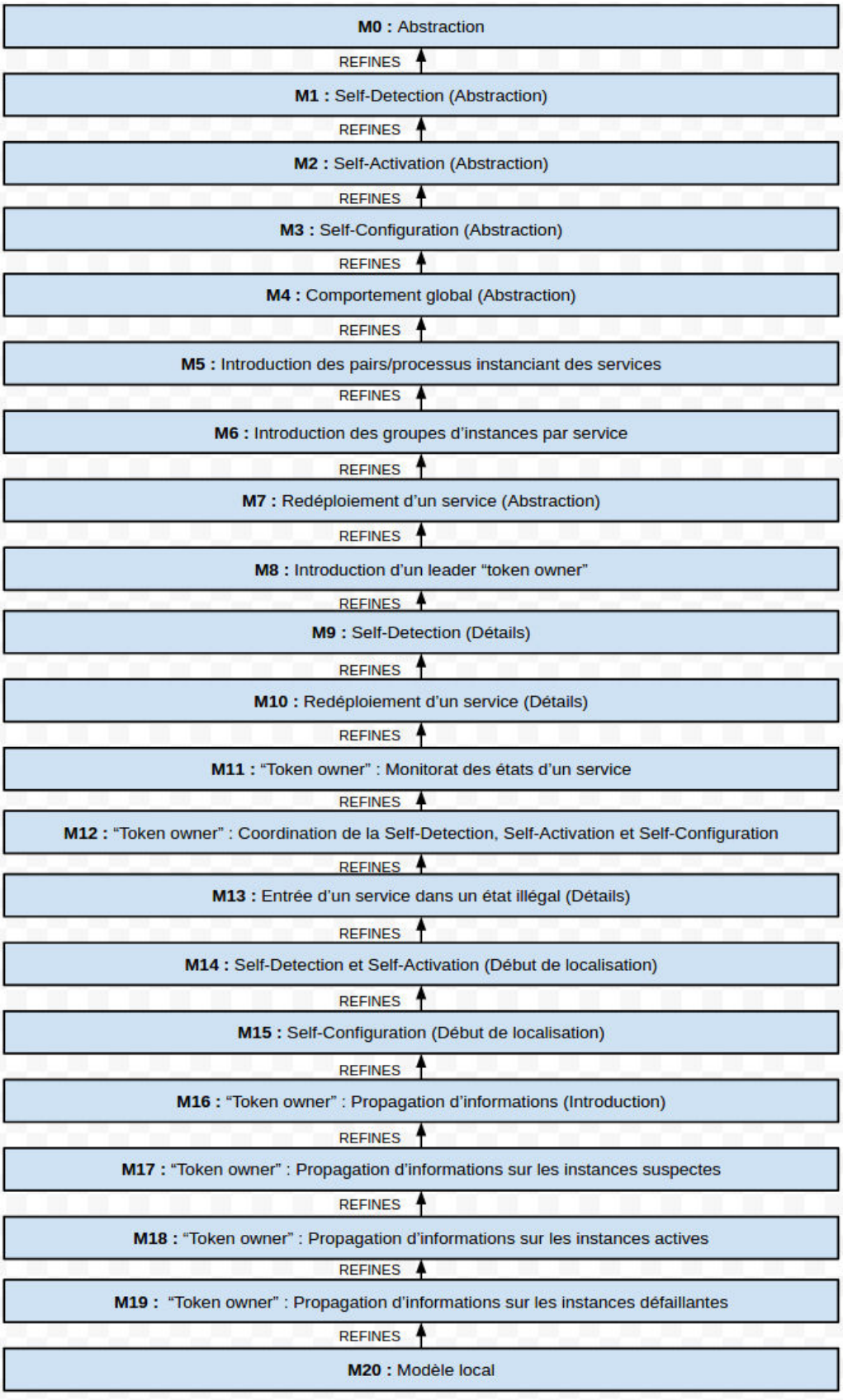


FIGURE 8.2 – Plan de développement du problème de *self-healing*

Algorithm 12 Self-configuration - Procédure de récupération

Ensure: $failed\_mngt\_service$ contient l'identité du service de gestion défaillant
Ensure: $num\_retries$ contient le nombre maximal de tentatives pour trouver un pair disponible

```
1: attempts ← 0
2: deployed ← false
3: while (attempts < num_retries) ∨ (deployed = false) do
4:    if FindAvailablePeer(failed_mngt_service) = true then
5:       DeployInstance(failed_mngt_service)
6:       deployed ← true
7:    else
8:       attempts ← attempts + 1
9:    end if
10: end while
11: if deployed = true then
12:    return true
13: else
14:    return false
15: end if
```

— Le deuxième raffinement M2 introduit abstraitement la phase de **self-activation**. L'état illégal d'un service est détecté, puis évalué : il s'agit de déterminer si l'état illégal et sa cause sont critiques, avant l'activation de la procédure de self-healing.
— Le troisième raffinement M3 introduit abstraitement la phase de **self-configuration**. L'évaluation de l'état illégal d'un service conduit à deux cas : soit l'état illégal et sa cause sont critiques, auquel cas une reconfiguration du service est nécessaire, pour le retour dans un état légal, soit l'état illégal et sa cause ne sont pas critique et le service est considéré comme étant toujours dans un état légal.
— Le quatrième raffinement M4 présente globalement le comportement du système P2P :
  — Une suspicion de l'existence d'un état illégal d'un service fourni par le système P2P entraîne l'activation de la procédure de self-detection.
  — La phase de self-detection établit si oui ou non l'état illégal du service est réel.
  — Si l'état illégal est réel, la phase de self-activation évalue si l'état du service et ses possibles causes sont critiques, sinon le service ignore l'état illégal et retourne dans un état légal.
  — Si l'état illégal du service s'avère critique, alors la procédure de self-configuration est activée, réparant le service. Autrement, l'état illégal du service est ignoré.
— Les raffinements de M5 à M19 permettent la localisation des modèles, dont les étapes principales sont :
  — L'introduction des pairs/processus instanciant les services fournis par le système P2P (M5 et M6 principalement).
  — L'introduction des instances « token owner » dirigeant et coordonnant les différentes phases de la procédure de self-healing, diffusant des informations sur les états des pairs/instances, services, etc (M8, M11, M12, M16 à M19 principalement).
  — La localisation progressive des trois phases de la procédure de self-healing : self-detection, self-activation, self-configuration ; l'ajout de détails concernant l'entrée d'un service dans un état illégal (les autres modèles non-cités plus haut).
  — L'ajout d'hypothèses permettant le fonctionnement correct de la procédure de self-healing : **(1)** *disponibilité constante d'instances de type « token owner »*, **(2)** *disponibilité d'un nombre suffisant de pairs pour pallier aux possibles défaillances de ceux instanciant un service.*
— Le dernier raffinement M20 est un modèle concret présentant les différentes phases algorithmiques composant la procédure de self-healing (algorithmes 9, 10, 11, 12).

Les sous-sections suivantes détaillent ces différents niveaux de raffinement.

8.3.2 Abstraction du problème de self-healing

Nous présentons dans cette sous-section une abstraction de la procédure appelée « self-healing » d'un service fourni par le système P2P, qui est entré dans un état fautif ou illégal. Chaque service (s) fourni par le système P2P est caractérisé dans cette abstraction par deux états : un état RUN, état légal dans lequel un service fonctionne correctement, et un état $FAIL$, état illégal ou fautif, faisant suite à une faute. Nous décrivons ces services, états et leurs caractéristiques dans un contexte C0 :

CONTEXT C0
SETS
$SERVICES, STATES$
CONSTANTS
$RUN, FAIL, InitState$
AXIOMS
$axm1$: $SERVICES \neq \varnothing$
$axm2$: $STATES = \{RUN, FAIL\}$
$axm3$: $RUN \neq FAIL$
$axm4$: $InitState \in SERVICES \rightarrow STATES$
$axm5$: $\forall s \cdot s \in SERVICES \Rightarrow s \mapsto RUN \in InitState$
$axm6$: $\forall s, st1, st2 \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st1 \in STATES \\ \wedge st2 \in STATES \\ \wedge s \mapsto st1 \in InitState \\ \wedge s \mapsto st2 \in InitState \end{array} \right) \Rightarrow st1 = st2$
END

— $axm1$: nous définissons un ensemble de $SERVICES$ non-vide.
— $axm2$: nous définissons un ensemble d'état $STATES$ contenant les états RUN et $FAIL$.
— $axm3$: les deux états RUN et $FAIL$ sont différents.
— $axm4$: nous définissons une constante $InitState$ qui contient les états initiaux de chaque service $s \in SERVICES$.
— $axm5$: l'état initial de chaque service est l'état légal RUN.
— $axm6$: cet axiome exprime que chaque service a un état initial unique.

Ce contexte C0 est utilisé par une machine M0, modélisant abstraitement la procédure « self-healing » d'un service s entré dans un état illégal.

La machine M0 contient une variable *serviceState*, définissant l'état courant de chacun des services fournis par le système P2P. L'état initial de cette variable est défini par la constante $InitState$:

INITIALISATION $\widehat{=}$
BEGIN
$act1$: $serviceState := InitState$
END

Cette variable est contrainte par les invariants suivants :

$inv1$: $serviceState \in SERVICES \rightarrow STATES$
$inv2$: $\forall s, st1, st2 \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st1 \in STATES \\ \wedge st2 \in STATES \\ \wedge s \mapsto st1 \in serviceState \\ \wedge s \mapsto st2 \in serviceState \end{array} \right) \Rightarrow st1 = st2$

— $inv1$ exprime que nous associons dans *serviceState* des services à des états.
— $inv2$ exprime que l'état d'un service s dans *serviceState* est unique.

Nous notons x l'ensemble des variables de l'abstraction M0 et $I(x)$ la conjonction des invariants caractérisant ces variables.

Nous définissons deux propriétés qui nous aiderons à exprimer la procédure « self-healing » d'un service s :

— $P \ \widehat{=} \ (s \mapsto RUN \in serviceState)$ exprime qu'un service s se trouve dans un état légal.
— $(\neg P) \ \widehat{=} \ (s \mapsto FAIL \in serviceState)$ exprime qu'un service s se trouve dans un état illégal.

La procédure « self-healing » d'un service s peut être exprimée simplement à l'aide de la propriété de vivacité suivante : $(\neg P) \rightsquigarrow P$, c'est-à-dire que si le service s se trouve dans un état illégal, une procédure le ramène fatalement dans un état légal. La liste Φ_0 des propriétés de vivacité caractérisant M0 est la suivante :

$$\Phi_0 \ \widehat{=} \ \{(\neg P) \rightsquigarrow P\}$$

Nous introduisons dans cette machine M0 deux événements HEAL_0 et FAILURE, que nous détaillerons un peu plus bas. L'hypothèse d'équité L_0 que nous posons sur M0 est la suivante :

$$L_0 \ \widehat{=} \ WF_x(\mathsf{HEAL}_0)$$

Nous posons une hypothèse d'équité faible sur l'événement HEAL_0 ($WF_x(\mathsf{HEAL}_0)$) et aucune hypothèse d'équité sur l'événement FAILURE. Nous définissons aussi la relation NEXT associée à M0 :

$$\begin{array}{lcl} \mathsf{NEXT} & \widehat{=} & \vee BA(\mathsf{HEAL}_0)(x, x') \\ & & \vee BA(\mathsf{FAILURE})(x, x') \end{array}$$

Nous utilisons les événements de M0 pour modéliser abstraitement l'idée générale du « self-healing » d'un service s, décrit par le diagramme suivant :

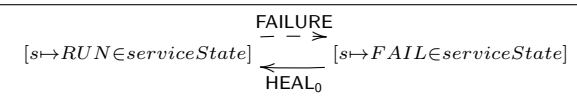


FIGURE 8.3 – Abstraction

Démonstration. Nous montrons maintenant que la machine M0 satisfait les propriétés de Φ_0.

⟨1⟩1. Nous montrons que M0 satisfait $(\neg P) \rightsquigarrow P$. Une hypothèse d'équité faible est posée sur l'événement HEAL_0 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de HEAL_0 sont observables lorsque $(\neg P)$ est vrai, leurs observations ne falsifient pas le prédicat $(\neg P)$, condition d'observation de HEAL_0, ou conduisent à P.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(\neg P) \wedge [\mathsf{NEXT}]_x \Rightarrow ((\neg P)' \vee P')$$

Soit :
— $(\neg P) \wedge BA(\mathsf{HEAL}_0)(x, x') \Rightarrow ((\neg P)' \vee P')$
— $(\neg P) \wedge BA(\mathsf{FAILURE})(x, x') \Rightarrow ((\neg P)' \vee P')$
— $(\neg P) \wedge (x = x') \Rightarrow ((\neg P)' \vee P')$
La propriété suivante **(3)** : $(\neg P)(x) \wedge BA(\mathsf{HEAL}_0)(x, x') \Rightarrow P(x')$, et la condition de faisabilité **(4)** : $(\neg P)(x) \Rightarrow (\exists x' \cdot BA(\mathsf{HEAL}_0)(x, x'))$ sont satisfaites par l'événement HEAL_0. La propriété **(3)** nous permet de déduire $(\neg P) \wedge \langle \mathsf{NEXT} \wedge \mathsf{HEAL}_0 \rangle_x \Rightarrow P'$ et **(4)** nous permet de déduire $(\neg P) \Rightarrow ENABLED \langle \mathsf{HEAL}_0 \rangle_x$, où $ENABLED \langle \mathsf{HEAL}_0 \rangle_x \ \widehat{=} \ (\exists y \cdot BA(\mathsf{HEAL}_0)(x, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M0}) \vdash (\neg P) \rightsquigarrow P$, donc que M0 satisfait $(\neg P) \rightsquigarrow P$. □

⟨1⟩2. L'étape ⟨1⟩1 nous permet de déduire que M0 satisfait les propriétés de vivacité de Φ_0. QED. □

Nous nous servons des propriétés de Φ_0 et du diagramme 8.3 pour définir les événements FAILURE et HEAL_0 :

— L'événement FAILURE modélise une faute concernant un service (s) : le service (s) passe d'un état légal (RUN) ($grd2$) à un état fautif ($FAIL$) ($act1$).

```
EVENT FAILURE ≙
  ANY
    s
  WHERE
    grd1 : s ∈ SERVICES
    grd2 : s ↦ RUN ∈ serviceState
  THEN
    act1 : serviceState := ({s} ⩤ serviceState) ∪ {s ↦ FAIL}
  END
```

— La procédure « self-healing » du service s est exprimée par l'événement HEAL_0 : le service (s) retourne dans un état légal légal (RUN) ($act1$) à partir d'un état fautif ($FAIL$) ($grd2$).

EVENT HEAL_0 $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FAIL \in serviceState$
THEN
$act1$: $serviceState := (serviceState \setminus \{s \mapsto FAIL\}) \cup \{s \mapsto RUN\}$
END

Cette première abstraction peut se résumer comme suit :

$$\mathsf{C0} \xleftarrow{\text{SEES}} \mathsf{M0} \xrightarrow{\text{LIVE}} \left(\left(\ WF_x(\mathsf{HEAL}_0)\ \right), \left\{\ (\neg P) \rightsquigarrow P\ \right\}\right)$$

FIGURE 8.4 – Abstraction du self-healing

La prochaine sous-section introduit la phase de self-detection.

8.3.3 Abstraction et introduction des phases de « self-healing »

8.3.3.1 Introduction de la self-detection

Dans ce premier raffinement, nous décomposons et détaillons la procédure de « self-healing » : nous introduisons entre les états illégaux d'un service et ces états légaux, un état supplémentaire qui est la détection d'un état illégal. Cette détection des états fautifs et illégaux d'un service est connue sous le nom de **self-detection**. Nous commençons par définir un contexte C1, étendant le contexte C0 et qui introduit ces notions.

CONTEXT C0 EXTENDS C1
SETS
$STATES\_1$
CONSTANTS
$RUN\_1, FAIL\_1, FL\_DT\_1, InitState\_1$
AXIOMS
$axm1$: $partition(STATES\_1, \{RUN\_1\}, \{FAIL\_1\}, \{FL\_DT\_1\})$
$axm2$: $InitState\_1 \in SERVICES \to STATES\_1$
$axm3$: $\forall s \cdot s \in SERVICES \Rightarrow s \mapsto RUN\_1 \in InitState\_1$
$axm4$: $\forall s, st1, st2 \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st1 \in STATES\_1 \\ \wedge st2 \in STATES\_1 \\ \wedge s \mapsto st1 \in InitState\_1 \\ \wedge s \mapsto st2 \in InitState\_1 \end{array}\right) \Rightarrow st1 = st2$
END

— $axm1$: un ensemble $STATES\_1$ d'états possibles pour chaque service est défini. Il est composé de trois états distincts :
  — RUN_1 représente un état de fonctionnement légal d'un service.
  — $FAIL\_1$ représente un état illégal, consécutif à une faute.
  — FL_DT_1 représente la détection d'un état illégal.
  Ce nouvel ensemble d'états $STATES\_1$ nous servira à remplacer l'ensemble $STATES$ de l'abstraction.
— $axm2$: $InitState\_1$ est une constante associant à chaque service un état de l'ensemble $STATES\_1$. Elle nous permettra de décrire les états initiaux des services fournis par le système P2P.
— $axm3$: l'état initial de chaque service est RUN_1.
— $axm4$: chaque service a un état initial unique.

Une machine M1, raffinant la machine M0 vue précédemment, utilise ce contexte C1.

Cette machine M1 introduit une nouvelle variable $serviceState\_1$, remplaçant la variable $serviceState$: cette nouvelle variable nous permet de prendre en compte le nouvel état intermédiaire FL_DT_1. L'état initial de cette variable est défini à l'aide de la constante $InitState\_1$: à l'état initial, tous les services sont dans l'état légal RUN_1.

$$\begin{array}{l}\mathsf{INITIALISATION} \ \widehat{=} \\ \quad \mathrm{BEGIN} \\ \quad\quad \ominus\ act1 \ : \ serviceState := InitState \\ \quad\quad \oplus\ act1 \ : \ serviceState\_1 := InitState\_1 \\ \quad \mathrm{END}\end{array}$$

Nous notons x_1 l'ensemble des variables de la machine M1.

Un invariant que nous notons $I_1(x_1)$ contraint ces variables et est défini comme suit :

— Nous avons du typage.

$$inv1 \ : \ serviceState\_1 \in SERVICES \rightarrow STATES\_1$$

La variable *serviceState*_1 associe des services et des états.

— Nous avons des propriétés de sûreté.

$$inv2 \ : \ \forall s, st1, st2 \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st1 \in STATES\_1 \\ \wedge st2 \in STATES\_1 \\ \wedge s \mapsto st1 \in serviceState\_1 \\ \wedge s \mapsto st2 \in serviceState\_1 \end{array} \right) \Rightarrow st1 = st2$$

Chaque service s a un état courant unique.

— Nous avons des invariants de collage, permettant de faire le lien entre les variables concrète *serviceState*_1 et abstraite *serviceState*, ainsi que les liens entre les états concrets RUN_1, $FAIL\_1$, FL_DT_1 et abstraits RUN, $FAIL$.

$$\begin{array}{l} inv3 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge s \mapsto RUN \in serviceState \end{array} \right) \Rightarrow \left(\begin{array}{l} \vee s \mapsto RUN\_1 \in serviceState\_1 \\ \vee s \mapsto FL\_DT\_1 \in serviceState\_1 \end{array} \right) \\ inv4 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge \left(\begin{array}{l} \vee s \mapsto RUN\_1 \in serviceState\_1 \\ \vee s \mapsto FL\_DT\_1 \in serviceState\_1 \end{array} \right) \end{array} \right) \Rightarrow s \mapsto RUN \in serviceState \\ inv5 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FAIL \in serviceState \Rightarrow s \mapsto FAIL\_1 \in serviceState\_1 \\ inv6 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FAIL\_1 \in serviceState\_1 \Rightarrow s \mapsto FAIL \in serviceState \end{array}$$

— Les propriétés $inv3$ et $inv4$ expriment que l'état abstrait RUN d'un service s est raffiné en deux états concrets : un état RUN_1 et un état FL_DT_1.

— Les propriétés $inv5$ et $inv6$ expriment que l'état abstrait $FAIL$ d'un service s est raffiné en un état concret $FAIL\_1$.

Trois événements, que nous détaillerons plus bas, sont présents dans cette machine M1 : FAILURE, FAIL_DETECT, HEAL_1 ; et l'hypothèse d'équité L_1 que nous posons sur la machine M1 est la suivante :

$$L_1 \ \widehat{=} \ WF(\mathsf{FAIL\_DETECT}) \wedge WF(\mathsf{HEAL}_1)$$

Nous définissons ensuite la relation NEXT caractérisant cette machine M1 :

$$\begin{array}{lll} \mathsf{NEXT} & \widehat{=} & \vee BA(\mathsf{HEAL}_1)(x_1, {x_1}') \\ & & \vee BA(\mathsf{FAILURE})(x_1, {x_1}') \\ & & \vee BA(\mathsf{FAIL\_DETECT})(x_1, {x_1}') \end{array}$$

Nous définissons ensuite trois propriétés :

— $P_1 \ \widehat{=} \ (s \mapsto RUN\_1 \in serviceState\_1)$. Cette propriété signifie qu'un service s se trouve dans un état légal RUN_1.

— $(\neg P_1) \ \widehat{=} \ (s \mapsto FAIL\_1 \in serviceState\_1)$. Cette propriété signifie qu'un service s se trouve dans un état fautif/illégal $FAIL\_1$.

— $R_1 \ \widehat{=} \ (s \mapsto FL\_DT\_1 \in serviceState\_1)$. Cette propriété signifie qu'un service s a identifié/détecté qu'il est entré dans un état fautif/illégal $FAIL\_1$ (**self-detection**).

Le mécanisme de **self-detection**, ainsi que les propriétés de vivacité satisfaites par la machine M1 sont exprimées par le diagramme suivant :

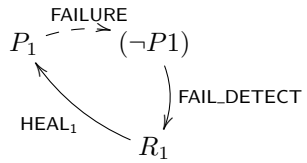


FIGURE 8.5 – Self-Detection

Il s'agit ici de détailler la procédure de « self-healing » d'un service, caractérisée par $(\neg P_1) \rightsquigarrow P_1$, en introduisant la détection des états fautifs caractérisée par la propriété R_1 entre $(\neg P_1)$ et P_1. Pour cela, nous utilisons la règle d'inférence **transitivité** :

$$\frac{(\neg P_1) \rightsquigarrow R_1 \qquad R_1 \rightsquigarrow P_1}{(\neg P_1) \rightsquigarrow P_1}\ (trans)$$

Le service s est capable **(phase 1)** de suspecter et d'identifier un état illégal ($(\neg P_1) \rightsquigarrow R_1$), puis **(phase 2)** de déclencher une procédure de guérison du service ($R_1 \rightsquigarrow P_1$). Nous définissons alors la liste Φ_1 des propriétés de vivacité caractérisant M1 comme suit :

$$\Phi_1 \ \widehat{=}\ \{(\neg P_1) \rightsquigarrow R_1, R_1 \rightsquigarrow P_1\}$$

Démonstration. Nous montrons que M1 satisfait les propriétés de Φ_1.

⟨1⟩1. Nous montrons que M1 satisfait $(\neg P_1) \rightsquigarrow R_1$. Une hypothèse d'équité faible est posée sur l'événement FAIL_DETECT et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de FAIL_DETECT sont observables lorsque $(\neg P_1)$ est vrai, leurs observations ne falsifient pas le prédicat $(\neg P_1)$, condition d'observation de FAIL_DETECT, ou conduisent à R_1.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$(\neg P_1) \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow ((\neg P_1)' \vee {R_1}')$$

Soit :
— $(\neg P_1) \wedge BA(\mathsf{HEAL}_1)(x_1, {x_1}') \Rightarrow ((\neg P_1)' \vee {R_1}')$
— $(\neg P_1) \wedge BA(\mathsf{FAILURE})(x_1, {x_1}') \Rightarrow ((\neg P_1)' \vee {R_1}')$
— $(\neg P_1) \wedge BA(\mathsf{FAIL\_DETECT})(x_1, {x_1}') \Rightarrow ((\neg P_1)' \vee {R_1}')$
— $(\neg P_1) \wedge (x_1 = {x_1}') \Rightarrow ((\neg P_1)' \vee {R_1}')$
La propriété suivante **(3)** : $(\neg P_1)(x_1) \wedge BA(\mathsf{FAIL\_DETECT})(x_1, {x_1}') \Rightarrow R_1({x_1}')$, et la condition de faisabilité **(4)** : $(\neg P_1)(x_1) \Rightarrow (\exists {x_1}' \cdot BA(\mathsf{FAIL\_DETECT})(x_1, {x_1}'))$ sont satisfaites par l'événement FAIL_DETECT. La propriété **(3)** nous permet de déduire $(\neg P_1) \wedge \langle \mathsf{NEXT} \wedge \mathsf{FAIL\_DETECT} \rangle_{x_1} \Rightarrow {R_1}'$ et **(4)** permet de déduire $(\neg P_1) \Rightarrow ENABLED \langle \mathsf{FAIL\_DETECT} \rangle_{x_1}$, où $ENABLED \langle \mathsf{FAIL\_DETECT} \rangle_{x_1} \ \widehat{=}\ (\exists y \cdot BA(\mathsf{FAIL\_DETECT})(x_1, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M1}) \vdash (\neg P_1) \rightsquigarrow R_1$, donc que M1 satisfait $(\neg P_1) \rightsquigarrow R_1$. □

⟨1⟩2. Nous montrons que M1 satisfait $R_1 \rightsquigarrow P_1$. Une hypothèse d'équité faible est posée sur l'événement HEAL_1 et nous nous trouvons dans le cas suivant **(1)** : si les autres événements différents de HEAL_1 sont observables lorsque R_1 est vrai, leurs observations ne falsifient pas le prédicat R_1, condition d'observation de HEAL_1, ou conduisent à P_1.
Nous pouvons déduire de **(1)** la propriété suivante **(2)** :

$$R_1 \wedge [\mathsf{NEXT}]_{x_1} \Rightarrow (R_1' \vee {P_1}')$$

Soit :
— $R_1 \wedge BA(\mathsf{HEAL}_1)(x_1, {x_1}') \Rightarrow (R_1' \vee {P_1}')$
— $R_1 \wedge BA(\mathsf{FAILURE})(x_1, {x_1}') \Rightarrow (R_1' \vee {P_1}')$
— $R_1 \wedge BA(\mathsf{FAIL\_DETECT})(x_1, {x_1}') \Rightarrow (R_1' \vee {P_1}')$
— $R_1 \wedge (x_1 = {x_1}') \Rightarrow (R_1' \vee {P_1}')$
La propriété suivante **(3)** : $R_1(x_1) \wedge BA(\mathsf{HEAL}_1)(x_1, {x_1}') \Rightarrow P_1({x_1}')$, et la condition de faisabilité **(4)** : $R_1(x_1) \Rightarrow (\exists x' \cdot BA(\mathsf{HEAL}_1)(x_1, {x_1}'))$ sont satisfaites par l'événement HEAL_1. La propriété **(3)** nous permet de déduire $R_1 \wedge \langle \mathsf{NEXT} \wedge \mathsf{HEAL}_1 \rangle_{x_1} \Rightarrow {P_1}'$ et **(4)** nous permet de déduire $R_1 \Rightarrow ENABLED \langle \mathsf{HEAL}_1 \rangle_{x_1}$, où $ENABLED \langle \mathsf{HEAL}_1 \rangle_{x_1} \ \widehat{=}\ (\exists y \cdot BA(\mathsf{HEAL}_1)(x_1, y))$. L'application de la règle WF1 nous permet de montrer que $\mathcal{S}pec(\mathsf{M1}) \vdash R_1 \rightsquigarrow P_1$, donc que M1 satisfait $R_1 \rightsquigarrow P_1$. □

⟨1⟩3. Les étapes ⟨1⟩1 et ⟨1⟩2 nous permettent de déduire que M1 satisfait les propriétés de vivacité de Φ_1. QED.

□

Soit Φ_0 la liste des propriétés de vivacité caractérisant la machine M0 précédente suivante :

$$\Phi_0 \ \widehat{=}\ \{(\neg P) \rightsquigarrow P\}$$

Démonstration. Nous montrons que M1 satisfait les propriétés de Φ_1.

⟨1⟩1. Nous montrons que M1 satisfait $(\neg P) \rightsquigarrow P$. Nous savons que M1 satisfait $(\neg P_1) \rightsquigarrow P_1$ (par transitivité (*trans*)) et que nous avons $(\neg P) \Leftrightarrow (\neg P_1)$ (grâce aux invariants *inv*5 et *inv*6 de M1), ainsi que $(P_1 \vee R_1) \Leftrightarrow P$ (grâce aux invariants *inv*3 et *inv*4 de M1). En utilisant la règle de déduction (*dedu*), nous en déduisant $(\neg P) \rightsquigarrow (\neg P_1)$ et $P_1 \rightsquigarrow P$. Par transitivité (*trans*), nous montrons que M1 satisfait $(\neg P) \rightsquigarrow P$.
⟨1⟩3. L'étape ⟨1⟩1 nous permet de déduire que M1 satisfait les propriétés de vivacité de Φ_0. QED.
□

Nous nous servons des propriétés de Φ_1 et du diagramme 8.5 pour définir les événements FAIL_DETECT, HEAL_1 et FAILURE :

— Evénement FAIL_DETECT, raffinement de l'événement abstrait HEAL_0 : un service s dont l'état est $FAIL\_1$ ($grd2$), détecte cet état et passe dans un état FL_DT_1 ($act1$).

EVENT FAIL_DETECT REFINES HEAL_0 $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FAIL\_1 \in serviceState\_1$
THEN
$act1$: $serviceState\_1 := (serviceState\_1 \setminus \{s \mapsto FAIL\_1\}) \cup \{s \mapsto FL\_DT\_1\}$
END

— Evénement HEAL_1 : un service s ayant détecté un état illégal et étant l'état est FL_DT_1 ($grd2$), déclenche la procédure de guérison et repasse dans un état RUN_1 ($act1$).

EVENT HEAL_1 $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_DT\_1 \in serviceState\_1$
THEN
$act1$: $serviceState\_1 := (serviceState\_1 \setminus \{s \mapsto FL\_DT\_1\}) \cup \{s \mapsto RUN\_1\}$
END

— L'événement FAILURE est raffiné, de manière à prendre en compte les nouvelles variables et nouveaux états :

EVENT FAILURE REFINES FAILURE $\widehat{=}$
...
WHERE
$\ominus$ $grd2$: $s \mapsto RUN \in serviceState$
$\oplus$ $grd2$: $s \mapsto RUN\_1 \in serviceState\_1$
THEN
$\ominus$ $act1$: ...
$\oplus$ $act1$: $serviceState\_1 := (serviceState\_1 \setminus \{s \mapsto RUN\_1\}) \cup \{s \mapsto FAIL\_1\}$
END

Ce premier raffinement peut se résumer comme suit :

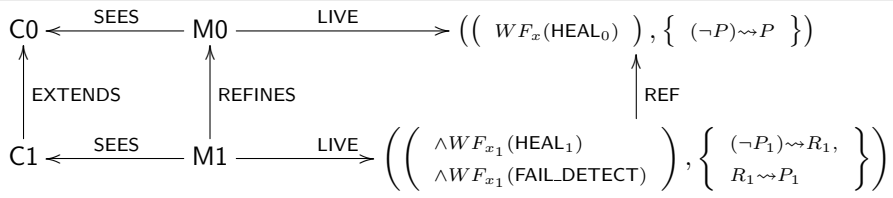


FIGURE 8.6 – Premier raffinement du self-healing

La prochaine sous-section introduit la phase de self-activation. Nous ne détaillons pas les preuves et toutes les étapes du développement pour les niveaux de raffinement suivants. Celles-ci peuvent être trouvés en annexe [11].

8.3.3.2 Introduction de la self-activation

Ce second raffinement nous permet d'introduire une nouvelle phase de la procédure de « self-healing » d'un service fourni par le système P2P : il s'agit de la phase de **self-activation** ayant lieu après celle de **self-detection**. La gravité d'une erreur (pannes, fautes, etc) ayant entraîné le service dans un état illégal est évaluée : s'il s'agit d'une fausse alerte ou si l'erreur n'est pas critique, elle est ignorée, sinon une procédure de guérison du service est déclenchée. Nous commençons par définir un contexte C2, étendant le contexte C1 vu précédemment, et introduisant des éléments (états, etc) relatifs à l'étape de **self-activation**.

CONTEXT C2 EXTENDS C1
SETS
$STATES\_2$
CONSTANTS
$RUN\_2, FAIL\_2, FL\_DT\_2, FL\_ACT\_2, InitState\_2$
AXIOMS
$axm1$: $partition(STATES\_2, \{RUN\_2\}, \{FAIL\_2\}, \{FL\_DT\_2\}, \{FL\_ACT\_2\})$
$axm2$: $InitState\_2 \in SERVICES \rightarrow STATES\_2$
$axm3$: $\forall s \cdot s \in SERVICES \Rightarrow s \mapsto RUN\_2 \in InitState\_2$
$axm4$: $\forall s, st1, st2 \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st1 \in STATES\_2 \\ \wedge st2 \in STATES\_2 \\ \wedge s \mapsto st1 \in InitState\_2 \\ \wedge s \mapsto st2 \in InitState\_2 \end{array} \right) \Rightarrow st1 = st2$
END

- $axm1$: un ensemble $STATES\_2$ d'états possibles pour chaque service est défini. Il est composé de quatre états distincts :
  - RUN_2 représente un état de fonctionnement légal d'un service.
  - $FAIL\_2$ représente un état illégal, consécutif à une faute.
  - FL_DT_2 représente la détection d'un état illégal.
  - FL_ACT_2 représente l'évaluation de la gravité de la cause (erreurs, fautes, pannes, etc) d'un état illégal.

  Ce nouvel ensemble d'états $STATES\_2$ nous servira à remplacer l'ensemble $STATES\_1$ du premier raffinement.
- $axm2$: $InitState\_2$ est une constante associant à chaque service un état de l'ensemble $STATES\_2$. Elle nous permettra de décrire les états initiaux des services fournis par le système P2P.
- $axm3$: l'état initial de chaque service est RUN_2.
- $axm4$: chaque service a un état initial unique.

Une machine M2, raffinant la machine M1 vue précédemment, utilise ce contexte C2.

Cette machine M2 introduit une nouvelle variable $serviceState\_2$, remplaçant la variable $serviceState\_1$: cette nouvelle variable nous permet de prendre en compte le nouvel état intermédiaire FL_ACT_2. L'état initial de cette variable est défini à l'aide de la constante $InitState\_2$: à l'état initial, tous les services sont dans l'état légal RUN_2.

INITIALISATION $\widehat{=}$
BEGIN
$\ominus$ $act1$: $serviceState\_1 := InitState\_1$
$\oplus$ $act1$: $serviceState\_2 := InitState\_2$
END

Nous x_2 l'ensemble des variables de cette machine M2.

Un invariant que nous notons $I_2(x_2)$ caractérisent ces variables et défini comme suit :

- Nous avons du typage.

$inv1$: $serviceState\_2 \in SERVICES \rightarrow STATES\_2$

11. `http://andriami.bruno.free.fr/andriami/main/short/annexe.pdf`

La variable *serviceState_2* est une relation associant des services et des états.

— Nous avons des propriétés de sûreté.

$$inv2 \ : \ \forall s, st1, st2 \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st1 \in STATES\_2 \\ \wedge st2 \in STATES\_2 \\ \wedge s \mapsto st1 \in serviceState\_2 \\ \wedge s \mapsto st2 \in serviceState\_2 \end{array} \right) \Rightarrow st1 = st2$$

La propriété *inv*2 exprime l'unicité de l'état courant d'un service s.

— Nous avons des invariants de collage, établissant les relations entre les variables concrète *serviceState_2* et abstraite *serviceState_1*, ainsi qu'entre les états concrets RUN_2, $FAIL\_2$, FL_DT_2, FL_ACT_2 et abstraits RUN_1, $FAIL\_1$, FL_DT_1.

$$inv3 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge s \mapsto RUN\_1 \in serviceState\_1 \end{array} \right) \Rightarrow \left(\begin{array}{l} \vee s \mapsto RUN\_2 \in serviceState\_2 \\ \vee s \mapsto FL\_ACT\_2 \in serviceState\_2 \end{array} \right)$$

$$inv4 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge \left(\begin{array}{l} \vee s \mapsto RUN\_2 \in serviceState\_2 \\ \vee s \mapsto FL\_ACT\_2 \in serviceState\_2 \end{array} \right) \end{array} \right) \Rightarrow s \mapsto RUN\_1 \in serviceState\_1$$

$$inv5 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_DT\_1 \in serviceState\_1 \Rightarrow s \mapsto FL\_DT\_2 \in serviceState\_2$$

$$inv6 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_DT\_2 \in serviceState\_2 \Rightarrow s \mapsto FL\_DT\_1 \in serviceState\_1$$

$$inv7 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FAIL\_1 \in serviceState\_1 \Rightarrow s \mapsto FAIL\_2 \in serviceState\_2$$

$$inv8 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FAIL\_2 \in serviceState\_2 \Rightarrow s \mapsto FAIL\_1 \in serviceState\_1$$

— Les propriétés *inv*3 et *inv*4 expriment que l'état abstrait RUN_1 d'un service s est raffiné en deux états concrets : un état RUN_2 et un état FL_ACT_2.

— Les propriétés *inv*5 et *inv*6 expriment que l'état abstrait FL_DT_1 d'un service s est raffiné en un état concret FL_DT_2.

— Les propriétés *inv*7 et *inv*8 expriment que l'état abstrait $FAIL\_1$ d'un service s est raffiné en un état concret $FAIL\_2$.

Cinq événements, que nous détaillerons plus bas, sont présents dans cette machine M2 : FAILURE, FAIL_DETECT, FAIL_ACTIV, IS_OK, HEAL_2. Une liste Φ_2 de propriétés de vivacité, détaillée en annexe [12] nous permet de définir les événements de M2 :

— Phase de **self-detection** : un service s détecte qu'il est entré dans un état illégal. Cette phase est exprimée par l'événement FAIL_DETECT suivant :

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
  ANY
    s
  WHERE
    $grd1$: $s \in SERVICES$
    $grd2$: $s \mapsto FAIL\_2 \in serviceState\_2$
  THEN
    $act1$: $serviceState\_1 := (serviceState\_1 \setminus \{s \mapsto FAIL\_2\}) \cup \{s \mapsto FL\_DT\_2\}$
  END

Le service s passe de l'état illégal $FAIL\_2$ ($grd2$) à l'état de détection FL_DT_2 ($act1$).

— Nous distinguons ensuite deux cas après la détection d'un état illégal par le service s :

— Le premier cas : l'erreur ayant entraîné l'état illégal du service s est une fausse alerte. Ce dernier peut donc revenir à un état légal. L'événement IS_OK, raffinant HEAL_1, exprime ce mécanisme :

EVENT IS_OK REFINES HEAL_1 $\widehat{=}$
  ANY
    s
  WHERE
    $grd1$: $s \in SERVICES$
    $grd2$: $s \mapsto FL\_DT\_2 \in serviceState\_2$
  THEN
    $act1$: $serviceState\_1 := (serviceState\_1 \setminus \{s \mapsto FL\_DT\_2\}) \cup \{s \mapsto RUN\_2\}$
  END

Le service s passe de l'état de détection FL_DT_2 ($grd2$) à l'état légal RUN_2 ($act1$).

— Deuxième cas :

— L'erreur ayant entraîné l'état illégal du service s est réelle et elle est fatalement évaluée, pour voir si elle est critique ou non. L'évenement FAIL_ACTIV, raffinement de HEAL_1, exprime ce mécanisme :

12. http://andriami.bruno.free.fr/andriami/main/short/annexe.pdf

EVENT FAIL_ACTIV REFINES HEAL_1 $\widehat{=}$
  ANY
    s
  WHERE
    $grd1$: $s \in SERVICES$
    $grd2$: $s \mapsto FL\_DT\_2 \in serviceState\_2$
  THEN
    $act1$: $serviceState\_2 := (serviceState\_2 \setminus \{s \mapsto FL\_DT\_2\}) \cup \{s \mapsto FL\_ACT\_2\}$
  END

Le service s passe de l'état de détection FL_DT_2 ($grd2$) à l'état FL_ACT_2 d'évaluation de l'erreur ($act1$).

— Déclenchement de la procédure de guérison après l'évaluation d'une erreur réelle ayant entraîné l'état illégale du service s. Un nouvel événement HEAL_2 exprime abstraitement ce processus de guérison :

EVENT HEAL_2 $\widehat{=}$
  ANY
    s
  WHERE
    $grd1$: $s \in SERVICES$
    $grd2$: $s \mapsto FL\_ACT\_2 \in serviceState\_2$
  THEN
    $act1$: $serviceState\_2 := (serviceState\_2 \setminus \{s \mapsto FL\_ACT\_2\}) \cup \{s \mapsto RUN\_2\}$
  END

Le service s passe de l'état d'évaluation d'erreurs FL_ACT_2 ($grd2$) à l'état légal de fonctionnement RUN_2 ($act1$).

— L'événement FAILURE est raffiné, de manière à prendre en compte les nouvelles variables et nouveaux états :

EVENT FAILURE REFINES FAILURE $\widehat{=}$
  ...
  WHERE
    $\ominus$ $grd2$: $s \mapsto RUN\_1 \in serviceState\_1$
    $\oplus$ $grd2$: $s \mapsto RUN\_2 \in serviceState\_2$
  THEN
    $\ominus$ $act1$: ...
    $\oplus$ $act1$: $serviceState\_2 := (serviceState\_2 \setminus \{s \mapsto RUN\_2\}) \cup \{s \mapsto FAIL\_2\}$
  END

Ce deuxième raffinement peut se résumer comme suit :

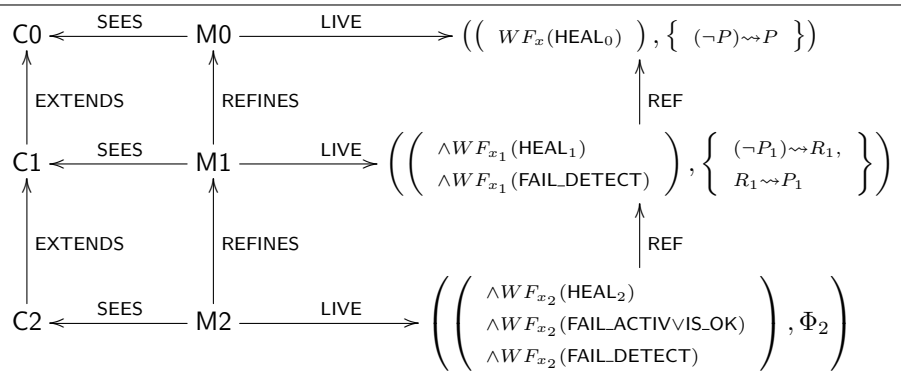


FIGURE 8.7 – Deuxième raffinement du self-healing

La prochaine sous-section introduit la phase de self-configuration.

8.3.3.3 Introduction de la self-configuration

Nous introduisons dans ce raffinement, des phases de la procédure de « self-healing » d'un service se déroulant après celle de **self-activation** : l'évaluation d'une erreur (faute, panne, etc), ayant conduit le service dans un état illégal, donne lieu soit au déclenchement de **(1)** la phase de **self-configuration** (reconfiguration d'un service pour un retour à un état légal) si l'erreur est critique, soit au fait que **(2)** l'erreur est ignorée, si elle n'est pas critique (le service est fourni correctement, malgré l'erreur). Un contexte C3, extension du précédent contexte C2, introduit des éléments (états, etc) relatifs à ces nouvelles phases de la procédure de « self-healing ».

CONTEXT C3 EXTENDS C2
SETS
$STATES\_3$
CONSTANTS
$RUN\_3, FAIL\_3, FL\_DT\_3, FL\_ACT\_3, FL\_CONF\_3, FL\_IGN\_3, InitState\_3$
AXIOMS
$axm1 : partition\left(\begin{array}{ll} STATES\_3, & \{RUN\_3\}\{FAIL\_3\}, \{FL\_DT\_3\}, \\ & \{FL\_ACT\_3\}, \{FL\_CONF\_3\}, \{FL\_IGN\_3\} \end{array}\right)$
$axm2 : InitState\_3 \in SERVICES \rightarrow STATES\_3$
$axm3 : \forall s \cdot s \in SERVICES \Rightarrow s \mapsto RUN\_3 \in InitState\_3$
$axm4 : \forall s, st1, st2 \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st1 \in STATES\_3 \\ \wedge st2 \in STATES\_3 \\ \wedge s \mapsto st1 \in InitState\_3 \\ \wedge s \mapsto st2 \in InitState\_3 \end{array}\right) \Rightarrow st1 = st2$
END

— $axm1$: un ensemble $STATES\_3$ d'états possibles pour chaque service est défini. Il est composé de six états distincts :
  — RUN_3 représente un état de fonctionnement légal d'un service.
  — $FAIL\_3$ représente un état illégal, consécutif à une faute.
  — FL_DT_3 représente la détection d'un état illégal.
  — FL_ACT_3 représente l'évaluation de la gravité de la cause (erreurs, fautes, pannes, etc) d'un état illégal.
  — FL_CONF_3 représente la configuration d'un service dans un état illégal pour un retour dans un état légal.
  — FL_IGN_3 représente le fait qu'une erreur ayant provoqué un état illégal est ignorée, car non-critique.

  Ce nouvel ensemble d'états $STATES\_3$ nous servira à remplacer l'ensemble $STATES\_2$ du précédent raffinement.
— $axm2$: $InitState\_3$ est une constante associant à chaque service un état de l'ensemble $STATES\_3$. Elle nous permettra de décrire les états initiaux des services fournis par le système P2P.
— $axm3$: l'état initial de chaque service est RUN_3.
— $axm4$: chaque service a un état initial unique.

Une machine M3, raffinant la machine M2 précédente, utilise ce contexte C3.

Cette machine M3 introduit une nouvelle variable $serviceState\_3$, remplaçant la variable $serviceState\_2$: cette nouvelle variable nous permet de prendre en compte les nouveaux états FL_CONF_3 et FL_IGN_3. L'état initial de cette variable est défini à l'aide de la constante $InitState\_3$: à l'état initial, tous les services sont dans l'état légal RUN_3.

INITIALISATION $\widehat{=}$
BEGIN
$\ominus\ act1 : serviceState\_2 := InitState\_2$
$\oplus\ act1 : serviceState\_3 := InitState\_3$
END

Nous notons x_3 l'ensemble des variables de cette machine M3.

Un invariant noté $I_3(x_3)$ contraint ces variables et est défini comme suit :
— Nous avons du typage.

$inv1 : serviceState\_3 \in SERVICES \rightarrow STATES\_3$

La variable $serviceState\_3$ est une relation associant des services et des états.

— Nous avons des propriétés de sûreté.

$$inv2 \ : \ \forall s, st1, st2 \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st1 \in STATES\_3 \\ \wedge st2 \in STATES\_3 \\ \wedge s \mapsto st1 \in serviceState\_3 \\ \wedge s \mapsto st2 \in serviceState\_3 \end{array} \right) \Rightarrow st1 = st2$$

$inv2$ exprime l'unicité de l'état courant d'un service s.

— Nous avons des invariants de collage qui définissent les relations entre les variables concrète $serviceState\_3$ et abstraite $serviceState\_2$, ainsi qu'entre les états concrets RUN_3, $FAIL\_3$, FL_DT_3, FL_ACT_3, FL_CONF_3, FL_IGN_3 et états abstraits RUN_2, $FAIL\_2$, FL_DT_2, FL_ACT_2.

$$inv3 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge s \mapsto RUN\_2 \in serviceState\_2 \end{array} \right) \Rightarrow \left(\begin{array}{l} \vee s \mapsto RUN\_3 \in serviceState\_3 \\ \vee s \mapsto FL\_CONF\_3 \in serviceState\_3 \\ \vee s \mapsto FL\_IGN\_3 \in serviceState\_3 \end{array} \right)$$

$$inv4 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge \left(\begin{array}{l} \vee s \mapsto RUN\_3 \in serviceState\_3 \\ \vee s \mapsto FL\_CONF\_3 \in serviceState\_3 \\ \vee s \mapsto FL\_IGN\_3 \in serviceState\_3 \end{array} \right) \end{array} \right) \Rightarrow s \mapsto RUN\_2 \in serviceState\_2$$

$$inv5 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_ACT\_2 \in serviceState\_2 \Rightarrow s \mapsto FL\_ACT\_3 \in serviceState\_3$$
$$inv6 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_ACT\_3 \in serviceState\_3 \Rightarrow s \mapsto FL\_ACT\_2 \in serviceState\_2$$
$$inv7 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_DT\_2 \in serviceState\_2 \Rightarrow s \mapsto FL\_DT\_3 \in serviceState\_3$$
$$inv8 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_DT\_3 \in serviceState\_3 \Rightarrow s \mapsto FL\_DT\_2 \in serviceState\_2$$
$$inv9 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FAIL\_2 \in serviceState\_2 \Rightarrow s \mapsto FAIL\_3 \in serviceState\_3$$
$$inv10 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FAIL\_3 \in serviceState\_3 \Rightarrow s \mapsto FAIL\_2 \in serviceState\_2$$

— Les propriétés $inv3$ et $inv4$ expriment que l'état abstrait RUN_2 d'un service s est raffiné en trois états concrets : un état RUN_3, un état FL_CONF_3 et un état FL_IGN_3.

— Les propriétés $inv5$ et $inv6$ expriment que l'état abstrait FL_ACT_2 d'un service s est raffiné en un état concret FL_ACT_3.

— Les propriétés $inv7$ et $inv8$ expriment que l'état abstrait FL_DT_2 d'un service s est raffiné en un état concret FL_DT_3.

— Les propriétés $inv9$ et $inv10$ expriment que l'état abstrait $FAIL\_2$ d'un service s est raffiné en un état concret $FAIL\_3$.

Six événements, que nous détaillerons plus bas, sont présents dans cette machine M3 : FAILURE, FAIL_DETECT, FAIL_ACTIV, IS_OK, FAIL_CF_IGN, HEAL_3. Une liste Φ_3 de propriétés de vivacité, détaillée en annexe, est utilisée pour définir les événements de la machine M3 :

— Un état illegal d'un service s est fatalement détecté (**self-detection**). L'événement FAIL_DETECT modélise cette phase : le service s passe de l'état $FAIL\_3$ ($grd2$) à l'état FL_DT_3 ($act1$).

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
  ANY
    s
  WHERE
    $grd1$: $s \in SERVICES$
    $grd2$: $s \mapsto FAIL\_3 \in serviceState\_3$
  THEN
    $act1$: $serviceState\_1 := (serviceState\_1 \setminus \{s \mapsto FAIL\_3\}) \cup \{s \mapsto FL\_DT\_3\}$
  END

— Un service s dont l'état illégal a été détecté va fatalement revenir à un état légal. Il s'agit ici de la procédure de guérison ayant lieu après celle de (**self-detection**). Nous distinguons deux cas.

— Premier cas : l'erreur (faute, panne, etc) ayant conduit un service s dans un état illégal est une fausse alerte, le service s revient donc directement dans un état de fonctionnement légal. Ce mécanisme est modélisé par l'événement IS_OK : l'état illégal du service s est détecté ($grd2$), mais étant donné que l'erreur ayant causé cet état est une fausse alerte, le nouvel état du service s est l'état légal RUN_3 ($act1$).

EVENT IS_OK REFINES IS_OK $\widehat{=}$
  ANY
    s
  WHERE
    $grd1$: $s \in SERVICES$
    $grd2$: $s \mapsto FL\_DT\_3 \in serviceState\_3$
  THEN
    $act1$: $serviceState\_3 := (serviceState\_3 \setminus \{s \mapsto FL\_DT\_3\}) \cup \{s \mapsto RUN\_3\}$
  END

— Second cas :
  — Si l'erreur (faute, panne, etc) ayant conduit un service s dans un état illégal est réelle, alors la gravité de celle-ci sera fatalement évalué (**self-activation**). L'événement FAIL_ACTIV modélise cette phase : le service s passe de la détection d'un état illégal (FL_DT_3) ($grd2$) à l'évaluation de l'erreur ayant causé cet état illégal ($act1$), pour voir si celle-ci est critique ou non.

EVENT FAIL_ACTIV REFINES FAIL_ACTIV $\widehat{=}$
  ANY
    s
  WHERE
    $grd1$: $s \in SERVICES$
    $grd2$: $s \mapsto FL\_DT\_3 \in serviceState\_3$
  THEN
    $act1$: $serviceState\_3 := (serviceState\_3 \setminus \{s \mapsto FL\_DT\_3\}) \cup \{s \mapsto FL\_ACT\_3\}$
  END

  — L'événement FAIL_CF_IGN (raffinement de HEAL_2) exprime que l'évaluation d'une erreur ayant causé l'état illégal d'un service s ($grd3$) produit fatalement un des résultats suivant ($grd2$, $act1$) : soit l'erreur est critique, auquel cas, le service s entre dans une phase de configuration (FL_CONF_3), soit l'erreur est bénigne et ne gène pas le fonctionnement du service, auquel cas, elle est ignorée (FL_IGN_3).

EVENT FAIL_CF_IGN REFINES HEAL_2 $\widehat{=}$
  ANY
    s, st
  WHERE
    $grd1$: $s \in SERVICES$
    $grd2$: $st \in \{FL\_CONF\_3, FL\_IGN\_3\}$
    $grd3$: $s \mapsto FL\_ACT\_3 \in serviceState\_3$
  THEN
    $act1$: $serviceState\_3 := (serviceState\_3 \setminus \{s \mapsto FL\_ACT\_3\}) \cup \{s \mapsto st\}$
  END

  — Un nouvel événement HEAL_3 exprime qu'un service s dans une phase de configuration ou ayant ignoré une erreur ($grd2$, $grd3$) revient fatalement dans un état légal ($act1$).

EVENT HEAL_3 $\widehat{=}$
  ANY
    s, st
  WHERE
    $grd1$: $s \in SERVICES$
    $grd2$: $st \in \{FL\_CONF\_3, FL\_IGN\_3\}$
    $grd3$: $s \mapsto st \in serviceState\_3$
  THEN
    $act1$: $serviceState\_3 := (serviceState\_3 \setminus \{s \mapsto st\}) \cup \{s \mapsto RUN\_3\}$
  END

— L'événement FAILURE est raffiné, de manière à prendre en compte les nouvelles variables et nouveaux états :

EVENT FAILURE REFINES FAILURE $\widehat{=}$
  ...
  WHERE
    $\ominus$ $grd2$: $s \mapsto RUN\_2 \in serviceState\_2$
    $\oplus$ $grd2$: $s \mapsto RUN\_3 \in serviceState\_3$
  THEN
    $\ominus$ $act1$: ...
    $\oplus$ $act1$: $serviceState\_3 := (serviceState\_3 \setminus \{s \mapsto RUN\_3\}) \cup \{s \mapsto FAIL\_3\}$
  END

Ce troisième raffinement peut se résumer comme suit :

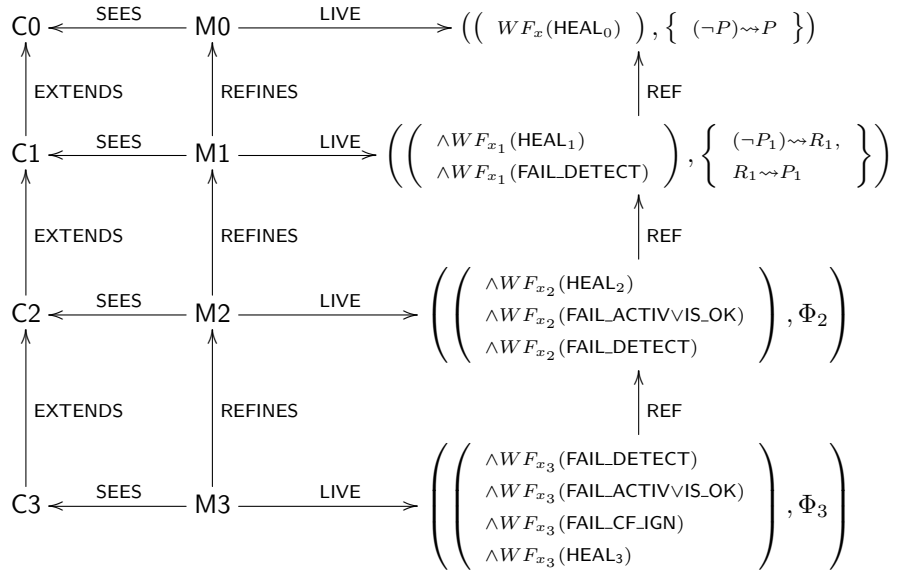


FIGURE 8.8 – Troisième raffinement du self-healing

La prochaine sous-section décrit le comportement global du système P2P.

8.3.3.4 Comportement global abstrait du système

Nous avons identifié dans l'abstraction et les trois premiers raffinements précédents les différentes phases de l'algorithme de « self-healing », à savoir, la phase de **self-detection**, celle de **self-activation** et celle de **self-configuration**. Ce quatrième raffinement nous permet de décomposer et de détailler encore plus ces phases abstraites, notamment celles de **self-activation** et celle de **self-configuration**. Nous obtenons ainsi un modèle décrivant de manière abstraite le comportement de l'algorithme de « self-healing », à un niveau global, qui est celui des services : nous ne focalisons pas dans ce raffinement sur les processus/pairs fournissant les services, nous ne nous intéressons pas encore à leurs interactions, états, etc pouvant influer sur les états des services. Un contexte C4, extension du précédent contexte C3, introduit des éléments (états, etc) relatifs au comportement global de la procédure de « self-healing ».

CONTEXT C4 EXTENDS C3
SETS
$STATES\_4$
CONSTANTS
$RUN\_4, FAIL\_4, FL\_DT\_4, FL\_ACT\_4, FL\_CONF\_4, FL\_IGN\_4, DPL\_4, InitState\_4$
AXIOMS
$axm1 : partition\left(STATES\_4, \begin{array}{l} \{RUN\_4\}\{FAIL\_4\}, \{FL\_DT\_4\}, \{DPL\_4\} \\ \{FL\_ACT\_4\}, \{FL\_CONF\_4\}, \{FL\_IGN\_4\} \end{array} \right)$
$axm2 : InitState\_4 \in SERVICES \rightarrow STATES\_4$
$axm3 : \forall s \cdot s \in SERVICES \Rightarrow s \mapsto RUN\_4 \in InitState\_4$
$axm4 : \forall s, st1, st2 \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st1 \in STATES\_4 \\ \wedge st2 \in STATES\_4 \\ \wedge s \mapsto st1 \in InitState\_4 \\ \wedge s \mapsto st2 \in InitState\_4 \end{array} \right) \Rightarrow st1 = st2$
END

— $axm1$: un ensemble $STATES\_4$ d'états possibles pour chaque service est défini. Il est composé de six états distincts :
  — RUN_4 représente un état de fonctionnement légal d'un service.
  — $FAIL\_4$ représente un état illégal, consécutif à une faute.
  — FL_DT_4 représente la détection d'un état illégal.
  — FL_ACT_4 représente l'évaluation de la gravité de la cause (erreurs, fautes, pannes, etc) d'un état illégal.

— FL_IGN_4 représente le fait qu'une erreur ayant provoqué un état illégal est ignorée, car non-critique.
— FL_CONF_4 représente la configuration d'un service dans un état illégal pour un retour dans un état légal.
— DPL_4 représente le redéploiement d'un service lors de sa configuration pour un retour dans un état légal.

Ce nouvel ensemble d'états $STATES\_4$ nous servira à remplacer l'ensemble $STATES\_4$ du précédent raffinement.

— $axm2 : InitState\_4$ est une constante associant à chaque service un état de l'ensemble $STATES\_4$. Elle nous permettra de décrire les états initiaux des services fournis par le système P2P.
— $axm3$: l'état initial de chaque service est RUN_4.
— $axm4$: chaque service a un état initial unique.

Une machine M4, raffinant la machine M3 précédente, utilise ce contexte C4.

Cette machine M4 introduit une nouvelle variable $serviceState\_4$, remplaçant la variable $serviceState\_3$: cette nouvelle variable nous permet de prendre en compte le nouvel état DPL_4. L'état initial de cette variable est défini à l'aide de la constante $InitState\_4$: à l'état initial, tous les services sont dans l'état légal RUN_4.

$$\begin{array}{l}\textsf{INITIALISATION} \ \widehat{=} \\ \quad \textsf{BEGIN} \\ \quad\quad \ominus\ act1 \ : \ serviceState\_3 := InitState\_3 \\ \quad\quad \oplus\ act1 \ : \ serviceState\_4 := InitState\_4 \\ \quad \textsf{END}\end{array}$$

Nous notons x_4 l'ensemble des variables de cette machine M4.

Un invariant noté $I_4(x_4)$ caractérise ces variables et est défini comme suit :

— Nous avons du typage.

$$inv1 \ : \ serviceState\_4 \in SERVICES \rightarrow STATES\_4$$

La variable $serviceState\_4$ est une relation associant des services et des états.

— Nous avons des propriétés de sûreté.

$$inv2 \ : \ \forall s, st1, st2 \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st1 \in STATES\_4 \\ \wedge st2 \in STATES\_4 \\ \wedge s \mapsto st1 \in serviceState\_4 \\ \wedge s \mapsto st2 \in serviceState\_4 \end{array} \right) \Rightarrow st1 = st2$$

Cette propriété exprime l'unicité de l'état courant d'un service s.

— Nous avons des invariants de collage qui définissent les relations entre les variables concrète $serviceState\_4$ et abstraite $serviceState\_3$, ainsi qu'entre les états concrets RUN_4, $FAIL\_4$, FL_DT_4, FL_ACT_4, FL_CONF_4, FL_IGN_4, DPL_4 et états abstraits RUN_3, $FAIL\_3$, FL_DT_3, FL_ACT_3, FL_CONF_3, FL_IGN_3.

$$\begin{array}{l} inv3 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge s \mapsto RUN\_3 \in serviceState\_3 \end{array} \right) \Rightarrow \left(\begin{array}{l} \vee s \mapsto RUN\_4 \in serviceState\_4 \\ \vee s \mapsto DPL\_4 \in serviceState\_4 \end{array} \right) \\ inv4 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge \left(\begin{array}{l} \vee s \mapsto RUN\_4 \in serviceState\_4 \\ \vee s \mapsto DPL\_4 \in serviceState\_4 \end{array} \right) \end{array} \right) \Rightarrow s \mapsto RUN\_3 \in serviceState\_3 \\ inv5 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_ACT\_3 \in serviceState\_3 \Rightarrow s \mapsto FL\_ACT\_4 \in serviceState\_4 \\ inv6 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_ACT\_4 \in serviceState\_4 \Rightarrow s \mapsto FL\_ACT\_3 \in serviceState\_3 \\ inv7 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_DT\_3 \in serviceState\_3 \Rightarrow s \mapsto FL\_DT\_4 \in serviceState\_4 \\ inv8 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_DT\_4 \in serviceState\_4 \Rightarrow s \mapsto FL\_DT\_3 \in serviceState\_3 \\ inv9 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_IGN\_3 \in serviceState\_3 \Rightarrow s \mapsto FL\_IGN\_4 \in serviceState\_4 \\ inv10 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_IGN\_4 \in serviceState\_4 \Rightarrow s \mapsto FL\_IGN\_3 \in serviceState\_3 \\ inv11 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_CONF\_3 \in serviceState\_3 \Rightarrow s \mapsto FL\_CONF\_4 \in serviceState\_4 \\ inv12 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_CONF\_4 \in serviceState\_4 \Rightarrow s \mapsto FL\_CONF\_3 \in serviceState\_3 \\ inv13 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FAIL\_3 \in serviceState\_3 \Rightarrow s \mapsto FAIL\_4 \in serviceState\_4 \\ inv14 \ : \ \forall s \cdot s \in SERVICES \wedge s \mapsto FAIL\_4 \in serviceState\_4 \Rightarrow s \mapsto FAIL\_3 \in serviceState\_3 \end{array}$$

— Les propriétés $inv3$ et $inv4$ expriment que l'état abstrait RUN_3 d'un service s est raffiné en deux états concrets : un état RUN_4 et un état DPL_4.
— Les propriétés $inv5$ et $inv6$ expriment que l'état abstrait FL_ACT_3 d'un service s est raffiné en un état concret FL_ACT_4.
— Les propriétés $inv7$ et $inv8$ expriment que l'état abstrait FL_DT_3 d'un service s est raffiné en un état concret FL_DT_4.

— Les propriétés $inv9$ et $inv10$ expriment que l'état abstrait FL_IGN_3 d'un service s est raffiné en un état concret FL_IGN_4.
— Les propriétés $inv11$ et $inv12$ expriment que l'état abstrait FL_CONF_3 d'un service s est raffiné en un état concret FL_CONF_4.
— Les propriétés $inv13$ et $inv14$ expriment que l'état abstrait $FAIL\_3$ d'un service s est raffiné en un état concret $FAIL\_4$.

Des événements, que nous détaillerons plus bas, sont présents dans cette machine M4 : FAILURE, FAIL_DETECT, FAIL_ACTIV, IS_OK, FAIL_IGN, IGNORE, FAIL_CONFIGURE, REDEPLOY, HEAL_4. Nous nous servons ensuite d'une liste Φ_4 de propriétés de vivacité, détaillée en annexe, pour raffiner M3 en M4 :

1. **Self-detection**

— Un service s détecte ($act1$) fatalement un état illégal ($grd2$). Ce mécanisme est modélisé par l'événement FAIL_DETECT.

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FAIL\_4 \in serviceState\_4$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FAIL\_4\}) \cup \{s \mapsto FL\_DT\_4\}$
END

— L'état illégal détecté du service s ($grd2$) est une fausse alerte ; le service s revient alors fatalement à un état légal RUN_4 ($act1$). L'événement IS_OK modélise ce mécanisme.

EVENT IS_OK REFINES IS_OK $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_DT\_4 \in serviceState\_4$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_DT\_4\}) \cup \{s \mapsto RUN\_4\}$
END

2. **Self-activation**

— L'erreur détectée ($grd2$) ayant causé l'état illégal du service s est réelle ; elle est alors fatalement évaluée ($act1$), pour que le service puisse savoir si elle est critique ou non. Ce mécanisme d'évaluation est modélisé par l'événement FAIL_ACTIV.

EVENT FAIL_ACTIV REFINES FAIL_ACTIV $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_DT\_4 \in serviceState\_4$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_DT\_4\}) \cup \{s \mapsto FL\_ACT\_4\}$
END

— L'erreur ayant causé l'état illégal du service s est évaluée ($grd2$) comme n'étant pas critique et ne perturbant pas le fonctionnement du service ; l'erreur est donc fatalement ignorée ($act1$). Ce mécanisme est modélisé par l'événement FAIL_IGN.

EVENT FAIL_IGN REFINES FAIL_CF_IGN $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_ACT\_4 \in serviceState\_4$
WITH
st : $st = FL\_IGN\_3$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_ACT\_4\}) \cup \{s \mapsto FL\_IGN\_4\}$
END

— Une erreur non-critique ($grd2$) d'un service s est fatalement ignorée ($act1$). L'événement IGNORE modélise ce comportement permettant d'ignorer une erreur bénigne.

EVENT IGNORE REFINES HEAL_3 $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_IGN\_4 \in serviceState\_4$
WITH
st : $st = FL\_IGN\_3$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_IGN\_4\}) \cup \{s \mapsto RUN\_4\}$
END

3. **Self-configuration**

— L'erreur ayant causé l'état illégal du service s est évaluée ($grd2$) comme étant critique et empêchant probablement le fonctionnement correct de ce dernier ; le service s entre alors fatalement dans une phase de **self-configuration** ($act1$). L'événement FAIL_CONF modélise cette entrée d'un service dans une phase de **self-configuration**.

EVENT FAIL_CONF REFINES FAIL_CF_IGN $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_ACT\_4 \in serviceState\_4$
WITH
st : $st = FL\_CONF\_3$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_ACT\_4\}) \cup \{s \mapsto FL\_CONF\_4\}$
END

— Le service s, dans un état illégal causé par une erreur critique et entré dans une phase de **self-configuration** ($grd2$), est fatalement redéployé ($act1$). L'événement REDEPLOY modélise cette procédure de redéploiement de manière abstraite.

EVENT REDEPLOY REFINES HEAL_3 $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_CONF\_4 \in serviceState\_4$
WITH
st : $st = FL\_CONF\_3$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_CONF\_4\}) \cup \{s \mapsto DPL\_4\}$
END

— Après son redéploiement ($grd2$), le service s revient fatalement à un état légal ($act1$). L'événement HEAL_4 modélise ce mécanisme de retour à un état légal.

EVENT HEAL_4 $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto DPL\_4 \in serviceState\_4$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto DPL\_4\}) \cup \{s \mapsto RUN\_4\}$
END

— L'événement FAILURE est raffiné, de manière à prendre en compte les nouvelles variables et nouveaux états :

EVENT FAILURE REFINES FAILURE $\widehat{=}$
...
WHERE
$\ominus$ $grd2$: $s \mapsto RUN\_3 \in serviceState\_3$
$\oplus$ $grd2$: $s \mapsto RUN\_4 \in serviceState\_4$
THEN
$\ominus$ $act1$: ...
$\oplus$ $act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto RUN\_4\}) \cup \{s \mapsto FAIL\_4\}$
END

Ce quatrième raffinement peut se résumer comme suit :

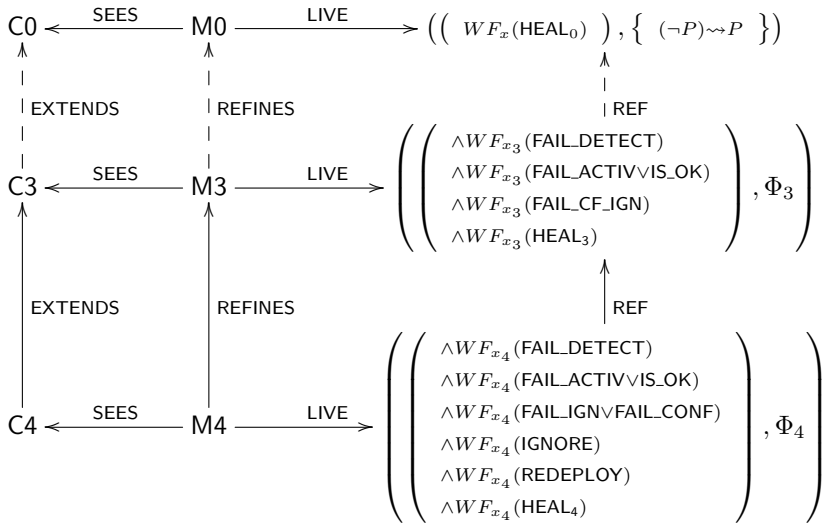


FIGURE 8.9 – Quatrième raffinement du self-healing

Les prochaines sous-sections introduisent les pairs/processus du système P2P fournissant les services.

8.3.4 Détails et début de localisation des phases

8.3.4.1 Pairs instances d'un service

Nous introduisons dans ce cinquième raffinement l'idée qu'un service est fourni par des pairs/processus, aussi appelés *instances* du service. Nous ne représentons pas encore concrètement les instances d'un service dans ce raffinement : nous caractérisons un service par le nombre d'instances qui le fournissent, sans donner des informations, détails sur ces instances. Nous utilisons le nombre d'instances par service pour expliquer notamment les phases de **self-detection**, **self-activation**, **self-configuration** et la sous-phase de redéploiement d'un service. Nous commençons par définir un contexte C5, raffinant le contexte C4 vu dans la sous-section précédente, prenant en compte le nombre d'instances fournissant un service.

CONTEXT C5 EXTENDS C4
CONSTANTS
$min\_inst, init\_inst$
AXIOMS
$axm1 : min\_inst \in SERVICES \rightarrow \mathbb{N}_1$
$axm2 : init\_inst \in SERVICES \rightarrow \mathbb{N}_1$
$axm3 : \forall s \cdot s \in SERVICES \Rightarrow min\_inst(s) \geq 2$
$axm4 : \forall s \cdot s \in SERVICES \Rightarrow init\_inst(s) \geq min\_inst(s)$
$axm5 : \forall s \cdot s \in SERVICES \Rightarrow init\_inst(s) \geq 2$
END

— $axm1$: nous définissons une constante min_inst. Cette constante associe à chaque service le nombre d'instances minimal (entier positif, non nul) requis pour le fonctionnement correct du service.
— $axm2$: nous définissons une constante $init\_inst$. Cette constante associe à chaque service le nombre initial d'instances (entier positif, non nul) fournissant le service.
— $axm3$ et $axm5$ expriment que le nombre d'instances respectivement minimal et initial par service est supérieur ou égal à 2.
— $axm4$ exprime que le nombre initial d'instance pour chaque service s est supérieur ou égal au nombre minimal d'instances du service.

Nous définissons ensuite une machine M5, raffinant la machine M4 précédente et utilisant ce contexte C5. Nous y introduisons deux nouvelles variables :

INITIALISATION $\widehat{=}$
BEGIN
...
$\oplus$ $act2 : num\_run := init\_inst$
$\oplus$ $act3 : num\_susp := SERVICES \times \{0\}$
END

— num_run est une fonction totale associant à chaque service le nombre courant d'instances fournissant le service.
— num_susp est une fonction totale associant à chaque service le nombre d'instance suspectées d'être dans un état illégal.

Nous notons x_5 l'ensemble des variables de M5.

L'invariant $I_5(x_5)$ caractérisant ces variables est défini comme suit :
— Nous avons des propriétés de typage :

$inv1 : num\_run \in SERVICES \rightarrow \mathbb{N}_1$
$inv2 : num\_susp \in SERVICES \rightarrow \mathbb{N}_1$

— Nous avons des propriétés de sûreté :

$$inv3 : \forall s, st \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st \in STATES\_4 \\ \wedge st \notin \{FAIL\_4, FL\_DT\_4\} \\ \wedge s \mapsto st \in serviceState\_4 \end{array} \right) \Rightarrow num\_susp(s) = 0$$
$$inv4 : \forall s \cdot s \in SERVICES \wedge s \mapsto RUN\_4 \in serviceState\_4 \Rightarrow num\_susp(s) = 0$$
$$inv5 : \forall s \cdot s \in SERVICES \wedge s \mapsto FL\_CONF\_4 \in serviceState\_4 \Rightarrow num\_run(s) < min\_inst(s)$$
$$inv6 : \forall s \cdot s \in SERVICES \Rightarrow num\_susp(s) < num\_run(s)$$

— $inv3$ exprime que si un service ne se trouve pas dans un état illégal ($FAIL\_4$) ou dans un état dans lequel il a détecté (FL_DT_4) un état illégal, le nombre d'instances suspectes (suspectées d'être dans un état illégal) est nul.
— $inv4$ est un théorème qui exprime que si un service s fonctionne correctement dans un état légal (RUN_4), le nombre d'instances suspectes (suspectées d'être défaillantes / dans un état illégal) est nul.
— $inv5$ exprime que si un service s doit être configuré (FL_CONF_4), cela signifie que la faute à l'origine de l'état illégal du service est critique : le nombre d'instances fournissant le service est inférieur au nombre minimal d'instances requis pour fournir correctement le service.

— $inv6$ exprime que le nombre d'instances suspectes d'un service s est toujours inférieur au nombre d'instances. Il s'agit d'une hypothèse que nous posons sur les services : un service ne peut jamais être complètement défaillant, un certain nombre d'instances doit toujours rester fonctionnelles, pour activer la procédure de *self-healing*.

Cette machine M5 contient les événements suivants : FAILURE, FAIL_DETECT, IS_OK, FAIL_ACTIV, IGNORE, FAIL_IGN, FAIL_CONF, REDEPLOY et HEAL_4. Une liste Φ_5 de propriétés de vivacité, détaillée en annexe, est utilisée pour construire le système d'événements de M5. Nous ne détaillons que les événements qui ont changé entre les machines M4 et M5 :

— Nous raffinons l'événement FAILURE, en y ajoutant un paramètre entier nb_fail, qui représente le nombre d'instances suspectées d'être défaillantes/dans un état illégal d'un service s ($act2$). Nous ajoutons une hypothèse sur ce nombre d'instances suspectes : il doit être inférieur au nombre d'instances fonctionnelles courant du service s ($grd4$).

EVENT FAILURE REFINES FAILURE $\widehat{=}$
ANY
...
$\oplus nb\_fail$
WHERE
...
$\oplus$ $grd3$: $nb\_fail \in \mathbb{N}_1$
$\oplus$ $grd4$: $nb\_fail < num\_run(s)$
THEN
...
$\oplus$ $act2$: $num\_susp(s) := nb\_fail$
END

— L'événement FAIL_DETECT est observé lorsque le nombre d'instances suspectes d'un service s est supérieur à 0. La procédure de **self-detection** détermine parmi les instances suspectes lesquels sont dans un état légal. Le nombre de ces instances suspectes dans un état légal est indiqué dans un entier num_safe ($grd3$, $grd4$). Les instances suspectes restantes ($act2$) sont celles qui sont vraiment dans un état illégal.

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
ANY
...
$\oplus num\_safe$
WHERE
...
$\oplus$ $grd3$: $num\_safe \in \mathbb{N}_1$
$\oplus$ $grd4$: $num\_safe \leq num\_susp(s)$
THEN
...
$\oplus$ $act2$: $num\_susp(s) := num\_susp(s) - num\_safe$
END

— L'événement IS_OK modélise le cas d'une fausse alerte : après la procédure de **self-detection**, le nombre d'instances suspectes d'un service s est de 0. Le service s peut donc revenir dans un état légal.

EVENT IS_OK REFINES IS_OK $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd3$: $num\_susp(s) = 0$
THEN
...
END

— L'événement FAIL_ACTIV modélise abstraitement la **self-activation** : le nombre d'instances d'un service s suspectes et qui sont vraiment dans un état illégal est non nul ($grd3$). Ce nombre d'instances défaillantes est donc soustrait du nombres d'instances fonctionnelles du service s ($act2$) et les instances défaillantes ne sont plus considérées comme des instances suspectes, mais comme des instances dans un état illégal ($act3$).

EVENT FAIL_ACTIV REFINES FAIL_ACTIV $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd3$: $num\_susp(s) > 0$
THEN
...
$\oplus$ $act2$: $num\_run(s) := num\_run(s) - num\_susp(s)$
$\oplus$ $act3$: $num\_susp(s) := 0$
END

— L'événement FAIL_IGN est observé lorsqu'un service s possède des instances dans un état illégal, et lorsque le nombre d'instances fonctionnelles du service s est supérieur ou égal au minimum requis pour fournir le service ($grd3$).

EVENT FAIL_IGN REFINES FAIL_IGN $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd3$: $num\_run(s) \geq min\_inst(s)$
THEN
...
END

— L'événement FAIL_CONF est observé lorsqu'un service s possède des instances dans un état illégal, et lorsque le nombre d'instances fonctionnelles du service s est inférieur au minimum requis pour fournir le service ($grd3$) : le service s entre alors dans une phase de *self-configuration* et va dans l'étape suivante, redéployer le service s en activant de nouvelles instances de ce dernier.

EVENT FAIL_CONF REFINES FAIL_CONF $\widehat{=}$
ANY
...
WHERE
...
$\oplus$ $grd3$: $num\_run(s) < min\_inst(s)$
THEN
...
END

— L'événement REDEPLOY modélise le redéploiement des instances d'un service s, entré dans un état illégal : de nouvelles instances du service s sont activées jusqu'à ce que le nombre total d'instances fonctionnelles fournissant le service s soit supérieur ou égal au nombre minimal d'instances requis pour fournir le service s ($grd4$, $act2$).

EVENT REDEPLOY REFINES REDEPLOY $\widehat{=}$
...
WHERE
...
$\oplus$ $grd3$: $new\_run \in \mathbb{N}_1$
$\oplus$ $grd4$: $new\_run \geq min\_inst(s)$
THEN
...
$\oplus$ $act2$: $num\_run(s) := new\_run$
END

Ce cinquième raffinement peut se résumer à l'aide du diagramme suivant :

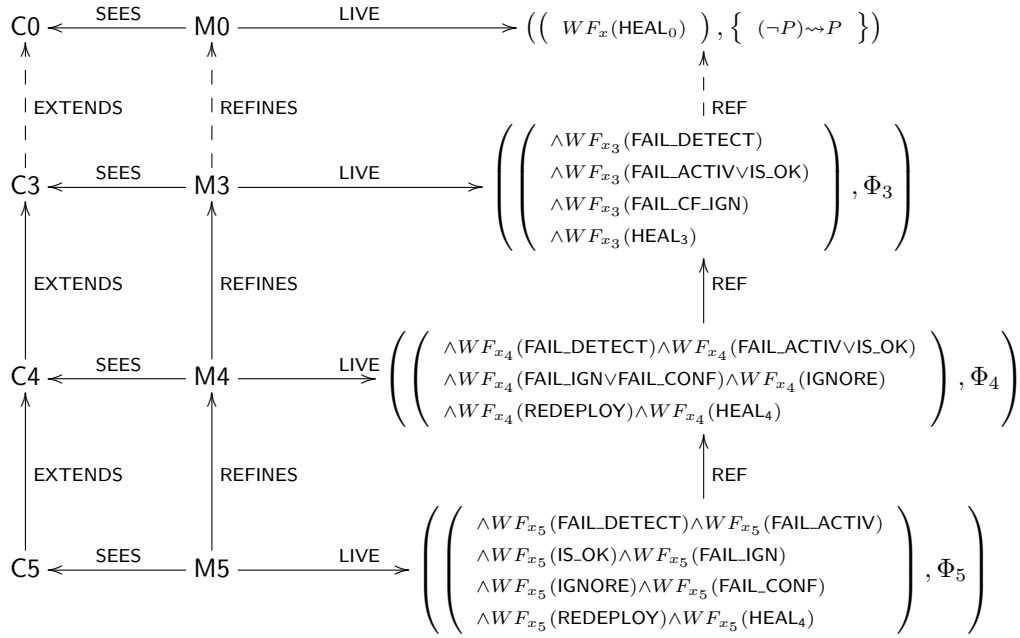


FIGURE 8.10 – Cinquième raffinement du self-healing

Ce cinquième raffinement centre les différentes phases de la procédure de *self-healing* d'un service sur le nombre des instances fonctionnelles, dans un état légal de ce dernier. En effet, les variations sur ce nombre permettent de déclencher selon les cas, les phases de **self-detection**, de **self-activation** et de **self-configuration** ; l'objectif de la procédure de *self-healing* d'un service décrit dans ce modèle peut se résumer de la manière suivante : restaurer le nombre d'instances du service, pour atteindre (ou dépasser) le nombre minimal requis pour fournir le service.

8.3.4.2 Groupe de pairs par service

Ce sixième raffinement introduit les groupes de pairs/processus fournissant les services. Nous commençons par définir un contexte C6 étendant le contexte C5 vu précédemment et introduisant les groupes de pairs pour chaque service, ainsi que quelques notions sur les ensembles :

CONTEXT C6 EXTENDS C5
SETS
$PEERS$
CONSTANTS
$InitSrvcPeers$
AXIOMS
$axm1 : InitSrvcPeers \in SERVICES \to \mathbb{P}_1(PEERS)$
$axm2 : \forall s \cdot s \in SERVICES \Rightarrow finite(InitSrvcPeers(s))$
$axm3 : \forall s \cdot s \in SERVICES \Rightarrow card(InitSrvcPeers(s)) = init\_inst(s)$
$axm4 : \forall s1, s2 \cdot \begin{pmatrix} \wedge s1 \subseteq PEERS \\ \wedge s2 \subseteq PEERS \\ \wedge s1 \neq \varnothing \\ \wedge s2 \neq \varnothing \\ \wedge finite(s1) \\ \wedge finite(s2) \\ s1 \subset s2 \end{pmatrix} \Rightarrow card(s1) \leq card(s2) - 1$
$axm5 : \forall s1 \cdot s1 \subseteq PEERS \wedge s1 \neq \varnothing \wedge finite(s1) \Rightarrow card(s1) > 0$
$axm6 : \forall s1, s2 \cdot \begin{pmatrix} \wedge s1 \subseteq PEERS \\ \wedge s2 \subseteq PEERS \\ \wedge finite(s1) \\ \wedge finite(s2) \\ \wedge s1 \subseteq s2 \end{pmatrix} \Rightarrow card(s2) - card(s1) = card(s2 \setminus s1)$
END

— Nous définissons un ensemble $PEERS$ qui contient les pairs/processus composant le système P2P.

— Nous introduisons une constante $InitSrvcPeers$ associant chaque service à un ensemble non-vide ($axm1$), fini ($axm2$) de pairs.
— $axm3$ exprime que le nombre de pairs pour chaque service s est donné par $init\_inst(s)$, représentant le nombre initial de pairs pour chaque service s.
— $axm4$ exprime que si deux ensembles de pairs $s1$ et $s2$ sont non-vides et finis, et que $s1$ est strictement inclus dans $s2$, alors le nombre d'éléments dans $s1$ est plus petit que le nombre d'éléments dans $s2$.
— $axm5$ exprime que si un ensemble $s1$ de pairs est non-vide et fini, alors le nombre d'éléments contenus dans $s1$ est non nul.
— $axm6$ exprime que si deux ensembles de pairs $s1$ et $s2$ sont non-vides et finis, et que $s1$ est inclus dans $s2$, alors le nombre donné par $card(s2) - card(s1)$ est égal au nombre d'éléments contenus dans l'ensemble $s2 \setminus s1$.

Une machine M6, raffinant la machine M5 vue précédemment, utilise ce contexte C6. Nous y supprimons des variables, telles que *num_run* et *num_susp*, indiquant respectivement le nombre de pairs dans un état légal fournissant un service et le nombre d'instances d'un service suspectés d'être défaillantes. De nouvelles variables y sont introduites :

INITIALISATION $\widehat{=}$
BEGIN
$\ominus$ $act2$: $num\_run := init\_inst$
$\ominus$ $act3$: $num\_susp := SERVICES \times \{0\}$
$\oplus$ $act2$: $run\_peers := InitSrvcPeers$
$\oplus$ $act3$: $susp\_peers := \varnothing$
$\oplus$ $act4$: $fail\_peers := \varnothing$
END

— run_peers est une fonction associant à chaque service un ensemble non-vide de pairs le fournissant. Cette variable est initialisée à l'aide de la constante $InitSrvcPeers$ et elle remplace celle abstraite num_run.
— $susp\_peers$ est une fonction associant à chaque service un ensemble d'instances suspectées d'être dans un état illégal. Cette variable est initialisée à l'aide de l'ensemble vide ($\varnothing$), parce qu'à l'état initial aucune instance d'un service n'est suspecte, et elle remplace la variable abstraite num_susp.
— $fail\_peers$ est une relation associant des services à des instances défaillantes, dans un état illégal. Cette variable est initialisée à l'aide de l'ensemble vide ($\varnothing$), parce qu'à l'état initial aucune instance d'un service n'est dans un état illégal.

Nous notons x_6 l'ensemble des variables de cette machine M6.

Un invariant noté $I_6(x_6)$ caractérise ces variables et est défini comme suit :
— Nous avons des propriétés de typage :

$inv1$: $run\_peers \in SERVICES \rightarrow \mathbb{P}_1(PEERS)$
$inv2$: $susp\_peers \in SERVICES \nrightarrow \mathbb{P}(PEERS)$
$inv3$: $fail\_peers \in SERVICES \leftrightarrow PEERS$

— Des invariants de collage permettent de remplacer les variables abstraites num_run et num_susp par celles plus concrètes run_peers et $susp\_peers$:

$inv4$: $\forall s \cdot s \in SERVICES \Rightarrow finite(run\_peers(s))$
$inv5$: $\forall s \cdot s \in SERVICES \Rightarrow num\_run(s) = card(run\_peers(s))$
$inv6$: $\forall s \cdot s \in SERVICES \wedge s \in dom(susp\_peers) \Rightarrow finite(susp\_peers(s))$
$inv7$: $\forall s \cdot s \in SERVICES \wedge s \in dom(susp\_peers) \Rightarrow num\_susp(s) = card(susp\_peers(s))$

— L'invariant $inv4$ exprime que le nombre d'instances fournissant un service s est fini ; et $inv5$ exprime que ce nombre d'instances d'un service s est égal à $num\_run(s)$.
— L'invariant $inv6$ exprime que le nombre d'instances suspectes d'un service s est fini ; et $inv7$ exprime que ce nombre d'instances d'un service s est égal à $num\_susp(s)$.
— D'autres propriétés de sûreté et d'invariance sont aussi présentes dans cette machine :

$$
\begin{array}{l}
inv8 \ : \ \forall s \cdot s \in SERVICES \Rightarrow run\_peers(s) \cap fail\_peers[\{s\}] = \varnothing \\
inv9 \ : \ \forall s \cdot s \in SERVICES \wedge s \in dom(susp\_peers) \Rightarrow susp\_peers(s) \subseteq run\_peers(s) \\
inv10 \ : \ \forall s, st \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st \in STATES\_4 \\ \wedge st \in \{FAIL\_4, FL\_DT\_4\} \\ \wedge s \mapsto st \in serviceState\_4 \end{array} \right) \Rightarrow s \in dom(susp\_peers) \\
inv11 \ : \ \forall s, st \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st \in STATES\_4 \\ \wedge st \in \{FAIL\_4, FL\_DT\_4\} \\ \wedge s \mapsto st \in serviceState\_4 \end{array} \right) \Rightarrow susp\_peers(s) \subset run\_peers(s)
\end{array}
$$

— $inv8$ exprime l'intersection entre l'ensemble des instances fonctionnelles fournissant un service s ($run\_peers(s)$) et l'ensemble des instances défaillantes du même service s ($fail\_peers[\{s\}]$) est vide. Les instances fonctionnelles sont donc différentes de celles défaillantes.

— $inv9$ exprime que les instances d'un service s considérées comme suspectes, font, dans un premier temps, partie des instances fonctionnelles.

— $inv10$ et $inv11$ exprime qu'un service s dans un état illégal ou ayant détecté un état illégal maintient une liste d'instances suspectées d'être défaillantes et que cette liste est un sous-ensemble strict de l'ensemble des instances fonctionnelles fournissant le service s.

Cette machine M6 contient les événements suivants : FAILURE, FAIL_DETECT, IS_OK, FAIL_ACTIV, IGNORE, FAIL_IGN, FAIL_CONF, REDEPLOY, UNFAIL_PEER et HEAL_4. Une liste Φ_6 de propriétés de vivacité, détaillée en annexe, est utilisée pour définir les événements de M6. Nous ne détaillons que les événements qui ont changé entre les machines M5 et M6, ainsi que le nouvel événement UNFAIL_PEER :

— Nous raffinons l'événement FAILURE : nous remplaçons le paramètre numérique nb_fail par un paramètre ensembliste fp, qui contient les instances d'un services s devenues suspectes. Le service s entre dans un état illégal ($act1$) quand un sous-ensemble non-vide d'instances auparavant fonctionnelles ($grd3$, $grd4$), semblent devenir défaillantes et deviennent ainsi suspectes ($act2$).

EVENT FAILURE REFINES FAILURE $\widehat{=}$
ANY
s, fp
WHERE
$grd1 \ : \ s \in SERVICES$
$grd2 \ : \ s \mapsto RUN\_4 \in serviceState\_4$
$grd3 \ : \ fp \subseteq PEERS$
$grd4 \ : \ fp \neq \varnothing$
$grd5 \ : \ fp \subset run\_peers(s)$
WITH
$nb\_fail \ : \ nb\_fail = card(fp)$
THEN
$act1 \ : \ serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto RUN\_4\}) \cup \{s \mapsto FAIL\_4\}$
$act2 \ : \ susp\_peers(s) := fp$
END

— L'événement FAIL_DETECT est observé lorsque l'ensemble des instances suspectes d'un service s n'est pas vide ($grd3$). La phase de **self-detection** permet ensuite au service s de déterminer un sous-ensemble sf des instances suspectes ($grd5$) qui sont dans un état légal et de retirer ce sous-ensemble de l'ensemble des instances suspectes ($act2$).

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
ANY
s, sf
WHERE
$grd1 \ : \ s \in SERVICES$
$grd2 \ : \ s \mapsto FAIL\_4 \in serviceState\_4$
$grd3 \ : \ susp\_peers(s) \neq \varnothing$
$grd4 \ : \ sf \subseteq PEERS$
$grd5 \ : \ sf \subseteq susp\_peers(s)$
WITH
$num\_safe \ : \ num\_safe = card(sf)$
THEN
$act1 \ : \ serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FAIL\_4\}) \cup \{s \mapsto FL\_DT\_4\}$
$act2 \ : \ susp\_peers(s) := susp\_peers(s) \setminus sf$
END

— L'événement IS_OK modélise le cas d'une fausse alerte pour un service s : l'ensemble des instances suspectes est vide. Le service peut donc revenir à un état légal.

EVENT IS_OK REFINES IS_OK $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_DT\_4 \in serviceState\_4$
$grd3$: $susp\_peers(s) = \varnothing$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_DT\_4\}) \cup \{s \mapsto RUN\_4\}$
END

— L'événement FAIL_ACTIV modélise le comportement d'un service s quand une erreur ayant causé un état illégal est réelle : si le nombre d'instances suspectes après la phase de **self-detection** est non-vide ($grd3$), alors, ces instances sont retirées de la liste des instances dans un état légal ($act2$) et sont considérées comme n'étant plus suspectes ($act3$), mais vraiment défaillantes ($act4$), dans un état illégal. La gravité de l'erreur est alors évaluée.

EVENT FAIL_ACTIV REFINES FAIL_ACTIV $\widehat{=}$
ANY
s WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_DT\_4 \in serviceState\_4$
$grd3$: $susp\_peers(s) \neq \varnothing$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_DT\_4\}) \cup \{s \mapsto FL\_ACT\_4\}$
$act2$: $run\_peers(s) := run\_peers(s) \setminus susp\_peers(s)$
$act3$: $susp\_peers(s) := \varnothing$
$act4$: $fail\_peers := fail\_peers \cup (\{s\} \times susp\_peers(s))$
END

— L'événement FAIL_IGN est observé lorsque le nombre d'instances pouvant fournir le service s est supérieur ou égal au minimum requis pour le service s ($grd3$). L'erreur à l'origine de l'état illégal du service s peut alors être ignorée ($act1$).

EVENT FAIL_IGN REFINES FAIL_IGN $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_ACT\_4 \in serviceState\_4$
$grd3$: $card(run\_peers(s)) \geq min\_inst(s)$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_ACT\_4\}) \cup \{s \mapsto FAIL\_IGN\_4\}$
END

— Dans le cas où l'erreur à l'origine de l'état illégal du service s est critique, c'est-à-dire lorsque d'instances pouvant fournir le service s est inférieur au minimum requis pour le service s ($grd3$), l'événement FAIL_CONF est observé, et le service entre dans la phase de **self-configuration** ($act1$) et attend son redéploiement ($act1$).

EVENT FAIL_CONF REFINES FAIL_CONF $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_ACT\_4 \in serviceState\_4$
$grd3$: $card(run\_peers(s)) < min\_inst(s)$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_ACT\_4\}) \cup \{s \mapsto FL\_CONF\_4\}$
END

— L'évenement REDEPLOY modélise le redéploiement des instances d'un service s : un ensemble de nouvelles instances ($grd7$) est déployé, de telle manière que le nombre d'instances fournissant le service s dépasse le minimum requis ($grd8$, $act2$). Le service peut ainsi revenir dans un état légal.

EVENT REDEPLOY REFINES REDEPLOY $\widehat{=}$
ANY
s, new_inst
WHERE
$grd1 : s \in SERVICES$
$grd2 : s \mapsto FL\_CONF\_4 \in serviceState\_4$
$grd3 : new\_inst \subseteq PEERS$
$grd4 : new\_inst \neq \varnothing$
$grd5 : finite(new\_inst)$
$grd6 : run\_peers(s) \cap new\_inst = \varnothing$
$grd7 : fail\_peers[\{s\}] \cap new\_inst = \varnothing$
$grd8 : card(run\_peers(s)) + card(new\_inst) \geq min\_inst(s)$
WITH
$new\_run : new\_run = card(run\_peers(s)) + card(new\_inst)$
THEN
$act1 : serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_CONF\_4\}) \cup \{s \mapsto DPL\_4\}$
$act2 : run\_peers(s) := run\_peers(s) \cup new\_inst$
END

— Nous avons aussi dans la machine M6 un événement de l'environnement UNFAIL_PEER, permettant de réparer ($act1$) un pair défaillant ($grd3$).

EVENT UNFAIL_PEER $\widehat{=}$
ANY
s, p
WHERE
$grd1 : s \in SERVICES$
$grd2 : p \in PEERS$
$grd3 : s \mapsto p \in fail\_peers$
THEN
$act1 : fail\_peers := fail\_peers \setminus \{s \mapsto p\}$
END

Ce sixième raffinement peut se résumer à l'aide du diagramme suivant :

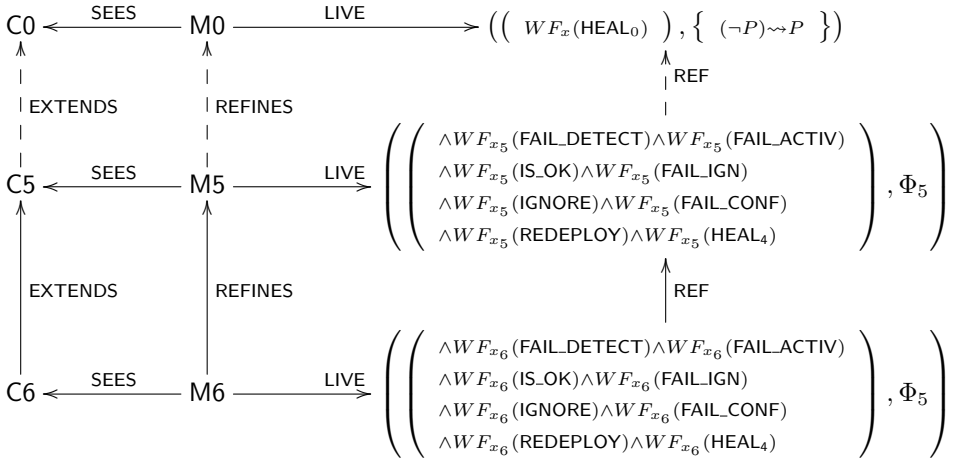


FIGURE 8.11 – Sixième raffinement du self-healing

Ce sixième raffinement introduit les pairs fournissant les services. Nous montrons dans ce raffinement que l'état d'un service est lié aux états des pairs qui le fournissent : par exemple, la défaillance d'un pair peut conduire à celle d'un service.

8.3.4.3 Redéploiement d'un service : abstraction

Ce septième raffinement présente une abstraction un peu plus détaillée de la procédure de redéploiement d'un service. Lors des précédents raffinements, cette procédure s'effectuait en un coup : l'observation d'un événement REDEPLOY modélisait le fait qu'un nombre suffisant d'instances d'un service étaient déployés pour que ce dernier atteigne le minimum requis pour fournir le service. Dans ce raffinement, des

instances d'un service dans un état illégal critique sont déployées, tant que le minimum n'est pas atteint, par une procédure inductive. Nous commençons dans un premier temps par définir un contexte C7, raffinant le contexte C6 du raffinement précédent et qui contient une constante *deplo_inst* :

CONTEXT C7 EXTENDS C6
CONSTANTS
deplo_inst
AXIOMS
$axm1$: $deplo\_inst \in SERVICES \rightarrow \mathbb{N}_1$
END

— $axm1$: la constante *deplo_inst* est une fonction qui associe à chaque service le nombre d'instances qu'il peut déployer en un coup. Ce sera le nombre d'instances à redéployer à chaque appel de la procédure inductive de redéploiement, tant que le nombre minimal d'instances nécessaires pour la fourniture du service n'est pas atteint ou dépassé.

Ce contexte C7 est utilisé par une machine M7, raffinant la machine M6 du raffinement précédent. Nous définissons dans cette machine M7 une nouvelle variables *dep_inst*, qui associe des services aux pairs (instances dans notre cas) que les services ont déployés ($inv1$), durant la phase de redéploiement.

INITIALISATION $\widehat{=}$
BEGIN
...
$\oplus$ $act5$: $dep\_inst := \varnothing$
END

Nous notons x_7 l'ensemble des variables de ce modèle M7.

Un invariant que nous notons $I_7(x_7)$ caractérise ces variables et est défini comme suit :

— Nous avons des propriétés de typage :

$inv1$: $dep\_inst \in SERVICES \leftrightarrow PEERS$

— Nous avons des propriétés de sûreté :

$$
\begin{array}{ll}
inv2 & : \forall s \cdot s \in SERVICES \Rightarrow dep\_inst[\{s\}] \cap fail\_peers[\{s\}] = \varnothing \\
inv3 & : \forall s, st \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge st \in STATES\_4 \\ \wedge s \mapsto st \in serviceState\_4 \\ \wedge st \neq FL\_CONF\_4 \end{array} \right) \Rightarrow dep\_inst[\{s\}] = \varnothing \\
inv4 & : \forall s \cdot s \in SERVICES \Rightarrow finite(dep\_inst[\{s\}]) \\
inv5 & : \forall s \cdot s \in SERVICES \Rightarrow dep\_inst[\{s\}] \cap run\_peers(s) = \varnothing
\end{array}
$$

— $inv2$ exprime que les instances déployées par un service s ne sont pas des pairs défaillants/dans un état illégal : ce sont des pairs fonctionnels qui sont déployés.
— $inv3$ exprime que de nouvelles instances d'un services sont déployées seulement durant la phase de **self-configuration** (qui inclue la phase de redéploiement) de ce dernier.
— $inv4$ exprime que l'ensemble des nouvelles instances qu'un service peut déployer en un coup est fini.
— $inv5$ exprime que les instances déployées par un service s ne sont pas des instances dans un état légal fournissant déjà le service s : il s'agit de pairs n'ayant pas encore fourni le service s.

Cette machine M7 rajoute, par rapport à la machine M6 précédente, un nouvel événement qui est l'événement REDEPLOY_INST. Nous nous servons des propriétés de vivacité d'une liste Φ_7, détaillée en annexe, pour définir les événements de M7. Nous ne détaillons que les nouveaux événements et ceux modifiés par raffinement :

— L'événement REDEPLOY_INST est un nouvel événement, modélisant le déploiement de nouvelles instances d'un service s. Durant la phase de reconfiguration d'un service s ($grd8$), un ensemble *dep* fini ($grd3$) de pairs ($grd2$), n'instanciant pas encore le service s ($grd4$), n'étant pas défaillantes ($grd5$) et contenant le nombre d'éléments que le service s peut déployer en un coup ($grd6$), est déployé par le service s ($act1$), tant que le nombre minimal d'instances requis pour fournir le service s n'est pas atteint ($grd7$).

EVENT REDEPLOY_INST $\widehat{=}$
ANY
s, dep
WHERE
$grd1 : s \in SERVICES$
$grd2 : dep \subseteq PEERS$
$grd3 : finite(dep)$
$grd4 : dep \cap run\_peers(s) = \varnothing$
$grd5 : dep \cap fail\_peers[\{s\}] = \varnothing$
$grd6 : card(dep) = deplo\_inst(s)$
$grd7 : card(dep\_inst[\{s\}]) + card(run\_peers(s)) < min\_inst(s)$
$grd8 : s \mapsto FL\_CONF\_4 \in serviceState\_4$
THEN
$act1 : dep\_inst := dep\_inst \cup (\{s\} \times dep)$
END

— L'événement REDEPLOY est observé lorsque le nombre d'instances dans un état légal, auquel on ajoute celles nouvellement déployées ($grd4$), d'un service s dépasse le minimum requis pour fournir le service s, c'est-à-dire lorsque la procédure de redéploiement du service s prend fin ($act1$). Les nouvelles instances déployés font alors partie de celles fournissant le service s ($act2$) et le service s est à nouveau libre de déployer d'autres instances ($act3$).

EVENT REDEPLOY REFINES REDEPLOY $\widehat{=}$
ANY
s
WHERE
$grd1 : s \in SERVICES$
$grd2 : s \mapsto FL\_CONF\_4 \in serviceState\_4$
$grd3 : dep\_inst[\{s\}] \neq \varnothing$
$grd4 : card(run\_peers(s)) + card(dep\_inst[\{s\}]) \geq min\_inst(s)$
WITH
$new\_inst : new\_inst = dep\_inst[\{s\}]$
THEN
$act1 : serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_CONF\_4\}) \cup \{s \mapsto DPL\_4\}$
$act2 : run\_peers(s) := run\_peers(s) \cup dep\_inst[\{s\}]$
$act3 : dep\_inst := \{s\} \lhd dep\_inst$
END

Ce septième raffinement peut se résumer à l'aide du diagramme suivant :

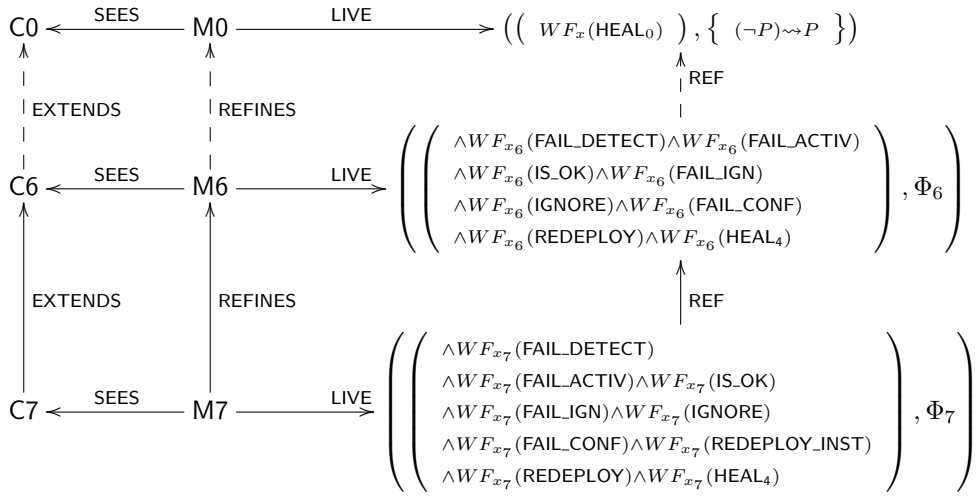


FIGURE 8.12 – Septième raffinement du self-healing

La sous-section suivante présente l'introduction d'une instance appelée « token owner », dirigeant les différentes phases de la procédure de self-healing.

8.3.4.4 Leader des phases du self-healing : le « token owner »

Ce raffinement introduit une particularité de l'algorithme de *self-healing* présenté dans [4] : chaque phase de l'algorithme (i.e. **self-detection**, **self-activation**, **self-configuration**) est régulée par des instances particulières, appelées « **token owner** ». Ces instances « **token owner** » gèrent les entrées et sorties d'un service des phases de l'algorithme de *self-healing* et permettent de coordonner ces dernières. Avant de pouvoir déclencher l'algorithme de self-healing, les instances « **token owner** » relatives à un service doivent dans un premier temps constater l'entrée du service dans un état illégal : ce raffinement modélise la suspicion par une instance « **token owner** » des états illégaux de certaines instances du service.

Nous commençons par définir un contexte C8 raffinant le contexte C7 et introduisant la notion de « **token owner** » :

CONTEXT C8 EXTENDS C7
CONSTANTS
$init\_tok, InitStatus, InitSuspPeers, InitFail$
AXIOMS
$axm1$: $init\_tok \in SERVICES \rightarrow PEERS$
$axm2$: $\forall s \cdot s \in SERVICES \Rightarrow init\_tok(s) \in InitSrvcPeers(s)$
$axm3$: $\forall a1, a2 \cdot \left(\begin{array}{l} \land a1 \in PEERS \leftrightarrow (SERVICES \times PEERS) \\ \land a2 \in PEERS \leftrightarrow (SERVICES \times PEERS) \\ \land finite(a1) \\ \land a2 \subseteq a1 \end{array} \right) \Rightarrow finite(a2)$
$axm4$: $InitStatus \in (PEERS \times SERVICES) \nrightarrow STATES\_4$
$axm5$: $\forall s, p \cdot \left(\begin{array}{l} \land s \in SERVICES \\ \land p \in PEERS \\ \land p = init\_tok(s) \end{array} \right) \Rightarrow (p \mapsto s) \mapsto RUN\_4 \in InitStatus$
$axm6$: $\forall s, p, stt \cdot \left(\begin{array}{l} \land s \in SERVICES \\ \land p \in PEERS \\ \land stt \in STATES\_4 \\ \land (p \mapsto s) \mapsto stt \in InitStatus \end{array} \right) \Rightarrow p = init\_tok(s) \land stt = RUN\_4$
$axm7$: $InitSuspPeers \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS)$
$axm8$: $\forall p, s, sp \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land sp \subseteq PEERS \\ \land (p \mapsto s) \mapsto sp \in InitSuspPeers \end{array} \right) \Rightarrow p = init\_tok(s) \land sp = \varnothing$
$axm9$: $\forall p, s \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land p = init\_tok(s) \end{array} \right) \Rightarrow (p \mapsto s) \mapsto \varnothing \in InitSuspPeers$
$axm10$: $InitFail \in SERVICES \rightarrow \mathbb{P}(PEERS)$
$axm11$: $\forall s \cdot s \in SERVICES \Rightarrow InitFail(s) = \varnothing$
END

— $axm1$: une constante *init_tok* définit les instances « token owner » initiales pour chaque service.
— $axm2$: cet axiome exprime que les instances « token owner » initiales de chaque service font partie des instances qui fournissent le service.
— $axm3$: cet axiome exprime une propriété concernant les ensembles. Si un ensemble $a1$ contenant des instances « token owner » d'un service associées aux paires fournissant le service est fini, et si un ensemble $a2$ est un sous ensemble de $a1$, alors, l'ensemble $a2$ est aussi fini.
— $axm4$: une constante $InitStatus$ associe à chaque « token owner » de chaque service, l'état de ce dernier.
— $axm5$, $axm6$: la constante $InitStatus$ va nous servir à définir les états initiaux associés aux instances « token owner » d'un service. Initiallement, les services sont dans un état légal RUN_4.
— $axm7$: une constante $InitSuspPeers$ associe chaque « token owner » de chaque service à un sous-ensemble de pairs (ici des instances du service) suspectés d'être défaillants/dans un état illégal.
— $axm8$, $axm9$: la constante $InitSuspPeers$ nous permettra de définir les pairs suspects par service, à l'état initial. Initiallement, nous n'avons aucun pair suspect par service. Le sous-ensemble de pairs défaillants associé à chaque « token owner » est vide ($\varnothing$).
— $axm10$: une constante $InitFail$ associant chaque service à un ensemble de pairs (ici des instances du service) défaillants/dans un état illégal est définie.
— $axm11$: la constante $InitFail$ va nous servir à définir les instances initiales défaillantes par service. Nous choisissons de n'avoir aucune instance défaillante à l'état initial.

Une machine M8, raffinant la machine M7 vue dans la sous-section précédente, utilise ce contexte C8 pour introduire les instances « token owner » des services, dans un premier temps, lorsque ces der-

niers suspectent qu'ils sont entrés dans des états illégaux. Pour ce faire, trois nouvelles variables sont introduites :

INITIALISATION $\widehat{=}$
BEGIN
$\ldots$
$\oplus act6 \ : \ token\_owner := init\_tok$
$\oplus act7 \ : \ unav\_peers := \varnothing$
$\oplus act8 \ : \ susp\_inst := \varnothing$
END

Nous notons x_8 l'ensemble des variables de la machine M8.

Un invariant noté $I_8(x_8)$ caractérise ces variables et est défini comme suit :

— Nous avons du typage :

$$\begin{array}{l} inv1 \ : \ token\_owner \in SERVICES \rightarrow PEERS \\ inv2 \ : \ unav\_peers \subseteq PEERS \\ inv3 \ : \ susp\_inst \in PEERS \leftrightarrow (SERVICES \times PEERS) \end{array}$$

— *token_owner* est une fonction associant à chaque service un pair/une instance appelée « token owner » et qui coordonnera les différentes phases de l'algorithme de *self-healing*.

— *unav_peers* contient les pairs/instances qui deviennent non-disponibles à un moment donné : ces pairs/instances ne fournissent plus les services qu'ils doivent fournir et seront ensuite suspectés d'être défaillants/dans un état illégal.

— *susp_inst* associe à chaque instance « token owner » d'un service, les pairs/instances suspectés d'être défaillants/dans un état illégal de ce service.

— Nous avons aussi des propriétés de sûreté :

$$\begin{array}{l} inv4 \ : \ \forall s \cdot s \in SERVICES \Rightarrow token\_owner(s) \in run\_peers(s) \setminus unav\_peers \\ inv5 \ : \ \forall s \cdot s \in SERVICES \wedge s \in dom(susp\_peers) \Rightarrow token\_owner(s) \notin susp\_peers(s) \\ inv6 \ : \ \forall ld, s \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge s \in dom(susp\_inst[\{ld\}]) \end{array} \right) \Rightarrow ld = token\_owner(s) \\ inv7 \ : \ \forall ld, s \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge s \in dom(susp\_inst[\{ld\}]) \\ \wedge ld = token\_owner(s) \end{array} \right) \Rightarrow ld \notin susp\_inst[\{ld\}][\{s\}] \\ inv8 \ : \ \forall ld, s \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge s \in dom(susp\_inst[\{ld\}]) \\ \wedge ld = token\_owner(s) \end{array} \right) \Rightarrow susp\_inst[\{ld\}][\{s\}] \subset run\_peers(s) \\ inv9 \ : \ \forall ld, s, stt \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge stt \in STATES\_4 \\ \wedge s \mapsto stt \in serviceState\_4 \\ \wedge ld = token\_owner(s) \\ \wedge stt \neq RUN\_4 \end{array} \right) \Rightarrow susp\_inst[\{ld\}][\{s\}] = \varnothing \end{array}$$

— *inv*4 exprime que l'instance « token owner » d'un service s fait partie des pairs fonctionnels et disponibles.

— *inv*5 exprime que pour chaque service s, l'instance « token owner » ne fait partie des pairs suspectés d'être défaillants.

— *inv*6 exprime que si une instance ld d'un service s maintient une liste d'instances suspectes, alors cette instance ld est l'instance « token owner » du service s.

— *inv*7 exprime que l'instance ld « token owner » d'un service s, qui maintient une liste d'instances suspectes, ne fait pas partie de cette liste d'instances suspectes.

— *inv*8 exprime que la liste d'instances suspectes d'un service s, maintenue par son instance « token owner » ld, est incluse dans la liste des instances fonctionnels du service s

— *inv*9 exprime que si un service s se trouve dans un état différent de l'état légal RUN_4, alors la liste d'instances suspectes de ce service s, maintenue par son instance « token owner » ld, est vide ($\varnothing$).

Cette machine M8 rajoute, par rapport à la machine M7 précédente, de nouveaux événements SUSPECT_INST, MAKE_PEER_UNAVAIL, MAKE_PEER_AVAIL. Une liste Φ_8 de propriétés de vivacité, détailler en annexe, nous guide dans le raffinement de M7 en M8, en présentant les nouveaux événements ou ceux ayant changé par raffinement :

— L'événement MAKE_PEER_UNAVAIL est un nouvel événement permettant de rendre les pairs d'un ensemble prs indisponibles ($act1$), c'est-à-dire, incapables temporairement de fournir des services. Nous considérons dans les gardes les cas suivants : $grd4$ exprime que pour un service srv, si l'instance « token owner » de ce service ne fait pas partie de l'ensemble prs, alors cette instance ne change pas ; sinon, si elle fait partie de l'ensemble prs ($grd5$ et $grd6$), une nouvelle instance « token owner » est choisie parmi les instances fonctionnelles, dans un état légal, fournissant le service srv, en dehors des instances indisponibles, de celles contenues dans prs et de celles suspectes et défaillantes. Nous exprimons le fait qu'un « token owner » doit toujours être disponible pour chaque service. En plus de rendre les membres de l'ensemble prs indisponibles ($act1$), cet événement permet de mettre à jour les instances « token owner » par service ($act2$), en cas d'indisponibilité de ces derniers et de faire en sorte que les instances de services membres de prs ne puissent plus maintenir de listes d'instances suspectes ($act3$).

EVENT MAKE_PEER_UNAVAIL $\widehat{=}$
ANY
prs, E
WHERE
$grd1 : prs \subseteq PEERS$
$grd2 : prs \not\subseteq unav\_peers$
$grd3 : E \in SERVICES \rightarrow PEERS$
$grd4 : \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \notin prs \end{array} \right) \Rightarrow E(srv) = token\_owner(srv)$
$grd5 : \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \in prs \\ \wedge srv \notin dom(susp\_peers) \end{array} \right) \Rightarrow E(srv) \in \left(\begin{array}{c} run\_peers(srv) \\ \setminus \\ \left(\begin{array}{l} \cup unav\_peers \\ \cup prs \\ \cup fail\_peers[\{srv\}] \end{array} \right) \end{array} \right)$
$grd6 : \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \in prs \\ \wedge srv \in dom(susp\_peers) \end{array} \right) \Rightarrow E(srv) \in \left(\begin{array}{c} run\_peers(srv) \\ \setminus \\ \left(\begin{array}{l} \cup unav\_peers \\ \cup prs \\ \cup susp\_peers(srv) \\ \cup fail\_peers[\{srv\}] \end{array} \right) \end{array} \right)$
THEN
$act1 : unav\_peers := unav\_peers \cup prs$
$act2 : token\_owner := token\_owner \lhd\!\!\!+ E$
$act3 : susp\_inst := prs \lhd susp\_inst$
END

— Un nouvel événement SUSPECT_INST permet à l'instance « token owner » d'un service s, dans un état légal ($grd5$), de constituer une première ($grd4$) liste d'instances suspectes $susp$ ($act1$). La liste $susp$ d'instances suspectes du service s est constituée des instances fournissant s et étant indisponibles ($grd3$).

EVENT SUSPECT_INST $\widehat{=}$
ANY
$s, susp$
WHERE
$grd1 : s \in SERVICES$
$grd2 : susp \subseteq PEERS$
$grd3 : susp = run\_peers(s) \cap unav\_peers$
$grd4 : s \notin dom(susp\_inst[\{token\_owner(s)\}])$
$grd5 : s \mapsto RUN\_4 \in serviceState\_4$
THEN
$act1 : susp\_inst := susp\_inst \cup (\{token\_owner(s)\} \times (\{s\} \times susp))$
END

— L'événement FAILURE modélise l'entrée d'un service s dans un état illégal. Un service s entre dans un état illégal $FAIL\_4$ ($act1$), lorsque la liste des instances suspectes maintenues par l'instance « token owner » du service s n'est pas vide. Les instances suspectées par l'instance « token owner » deviennent celles suspectées par le service s ($act2$). Nous permettons ensuite à l'instance « token

owner » du service s de reconstituer une liste d'instances suspectes en vidant la liste courante des instances suspectes maintenue par cette instance « token owner » ($act3$).

EVENT FAILURE REFINES FAILURE $\widehat{=}$
ANY
s
WHERE
$grd1 : s \in SERVICES$
$grd2 : s \mapsto RUN\_4 \in serviceState\_4$
$grd3 : susp\_inst[\{token\_owner(s)\}][\{s\}] \neq \varnothing$
WITH
$fp : fp = susp\_inst[\{token\_owner(s)\}][\{s\}]$
THEN
$act1 : serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto RUN\_4\}) \cup \{s \mapsto FAIL\_4\}$
$act2 : susp\_peers(s) := susp\_inst[\{token\_owner(s)\}][\{s\}]$
$act3 : susp\_inst := susp\_inst \rhd (\{s\} \lhd ran(susp\_inst))$
END

— Un nouvel événement MAKE_PEER_AVAIL permet à un pair indisponible (incapable d'instancier et de fournir des services) de redevenir disponible et de refournir et instancier les services qu'il proposait précédemment.

EVENT MAKE_PEER_AVAIL $\widehat{=}$
ANY
p
WHERE
$grd1 : p \in PEERS$
$grd2 : p \in unav\_peers$
THEN
$act1 : unav\_peers := unav\_peers \setminus \{p\}$
END

Ce huitième raffinement peut se résumer à l'aide du diagramme suivant :

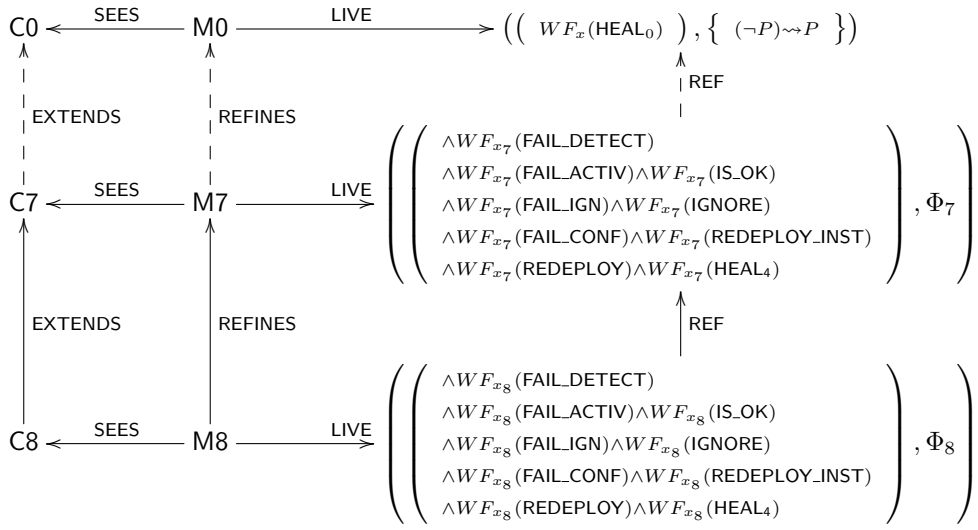


FIGURE 8.13 – Huitième raffinement du self-healing

Le prochain raffinement introduit la notion de « token owner » dans la procédure de **self-detection** d'un service : cette procédure de détection et confirmation d'un état illégal d'un service devient alors prise en charge par l'instance « token owner » de ce dernier.

8.3.4.5 Phase de self-detection : détails

Ce raffinement M9 détaille la première phase de la procédure *self-healing*, appelée **self-detection** : un service, dont certaines instances deviennent indisponibles, confirme grâce à la phase de **self-detection**,

si ces instances sont bien dans des états illégaux, auquel cas, le service est effectivement dans un état illégal.

Cette machine M9 introduit deux nouvelles variables :

$$\begin{array}{l} \text{INITIALISATION } \widehat{=} \\ \quad \text{BEGIN} \\ \quad\quad \oplus act10 \ : \ rec\_inst := \varnothing \\ \quad\quad \oplus act11 \ : \ rct\_inst := \varnothing \\ \quad \text{END} \end{array}$$

— *rec_inst* associe à chaque instance « token owner » d'un service, les pairs que cette instance a essayé de recontacter, pour confirmer leur statut (légal ou illégal).

— *rct_inst* associe à chaque instance « token owner » d'un service, les pairs que cette instance a essayé de recontacter et qu'elle a effectivement pu recontacter, confirmant ainsi leur statut fonctionnel et légal.

Nous notons x_9 l'ensemble des variables de la machine M9.

Un invariant noté $I_9(x_9)$ caractérise ces variables et est défini comme suit :

— Nous avons des propriétés de typage :

$$\begin{array}{l} inv1 \ : \ rec\_inst \in PEERS \leftrightarrow (SERVICES \times PEERS) \\ inv2 \ : \ rct\_inst \in PEERS \leftrightarrow (SERVICES \times PEERS) \end{array}$$

— Les propriétés de $inv3$ à $inv9$ expriment des propriétés de sûreté satisfaites par la machine M9 :

$$\begin{array}{ll} inv3 \ : & \forall ld, s \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge s \in dom(rec\_inst[\{ld\}]) \end{array} \right) \Rightarrow ld = token\_owner(s) \\ inv4 \ : & \forall ld, s \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge rct\_inst[\{ld\}][\{s\}] \neq \varnothing \end{array} \right) \Rightarrow rec\_inst[\{ld\}][\{s\}] \neq \varnothing \\ inv5 \ : & \forall ld, s \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge rct\_inst[\{ld\}][\{s\}] \neq \varnothing \end{array} \right) \Rightarrow rct\_inst[\{ld\}][\{s\}] \subseteq rec\_inst[\{ld\}][\{s\}] \\ inv6 \ : & \forall ld, s \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge s \in dom(rec\_inst[\{ld\}]) \\ \wedge ld = token\_owner(s) \end{array} \right) \Rightarrow ld \notin rec\_inst[\{ld\}][\{s\}] \\ inv7 \ : & \forall ld, s \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge s \in dom(rct\_inst[\{ld\}]) \end{array} \right) \Rightarrow ld = token\_owner(s) \\ inv8 \ : & \forall ld, s \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge s \in dom(rct\_inst[\{ld\}]) \\ \wedge ld = token\_owner(s) \end{array} \right) \Rightarrow ld \notin rct\_inst[\{ld\}][\{s\}] \\ inv9 \ : & dom(rct\_inst) \subseteq dom(rec\_inst) \end{array}$$

— $inv3$ et $inv7$ expriment que l'instance ld d'un service s qui essaye de (ou réussit à) contacter les instances suspectes est l'instance « token owner » du service s.

— $inv4$ et $inv9$ expriment que si l'instance « token owner » ld d'un service s a réussi à contacter des instances suspectes, alors cela signifie que l'instance « token owner » a déjà essayé de contacter des instances indisponibles.

— $inv5$ exprime que les instances que l'instance « token owner » ld d'un service s a réussi à contacter des instances font partie d'un sous-ensemble des instances que l'instance « token owner » a déjà essayé de contacter.

— $inv6$ et $inv8$ expriment que l'instance « token owner » ld ne fait pas partie des instances à contacter.

Cette machine M9 rajoute, par rapport à la machine M8 précédente, deux nouveaux événements RECONTACT_INST_OK et RECONTACT_INST_KO. Une liste Φ_9 de propriétés de vivacité (cf. annexe) permet de guider le raffinement de M8 en M9, en présentant les nouveaux événements ou ceux ayant changé par raffinement :

— Nous modifions par raffinement l'événement MAKE_PEER_UNAVAIL : maintenant, les paires membres de l'ensemble prs des paires indisponibles ne peuvent plus recontacter des instances ($act4$ et $act5$).

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL $\widehat{=}$
ANY
...
WHERE
...
THEN
$\oplus act4$: $rec\_inst := prs \lhd rec\_inst$
$\oplus act5$: $rct\_inst := prs \lhd rct\_inst$
END

— Nous définissons deux événements modélisant l'opération permettant à une instance « token owner » d'un service s de recontacter les instances de s devenues suspectes :
— RECONTACT_INST_OK modélise le fait que l'instance « token owner » du service s a essayé de contacter ($grd6$, $act1$) une instance i de s suspecte ($grd5$), et que l'instance « token owner » a réussi à contacter cette instance i parce que cette dernière est redevenue disponible ($grd5$, $act2$).

EVENT RECONTACT_INST_OK $\widehat{=}$
ANY
s, i
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $i \in PEERS$
$grd3$: $s \mapsto FAIL\_4 \in serviceState\_4$
$grd4$: $susp\_peers(s) \neq \varnothing$
$grd5$: $i \in susp\_peers(s) \setminus unav\_peers$
$grd6$: $token\_owner(s) \mapsto (s \mapsto i) \notin rec\_inst$
$grd7$: $rec\_inst[\{token\_owner(s)\}][\{s\}] \subset susp\_peers(s)$
THEN
$act1$: $rec\_inst := rec\_inst \cup \{token\_owner(s) \mapsto (s \mapsto i)\}$
$act2$: $rct\_inst := rct\_inst \cup \{token\_owner(s) \mapsto (s \mapsto i)\}$
END

— RECONTACT_INST_KO modélise le fait que l'instance « token owner » du service s a essayé de contacter ($grd6$, $act1$) une instance i de s suspecte ($grd5$), et que l'instance « token owner » n'a pas réussi à contacter cette instance i parce que cette dernière est demeurée disponible ($grd5$, $act2$).

EVENT RECONTACT_INST_KO $\widehat{=}$
ANY
s, i
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $i \in PEERS$
$grd3$: $s \mapsto FAIL\_4 \in serviceState\_4$
$grd4$: $susp\_peers(s) \neq \varnothing$
$grd5$: $i \in susp\_peers(s) \cap unav\_peers$
$grd6$: $token\_owner(s) \mapsto (s \mapsto i) \notin rec\_inst$
$grd7$: $rec\_inst[\{token\_owner(s)\}][\{s\}] \subset susp\_peers(s)$
THEN
$act1$: $rec\_inst := rec\_inst \cup \{token\_owner(s) \mapsto (s \mapsto i)\}$
END

— L'événement FAIL_DETECT modélise l'étape de la phase de **self-detection** au cours de laquelle l'état illégal d'un service s est confirmé ou non : le service s a détecté des instances suspectes ($grd3$), que son instance « token owner » a essayé de contacter ($grd4$). Les pairs que l'instance « token owner » a réussi à contacter sont retirés de la liste des pairs (instances) suspects ($act2$) et les listes d'instances suspectes du services s que l'instance « token owner » de ce dernier doit contacter sont vidées ($act3$, $act4$), afin que celui puisse recontacter à nouveau d'autres instances.

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
ANY
s
WHERE
$grd1 : s \in SERVICES$
$grd2 : s \mapsto FAIL\_4 \in serviceState\_4$
$grd3 : susp\_peers(s) \neq \varnothing$
$grd4 : rec\_inst[\{token\_owner(s)\}][\{s\}] = susp\_peers(s)$
WITH
$sf : sf = rct\_inst[\{token\_owner(s)\}][\{s\}]$
THEN
$act1 : serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FAIL\_4\}) \cup \{s \mapsto FL\_DT\_4\}$
$act2 : susp\_peers(s) := susp\_peers(s) \setminus rct\_inst[\{token\_owner(s)\}][\{s\}]$
$act3 : rec\_inst := rec\_inst \rhd (\{s\} \lhd ran(rec\_inst))$
$act4 : rct\_inst := rct\_inst \rhd (\{s\} \lhd ran(rct\_inst))$
END

Ce neuvième raffinement peut se résumer à l'aide du diagramme suivant :

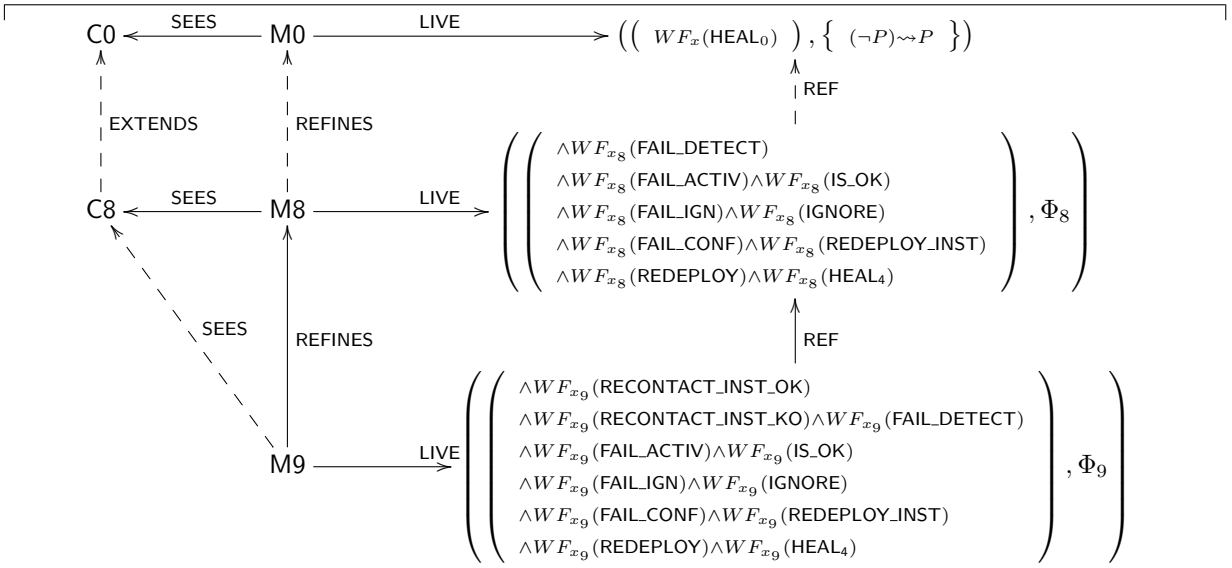


FIGURE 8.14 – Neuvième raffinement du self-healing

Le prochain raffinement introduit plus de détails dans la procédure de **self-configuration** d'un service : nous continuons le développement de la phase de redéploiement, sous-phase de **self-configuration**.

8.3.4.6 Redéploiement d'un service : détails

Dans ce dixième raffinement, nous détaillons un peu plus le redéploiement des instances d'un service dans un état illégal, qui consiste à déployer de nouvelles instances du service une à une, jusqu'à ce que le nombre d'instances minimal requis pour fournir le service soit atteint.

Une machine M10 introduit une nouvelle variable *actv_inst* :

INITIALISATION $\widehat{=}$
BEGIN
...
$\oplus act12 : actv\_inst := \varnothing$
END

— *actv_inst* associe des pairs (ici les instances « token owner ») d'un services à d'autres pairs (nouvellement déployés pour ce service).

Nous notons x_{10} l'ensemble des variables de la machine M10.

Un invariant noté $I_{10}(x_{10})$ caractérisant ces variables est défini comme suit :

— Nous avons une propriété de typage :

$$inv1 \ : \ actv\_inst \in PEERS \leftrightarrow (SERVICES \times PEERS)$$

— Nous avons des propriétés de sûreté :

$$\begin{array}{ll}
inv2 & : \ \forall s, i \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge i \in PEERS \end{array} \right) \Rightarrow finite(actv_inst[\{i\}][\{s\}]) \\
inv3 & : \ \forall ld, s \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge s \in dom(actv_inst[\{ld\}]) \end{array} \right) \Rightarrow ld = token_owner(s) \\
inv4 & : \ \forall s, i \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge i \in PEERS \end{array} \right) \Rightarrow actv_inst[\{i\}][\{s\}] \cap run_peers(s) = \varnothing \\
inv5 & : \ \forall s, i \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge i \in PEERS \end{array} \right) \Rightarrow actv_inst[\{i\}][\{s\}] \cap dep_inst[\{s\}] = \varnothing \\
inv6 & : \ \forall s, i \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge i \in PEERS \end{array} \right) \Rightarrow actv_inst[\{i\}][\{s\}] \cap fail_peers[\{s\}] = \varnothing \\
inv7 & : \ \forall ld, s, stt \cdot \left(\begin{array}{l} \wedge ld \in PEERS \\ \wedge s \in SERVICES \\ \wedge stt \in STATES_4 \\ \wedge s \mapsto stt \in serviceState_4 \\ \wedge ld = token_owner(s) \\ \wedge stt \neq FL_CONF_4 \end{array} \right) \Rightarrow actv_inst[\{ld\}][\{s\}] = \varnothing \\
inv8 & : \ finite(actv_inst)
\end{array}$$

— $inv2$ exprime que l'ensemble des nouvelles instances d'un service s, déployées par une instance i de s, est fini.
— $inv3$ exprime que l'instance ld ayant déployé de nouvelles instances d'un service s est l'instance « token owner » du service s.
— $inv4$, $inv5$, $inv6$ expriment respectivement que les nouvelles instances déployées pour un service s ne sont pas encore des instances fonctionnelles de ce service, n'ont pas été déployées pour ce service auparavant et ne sont pas des instances dans un état illégal/défaillant.
— $inv7$ exprime que dans les phases différentes de celle de **self-configuration**, l'instance « token owner » d'un service s ne déploie aucune nouvelle instance de ce dernier.
— $inv8$ exprime que l'ensemble $actv\_inst$ est fini.

Nous introduisons dans ce modèle un nouvel événement REDEPLOY_INSTC. Une liste Φ_{10} de propriétés de vivacité, détaillée en annexe [13], guide le raffinement de M9 en M10, en présentant les nouveaux événements ou ceux ayant changé par raffinement :

— Nous modifions l'événement MAKE_PEER_UNAVAIL : les pairs membres de l'ensemble prs contenant les pairs devenus indisponibles (incapables de fournir des services) ne sont plus en mesure de déployer des instances pour des services ($act6$).

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL $\widehat{=}$
ANY
...
WHERE
...
THEN
...
$\oplus act6 \ : \ actv\_inst := prs \lhd actv\_inst$
END

— De nouveaux événements relatifs à la phase de redéploiement d'un service sont ajoutés et ceux déjà présents sont détaillés par raffinement :
— Un nouvel événement REDEPLOY_INSTC est défini : tant que le nombre d'instance $deplo\_inst(s)$ qu'un service s peut déployer en un coup n'est pas atteint ($grd6$) et tant que le nombre total d'instances nouvellement déployées pour le service s ne permet pas d'atteindre le minimum requis pour fournir le service s ($grd7$), une nouvelle instance i (ne faisant pas partie des instances déjà fonctionnelles de s, des instances dans un état illégal, indisponibles ou déjà déployée) ($grd2$, $grd3$) est déployée par l'instance « token owner » du service s.

13. `http://andriami.bruno.free.fr/andriami/main/short/annexe.pdf`

EVENT REDEPLOY_INSTC $\widehat{=}$
ANY
s, i
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $i \in PEERS$
$grd3$: $i \notin run\_peers(s) \cup fail\_peers[\{s\}] \cup unav\_peers \cup dep\_inst[\{s\}]$
$grd4$: $token\_owner(s) \mapsto (s \mapsto i) \notin actv\_inst$
$grd5$: $s \mapsto FL\_CONF\_4 \in serviceState\_4$
$grd6$: $card(actv\_inst[\{token\_owner(s)\}][\{s\}]) < deplo\_inst(s)$
$grd7$: $card(dep\_inst[\{s\}]) + card(run\_peers(s)) < min\_inst(s)$
THEN
$act1$: $actv\_inst := actv\_inst \cup \{token\_owner(s) \mapsto (s \mapsto i)\}$
END

— L'événement REDEPLOY_INSTS, raffinement de REDEPLOY_INST, est observé lorsque le nombre de nouvelles instances déployées par l'instance « token owner » d'un service s atteint le nombre d'instance qu'un service s peut déployer en un coup ($grd2$) et tant que le nombre total d'instances nouvellement déployées pour le service s ne permet pas d'atteindre le minimum requis pour fournir le service s ($grd4$). Les instances nouvellement déployées sont alors considérées comme des instances destinées à fournir le service s ($act1$) et la liste des nouvelles instances maintenue par l'instance « token owner » est remise à zéro ($act2$) afin que cette instance puisse en redéployer d'autres.

EVENT REDEPLOY_INSTS REFINES REDEPLOY_INST $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $card(actv\_inst[\{token\_owner(s)\}][\{s\}]) = deplo\_inst(s)$
$grd3$: $card(dep\_inst[\{s\}]) + card(run\_peers(s)) < min\_inst(s)$
$grd4$: $s \mapsto FL\_CONF\_4 \in serviceState\_4$
WITH
dep : $dep = actv\_inst[\{token\_owner(s)\}][\{s\}]$
THEN
$act1$: $dep\_inst := dep\_inst \cup (\{s\} \times actv\_inst[\{token\_owner(s)\}][\{s\}])$
$act2$: $actv\_inst := actv\_inst \rhd (\{s\} \lhd ran(actv\_inst))$
END

— L'événement REDEPLOY est observé lorsque le nombre d'instances dans un état légal, auquel on ajoute celles nouvellement déployées ($grd5$), d'un service s dépasse le minimum requis pour fournir le service s, et lorsque l'instance « token owner » du service s a terminé le déploiement de nouvelles instances ($grd3$), c'est-à-dire lorsque la procédure de redéploiement du service s prend fin ($act1$). Les nouvelles instances déployés font alors partie de celles fournissant le service s ($act2$) et le service s est à nouveau libre de déployer d'autres instances ($act3$).

EVENT REDEPLOY REFINES REDEPLOY $\widehat{=}$
ANY
s
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $s \mapsto FL\_CONF\_4 \in serviceState\_4$
$grd3$: $actv\_inst[\{token\_owner(s)\}][\{s\}] = \varnothing$
$grd4$: $dep\_inst[\{s\}] \neq \varnothing$
$grd5$: $card(run\_peers(s)) + card(dep\_inst[\{s\}]) \geq min\_inst(s)$
THEN
$act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_CONF\_4\}) \cup \{s \mapsto DPL\_4\}$
$act2$: $run\_peers(s) := run\_peers(s) \cup dep\_inst[\{s\}]$
$act3$: $dep\_inst := \{s\} \lhd\!\!\!- dep\_inst$
END

Ce dixième raffinement peut se résumer à l'aide du diagramme suivant :

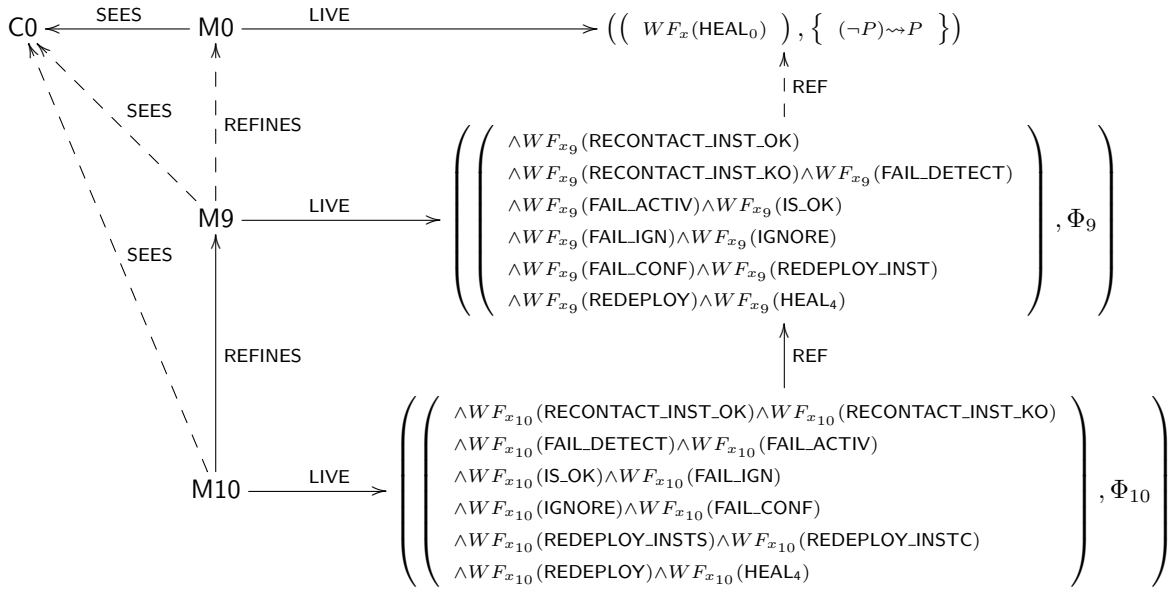


FIGURE 8.15 – Dixième raffinement du self-healing

Le prochain raffinement introduit plus de détails sur les fonctions des instances « token owner ».

8.3.4.7 L'instance « token owner » et les états d'un service

Ce raffinement introduit le fait que l'état d'un service est monitoré par son instance « token owner » : c'est cette dernière, suivant les états, les évolutions du nombre des instances du service, qui associe un état global au service. Une machine M11, raffinant la machine M10 précédente et utilisant le contexte C8, exprime ce monitorat de l'état global d'un service par son instance « token owner ». Nous commençons par rappeler la partie du contexte C8 utilisée par la machine M11 :

CONTEXT C8 EXTENDS C7
CONSTANTS
$\ldots, InitStatus, \ldots$
AXIOMS
$\ldots$
$axm4 \ : \ InitStatus \in (PEERS \times SERVICES) \pfun STATES\_4$
$axm5 \ : \ \forall s, p \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge p \in PEERS \\ \wedge p = init\_tok(s) \end{array} \right) \Rightarrow (p \mapsto s) \mapsto RUN\_4 \in InitStatus$
$axm6 \ : \ \forall s, p, stt \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge p \in PEERS \\ \wedge stt \in STATES\_4 \\ \wedge (p \mapsto s) \mapsto stt \in InitStatus \end{array} \right) \Rightarrow p = init\_tok(s) \wedge stt = RUN\_4$
$\ldots$
END

— $axm4$, $axm5$, $axm6$ expriment que la constante $InitStatus$ associe à chaque pair qui est instance « token owner » d'un service s, un état initial, qui est l'état légal RUN_4.

Nous détaillons maintenant la machine M11. Nous introduisons dans cette machine une variable *i_state* :

INITIALISATION $\widehat{=}$
BEGIN
$\ominus act1 \ : \ serviceState\_4 := InitState\_4$
$\oplus act1 \ : \ i\_state := InitStatus$
$\ldots$
END

— Cette variable *i_state* associe à chaque instance « token owner » d'un service ($axm2$, $axm3$), l'état global courant de ce dernier ($axm1$). La variable *i_state* est initialisée à l'aide de la constante $InitStatus$: chaque service est initialement dans un état légal RUN_4.

Nous notons x_{11} l'ensemble des variables de la machine M11.

Un invariant noté $I_{11}(x_{11})$ caractérise ces variables et est défini comme suit :

— Nous avons une propriété de typage :

$$inv1 \ : \ i\_state \in (PEERS \times SERVICES) \nrightarrow STATES\_4$$

— Nous avons des propriétés de sûreté :

$$\begin{array}{l} inv2 \ : \ \forall s \cdot s \in SERVICES \Rightarrow token\_owner(s) \mapsto s \in dom(i\_state) \\ inv3 \ : \ \forall p, s \cdot \left(\begin{array}{l} \wedge p \in PEERS \\ \wedge s \in SERVICES \\ \wedge (p \mapsto s) \in dom(i\_state) \end{array} \right) \Rightarrow p = token\_owner(s) \end{array}$$

— $inv2$ et $inv3$ expriment que pour chaque service s, une instance « token owner » monitore l'état du service.

— Nous rappelons que la variable *i_state* remplace la variable abstraite *serviceState_4*. Les propriétés $inv4$ et $inv5$ suivants définissent les liens entre ces deux variables :

$$\begin{array}{l} inv4 \ : \ \forall s, stt \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge stt \in STATES\_4 \\ \wedge s \mapsto stt \in serviceState\_4 \end{array} \right) \Rightarrow (token\_owner(s) \mapsto s) \mapsto stt \in i\_state \\ inv5 \ : \ \forall s, stt \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge stt \in STATES\_4 \\ \wedge (token\_owner(s) \mapsto s) \mapsto stt \in i\_state \end{array} \right) \Rightarrow s \mapsto stt \in serviceState\_4 \end{array}$$

— $inv4$ et $inv5$ exprime que l'état global d'un service s est le même que celui monitoré par son instance « token owner ».

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
serviceState_4	*i_state*

Les axiomes présentés précédemment du contexte C8, ainsi que l'invariant de collage I_{11} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M10 en M11, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M11, ainsi que pour justifier la préservation des propriétés Φ_{10} lors du raffinement.

Une liste Φ_{11} de propriétés de vivacité, détaillée en annexe, nous permet d'expliciter le raffinement de M10 en M11, en présentant les nouveaux événements ou ceux ayant changé par raffinement :

— Evénement MAKE_PEER_UNAVAIL : l'instance « token owner » de chaque service srv ($token\_owner(srv)$) pouvant être rendu indisponible, nous sélectionnons un remplaçant ($E(srv)$) qui a monitoré l'état du service de manière identique. Nous utilisons un paramètre i_s pour stocker ces remplaçants et nous utilisons ce paramètre pour mettre à jour les instances « token owner » monitorant les états de chaque service ($act7$).

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL $\widehat{=}$
ANY
$\ldots$
$\oplus$ *i_s*
WHERE
$\ldots$
$\oplus grd7 \ : \ i\_s \in (PEERS \times SERVICES) \nrightarrow STATES\_4$
$\oplus grd8 \ : \ \forall p, s \cdot \left(\begin{array}{l} \wedge p \in PEERS \\ \wedge s \in SERVICES \\ \wedge p \mapsto s \in dom(i\_s) \end{array} \right) \Rightarrow p = E(s)$
$\oplus grd9 \ : \ \forall srv \cdot srv \in SERVICES \Rightarrow (E(srv) \mapsto srv) \mapsto i\_state(token\_owner(srv) \mapsto srv) \in i\_s$
THEN
$\ldots$
$\oplus act7 \ : \ i\_state := i\_s$
END

— Evénement SUSPECT_INST : l'état légal RUN_4 d'un service s, condition nécessaire à la découverte d'instances suspectes du service s, n'est plus constaté globalement au niveau du service, mais plutôt au niveau d'une instance particulière « token owner » qui monitore les évolutions de l'état du service s ($grd5$).

EVENT SUSPECT_INST REFINES SUSPECT_INST $\widehat{=}$
ANY
$\ldots$
WHERE
$\ldots$
$\ominus grd5 \ : \ s \mapsto RUN\_4 \in serviceState\_4$
$\oplus grd5 \ : \ i\_state(token\_owner(s) \mapsto s) = RUN\_4$
THEN
$\ldots$
END

— Evénement FAILURE : un service s ayant suspecté certaines de ses instances d'être dans un état illégal et dont l'état monitoré par son instance « token owner » est l'état légal RUN_4, entre dans un état illégal, c'est-à-dire que le nouvel état du service monitoré par son instance « token owner » devient $FAIL\_4$.

EVENT FAILURE REFINES FAILURE $\widehat{=}$
ANY
$\ldots$
WHERE
$\ldots$
$\ominus grd2 \ : \ s \mapsto RUN\_4 \in serviceState\_4$
$\oplus grd2 \ : \ i\_state(token\_owner(s) \mapsto s) = RUN\_4$
THEN
$\ominus act1 \ : \ serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto RUN\_4\}) \cup \{s \mapsto FAIL\_4\}$
$\oplus act1 \ : \ i\_state(token\_owner(s) \mapsto s) := FAIL\_4$
$\ldots$
END

— Evénements RECONTACT_INST_OK et RECONTACT_INST_KO : L'instance « token owner » d'un service s a constaté un état illégal $FAIL\_4$ ($grd3$), ainsi que l'existance d'instances suspectes. Ces instances suspectes sont alors contactées, afin de déterminer si elles sont vraiment dans un état illégal.

EVENT RECONTACT_INST_OK REFINES RECONTACT_INST_OK $\widehat{=}$
ANY
$\ldots$
WHERE
$\ldots$
$\ominus grd3 \ : \ s \mapsto FAIL\_4 \in serviceState\_4$
$\oplus grd3 \ : \ i\_state(token\_owner(s) \mapsto s) = FAIL\_4$
$\ldots$
THEN
$\ldots$
END

EVENT RECONTACT_INST_KO REFINES RECONTACT_INST_KO $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus grd3$: $s \mapsto FAIL\_4 \in serviceState\_4$
    $\oplus grd3$: $i\_state(token\_owner(s) \mapsto s) = FAIL\_4$
    ...
  THEN
    ...
  END

— L'événement FAIL_DETECT est observé après les événements de contacts décrits ci-dessus : des instances d'un service s n'ont pas répondu aux demandes de contacts, et l'instance « token owner » constate que ces instances sont demeurées suspectes, et par conséquent fait passer l'état du service s de $FAIL\_4$ à FL_DT_4 ($grd2$, $act1$). L'instance « token owner » du service s guide le service vers la phase de **self-detection**.

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus grd2$: $s \mapsto FAIL\_4 \in serviceState\_4$
    $\oplus grd2$: $i\_state(token\_owner(s) \mapsto s) = FAIL\_4$
    ...
  THEN
    $\ominus act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FAIL\_4\}) \cup \{s \mapsto FL\_DT\_4\}$
    $\oplus act1$: $i\_state(token\_owner(s) \mapsto s) := FL\_DT\_4$
    ...
  END

— Evénement IS_OK : l'état illégal détecté d'un service s est une fausse alerte ; l'instance « token owner » du service s fait alors passer l'état du service s de l'état de détection FL_DT_4 d'un état illégal ($grd2$) à un état légal RUN_4 ($act1$).

EVENT IS_OK REFINES IS_OK $\widehat{=}$
  ANY
    ...
  WHERE
...
    $\ominus grd2$: $s \mapsto FL\_DT\_4 \in serviceState\_4$
    $\oplus grd2$: $i\_state(token\_owner(s) \mapsto s) = FL\_DT\_4$
...
  THEN
    $\ominus act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_DT\_4\}) \cup \{s \mapsto RUN\_4\}$
    $\oplus act1$: $i\_state(token\_owner(s) \mapsto s) := RUN\_4$
  END

— Evénement FAIL_ACTIV : l'instance « token owner » d'un service s a constaté que l'état illégal qu'elle a détecté ($grd2$) est réel ; l'instance « token owner » provoque alors l'entrée du service s dans la phase de **self-activation** ($act1$), pour l'évaluation de la gravité de la cause de l'état illégal.

EVENT FAIL_ACTIV REFINES FAIL_ACTIV $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus grd2$: $s \mapsto FL\_DT\_4 \in serviceState\_4$
    $\oplus grd2$: $i\_state(token\_owner(s) \mapsto s) = FL\_DT\_4$
    ...
  THEN
    $\ominus act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_DT\_4\}) \cup \{s \mapsto FL\_ACT\_4\}$
    $\oplus act1$: $i\_state(token\_owner(s) \mapsto s) := FL\_ACT\_4$
    ...
  END

— Evénement FAIL_IGN : l'instance « token owner » d'un service s a décidé que l'état illégal de ce dernier pouvait être ignoré ($grd2$, $act1$), car l'état illégal n'est pas critique et ne perturbe pas le fonctionnement du service s.

EVENT FAIL_IGN REFINES FAIL_IGN $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd2$: $s \mapsto FL\_ACT\_4 \in serviceState\_4$
$\oplus grd2$: $i\_state(token\_owner(s) \mapsto s) = FL\_ACT\_4$
...
THEN
$\ominus act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_ACT\_4\}) \cup \{s \mapsto FAIL\_IGN\_4\}$
$\oplus act1$: $i\_state(token\_owner(s) \mapsto s) := FAIL\_IGN\_4$
END

— Evénement IGNORE : l'instance « token owner » d'un service s ignore l'état illégal de ce dernier et le fait passer à un état légal ($grd2$, $act1$).

EVENT IGNORE REFINES IGNORE $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd2$: $s \mapsto FAIL\_IGN\_4 \in serviceState\_4$
$\oplus grd2$: $i\_state(token\_owner(s) \mapsto s) = FAIL\_IGN\_4$
...
THEN
$\ominus act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FAIL\_IGN\_4\}) \cup \{s \mapsto RUN\_4\}$
$\oplus act1$: $i\_state(token\_owner(s) \mapsto s) := RUN\_4$
END

— Evénement FAIL_CONF : l'instance « token owner » d'un service s constate que l'état illégal de ce dernier est critique ($grd2$) et le fait passer à la phase de **self-configuration** ($act1$).

EVENT FAIL_CONF REFINES FAIL_CONF $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd2$: $s \mapsto FL\_ACT\_4 \in serviceState\_4$
$\oplus grd2$: $i\_state(token\_owner(s) \mapsto s) = FL\_ACT\_4$
...
THEN
$\ominus act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_ACT\_4\}) \cup \{s \mapsto FL\_CONF\_4\}$
$\oplus act1$: $i\_state(token\_owner(s) \mapsto s) := FL\_CONF\_4$
END

— Evénements REDEPLOY_INSTC, REDEPLOY_INSTS et REDEPLOY : nous pouvons voir que pour un service s, c'est l'instance « token owner » de ce dernier qui le guide durant toute la phase de redéploiement (sous-phase de **self-detection**), en le faisant passer de l'état FL_CONF_4 à l'état DPL_4, indiquant la fin de la phase de redéploiement.

EVENT REDEPLOY_INSTC REFINES REDEPLOY_INSTC $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd5$: $s \mapsto FL\_CONF\_4 \in serviceState\_4$
$\oplus grd5$: $i\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
...
THEN
...
END

EVENT REDEPLOY_INSTS REFINES REDEPLOY_INSTS $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd8$: $s \mapsto FL\_CONF\_4 \in serviceState\_4$
$\oplus grd8$: $i\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
THEN
...
END

EVENT REDEPLOY REFINES REDEPLOY $\widehat{=}$
ANY
...
WHERE ...
$\ominus grd2$: $s \mapsto FL\_CONF\_4 \in serviceState\_4$
$\oplus grd2$: $i\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
...
THEN
$\ominus act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto FL\_CONF\_4\}) \cup \{s \mapsto DPL\_4\}$
$\oplus act1$: $i\_state(token\_owner(s) \mapsto s) := DPL\_4$
...
END

— Evénement HEAL_4 : l'instance « token owner » d'un service s constate la fin du redéploiement de ce dernier ($grd2$) et le fait passer dans un état légal RUN_4 ($act1$).

EVENT HEAL_4 REFINES HEAL_4 $\widehat{=}$
ANY
...
WHERE
$\ominus grd2$: $s \mapsto DPL\_4 \in serviceState\_4$
$\oplus grd2$: $i\_state(token\_owner(s) \mapsto s) = DPL\_4$
THEN
$\ominus act1$: $serviceState\_4 := (serviceState\_4 \setminus \{s \mapsto DPL\_4\}) \cup \{s \mapsto RUN\_4\}$
$\oplus act1$: $i\_state(token\_owner(s) \mapsto s) := RUN\_4$
END

Ce onzième raffinement peut se résumer par le diagramme suivant :

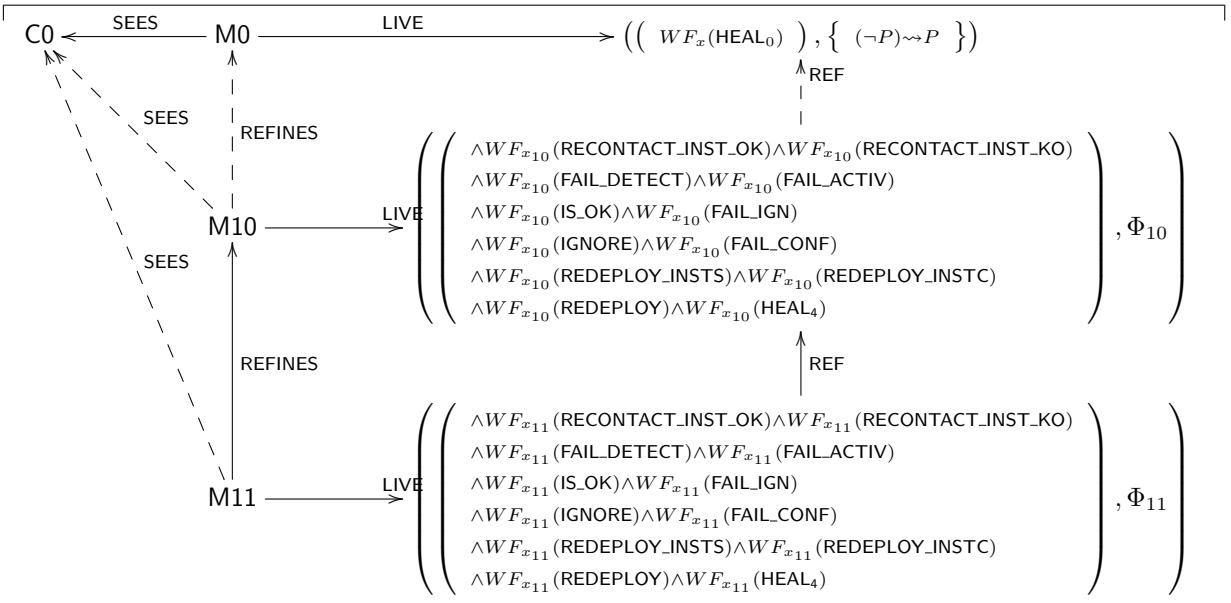


FIGURE 8.16 – Onzième raffinement du self-healing

Ce raffinement nous montre que chaque phase de la procédure de *self-healing* d'un service fourni par le système P2P (i.e. les phases de **self-detection**, **self-activation**, **self-configuration**) est guidé/dirigé par une instance spéciale de ce service appelé « token owner » : cette instance spéciale décide des entrées et sorties du service de chacune de ces phases.

8.3.4.8 « Token owner » : coordination des phases

Ce douzième raffinement étend le champ d'action des instances « token owner » de chaque service. Ces dernières ne font pas que maintenir des listes des instances indisponibles (ne fournissant plus le(s)

service(s) demandé(s)), monitorer les états des services, etc ; elles ont aussi la charge de détecter si les instances indisponibles sont vraiment défaillantes/dans un état illégal ou sont seulement victimes de fausses alertes. Pour modéliser ces nouveaux comportements, nous réutilisons une constante $InitSuspPeers$, définie dans le contexte C8 vu dans les sections précédentes. Nous rappelons ici que cette constante $InitSuspPeers$ associe à chaque instance « token owner » de chaque service une liste initialement vide ($\varnothing$) d'instances.

Une machine M12, raffinant la machine M11 vue dans la section précédent, utilise cette constante $InitSuspPeers$, notamment pour définir l'état initial d'une nouvelle variable $suspc\_peers$:

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus act3$: $susp\_peers := \varnothing$
$\oplus act3$: $suspc\_peers := InitSuspPeers$
...
END

Nous notons x_{12} l'ensemble des variables de la machine M12.

Ces variables sont caractérisées par un invariant $I_{12}(x_{12})$, défini comme suit :

— Nous avons des propriété de typages et des propriétés relatives à la nouvelle variable $suspc\_peers$:

$$\begin{array}{l} inv1 \ : \ suspc\_peers \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS) \\ inv2 \ : \ \forall p, s \cdot \left(\begin{array}{l} \wedge p \in PEERS \\ \wedge s \in SERVICES \\ \wedge (p \mapsto s) \in dom(suspc\_peers) \end{array} \right) \Rightarrow p = token\_owner(s) \\ inv3 \ : \ \forall p, s \cdot \left(\begin{array}{l} \wedge p \in PEERS \\ \wedge s \in SERVICES \\ \wedge p = token\_owner(s) \end{array} \right) \Rightarrow (p \mapsto s) \in dom(suspc\_peers) \end{array}$$

— la nouvelle variable $suspc\_peers$ associe à chaque instance « token owner » ($inv2$, $inv3$) de chaque service s, un sous-ensemble de pairs (instances) contenant les instances indiponibles ($inv1$) (incapables de fournir le service demander), que l'instance « token owner » n'a pas réussi à contacter et qui sont par conséquent suspectées d'être réellement dans des états illégaux. La variable $suspc\_peers$ est initialisée à l'aide la constante $InitSuspPeers$, parce qu'à l'état initial, la liste des instances suspectes par « token owner » est vide ($\varnothing$).

— La variable $suspc\_peers$ remplace la variable abstraite $susp\_peers$. L'invariant $inv4$ établit les relations existant entre ces deux variables :

$$inv4 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge s \in dom(susp\_peers) \end{array} \right) \Rightarrow susp\_peers(s) = suspc\_peers(token\_owner(s) \mapsto s)$$

— L'invariant $inv4$ exprime que pour chaque service s, la liste des instances indiponibles suspectées d'être réellement dans des états illégaux, maintenue par le service est la même que celle maintenue par l'instance « token owner » du service s. Nous pouvons par conséquent utiliser la nouvelle variable $suspc\_peers$ à la place de $susp\_peers$.

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
$susp\_peers$	$suspc\_peers$

Les axiomes présentés précédemment du contexte C8, ainsi que l'invariant de collage I_{12} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M11 en M12, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M12, ainsi que pour justifier la préservation des propriétés Φ_{11} lors du raffinement.

Une liste Φ_{12} de propriétés de vivacité, détaillée en annexe, guide le raffinement de M11 en M12, en présentant les nouveaux événements ou ceux ayant changé par raffinement :

— Evénement MAKE_PEER_UNAVAIL : les gardes et actions faisant référence ou modifiant la variable *susp_peers* sont modifiées. Le possible remplaçant d'une instance « token owner » d'un service *srv* est choisi en dehors des instances indisponibles, de celles qui vont être rendues indisponibles, de celles dans défaillantes/dans un état illégal et de celles suspectées d'être réellement dans des états illégaux (*grd*6, *grd*7). Désormais, cette liste d'instances suspectes n'est plus une liste globale maintenue par le service lui-même, mais plutôt une liste locale maintenue par son instance « token owner » (*grd*6, *grd*7). Le possible remplaçant de l'instance « token owner » d'un service *srv* maintient également la même liste d'instances suspectes que ce dernier (*grd*10, *grd*11, *grd*12, *act*1).

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL $\widehat{=}$
ANY
...
$\oplus p\_s$
WHERE
...

$$\ominus grd6 \ : \ \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \in prs \\ \wedge srv \notin dom(susp\_peers) \end{array} \right) \Rightarrow E(srv) \in \left(\begin{array}{c} run\_peers(srv) \\ \setminus \\ \left(\begin{array}{l} \cup unav\_peers \\ \cup prs \\ \cup fail\_peers[\{srv\}] \end{array} \right) \end{array} \right)$$

$$\oplus grd6 \ : \ \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \in prs \\ \wedge (token\_owner(srv) \mapsto srv) \notin dom(suspc\_peers) \end{array} \right) \Rightarrow E(srv) \in \left(\begin{array}{c} run\_peers(srv) \\ \setminus \\ \left(\begin{array}{l} \cup unav\_peers \\ \cup prs \\ \cup fail\_peers[\{srv\}] \end{array} \right) \end{array} \right)$$

$$\ominus grd7 \ : \ \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \in prs \\ \wedge srv \in dom(susp\_peers) \end{array} \right) \Rightarrow E(srv) \in \left(\begin{array}{c} run\_peers(srv) \\ \setminus \\ \left(\begin{array}{l} \cup unav\_peers \\ \cup prs \\ \cup fail\_peers[\{srv\}] \\ \cup susp\_peers(srv) \end{array} \right) \end{array} \right)$$

$$\oplus grd7 \ : \ \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \in prs \\ \wedge \left(\begin{array}{c} \left(\begin{array}{c} token\_owner(srv) \\ \mapsto \\ srv \end{array} \right) \\ \in \\ dom(suspc\_peers) \end{array} \right) \end{array} \right) \Rightarrow E(srv) \in \left(\begin{array}{c} run\_peers(srv) \\ \setminus \\ \left(\begin{array}{l} \cup unav\_peers \\ \cup prs \\ \cup fail\_peers[\{srv\}] \\ \cup suspc\_peers \left(\begin{array}{c} token\_owner(srv) \\ \mapsto \\ srv \end{array} \right) \end{array} \right) \end{array} \right)$$

...
$\oplus grd10 \ : \ p\_s \in (PEERS \times SERVICES) \pfun \mathbb{P}(PEERS)$
$\oplus grd11 \ : \ \forall p, s \cdot (\ p \in PEERS \wedge s \in SERVICES \wedge p \mapsto s \in dom(p\_s) \ ) \Rightarrow p = E(s)$
$\oplus grd12 \ : \ \forall srv \cdot srv \in SERVICES \Rightarrow (E(srv) \mapsto srv) \mapsto suspc\_peers(token\_owner(srv) \mapsto srv) \in p\_s$
THEN
...
$\oplus act8 \ : \ suspc\_peers := p\_s$
END

— Evénement FAILURE : l'instance « token owner » d'un service *s* a détecté des instances du service indisponibles. L'instance « token owner » initialise sa liste d'instances suspectées d'être réellement dans un état illégal à l'aide de ces instances indisponibles (*act*2).

EVENT FAILURE REFINES FAILURE $\widehat{=}$
ANY
...
WHERE
...
THEN
...
$\ominus act2 \ : \ susp\_peers(s) := susp\_inst[\{token\_owner(s)\}][\{s\}]$
$\oplus act2 \ : \ suspc\_peers(token\_owner(s) \mapsto s) := susp\_inst[\{token\_owner(s)\}][\{s\}]$
...
END

— Evénements RECONTACT_INST_OK et RECONTACT_INST_KO : l'instance « token owner » d'un service *s* essaie de recontacter tous les membres de la liste des instances suspectées d'être réellement dans un état illégal (*grd*4, *grd*5, *grd*7). Selon le cas, les instances recontactées peuvent répondre car

elles sont finalement redevenues disponibles ($grd5$ de l'événement RECONTACT_INST_OK) ou elles ne peuvent être recontactées car elles demeurent indisponibles ($grd5$ de RECONTACT_INST_KO).

EVENT RECONTACT_INST_OK REFINES RECONTACT_INST_OK $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd4$: $susp\_peers(s) \neq \varnothing$
$\oplus grd4$: $suspc\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
$\ominus grd5$: $i \in susp\_peers(s) \setminus unav\_peers$
$\oplus grd5$: $i \in suspc\_peers(token\_owner(s) \mapsto s) \setminus unav\_peers$
...
$\ominus grd7$: $rec\_inst[\{token\_owner(s)\}][\{s\}] \subset susp\_peers(s)$
$\oplus grd7$: $rec\_inst[\{token\_owner(s)\}][\{s\}] \subset suspc\_peers(token\_owner(s) \mapsto s)$
THEN
...
END

EVENT RECONTACT_INST_KO REFINES RECONTACT_INST_KO $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd4$: $susp\_peers(s) \neq \varnothing$
$\oplus grd4$: $suspc\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
$\ominus grd5$: $i \in susp\_peers(s) \cap unav\_peers$
$\oplus grd5$: $i \in suspc\_peers(token\_owner(s) \mapsto s) \cap unav\_peers$
...
$\ominus grd7$: $rec\_inst[\{token\_owner(s)\}][\{s\}] \subset susp\_peers(s)$
$\oplus grd7$: $rec\_inst[\{token\_owner(s)\}][\{s\}] \subset suspc\_peers(token\_owner(s) \mapsto s)$
THEN
...
END

— Evénement FAIL_DETECT : les instances suspectes d'un service s que son instance « token owner » a réussi à recontacter, sont retirées de la liste des instances suspectes maintenues par l'instance « token owner » du service s ($act2$). Il s'agit de l'entrée dans la phase de **self-detection**.

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd3$: $susp\_peers(s) \neq \varnothing$
$\oplus grd3$: $suspc\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
...
THEN
...
$\ominus act2$: $susp\_peers(s) := susp\_peers(s) \setminus rct\_inst[\{token\_owner(s)\}][\{s\}]$
$\oplus act2$: $suspc\_peers(token\_owner(s) \mapsto s)$
$:= suspc\_peers(token\_owner(s) \mapsto s) \setminus rct\_inst[\{token\_owner(s)\}][\{s\}]$
...
END

— Evénement IS_OK : cet événement est observé quand l'instance « token owner » d'un service s constate que la liste des instances suspectes est vide, car ces instances ont toutes pu être recontactées ($grd3$). L'instance « token owner » fait alors passer le service s de la phase de **self-detection** à un état légal.

EVENT IS_OK REFINES IS_OK $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd3$: $susp\_peers(s) = \varnothing$
$\oplus grd3$: $suspc\_peers(token\_owner(s) \mapsto s) = \varnothing$
THEN
...
END

— Evénement FAIL_ACTIV : Cet événement est le dual de l'événement IS_OK ; en effet, ici l'instance « token owner » d'un service s constate que la liste des instances suspectes qu'elle maintient n'est pas vide, car ces instances n'ont pas toutes pu être recontactées ($grd3$). Elles sont donc retirées de la liste des instances fournissant le service s ($act2$), elles sont considérées comme étant défaillantes/dans un état illégal ($act3$). Il s'agit ici de l'entrée du service s dans la phase de **self-activation**. L'instance « token owner » remet ensuite à zéro sa liste d'instances suspectes ($act4$).

EVENT FAIL_ACTIV REFINES FAIL_ACTIV $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd3 \ : \ susp\_peers(s) \neq \varnothing$
$\oplus grd3 \ : \ suspc\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
THEN
...
$\ominus act2 \ : \ run\_peers(s) := run\_peers(s) \setminus susp\_peers(s)$
$\oplus act2 \ : \ run\_peers(s) := run\_peers(s) \setminus suspc\_peers(token\_owner(s) \mapsto s)$
$\ominus act3 \ : \ fail\_peers := fail\_peers \cup (\{s\} \times susp\_peers(s))$
$\oplus act3 \ : \ fail\_peers := fail\_peers \cup (\{s\} \times suspc\_peers(token\_owner(s) \mapsto s))$
$\ominus act4 \ : \ susp\_peers(s) := \varnothing$
$\oplus act4 \ : \ suspc\_peers(token\_owner(s) \mapsto s) := \varnothing$
...
END

Ce douzième raffinement peut se résumer par le diagramme suivant :

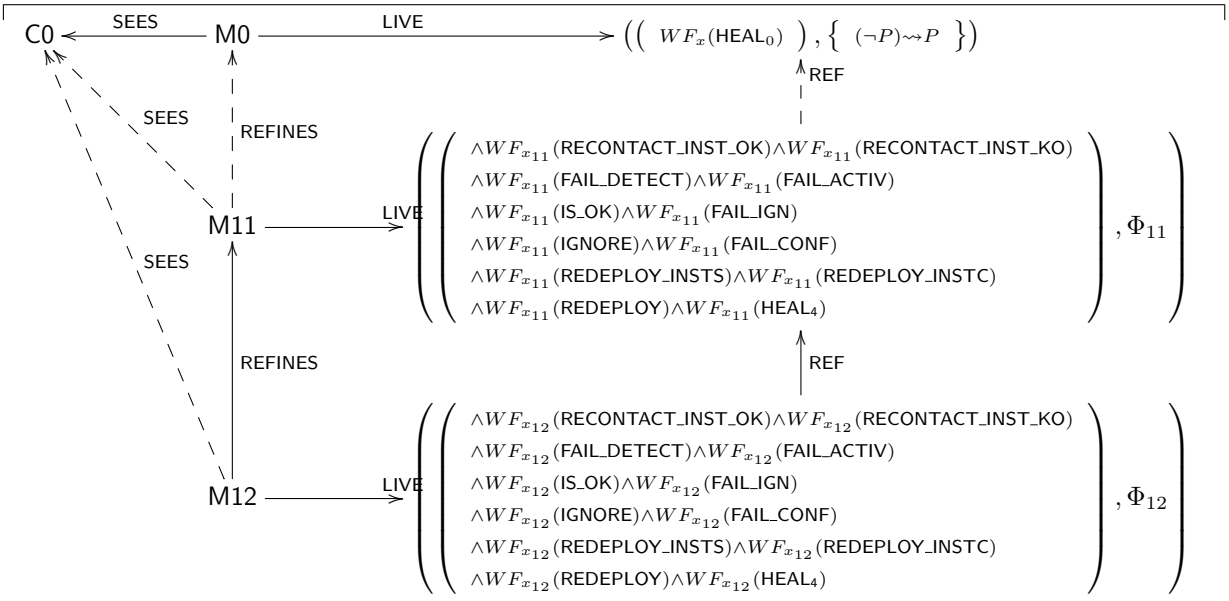


FIGURE 8.17 – Douzième raffinement du self-healing

Ce douzième raffinement nous montre que les instances « token owner » de chaque service sont au cœur des phases de détection d'états illégaux (**self-detection**) et d'évaluation des états illégaux, si ceux-ci sont critiques ou non (**self-activation**). Les raffinements suivant étendent l'influence des instances « token owner » durant toute la procédure de *self-healing* d'un service.

8.3.4.9 Entrée d'un service dans un état illégal : détails

Ce treizième raffinement introduit une modification de l'entrée d'un service dans un état illégal, modélisée par les événements MAKE_PEER_UNAVAIL, SUSPECT_INST et FAILURE. Il s'agit ici de rendre plus concrète la relation *susp_inst* couplant les instances indisponibles (ne pouvant fournir aucun service)

aux instances « token owner » des service les ayant détectées, en la remplaçant par une fonction *suspc_inst* associant à chaque instance « token owner » de chaque service, un ensemble d'instances indisponibles. Nous rappelons ici que nous utilisons une partie du contexte C8 dans ce raffinement : nous réutilisons la constante $InitSuspPeers$ associant à chaque instance « token owner » de chaque service une liste initialement vide ($\varnothing$) d'instances.

Une machine M13, raffinant la machine M12 vue dans la sous-section précédente, introduit la nouvelle variable *suspc_inst* :

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus act8$: $susp\_inst := \varnothing$
$\oplus act8$: $suspc\_inst := InitSuspPeers$
...
END

Nous notons x_{13} l'ensemble des variables de cette machine M13.

Un invariant noté $I_{13}(x_{13})$ caractérise ces variables et est défini comme suit :

— Nous avons des propriétés de typage et d'autres propriétés relatives à la nouvelle variable *suspc_inst* :

$$inv1 : suspc\_inst \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS)$$
$$inv2 : \forall p, s \cdot \left(\begin{array}{l} \wedge p \in PEERS \\ \wedge s \in SERVICES \\ \wedge (p \mapsto s) \in dom(suspc\_inst) \end{array} \right) \Rightarrow p = token\_owner(s)$$
$$inv3 : \forall p, s \cdot \left(\begin{array}{l} \wedge p \in PEERS \\ \wedge s \in SERVICES \\ \wedge p = token\_owner(s) \end{array} \right) \Rightarrow (p \mapsto s) \in dom(suspc\_inst)$$

  — La variable *suspc_inst* associe à chaque instance « token owner » d'un service s ($inv2$, $inv3$), un sous-ensemble de pairs (instances) devenus indisponibles (incapables de fournir le service s) ($inv1$). Elle est initialisée à l'aide de la constante $InitSuspPeers$, car initialement, la liste des pairs indisponibles par « token owner » est vide : tous les pairs sont initialement disponibles et fournissent les services qui leur ont été demandés.

— Nous remplaçons la variable abstraite *susp_inst* par la variable plus concrète *suspc_inst*. L'invariant $inv4$, aussi appelé *invariant de collage*, établit les liens entre ces deux variables et permet le remplacement :

$$inv4 : \forall p, s \cdot \left(\begin{array}{l} \wedge p \in PEERS \\ \wedge s \in SERVICES \\ \wedge (p \mapsto s) \in dom(suspc\_inst) \end{array} \right) \Rightarrow susp\_inst[\{p\}][\{s\}] = suspc\_inst(p \mapsto s)$$

  — $inv4$ exprime que les instances indisponibles d'un service s (incapables de fournir le service s) au niveau abstrait sont les mêmes que celles contenues dans l'ensemble maintenu par l'instance « token owner » de ce dernier, au niveau concret. Nous pouvons donc utiliser la variable concrète *suspc_inst* à la place de celle abstraite *susp_inst*.

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
susp_inst	*suspc_inst*

Les axiomes du contexte C8, ainsi que l'invariant de collage I_{13} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M12 en M13, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M13, ainsi que pour justifier la préservation des propriétés Φ_{12} lors du raffinement.

Une liste Φ_{13} de propriétés de vivacité, détaillée en annexe, guide le raffinement de M12 en M13, en présentant les nouveaux événements ou ceux ayant changé par raffinement :

— Evénement MAKE_PEER_UNAVAIL : Nous considérons ici deux cas. **(1)** Si l'instance « token owner » d'un service *srv* ne fait pas partie des instances qui seront rendues indisponibles, alors cette

instance n'est pas remplacée et elle continue de maintenir la même liste d'instances indisponibles ($grd15$, $act3$), **(2)** sinon, l'instance « token owner » du service srv est remplacée par une instance dont la liste d'instances indisponibles est vide ($grd16$, $act3$), signifiant que cette liste doit être reconstruite ou que le service srv se trouve dans une phase de la procédure de *self-healing* ne nécessitant pas cette liste d'instances indisponibles.

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL $\widehat{=}$
ANY
$\ldots$
$\oplus s\_i$
WHERE
$\ldots$
$\oplus grd13 \ : \ s\_i \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS)$
$\oplus grd14 \ : \ \forall p, s \cdot \left(\begin{array}{l} \wedge p \in PEERS \\ \wedge s \in SERVICES \\ \wedge p \mapsto s \in dom(s\_i) \end{array} \right) \Rightarrow p = E(s)$
$\oplus grd15 \ : \ \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \notin prs \end{array} \right) \Rightarrow \left(\begin{array}{c} (E(srv) \mapsto srv) \\ \mapsto \\ suspc\_inst(E(srv) \mapsto srv) \end{array} \right) \in s\_i$
$\oplus grd16 \ : \ \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \in prs \end{array} \right) \Rightarrow \left(\begin{array}{c} (E(srv) \mapsto srv) \\ \mapsto \\ \varnothing \end{array} \right) \in s\_i$
THEN
$\ldots$
$\ominus act3 \ : \ susp\_inst := prs \lhd susp\_inst$
$\oplus act3 \ : \ suspc\_inst := s\_i$
$\ldots$
END

— Evénement SUSPECT_INST : cet événement modélise la détection par une instance « token owner » d'un service s d'un ensemble $susp$ d'instances indisponibles non-vide. L'instance « token owner » initialise alors sa liste d'instances indisponibles pour le service s à l'aide de cet ensemble $susp$ ($grd2$, $act1$).

EVENT SUSPECT_INST REFINES SUSPECT_INST $\widehat{=}$
ANY
$\ldots$
WHERE
$\ldots$
$\ominus grd4 \ : \ s \notin dom(susp\_inst[\{token\_owner(s)\}])$
$\oplus grd4 \ : \ suspc\_inst(token\_owner(s) \mapsto s) = \varnothing$
$\ldots$
THEN
$\ominus act1 \ : \ susp\_inst := susp\_inst \cup (\{token\_owner(s)\} \times (\{s\} \times susp))$
$\oplus act1 \ : \ suspc\_inst(token\_owner(s) \mapsto s) := susp$
END

— Evénement FAILURE : la liste des instances indisponibles d'un service s, maintenue par son instance « token owner » n'est pas vide ($grd3$). Les instances indisponibles sont alors suspectées comme étant réellement dans des états illégaux ($act2$) et la liste des instances indisponibles est remise à zéro ($act3$). Le service entre par conséquent dans un état illégal.

EVENT FAILURE REFINES FAILURE $\widehat{=}$
ANY
$\ldots$
WHERE
$\ldots$
$\ominus grd3 \ : \ susp\_inst[\{token\_owner(s)\}][\{s\}] \neq \varnothing$
$\oplus grd3 \ : \ suspc\_inst(token\_owner(s) \mapsto s) \neq \varnothing$
THEN
$\ldots$
$\ominus act2 \ : \ suspc\_peers(token\_owner(s) \mapsto s) := susp\_inst[\{token\_owner(s)\}][\{s\}]$
$\oplus act2 \ : \ suspc\_peers(token\_owner(s) \mapsto s) := suspc\_inst(token\_owner(s) \mapsto s)$
$\ominus act3 \ : \ susp\_inst := susp\_inst \rhd (\{s\} \lhd ran(susp\_inst))$
$\oplus act3 \ : \ suspc\_inst(token\_owner(s) \mapsto s) := \varnothing$
END

Ce treizième raffinement peut se résumer par le diagramme suivant :

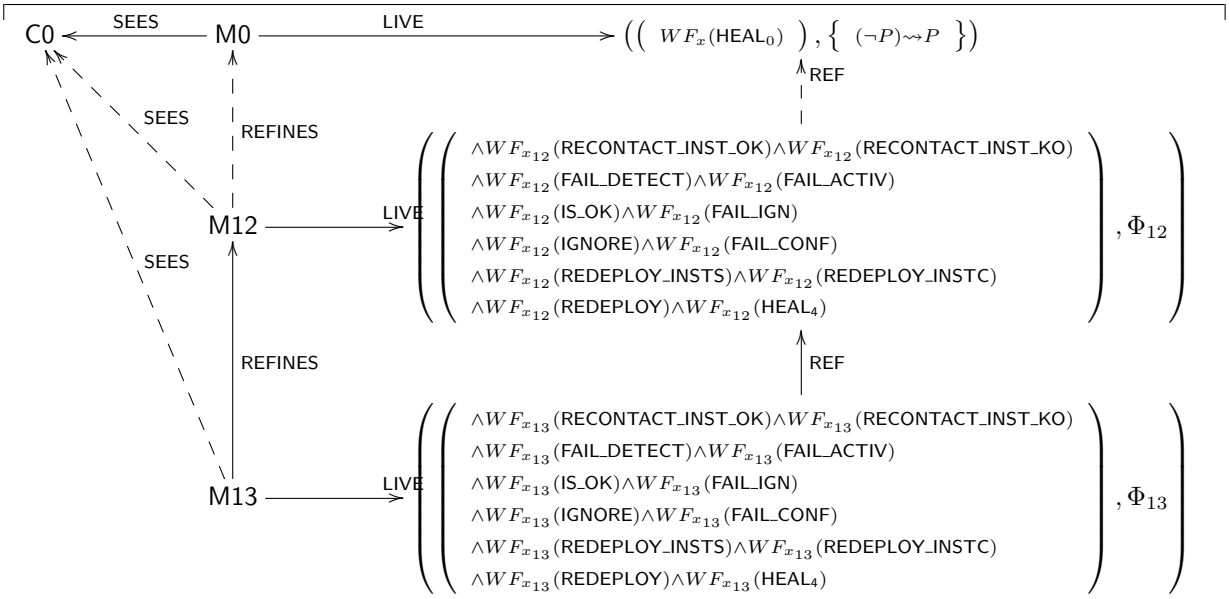


FIGURE 8.18 – Treizième raffinement du self-healing

Ce raffinement recentre une partie du modèle qui ne modélise pas une des phases des la procédure de *self-healing* d'un service sur l'instance « token owner » du service : en effet, c'est cette dernière qui maintient les listes d'instances défaillantes, suspectées d'être dans des états illégaux du service. Elle est donc celle qui constate l'entrée du service dans un état illégal.

8.3.4.10 Self-detection et self-activation : début de localisation

Ce quatorzième raffinement reprend les idées du précédent : il s'agit ici de rendre les modèles de plus en plus concrets en transformant des relations couplant des services ou des instances « token owner » à des instances, en fonctions qui associent soit des services, soit des instances « token owner » à des ensembles d'instances. Nous commençons à localiser ici les phases de **self-detection** et **self-activation**, mais nous nous concentrons principalement sur la phase de **self-detection**. Nous rappelons ici que nous utilisons des éléments du contexte C8, qui sont les suivants :

— Une constante $InitFail$ qui associe à chaque service un ensemble de pairs (instances) vide ($\varnothing$).
— Une constante $InitSuspPeers$ qui associe à chaque instance « token owner » de chaque service un ensemble de pairs (instances) vide ($\varnothing$).

La machine M14 qui utilise ces éléments introduit quatre nouvelles variables $failr\_peers$, dep_instc, $rect\_inst$ et $rctt\_inst$:

```
INITIALISATION ≙
  BEGIN
    ...
    ⊖act3 : fail_peers := ∅
    ⊕act3 : failr_peers := InitFail
    ⊖act4 : dep_inst := ∅
    ⊕act4 : dep_instc := InitFail
    ...
    ⊖act7 : rec_inst := ∅
    ⊕act7 : rect_inst := InitSuspPeers
    ⊖act8 : rct_inst := ∅
    ⊕act8 : rctt_inst := InitSuspPeers
    ...
  END
```

— $failr\_peers$ associe à chaque service un ensemble de pairs (instances) dans un état illégal/défaillants ($inv4$). Cette variable est initialisée à l'aide de la constante $InitFail$: à l'état initial, aucune instance n'est dans un état illégal.

— *dep_instc* associe à chaque service un ensemble de pairs (instances) déployés ($inv2$). Cette variable est initialisée à l'aide de la constante $InitFail$: à l'état initial, aucun service n'est encore passé par la phase de redéploiement.
— *rect_inst* associe à chaque instance « token owner » de chaque service ($inv5$, $inv6$) un ensemble de pairs (instances) que l'instance « token owner » a essayé de contacter. Cette variable est initialisée à l'aide de la constante $InitSuspPeers$: à l'état initial, aucune instance « token owner » n'a essayé de contacter des instances.
— *rctt_inst* associe à chaque instance « token owner » de chaque service ($inv5$, $inv6$) un ensemble de pairs (instances) que l'instance « token owner » a essayé et réussi à contacter. Cette variable est initialisée à l'aide de la constante $InitSuspPeers$: à l'état initial, aucune instance « token owner » n'a contacté des instances.

Nous notons x_{14} l'ensemble des variables de M14.

Un invariant que nous notons $I_{14}(x_{14})$ caractérise ces variables et est défini comme suit :

— Nous avons des propriétés de typage ($inv1$ à $inv4$) et des propriétés relatives aux nouvelles variables ($inv6$ à $inv8$), exprimant les descriptions et fonctionnalités, vues précédemment, de ces variables.

$$
\begin{array}{l}
inv1 \ : \ rect\_inst \in (PEERS \times SERVICES) \pfun \mathbb{P}(PEERS) \\
inv2 \ : \ dep\_instc \in SERVICES \to \mathbb{P}(PEERS) \\
inv3 \ : \ rctt\_inst \in (PEERS \times SERVICES) \pfun \mathbb{P}(PEERS) \\
inv4 \ : \ failr\_peers \in SERVICES \to \mathbb{P}(PEERS) \\
inv5 \ : \ \forall p, s \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \land (p \mapsto s) \in dom(rect\_inst) \end{array} \right) \Rightarrow p = token\_owner(s) \\
inv6 \ : \ \forall p, s \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land p = token\_owner(s) \end{array} \right) \Rightarrow (p \mapsto s) \in dom(rect\_inst) \\
inv7 \ : \ \forall p, s \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land (p \mapsto s) \in dom(rctt\_inst) \end{array} \right) \Rightarrow p = token\_owner(s) \\
inv8 \ : \ \forall p, s \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land p = token\_owner(s) \end{array} \right) \Rightarrow (p \mapsto s) \in dom(rctt\_inst)
\end{array}
$$

— Les propriétés de $inv9$ à $inv12$ suivants font le lien entre les variables concrètes et celles abstraites, permettant ainsi le remplacement de ces dernières par les variables concrètes :

$$
\begin{array}{l}
inv9 \ : \ \forall p, s \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land (p \mapsto s) \in dom(rect\_inst) \end{array} \right) \Rightarrow rec\_inst[\{p\}][\{s\}] = rect\_inst(p \mapsto s) \\
inv10 \ : \ \forall p, s \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land (p \mapsto s) \in dom(rctt\_inst) \end{array} \right) \Rightarrow rct\_inst[\{p\}][\{s\}] = rctt\_inst(p \mapsto s) \\
inv11 \ : \ \forall s \cdot s \in SERVICES \Rightarrow fail\_peers[\{s\}] = failr\_peers(s) \\
inv12 \ : \ \forall s \cdot s \in SERVICES \Rightarrow dep\_inst[\{s\}] = dep\_instc(s)
\end{array}
$$

— $inv9$ exprime que les instances d'un service s, que l'instance « token owner » a essayé de recontacter, sont les mêmes, aussi bien au niveau abstrait, qu'au niveau concret. Le remplacement de *rec_inst* par *rect_inst* est donc possible.
— $inv10$ exprime que les instances d'un service s, que l'instance « token owner » a recontacté avec succès, sont les mêmes, aussi bien au niveau abstrait, qu'au niveau concret. Le remplacement de *rct_inst* par *rctt_inst* est donc possible.
— $inv11$ exprime que les instances dans un état illégal d'un service s sont les mêmes, aussi bien au niveau abstrait, qu'au niveau concret. Le remplacement de $fail\_peers$ par $failr\_peers$ est donc possible.
— $inv11$ exprime que les instances déployées (lors de la phase de configuration) d'un service s sont les mêmes, aussi bien au niveau abstrait, qu'au niveau concret. Le remplacement de *dep_inst* par *dep_instc* est donc possible.

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
rec_inst	$rect\_inst$
rct_inst	$rctt\_inst$
$fail\_peers$	$failr\_peers$
dep_inst	dep_instc

Les axiomes du contexte C8, ainsi que l'invariant de collage I_{14} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M13 en M14, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M14, ainsi que pour justifier la préservation des propriétés Φ_{13} lors du raffinement.

Une liste Φ_{14} de propriétés de vivacité, détaillée en annexe [14], permet de guider le raffinement de M13 en M14, en présentant les nouveaux événements ou ceux ayant changé par raffinement :

— L'événement MAKE_PEER_UNAVAIL modélise le fait qu'un ensemble de pairs prs deviennent indisponibles (ne fournissent plus de services) à un moment donné. Nous insistons dans cet événement sur l'importance des instances « token owner » de chaque service : **(1)** si une instance « token owner » d'un service srv venait à être indisponible, un remplaçant capable d'effectuer les tâches en tant que « token owner » (maintenir une liste de pairs suspects, à recontacter, à redéployer, etc) ($grd10$, $grd11$, $grd14$) et ayant monitoré les mêmes états du service srv ($grd12$), est choisi en tant que nouveau « token owner » pour le service srv, **(2)** sinon, l'instance « token owner » du service srv ne change pas ($grd9$, $grd12$, $grd14$). En fonction de ces deux cas, de nouvelles valeurs sont affectées aux variables $token\_owner$, $rctt\_inst$, $rect\_inst$, $actv\_inst$, i_state, $suspc\_peers$ et $suspc\_inst$ ($act2$ à $act8$) : les valeurs de ces variables sont liées aux changements d'instances « token owner » des services, car ceux sont ces dernières qui maintiennent les listes d'instances suspectes, à recontacter, à redéployer, etc.

14. `http://andriami.bruno.free.fr/andriami/main/short/annexe.pdf`

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL
ANY
$prs, E, i\_s, p\_s, s\_i, rc\_s, rt\_s$
WHERE
$grd1 : prs \subseteq PEERS$
$grd2 : prs \not\subseteq unav\_peers$
$grd3 : E \in SERVICES \rightarrow PEERS$
$grd4 : i\_s \in (PEERS \times SERVICES) \nrightarrow STATES\_4$
$grd5 : p\_s \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS)$
$grd6 : s\_i \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS)$
$grd7 : rt\_s \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS)$
$grd8 : rc\_s \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS)$
$grd9 : \forall srv \cdot (srv \in SERVICES \wedge token\_owner(srv) \notin prs) \Rightarrow E(srv) = token\_owner(srv)$
$$grd10 : \forall srv \cdot \begin{pmatrix} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \in prs \end{pmatrix} \Rightarrow E(srv) \in \begin{pmatrix} run\_peers(srv) \\ \setminus \\ \begin{pmatrix} \cup unav\_peers \cup prs \cup failr\_peers(srv) \\ \cup suspc\_peers \begin{pmatrix} token\_owner(srv) \\ \mapsto \\ srv \end{pmatrix} \end{pmatrix} \end{pmatrix}$$
$grd11 : dom(i\_s) = E^{-1} \wedge dom(p\_s) = dom(i\_s) \wedge dom(s\_i) = dom(i\_s) \wedge dom(rc\_s) = dom(i\_s) \wedge dom(rt\_s) = dom(i\_s)$
$$grd12 : \forall srv \cdot srv \in SERVICES \Rightarrow \begin{pmatrix} \wedge \begin{pmatrix} (E(srv) \mapsto srv) \\ \mapsto \\ i\_state(token\_owner(srv) \mapsto srv) \end{pmatrix} \in i\_s \\ \wedge \begin{pmatrix} (E(srv) \mapsto srv) \\ \mapsto \\ suspc\_peers(token\_owner(srv) \mapsto srv) \end{pmatrix} \in p\_s \end{pmatrix}$$
$$grd13 : \forall srv \cdot \begin{pmatrix} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \notin prs \end{pmatrix} \Rightarrow \begin{pmatrix} \wedge ((E(srv) \mapsto srv) \mapsto suspc\_inst(E(srv) \mapsto srv)) \in s\_i \\ \wedge ((E(srv) \mapsto srv) \mapsto rctt\_inst(E(srv) \mapsto srv)) \in rt\_s \\ \wedge ((E(srv) \mapsto srv) \mapsto rect\_inst(E(srv) \mapsto srv)) \in rc\_s \end{pmatrix}$$
$$grd14 : \forall srv \cdot (srv \in SERVICES \wedge token\_owner(srv) \in prs) \Rightarrow \begin{pmatrix} \wedge ((E(srv) \mapsto srv) \mapsto \varnothing) \in s\_i \\ \wedge ((E(srv) \mapsto srv) \mapsto \varnothing) \in rt\_s \\ \wedge ((E(srv) \mapsto srv) \mapsto \varnothing) \in rc\_s \end{pmatrix}$$
THEN
$act1 : unav\_peers := unav\_peers \cup prs$
$act2 : token\_owner := token\_owner \Leftarrow E$
$act3 : rect\_inst := rc\_s$
$act4 : rctt\_inst := rt\_s$
$act5 : actv\_inst := prs \lhd actv\_inst$
$act6 : i\_state := i\_s$
$act7 : suspc\_peers := p\_s$
$act8 : suspc\_inst := s\_i$
END

— Evénements de la phase de **self-detection** :
  — Evénements RECONTACT_INST_OK et RECONTACT_INST_KO : une instance i suspecte d'un service s est recontactée par l'instance « token owner » de cette dernière ($grd6$, $grd7$, $act1$). Cette opération considère deux cas : **(1)** soit l' instance « token owner » réussit à recontacter l'instance i car cette dernière n'est plus indisponible ($grd7$ et $act2$ de RECONTACT_INST_OK), soit l'instance « token owner » n'a pas réussi à recontacter l'instance i car cette dernière est demeurée indisponible ($grd7$ de RECONTACT_INST_KO).

EVENT RECONTACT_INST_OK REFINES RECONTACT_INST_OK
ANY
...
WHERE
...
$\ominus grd6 : token\_owner(s) \mapsto (s \mapsto i) \notin rec\_inst$
$\oplus grd6 : i \notin rect\_inst(token\_owner(s) \mapsto s)$
$\ominus grd7 : rec\_inst[\{token\_owner(s)\}][\{s\}] \subset suspc\_peers(token\_owner(s) \mapsto s)$
$\oplus grd7 : rect\_inst(token\_owner(s) \mapsto s) \subset suspc\_peers(token\_owner(s) \mapsto s)$
THEN
$\ominus act1 : rec\_inst := rec\_inst \cup \{token\_owner(s) \mapsto (s \mapsto i)\}$
$\oplus act1 : rect\_inst(token\_owner(s) \mapsto s) := rect\_inst(token\_owner(s) \mapsto s) \cup \{i\}$
$\ominus act2 : rct\_inst := rct\_inst \cup \{token\_owner(s) \mapsto (s \mapsto i)\}$
$\oplus act2 : rctt\_inst(token\_owner(s) \mapsto s) := rctt\_inst(token\_owner(s) \mapsto s) \cup \{i\}$
END

EVENT RECONTACT_INST_KO REFINES RECONTACT_INST_KO
ANY
...
WHERE
...
$\ominus grd6$: $token\_owner(s) \mapsto (s \mapsto i) \notin rec\_inst$
$\oplus grd6$: $i \notin rect\_inst(token\_owner(s) \mapsto s)$
$\ominus grd7$: $rec\_inst[\{token\_owner(s)\}][\{s\}] \subset suspc\_peers(token\_owner(s) \mapsto s)$
$\oplus grd7$: $rect\_inst(token\_owner(s) \mapsto s) \subset suspc\_peers(token\_owner(s) \mapsto s)$
THEN
$\ominus act1$: $rec\_inst := rec\_inst \cup \{token\_owner(s) \mapsto (s \mapsto i)\}$
$\oplus act1$: $rect\_inst(token\_owner(s) \mapsto s) := rect\_inst(token\_owner(s) \mapsto s) \cup \{i\}$
END

— Evénement FAIL_DETECT : l'instance « token owner » d'un service s a essayé de recontacter toutes les instances qu'elle a considérées comme suspectes ($grd4$). Elle retire de la liste des instances suspectes, les instances qu'elle a réussi à recontacter ($act2$). Les listes des instances recontactées sont alors remises à zéro ($act3$, $act4$).

EVENT FAIL_DETECT REFINES FAIL_DETECT
ANY
...
WHERE
...
$\ominus grd4$: $rec\_inst[\{token\_owner(s)\}][\{s\}] = suspc\_peers(token\_owner(s) \mapsto s)$
$\oplus grd4$: $rect\_inst(token\_owner(s) \mapsto s) = suspc\_peers(token\_owner(s) \mapsto s)$
THEN
...
$\ominus act2$: $suspc\_peers(token\_owner(s) \mapsto s) := \left(\begin{array}{c} suspc\_peers(token\_owner(s) \mapsto s) \\ \setminus \\ rct\_inst[\{token\_owner(s)\}][\{s\}] \end{array} \right)$
$\oplus act2$: $suspc\_peers(token\_owner(s) \mapsto s) := \left(\begin{array}{c} suspc\_peers(token\_owner(s) \mapsto s) \\ \setminus \\ rctt\_inst(token\_owner(s) \mapsto s) \end{array} \right)$
$\ominus act3$: $rec\_inst := rec\_inst \rhd (\{s\} \lhd ran(rec\_inst))$
$\oplus act3$: $rect\_inst(token\_owner(s) \mapsto s) := \varnothing$
$\ominus act4$: $rct\_inst := rct\_inst \rhd (\{s\} \lhd ran(rct\_inst))$
$\oplus act4$: $rctt\_inst(token\_owner(s) \mapsto s) := \varnothing$
END

— Evénement FAIL_ACTIV : les instances du services s qui sont restées suspectes sont ajoutées à l'ensemble des pairs instanciant le service s dans un état illégal ($act3$).

EVENT FAIL_ACTIV REFINES FAIL_ACTIV
ANY
...
WHERE
...
THEN
...
$\ominus act3$: $fail\_peers := fail\_peers \cup (\{s\} \times suspc\_peers(token\_owner(s) \mapsto s))$
$\oplus act3$: $failr\_peers(s) := failr\_peers(s) \cup suspc\_peers(token\_owner(s) \mapsto s)$
...
END

— Evénement REDEPLOY_INSTC : l'instance i à déployer du service s doit être un pair n'instanciant pas encore le service s, n'étant pas dans un état illégal, ni indisponible, et n'ayant pas encore été déployé pour fournir le service s ($grd3$).

EVENT REDEPLOY_INSTC REFINES REDEPLOY_INSTC
ANY
...
WHERE
...
$\ominus grd3$: $i \notin run\_peers(s) \cup fail\_peers[\{s\}] \cup unav\_peers \cup dep\_inst[\{s\}]$
$\oplus grd3$: $i \notin run\_peers(s) \cup failr\_peers(s) \cup unav\_peers \cup dep\_instc(s)$
...
THEN
...
END

— Evénement UNFAIL_PEER : un pair p instanciant un service s et considéré comme étant dans un

état illégal (défaillant) ($grd3$), retourne dans un état légal : il est retiré de la liste des pairs dans un état illégal instanciant le service s ($act1$).

EVENT UNFAIL_PEER REFINES UNFAIL_PEER
ANY
...
WHERE
...
$\ominus grd3$: $s \mapsto p \in fail\_peers$
$\oplus grd3$: $p \in failr\_peers(s)$
THEN
$\ominus act1$: $fail\_peers := fail\_peers \setminus \{s \mapsto p\}$
$\oplus act1$: $failr\_peers(s) := failr\_peers(s) \setminus \{p\}$
END

Ce quatorzième raffinement peut se résumer par le diagramme suivant :

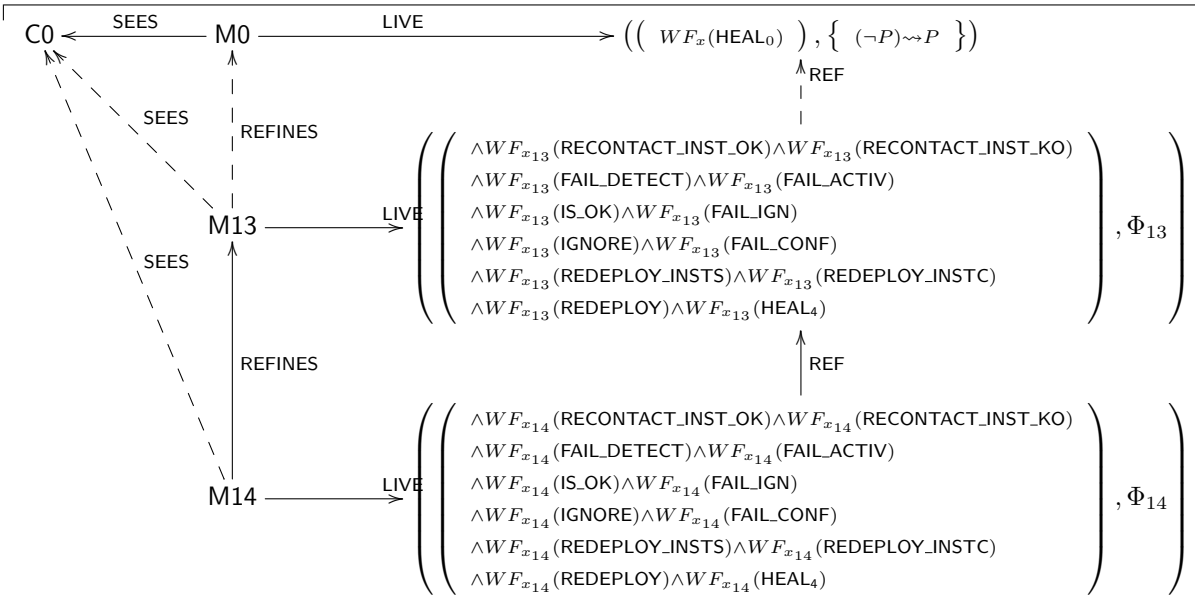


FIGURE 8.19 – Quatorzième raffinement du self-healing

Ce raffinement nous a permis d'étendre encore plus le champ d'action des instances « token owner » des services, notamment dans la phase de **self-detection** : nous remarquons que maintenant, tous les événements modélisant des procédures de cette phase (RECONTACT_INST_OK, RECONTACT_INST_KO, FAIL_DETECT, IS_OK) sont localisés et centrés sur les instances « token owner », qui déterminent les entrées et sorties des services de cette phase de **self-detection**.

8.3.4.11 Self-configuration : début de localisation

Ce raffinement détaille la phase de **self-configuration** d'un service : le nombre d'instances capables de fournir le service est inférieur au minimum requis pour le fournir. L'instance « token owner » du service choisit alors de déployer (activer) de nouvelles instances du service, jusqu'à ce que le minimum soit atteint. Nous rappelons ici que nous utilisons une partie du contexte C8 vu précédemment, notamment la constante $InitSuspPeers$ qui associe à chaque instance « token owner » de chaque service, un ensemble d'instances vide ($\varnothing$).

La machine M15, raffinant la machine M14 précédente, introduit une nouvelle variable $actv\_instc$:

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus act10$: $actv\_inst := \varnothing$
$\oplus act10$: $actv\_instc := InitSuspPeers$
...
END

— La variable *actv_instc* associe chaque instance « token owner » de chaque service ($inv2$, $inv3$) à un ensemble d'instances ($inv1$) que l'instance « token owner » a activé (déployé). Cette variable est initialisée à l'aide de la constante $InitSuspPeers$, parce qu'à l'état initial, aucune nouvelle instance d'un service n'est activé par les instances « token owner ».

Nous notons x_{15} l'ensemble des variables de la machine M15.

Un invariant noté $I_{15}(x_{15})$ caractérise ces variables et est défini comme suit :

— Nous avons des propriétés de typage ($inv1$) et des propriétés concernant la variables *actv_instc* exprimant les descriptions et fonctionnalités de cette dernière, telles que vues précédemment.

$$
\begin{array}{l}
inv1 \ : \ actv\_instc \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS) \\
inv2 \ : \ \forall p, s \cdot \left(\begin{array}{l} \wedge p \in PEERS \\ \wedge s \in SERVICES \\ \wedge (p \mapsto s) \in dom(actv\_instc) \end{array} \right) \Rightarrow p = token\_owner(s) \\
inv3 \ : \ \forall p, s \cdot \left(\begin{array}{l} \wedge p \in PEERS \\ \wedge s \in SERVICES \\ \wedge p = token\_owner(s) \end{array} \right) \Rightarrow (p \mapsto s) \in dom(actv\_instc)
\end{array}
$$

— La nouvelle variable *actv_instc* remplace une variable abstraite *actv_inst*. L'invariant de collage $inv4$ fait le lien entre ces deux variables :

$$
inv4 \ : \ \forall p, s \cdot \left(\begin{array}{l} \wedge p \in PEERS \\ \wedge s \in SERVICES \\ \wedge (p \mapsto s) \in dom(actv\_instc) \end{array} \right) \Rightarrow actv\_inst[\{p\}][\{s\}] = actv\_instc(p \mapsto s)
$$

— $inv4$ exprime que les nouvelles instances activées par l'instance « token owner » p d'un service s sont les mêmes aussi bien au niveau abstrait que concret. Nous pouvons par conséquent remplacer la variable *actv_inst* par *actv_instc*.

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
actv_inst	*actv_instc*

Les axiomes du contexte C8, ainsi que l'invariant de collage I_{15} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M14 en M15, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M15, ainsi que pour justifier la préservation des propriétés Φ_{14} lors du raffinement.

Une liste Φ_{15} de propriétés de vivacité, détaillée en annexe, permet de guider le raffinement de M14 en M15, en présentant les nouveaux événements ou ceux ayant changé par raffinement. Lors de ce raffinement, nous remplaçons dans les événements existants les occurrences de la variable *actv_inst* par la nouvelle variable *actv_instc* :

— Evénement MAKE_PEER_UNAVAIL : Cet événement modélise le fait qu'un ensemble prs d'instances de services deviennent indisponibles (incapable de fournir des services). Nous distinguons deux cas pour chaque service srv impacté par des indisponibilités : **(1)** si l'instance « token owner » est toujours disponible, cette instance est conservée, ainsi que ces caractéristiques, dont les instances qu'il a activées ($grd11$, $grd12$), sinon **(2)** une nouvelle instance « token owner » est choisie, les critères étant que cette nouvelle instance a monitoré les mêmes états du service srv que l'ancienne instance « token owner » et a suspecté les mêmes instances ($grd11$). Les variables servant à des calculs intermédiaires sont remises à zéro, signifiant que la nouvelle instance « token owner » doit refaire ces calculs ou qu'elle ne se trouve plus dans des phases nécessitant ces calculs ($grd13$). Les variables servant à des calculs intermédiaires sont mises à jour en fonction de ces deux cas, parce que leurs valeurs dépendent des instances « token owner » des services ($act5$).

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL

$$
\begin{array}{l}
\quad \ldots \\
\quad \oplus ac\_i \\
\text{WHERE} \\
\quad \ldots \\
\quad \ominus grd9 \ : \ \ldots \\
\quad \oplus grd9 \ : \ ac\_i \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS) \\
\quad \ominus grd10 \ : \ \ldots \\
\quad \oplus grd10 \ : \ \left(\begin{array}{l} \wedge dom(i\_s) = E^{-1} \\ \wedge dom(p\_s) = dom(i\_s) \\ \wedge dom(s\_i) = dom(i\_s) \\ \wedge dom(rc\_s) = dom(i\_s) \\ \wedge dom(rt\_s) = dom(i\_s) \\ \wedge dom(ac\_i) = dom(i\_s) \end{array} \right) \\
\quad \ominus grd11 \ : \ \ldots \\
\quad \oplus grd11 \ : \ \forall srv \cdot srv \in SERVICES \Rightarrow \left(\begin{array}{l} \wedge \left(\begin{array}{c} i\_s(E(srv) \mapsto srv) \\ = \\ i\_state(token\_owner(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{c} p\_s(E(srv) \mapsto srv) \\ = \\ suspc\_peers(token\_owner(srv) \mapsto srv) \end{array} \right) \end{array} \right) \\
\quad \ominus grd12 \ : \ \ldots \\
\quad \oplus grd12 \ : \ \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \notin prs \end{array} \right) \Rightarrow \left(\begin{array}{l} \wedge E(srv) = token\_owner(srv) \\ \wedge \left(\begin{array}{c} s\_i(E(srv) \mapsto srv) \\ = \\ suspc\_inst(E(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{c} rt\_s(E(srv) \mapsto srv) \\ = \\ rctt\_inst(E(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{c} rc\_s(E(srv) \mapsto srv) \\ = \\ rect\_inst(E(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{c} ac\_i(E(srv) \mapsto srv) \\ = \\ actv\_instc(E(srv) \mapsto srv) \end{array} \right) \end{array} \right) \\
\quad \ominus grd13 \ : \ \ldots \\
\quad \oplus grd13 \ : \ \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \in prs \end{array} \right) \\
\qquad \Rightarrow \left(\begin{array}{l} \wedge E(srv) \in \left(\begin{array}{l} run\_peers(srv) \\ \setminus \\ \left(\begin{array}{l} \cup unav\_peers \\ \cup prs \\ \cup failr\_peers(srv) \\ \cup suspc\_peers \left(\begin{array}{c} token\_owner(srv) \\ \mapsto \\ srv \end{array} \right) \end{array} \right) \end{array} \right) \\ \wedge (s\_i(E(srv) \mapsto srv) = \varnothing) \\ \wedge (rt\_s(E(srv) \mapsto srv) = \varnothing) \\ \wedge (rc\_s(E(srv) \mapsto srv) = \varnothing) \\ \wedge (ac\_i(E(srv) \mapsto srv) = \varnothing) \end{array} \right) \\
\quad \ominus grd14 \ : \ \ldots \\
\text{THEN} \\
\quad \ldots \\
\quad \ominus act5 \ : \ actv\_inst := prs \lhd actv\_inst \\
\quad \oplus act5 \ : \ actv\_instc := ac\_i \\
\quad \ldots \\
\text{END}
\end{array}
$$

— Evénements modélisant la phase de **self-configuration**, principalement la phase de redéploiement d'un service.

— Evénement REDEPLOY_INSTC : l'instance i que l'instance « token owner » d'un service s a choisi d'activer pour y instancier le service s, n'a pas encore été activée par l'instance « token owner » du service s ($grd4$). L'instance i est donc activée par l'instance « token owner » du service s ($act1$). Cette opération est renouvellée tant que le nombre d'instances déployables (activables) en un coup par le service s n'est pas atteint ($grd6$).

EVENT REDEPLOY_INSTC REFINES REDEPLOY_INSTC $\widehat{=}$
- ...
- $\ominus grd4$: $token\_owner(s) \mapsto (s \mapsto i) \notin actv\_inst$
- $\oplus grd4$: $i \notin actv\_instc(token\_owner(s) \mapsto s)$
- ...
- $\ominus grd6$: $card(actv\_inst[\{token\_owner(s)\}][\{s\}]) < deplo\_inst(s)$
- $\oplus grd6$: $card(actv\_instc(token\_owner(s) \mapsto s)) < deplo\_inst(s)$

THEN
- $\ominus act1$: $actv\_inst := actv\_inst \cup \{token\_owner(s) \mapsto (s \mapsto i)\}$
- $\oplus act1$: $actv\_instc(token\_owner(s) \mapsto s) := actv\_instc(token\_owner(s) \mapsto s) \cup \{i\}$

END

— Evénement REDEPLOY_INSTS : le nombre d'instances déployables (activables) en un coup par un service s n'est pas atteint ($grd2$). Les instances déployées par l'instance « token owner » du service s rejoignent le rang de celles déjà déployées pour le service s ($act1$). La liste des instances déployées par l'instance « token owner » du service s est remise à zéro ($act2$), afin que cette dernière puisse déployer d'autres instances si nécessaire.

EVENT REDEPLOY_INSTS REFINES REDEPLOY_INSTS $\widehat{=}$
- ...
- $\ominus grd2$: $card(actv\_inst[\{token\_owner(s)\}][\{s\}]) = deplo\_inst(s)$
- $\oplus grd2$: $card(actv\_instc(token\_owner(s) \mapsto s)) = deplo\_inst(s)$
- ...

THEN
- $\ominus act1$: $dep\_instc(s) := dep\_instc(s) \cup actv\_inst[\{token\_owner(s)\}][\{s\}]$
- $\oplus act1$: $dep\_instc(s) := dep\_instc(s) \cup actv\_instc(token\_owner(s) \mapsto s)$
- $\ominus act2$: $actv\_inst := actv\_inst \rhd (\{s\} \lhd ran(actv\_inst))$
- $\oplus act2$: $actv\_instc(token\_owner(s) \mapsto s) := \varnothing$

END

— Evénement REDEPLOY : le nombre d'instances totales fournissant un service s est égal au minimum requis ou le dépasse, et l'instance « token owner » du service s a fini de déployer de nouvelles instances ($grd7$). La procédure de redéploiement d'un service se termine alors.

EVENT REDEPLOY REFINES REDEPLOY $\widehat{=}$
- ...
- $\ominus grd7$: $actv\_inst[\{token\_owner(s)\}][\{s\}] = \varnothing$
- $\oplus grd7$: $actv\_instc(token\_owner(s) \mapsto s) = \varnothing$
- ...

THEN
- ...

END

Ce quinzième raffinement peut se résumer par le diagramme suivant :

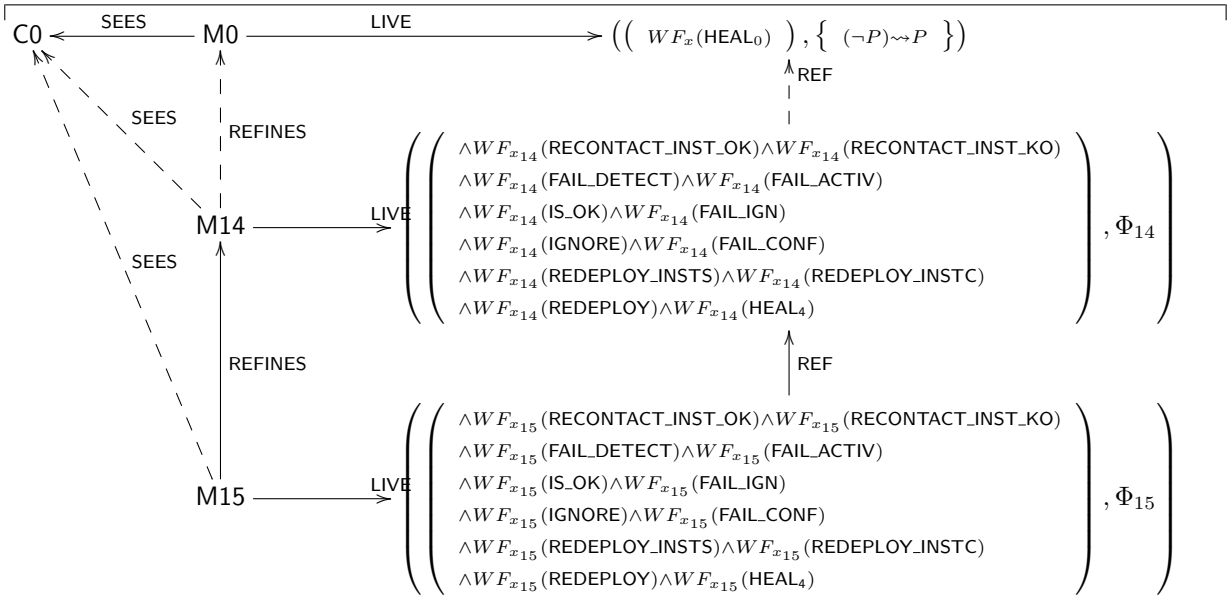


FIGURE 8.20 – Quinzième raffinement du self-healing

Ce raffinement nous montre l'importance de l'instance « token owner » d'un service durant la phase de **self-configuration** : c'est cette instance qui active et déploie de nouvelles instances pour corriger l'état illégal du service.

8.3.5 Rôle de l'instance « token owner » et localisation finale des phases

8.3.5.1 Propagation d'informations par le « token owner » : introduction

Par rapport aux raffinements précédents, qui se concentraient uniquement sur l'introduction d'une instance « token owner » par service, coordonnant les différentes phases de la procédure de *self-healing*, ce raffinement introduit le fait que l'instance « token owner » d'un service ne guide pas seulement la procédure de *self-healing*, mais propage aussi les informations concernant cette procédure (état courant du service, etc.) aux autres instances du service.

Pour mettre en œuvre ces idées, nous commençons par définir un contexte C9, étendant le contexte C8, vu dans les sous-sections précédentes :

CONTEXT C9 EXTENDS C8
CONSTANTS
$InitStateSrv, InitSuspPrs, InitRunPeers$
AXIOMS
$axm1 : InitStateSrv \in PEERS \times SERVICES \pfun STATES\_4$
$axm2 : \forall s, p \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land p \in InitSrvcPeers(s) \end{array} \right) \limp (p \mapsto s) \mapsto RUN\_4 \in InitStateSrv$
$axm3 : \forall s, p, stt \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land (p \mapsto s) \mapsto stt \in InitStateSrv \end{array} \right) \limp p \in InitSrvcPeers(s) \land stt = RUN\_4$
$axm4 : InitSuspPrs \in PEERS \times SERVICES \pfun \pow(PEERS)$
$axm5 : \forall s, p \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land p \in InitSrvcPeers(s) \end{array} \right) \limp (p \mapsto s) \mapsto \varnothing \in InitSuspPrs$
$axm6 : \forall s, p, stt \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land (p \mapsto s) \mapsto stt \in InitSuspPrs \end{array} \right) \limp \left(\begin{array}{l} \land p \in InitSrvcPeers(s) \\ \land stt = \varnothing \end{array} \right)$
$axm7 : InitRunPeers \in PEERS \times SERVICES \pfun \pow(PEERS)$
$axm8 : \forall s, p \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land p \in InitSrvcPeers(s) \end{array} \right) \limp (p \mapsto s) \mapsto InitSrvcPeers(s) \in InitRunPeers$
$axm9 : \forall s, p, stt \cdot \left(\begin{array}{l} \land p \in PEERS \\ \land s \in SERVICES \\ \land (p \mapsto s) \mapsto stt \in InitRunPeers \end{array} \right) \limp \left(\begin{array}{l} \land p \in InitSrvcPeers(s) \\ \land stt = InitSrvcPeers(s) \end{array} \right)$
END

— Une constante $InitStateSrv$ associant tous les pairs instanciant des services à des états est définie ($axm1$). Les pairs et les services qu'ils instancient sont associés à un état légal RUN_4 ($axm2$, $axm3$).
— Une constante $InitSrvcPeers$ associant chaque pair instanciant un service à un sous-ensemble de pairs est définie ($axm4$). Les pairs et les services qu'ils instancient sont associés à des sous-ensembles vides ($\varnothing$) ($axm5$, $axm6$).
— Une constante $InitRunPeers$ associant chaque pair instanciant un service à un sous-ensemble de pairs est définie ($axm7$). Chaque pair et le service qu'il instancie est associé à l'ensemble des pairs fournissant initialement le service ($axm8$, $axm9$).

Une machine M16, raffinant la machine M15 vue dans la sous-section précédente, introduit une nouvelle variable *inst_state*, modélisant l'état d'une instance d'un service :

INITIALISATION $\widehat{=}$
BEGIN
$\ominus act1 : \ldots$
$\oplus act1 : inst\_state := InitStateSrv$
$\ldots$
END

— *inst_state* associe les instances d'un service à l'état courant de ce dernier ($inv1$).

Nous notons x_{16} l'ensemble des variables de cette machine M16

Un invariant que nous notons $I_{16}(x_{16})$ contraint ces variables et est défini comme suit :

— Nous avons du typage :

$$inv1 \ : \ inst\_state \in (PEERS \times SERVICES) \nrightarrow STATES\_4$$

— La variable *inst_state* remplace la variable abstraite *i_state*. Les invariants *inv*2 et *inv*3 suivants font office d'invariants de collage et établissent les relations entre les deux variables :

$$\begin{array}{l} inv2 \ : \ \forall s \cdot s \in SERVICES \Rightarrow token\_owner(s) \mapsto s \in dom(inst\_state) \\ inv3 \ : \ \forall s \cdot s \in SERVICES \Rightarrow i\_state(token\_owner(s) \mapsto s) = inst\_state(token\_owner(s) \mapsto s) \end{array}$$

— *inv*2 exprime que l'instance « token owner » d'un service s monitore l'état courant du service s.
— *inv*3 exprime que l'état courant monitoré par l'instance « token owner » d'un service s est le même aussi bien au niveau abstrait qu'au niveau concret. Nous pouvons donc remplacer la variable *i_state* par *inst_state*, dans ce raffinement.

— D'autres invariants expriment d'autres propriétés de sûreté satisfaites par ce raffinement :
— Les invariants de *inv*4 à *inv*7 expriment que les instances recontactées (*inv*4, *inv*7), ainsi que celles suspectées d'être dans un état illégal (*inv*5, *inv*6) par l'instance « token owner » d'un service s, font encore partie des instances fonctionnelles du service car elles n'ont pas encore été déclarées comme étant dans un état illégal.

$$\begin{array}{l} inv4 \ : \ \forall s \cdot s \in SERVICES \Rightarrow rctt\_inst(token\_owner(s) \mapsto s) \subseteq run\_peers(s) \\ inv5 \ : \ \forall s \cdot s \in SERVICES \Rightarrow suspc\_peers(token\_owner(s) \mapsto s) \subseteq run\_peers(s) \\ inv6 \ : \ \forall s \cdot s \in SERVICES \Rightarrow suspc\_inst(token\_owner(s) \mapsto s) \subseteq run\_peers(s) \\ inv7 \ : \ \forall s \cdot s \in SERVICES \Rightarrow rect\_inst(token\_owner(s) \mapsto s) \subseteq run\_peers(s) \end{array}$$

— Les invariants de *inv*8 à *inv*11 expriment que l'instance « token owner » d'un service s n'est ni suspectée d'être dans un état illégal (*inv*8, *inv*9), ni recontactée (*inv*10, *inv*11) pour vérifier qu'elle est bien dans un état légal et n'est pas indisponible (*inv*10, *inv*11).

$$\begin{array}{l} inv8 \ : \ \forall s \cdot s \in SERVICES \Rightarrow token\_owner(s) \notin suspc\_inst(token\_owner(s) \mapsto s) \\ inv9 \ : \ \forall s \cdot s \in SERVICES \Rightarrow token\_owner(s) \notin suspc\_peers(token\_owner(s) \mapsto s) \\ inv10 \ : \ \forall s \cdot s \in SERVICES \Rightarrow token\_owner(s) \notin rctt\_inst(token\_owner(s) \mapsto s) \\ inv11 \ : \ \forall s \cdot s \in SERVICES \Rightarrow token\_owner(s) \notin rect\_inst(token\_owner(s) \mapsto s) \end{array}$$

— L'invariant *inv*12 exprime que les instances d'un service s suspectées d'être dans un état illégal par uniquement l'instance « token owner » lors d'un calcul intermédiaire (i.e. les instances dans $suspc\_inst(token\_owner(s) \mapsto s)$ ne sont pas encore incluses dans l'ensemble des instances suspectes déjà connues du service s (i.e. les instances dans $suspc\_peers(token\_owner(s) \mapsto s)$.

$$inv12 \ : \ \forall s \cdot s \in SERVICES \Rightarrow \left(\begin{array}{c} \cap suspc\_inst(token\_owner(s) \mapsto s) \\ \cap suspc\_peers(token\_owner(s) \mapsto s) \end{array} \right) = \varnothing$$

— Les invariants de *inv*13 à *inv*15 expriment des propriétés relatives à l'état courant d'un service s :
— *inv*13 exprime que si un service s ne se trouve pas dans un état illégal ($FAIL\_4$) ou dans la phase de **self-detection** (FL_DT_4), son instance « token owner » n'a pas de raison de maintenir une liste d'instances suspectées d'être dans un état illégal.
— *inv*14 et *inv*15 expriment que si un service s ne se trouve pas dans un état illégal ($FAIL\_4$), alors son instance « token owner » n'a aucune instance suspecte à recontacter, pour vérifier si l'état illégal est vraiment réel.

$$
\begin{array}{l}
inv13 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge \left(\begin{array}{c} inst\_state(token\_owner(s) \mapsto s) \\ \notin \\ \{FAIL\_4, FL\_DT\_4\} \end{array} \right) \end{array} \right) \Rightarrow suspc\_peers(token\_owner(s) \mapsto s) = \varnothing \\
inv14 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge inst\_state(token\_owner(s) \mapsto s) \neq FAIL\_4 \end{array} \right) \Rightarrow rctt\_inst(token\_owner(s) \mapsto s) = \varnothing \\
inv15 \ : \ \forall s \cdot \left(\begin{array}{l} \wedge s \in SERVICES \\ \wedge inst\_state(token\_owner(s) \mapsto s) \neq FAIL\_4 \end{array} \right) \Rightarrow rect\_inst(token\_owner(s) \mapsto s) = \varnothing
\end{array}
$$

Une liste Φ_{16} de propriétés de vivacité, détaillée en annexe, permet de guider le raffinement de M15 en M16, en présentant les nouveaux événements ou ceux ayant changé par raffinement :

— L'événement MAKE_PEER_UNAVAIL modélise le fait qu'un ensemble *prs* d'instances de service deviennent indisponibles (ne fournissent plus les services demandés). Nous n'expliquons que les nouvelles gardes et actions relatives à la variables *inst_state*, les autres étant déjà vues dans les raffinements précédents. Nous nous assurons dans un premier temps qu'il reste des instances de chaque service ayant monitoré les état de ces derniers ($grd4$) et dans ce cas, nous retirons aux pairs membres de *prs* la possibilité de monitorer les états courants des services qu'ils fournissaient avant de devenir indisponibles ($act6$).

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL
ANY
...
$\ominus i\_s$
WHERE
...
$\ominus grd4 \ : \ i\_s \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS)$
$\oplus grd4 \ : \ \forall srv \cdot srv \in SERVICES \Rightarrow dom(dom(inst\_state) \rhd \{srv\}) \setminus prs \neq \varnothing$
$\ominus grd10 \ : \ \ldots$
$\oplus grd10 \ : \ dom(s\_i) = E^{-1} \wedge dom(p\_s) = E^{-1} \wedge dom(rc\_s) = E^{-1} \wedge dom(rt\_s) = E^{-1} \wedge dom(ac\_i) = E^{-1}$
$\ominus grd11 \ : \ \ldots$
$\oplus grd11 \ : \ \forall srv \cdot srv \in SERVICES \Rightarrow (p\_s(E(srv) \mapsto srv) = suspc\_peers(token\_owner(srv) \mapsto srv))$
...
$\ominus grd13 \ : \ \ldots$

$$
\oplus grd13 \ : \ \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge token\_owner(srv) \in prs \end{array} \right) \Rightarrow \left(\begin{array}{l} \wedge E(srv) \in \left(\begin{array}{l} \left(\begin{array}{c} run\_peers(srv) \\ \setminus \\ \cup unav\_peers \cup prs \cup failr\_peers(srv) \end{array} \right) \\ \cup suspc\_peers \left(\begin{array}{c} token\_owner(srv) \\ \mapsto \\ srv \end{array} \right) \end{array} \right) \\ \wedge (s\_i(E(srv) \mapsto srv) = \varnothing) \wedge (rt\_s(E(srv) \mapsto srv) = \varnothing) \\ \wedge (rc\_s(E(srv) \mapsto srv) = \varnothing) \wedge (ac\_i(E(srv) \mapsto srv) = \varnothing) \end{array} \right)
$$

WITH
$i\_s \ : \ i\_s = E^{-1} \lhd inst\_state$
THEN
...
$\ominus act6 \ : \ i\_state := i\_s$
$\oplus act6 \ : \ inst\_state := (prs \times SERVICES) \lhd\!\!\!- inst\_state$
...
END

— La variable *inst_state* est utilisée dans les gardes de SUSPECT_INST, RECONTACT_INST_OK, RECONTACT_INST_KO, REDEPLOY_INSTC et REDEPLOY_INSTS : l'instance « token owner » d'un service s vérifie si l'état courant du service correspond à un état pré-requis, condition nécessaire pour l'observation de l'un de ces événements.
  — Evénement SUSPECT_INST : cet événement est observé si l'instance « token owner » d'un service s détecte que le service s est dans un état légal RUN_4 ($grd5$). L'instance « token owner » suspecte alors les instances du service s devenues indisponibles d'être dans un état illégal.

EVENT SUSPECT_INST REFINES SUSPECT_INST ≙
ANY
...
WHERE
...
$\ominus grd5 \ : \ i\_state(token\_owner(s) \mapsto s) = RUN\_4$
$\oplus grd5 \ : \ inst\_state(token\_owner(s) \mapsto s) = RUN\_4$
...
THEN
...
END

— Evénements RECONTACT_INST_OK et RECONTACT_INST_KO : ces événements sont observés si l'instance « token owner » d'un service s détecte que le service s est dans un état illégal $FAIL\_4$ ($grd3$). L'instance « token owner » essaie de recontacter les instances du service s devenues indisponibles, afin de vérifier si l'état illégal est avéré.

EVENT RECONTACT_INST_OK REFINES RECONTACT_INST_OK ≙
ANY
...
WHERE
...
$\ominus grd3 \ : \ i\_state(token\_owner(s) \mapsto s) = FAIL\_4$
$\oplus grd3 \ : \ inst\_state(token\_owner(s) \mapsto s) = FAIL\_4$
...
THEN
...
END

EVENT RECONTACT_INST_KO REFINES RECONTACT_INST_KO ≙
ANY
...
WHERE
...
$\ominus grd3 \ : \ i\_state(token\_owner(s) \mapsto s) = FAIL\_4$
$\oplus grd3 \ : \ inst\_state(token\_owner(s) \mapsto s) = FAIL\_4$
...
THEN
...
END

— Evénements REDEPLOY_INSTC et REDEPLOY_INSTS : ces événements sont observés si l'instance « token owner » d'un service s détecte que le service s est dans la phase de **self-configuration** (FL_CONF_4) ($grd5$ pour REDEPLOY_INSTC et $grd4$ pour REDEPLOY_INSTS). L'instance « token owner » redéploie alors de nouvelles instances du service s.

EVENT REDEPLOY_INSTC REFINES REDEPLOY_INSTC ≙
ANY
...
WHERE
...
$\ominus grd5 \ : \ i\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
$\oplus grd5 \ : \ inst\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
...
THEN
...
END

EVENT REDEPLOY_INSTS REFINES REDEPLOY_INSTS ≙
ANY
...
WHERE
...
$\ominus grd4 \ : \ i\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
$\oplus grd4 \ : \ inst\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
THEN
...
END

— D'autres événements modélisent les comportements suivants : l'instance « token owner » d'un service s vérifie si l'état courant du service correspond à un état pré-requis, condition nécessaire pour l'observation de l'un de ces événements ; si cette condition est vérifiée, l'instance « token owner » procède ensuite à une modification de cet état, qu'il propage à un ensemble *prop* des instances du service s considérées comme étant fonctionnelles (dans un état légal).

— Evénement FAILURE : le service s se trouve dans l'état RUN_4 ($grd2$) et l'instance « token owner » a détecté des instances suspectes. L'instance « token owner » fait passer le service dans un état illégal $FAIL\_4$ et propage cette information à un ensemble *prop* ($act1$) constitué des instances de s dans un état légal, sans les instances suspectes et celles indisponibles ($grd4$, $grd5$).

EVENT FAILURE REFINES FAILURE $\widehat{=}$
ANY
...
$\oplus prop$
WHERE
...
$\ominus grd2$: $i\_state(token\_owner(s) \mapsto s) = RUN\_4$
$\oplus grd2$: $inst\_state(token\_owner(s) \mapsto s) = RUN\_4$
...
$\oplus grd4$: $prop \subseteq PEERS$
$\oplus grd5$: $prop = run\_peers(s) \setminus (suspc\_inst(token\_owner(s) \mapsto s) \cup unav\_peers)$
THEN
$\ominus act1$: $i\_state(token\_owner(s) \mapsto s) := FAIL\_4$
$\oplus act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FAIL\_4\})$
...
END

— Evénement FAIL_DETECT : le service s se trouve dans un état illégal $FAIL\_4$ ($grd2$). L'instance « token owner » fait passer le service dans la phase de **self-detection** (FL_DT_4) et propage cette information à un ensemble *prop* ($act1$) constitué des instances de s dans un état légal, sans les instances suspectes et celles demeurées indisponibles, mais avec celles que l'instance « token owner » a réussi à recontacter ($grd5$, $grd6$).

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
ANY
...
$\oplus prop$
WHERE
...
$\ominus grd3$: $i\_state(token\_owner(s) \mapsto s) = FAIL\_4$
$\oplus grd3$: $inst\_state(token\_owner(s) \mapsto s) = FAIL\_4$
...
$\oplus grd5$: $prop \subseteq PEERS$
$\oplus grd6$: $prop = \left(\begin{array}{l} \cup((run\_peers(s) \setminus suspc\_peers(token\_owner(s) \mapsto s)) \\ \cup rctt\_inst(token\_owner(s) \mapsto s)) \setminus unav\_peers \end{array} \right)$
...
THEN
$\ominus act1$: $i\_state(token\_owner(s) \mapsto s) := FL\_DT\_4$
$\oplus act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FL\_DT\_4\})$
...
END

— Evénement IS_OK : le service s se trouve dans la phase de **self-detection** (FL_DT_4) ($grd2$). L'instance « token owner », ayant réussi à recontacter toutes les instances de s suspectes, fait passer le service dans un état légal (RUN_4) et propage cette information à un ensemble *prop* ($act1$) constitué des instances de s dans un état légal, sans les instances indisponibles du service s ($grd4$, $grd5$).

EVENT IS_OK REFINES IS_OK $\widehat{=}$
ANY
$\ldots$
$\oplus prop$
WHERE
$\ldots$
$\ominus grd2 \ : \ i\_state(token\_owner(s) \mapsto s) = FL\_DT\_4$
$\oplus grd2 \ : \ inst\_state(token\_owner(s) \mapsto s) = FL\_DT\_4$
$\ldots$
$\oplus grd4 \ : \ prop \subseteq PEERS$
$\oplus grd5 \ : \ prop = run\_peers(s) \setminus unav\_peers$
THEN
$\ominus act1 \ : \ i\_state(token\_owner(s) \mapsto s) := RUN\_4$
$\oplus act1 \ : \ inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{RUN\_4\})$
END

— Evénement FAIL_ACTIV : le service s se trouve dans la phase de **self-detection** (FL_DT_4) ($grd2$). L'instance « token owner », n'ayant pas réussi à recontacter toutes les instances de s suspectes, fait passer le service dans la phase de **self-activation** (FL_ACT_4) et propage cette information à un ensemble $prop$ ($act1$) constitué des instances de s dans un état légal, sans les instances indisponibles du service s ($grd4$, $grd5$).

EVENT FAIL_ACTIV REFINES FAIL_ACTIV $\widehat{=}$
ANY
$\ldots$
$\oplus prop$
WHERE
$\ldots$
$\ominus grd2 \ : \ i\_state(token\_owner(s) \mapsto s) = FL\_DT\_4$
$\oplus grd2 \ : \ inst\_state(token\_owner(s) \mapsto s) = FL\_DT\_4$
$\ldots$
$\oplus grd4 \ : \ prop \subseteq PEERS$
$\oplus grd5 \ : \ prop = run\_peers(s) \setminus (unav\_peers \cup suspc\_peers(token\_owner(s) \mapsto s))$
THEN
$\ominus act1 \ : \ i\_state(token\_owner(s) \mapsto s) := FL\_ACT\_4$
$\oplus act1 \ : \ inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FL\_ACT\_4\})$
$\ldots$
END

— Evénement FAIL_IGNORE : le service s se trouve dans la phase de **self-activation** (FL_ACT_4) ($grd2$). L'instance « token owner », ayant évalué que l'état illégal du service s n'était pas critique, fait passer le service dans un état $FAIL\_IGN\_4$ et propage cette information à un ensemble $prop$ ($act1$) constitué des instances de s dans un état légal, sans les instances indisponibles du service s ($grd4$, $grd5$).

EVENT FAIL_IGNORE REFINES FAIL_IGNORE $\widehat{=}$
ANY
$\ldots$
$\oplus prop$
WHERE
$\ldots$
$\ominus grd2 \ : \ i\_state(token\_owner(s) \mapsto s) = FL\_ACT\_4$
$\oplus grd2 \ : \ inst\_state(token\_owner(s) \mapsto s) = FL\_ACT\_4$
$\ldots$
$\oplus grd4 \ : \ prop \subseteq PEERS$
$\oplus grd5 \ : \ prop = run\_peers(s) \setminus unav\_peers$
THEN
$\ominus act1 \ : \ i\_state(token\_owner(s) \mapsto s) := FL\_IGN\_4$
$\oplus act1 \ : \ inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FL\_IGN\_4\})$
END

— Evénement IGNORE : le service s se trouve dans l'état FL_IGN_4 ($grd2$). L'instance « token owner », ayant ainsi décidé que l'état illégal du service s pouvait être ignoré, fait passer le

service dans un état légal RUN_4 et propage cette information à un ensemble $prop$ ($act1$) constitué des instances de s dans un état légal, sans les instances indisponibles du service s ($grd3$, $grd4$).

EVENT IGNORE REFINES IGNORE $\widehat{=}$
ANY
...
$\oplus prop$
WHERE
...
$\ominus grd2$: $i\_state(token\_owner(s) \mapsto s) = FL\_IGN\_4$
$\oplus grd2$: $inst\_state(token\_owner(s) \mapsto s) = FL\_IGN\_4$
$\oplus grd3$: $prop \subseteq PEERS$
$\oplus grd4$: $prop = run\_peers(s) \setminus unav\_peers$
THEN
$\ominus act1$: $i\_state(token\_owner(s) \mapsto s) := RUN\_4$
$\oplus act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{RUN\_4\})$
END

— Evénement FAIL_CONFIGURE : le service s se trouve dans la phase de **self-activation** (FL_ACT_4) ($grd2$). L'instance « token owner », ayant évalué que l'état illégal du service s était critique, fait passer le service dans un état $FAIL\_CONF\_4$ (phase de **self-configuration**) et propage cette information à un ensemble $prop$ ($act1$) constitué des instances de s dans un état légal, sans les instances indisponibles du service s ($grd4$, $grd5$).

EVENT FAIL_CONFIGURE REFINES FAIL_CONFIGURE $\widehat{=}$
ANY
...
$\oplus prop$
WHERE
...
$\ominus grd2$: $i\_state(token\_owner(s) \mapsto s) = FL\_ACT\_4$
$\oplus grd2$: $inst\_state(token\_owner(s) \mapsto s) = FL\_ACT\_4$
...
$\oplus grd4$: $prop \subseteq PEERS$
$\oplus grd5$: $prop = run\_peers(s) \setminus unav\_peers$
THEN
$\ominus act1$: $i\_state(token\_owner(s) \mapsto s) := FL\_CONF\_4$
$\oplus act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FL\_CONF\_4\})$
END

— Evénement REDEPLOY : le service s se trouve dans la phase de **self-configuration** (FL_CONF_4) ($grd2$). L'instance « token owner », ayant terminé le redéploiement du service s, fait passer le service dans un état DPL_4 (fin du redéploiement) et propage cette information à un ensemble $prop$ ($act1$) constitué des instances de s dans un état légal, sans les instances indisponibles du service s ($grd6$, $grd7$).

EVENT REDEPLOY REFINES REDEPLOY $\widehat{=}$
ANY
...
$\oplus prop$
WHERE
...
$\ominus grd2$: $i\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
$\oplus grd2$: $inst\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
...
$\oplus grd6$: $prop \subseteq PEERS$
$\oplus grd7$: $prop = run\_peers(s) \setminus unav\_peers$
THEN
$\ominus act1$: $i\_state(token\_owner(s) \mapsto s) := DPL\_4$
$\oplus act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{DPL\_4\})$
...
END

— Evénement HEAL_4 : le service s se trouve dans l'état DPL_4 ($grd2$). L'instance « token owner », ayant ainsi terminé la phase de redéploiement du service s, fait passer le service dans un état légal RUN_4 et propage cette information à un ensemble $prop$ ($act1$) constitué des instances de s dans un état légal, sans les instances indisponibles du service s ($grd3$, $grd4$).

EVENT HEAL_4 REFINES HEAL_4 $\widehat{=}$
ANY
$\ldots$
$\oplus prop$
WHERE
$\ldots$
$\ominus grd2 \ : \ i\_state(token\_owner(s) \mapsto s) = DPL\_4$
$\oplus grd2 \ : \ inst\_state(token\_owner(s) \mapsto s) = DPL\_4$
$\oplus grd3 \ : \ prop \subseteq PEERS$
$\oplus grd4 \ : \ prop = run\_peers(s) \setminus unav\_peers$
THEN
$\ominus act1 \ : \ i\_state(token\_owner(s) \mapsto s) := RUN\_4$
$\oplus act1 \ : \ inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{RUN\_4\})$
END

Ce seizième raffinement peut se résumer par le diagramme suivant :

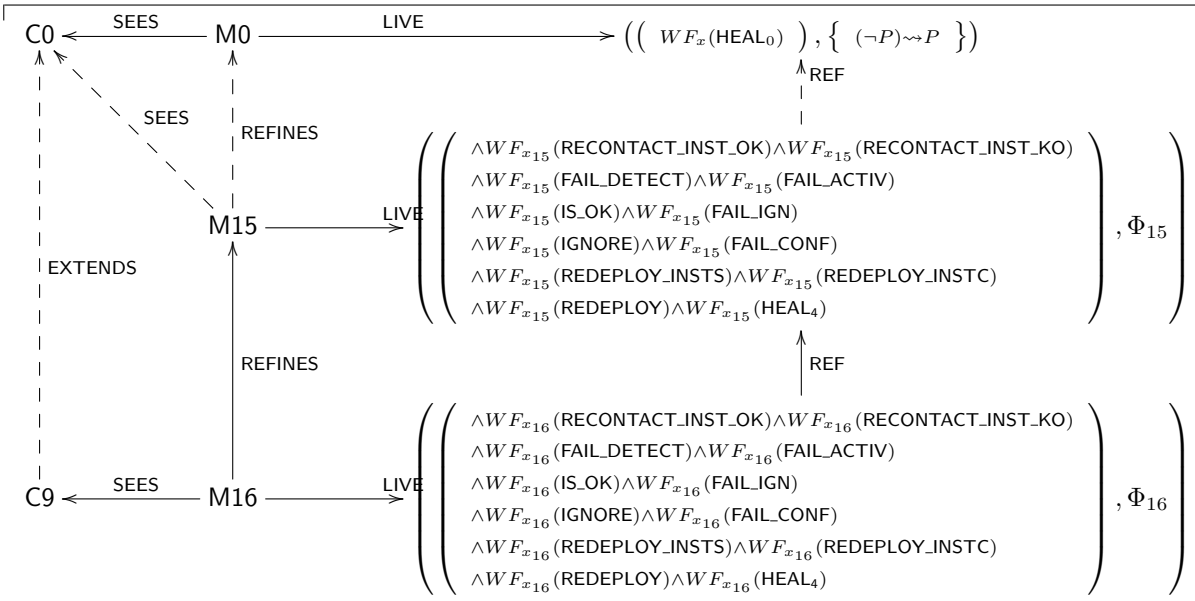


FIGURE 8.21 – Seizième raffinement du self-healing

Ce raffinement a introduit la notion de propagation : l'instance « token owner » d'un service fait connaître à toutes les instances fonctionnelles (dans un état légal) du service les entrées et sorties du service de chaque phase de la procédure de *self-healing*. Dans les raffinements suivants, nous généralisons cette notion de propagation, l'instance « token owner » d'un service service ne se contentant pas seulement de diriger les autres instances durant la procédure de *self-healing*, mais partageant aussi des informations importantes comme les identités des instances suspectes, défaillantes (dans un état illégal), etc.

8.3.5.2 Propagation d'informations par le « token owner » : suspicions

Ce raffinement étend la notion de propagation vue dans le raffinement précédent. Nous détaillons ici plus précisément la phase de **self-detection** : durant cette phase, l'instance « token owner » d'un service ne fait pas seulement passer ce dernier dans l'état FL_DT_4, signifiant qu'un état illégal a été détecté, mais il propage aussi les identités des pairs (instances) suspects qui l'ont poussé à la détection d'un état illégal. Nous utilisons pour modéliser ces aspects une constante $InitSuspPrs$ définie dans le contexte C9 vu précédemment : cette constante associe les pairs instanciant un service à un ensemble de pairs vide ($\varnothing$).

Une machine M17, raffinant la machine M16 vue dans la sous-section précédente, introduit une nouvelle variable *suspct_peers* :

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus act2$: $suspc\_peers := InitSuspPeers$
$\oplus act2$: $suspct\_peers := InitSuspPrs$
...
END

— La variable *suspct_peers* associe à chaque pair qui instancie un service, un sous-ensemble de pairs instanciant ce service et suspectées d'être dans des états illégaux. Cette variable est initialisée à l'aide de la constante $InitSuspPrs$, car à l'état initial, aucun pair instanciant un service n'est suspecté d'être dans un état illégal.

Nous notons x_{17} l'ensemble des variables de cette machine M17.

Un invariant que nous notons $I_{17}(x_{17})$ caractérise ces variables et est défini comme suit :

— Nous avons une propriété de typage :

$$inv1 \ : \ suspct\_peers \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS)$$

— Nous remplaçons dans ce raffinement la variable abstraite *suspc_peers* par la variable *suspct_peers*. Les deux propriétés suivants établissent les relations entre ces deux variables :

$$inv2 \ : \ \forall s \cdot s \in SERVICES \Rightarrow token\_owner(s) \mapsto s \in dom(suspct\_peers)$$
$$inv3 \ : \ \forall s \cdot s \in SERVICES \Rightarrow suspc\_peers(token\_owner(s) \mapsto s) = suspct\_peers(token\_owner(s) \mapsto s)$$

— $inv2$ exprime que l'instance « token owner » d'un service s fait partie des instances qui maintiennent une liste d'instances suspectes pour le service s.

— $inv3$ exprime que les instances d'un service s suspectées d'être dans des états illégaux, sont les mêmes, aussi bien au niveau abstrait qu'au niveau concret, permettant ainsi le remplacement de *suspc_peers* par la variable *suspct_peers*.

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
suspc_peers	*suspct_peers*

Les axiomes du contexte C9, ainsi que l'invariant de collage I_{17} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M16 en M17, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M17, ainsi que pour justifier la préservation des propriétés Φ_{16} lors du raffinement.

Une liste Φ_{17} de propriétés de vivacité, détaillée en annexe, guide le raffinement de M16 en M17, en présentant les nouveaux événements ou ceux ayant changé par raffinement :

— L'événement MAKE_PEER_UNAVAIL est affecté par l'introduction de la nouvelle variable *suspct_peers*. Nous modifions donc cet événement par raffinement. Cet événement modélise le fait de rendre un ensemble *prs* d'instances de services indisponibles ($act1$). Pour que cet événement puisse être observé, nous requérons que chaque service *srv* dispose au moins d'une instance monitorant l'état du service, non incluse dans *prs* ($grd3$). Nous distinguons deux cas : **(1)** pour un service *srv*, l'instance « token owner » ne fait pas partie de *prs*, auquel cas, cette instance ne change pas ($grd5$), sinon **(2)** si l'instance « token owner » fait partie de l'ensemble *prs*, un remplaçant $E(srv)$ pour l'instance « token owner » est choisi, avec la condition que celui-ci ait les mêmes caractéristiques que l'instance « token owner » à remplacer (même état de *srv* constaté, même liste d'instances suspecte) ($grd6$). Nous mettons alors à jour les instances « token owner » pour chaque service ($act2$) et interdisons ensuite aux membres de l'ensemble *prs* de suspecter des instances ($act6$, $act7$), de recontacter des instances suspectes ($act3$, $act4$), de monitorer des états ($act8$), de redéployer des instances de services ($act5$).

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL $\widehat{=}$
ANY
prs, E
WHERE
$grd1 : prs \subseteq PEERS$
$grd2 : prs \not\subseteq unav\_peers$
$grd3 : \forall srv \cdot srv \in SERVICES \Rightarrow dom(dom(inst\_state) \rhd \{srv\}) \setminus prs \neq \varnothing$
$grd4 : E \in SERVICES \rightarrow PEERS$
$grd5 : \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge \left(\begin{array}{c} token\_owner(srv) \\ \notin \\ prs \end{array} \right) \end{array} \right) \Rightarrow E(srv) = token\_owner(srv)$
$grd6 : \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge \left(\begin{array}{c} token\_owner(srv) \\ \in \\ prs \end{array} \right) \end{array} \right) \Rightarrow \left(\begin{array}{l} \wedge E(srv) \in \left(\begin{array}{c} run\_peers(srv) \\ \setminus \\ \left(\begin{array}{l} \cup unav\_peers \\ \cup prs \\ \cup failr\_peers(srv) \\ \cup suspct\_peers \left(\begin{array}{c} token\_owner(srv) \\ \mapsto \\ srv \end{array} \right) \end{array} \right) \end{array} \right) \\ \wedge E(srv) \mapsto srv \in dom(inst\_state) \\ \wedge E(srv) \mapsto srv \in dom(suspct\_peers) \\ \wedge \left(\begin{array}{c} inst\_state(E(srv) \mapsto srv) \\ = \\ inst\_state(token\_owner(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{c} suspct\_peers(E(srv) \mapsto srv) \\ = \\ suspct\_peers(token\_owner(srv) \mapsto srv) \end{array} \right) \end{array} \right)$
WITH
$p\_s : p\_s = E^{-1} \lhd suspct\_peers$
$rc\_s : rc\_s = ((prs \times SERVICES) \lhd rect\_inst) \lhd\!\!+ (E \setminus token\_owner)^{-1}) \times \{\varnothing\})$
$s\_i : s\_i = ((prs \times SERVICES) \lhd suspc\_inst) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$
$rt\_s : rt\_s = ((prs \times SERVICES) \lhd rctt\_inst) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$
$ac\_i : ac\_i = ((prs \times SERVICES) \lhd actv\_instc) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$
THEN
$act1 : unav\_peers := unav\_peers \cup prs$
$act2 : token\_owner := token\_owner \lhd\!\!+ E$
$act3 : rect\_inst := ((prs \times SERVICES) \lhd rect\_inst) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$
$act4 : rctt\_inst := ((prs \times SERVICES) \lhd rctt\_inst) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$
$act5 : actv\_instc := ((prs \times SERVICES) \lhd actv\_instc) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$
$act6 : suspct\_peers := (prs \times SERVICES) \lhd suspct\_peers$
$act7 : suspc\_inst := ((prs \times SERVICES) \lhd suspc\_inst) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$
$act8 : inst\_state := (prs \times SERVICES) \lhd inst\_state$
END

— L'événement FAILURE est aussi modifié par raffinement pour prendre en compte la nouvelle variable *suspct_peers* : l'instance « token owner » d'un service s suspecte des instances de s d'être dans un état illégal ($grd4$). Il propage alors cette information à un ensemble *prop* d'instances de s ($act2$) contenant les instances fonctionnelles (dans un état légal) du service s, sans celles suspectes et celles indisponibles ($grd5$). Les instances de s considérées comme fonctionnelles (dans un état légal) ont maintenant connaissance des identités des pairs suspectés d'être défaillants.

EVENT FAILURE REFINES FAILURE $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1 : s \in SERVICES$
$grd2 : prop \subseteq PEERS$
$grd3 : inst\_state(token\_owner(s) \mapsto s) = RUN\_4$
$grd4 : suspc\_inst(token\_owner(s) \mapsto s) \neq \varnothing$
$grd5 : prop = run\_peers(s) \setminus (suspc\_inst(token\_owner(s) \mapsto s) \cup unav\_peers)$
THEN
$act1 : inst\_state := inst\_state \lhd\!\!+ ((prop \times \{s\}) \times \{FAIL\_4\})$
$act2 : suspct\_peers := suspct\_peers \lhd\!\!+ ((prop \times \{s\}) \times \{suspc\_inst(token\_owner(s) \mapsto s)\})$
$act3 : suspc\_inst(token\_owner(s) \mapsto s) := \varnothing$
END

— Les événements modélisant les étapes de la phase de **self-detection** d'un service sont aussi modifiés par raffinement :
— Les événements RECONTACT_INST_OK et RECONTACT_INST_KO sont modifiés ils modélisent le fait que l'instance « token owner » d'un service s recontacte les instances suspectes du service

s ($grd4$, $grd5$), pour constater si l'état illégal du service est avéré ou non.

EVENT RECONTACT_INST_OK REFINES RECONTACT_INST_OK $\widehat{=}$
ANY
s, i
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $i \in PEERS$
$grd3$: $inst\_state(token\_owner(s) \mapsto s) = FAIL\_4$
$grd4$: $suspct\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
$grd5$: $i \in suspct\_peers(token\_owner(s) \mapsto s) \setminus unav\_peers$
$grd6$: $i \notin rect\_inst(token\_owner(s) \mapsto s)$
$grd7$: $rect\_inst(token\_owner(s) \mapsto s) \subset suspct\_peers(token\_owner(s) \mapsto s)$
THEN
$act1$: $rect\_inst(token\_owner(s) \mapsto s) := rect\_inst(token\_owner(s) \mapsto s) \cup \{i\}$
$act2$: $rctt\_inst(token\_owner(s) \mapsto s) := rctt\_inst(token\_owner(s) \mapsto s) \cup \{i\}$
END

EVENT RECONTACT_INST_KO REFINES RECONTACT_INST_KO $\widehat{=}$
ANY
s, i
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $i \in PEERS$
$grd3$: $inst\_state(token\_owner(s) \mapsto s) = FAIL\_4$
$grd4$: $suspct\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
$grd5$: $i \in suspct\_peers(token\_owner(s) \mapsto s) \cap unav\_peers$
$grd6$: $i \notin rect\_inst(token\_owner(s) \mapsto s)$
$grd7$: $rect\_inst(token\_owner(s) \mapsto s) \subset suspct\_peers(token\_owner(s) \mapsto s)$
THEN
$act1$: $rect\_inst(token\_owner(s) \mapsto s) := rect\_inst(token\_owner(s) \mapsto s) \cup \{i\}$
END

— Evénement FAIL_DETECT : l'instance « token owner » d'un service s a essayé de recontacter toutes les instances de s suspectées d'être dans un état illégal ($grd5$, $grd6$). Les instances réellement suspectes sont celles que l'instance « token owner » n'a pas réussi à recontacter et contenues dans un ensemble $susp$ ($grd8$). L'instance « token owner » communique alors les identités de ces instances contenues dans $susp$ à un ensemble $prop$ ($act2$), contenant les instances fonctionnelles (dans un état légal) de s, sans celles suspectes, celles indisponibles, mais avec celles que l'instance « token owner » du service s a réussi à recontacter ($grd7$).

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
ANY
$s, prop, susp$
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $prop \subseteq PEERS$
$grd3$: $susp \subseteq PEERS$
$grd4$: $inst\_state(token\_owner(s) \mapsto s) = FAIL\_4$
$grd5$: $suspct\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
$grd6$: $rect\_inst(token\_owner(s) \mapsto s) = suspct\_peers(token\_owner(s) \mapsto s)$
$grd7$: $prop = \left(\begin{array}{l} \cup((run\_peers(s) \setminus suspct\_peers(token\_owner(s) \mapsto s)) \\ \cup rctt\_inst(token\_owner(s) \mapsto s)) \setminus unav\_peers \end{array} \right)$
$grd8$: $susp = suspct\_peers(token\_owner(s) \mapsto s) \setminus rctt\_inst(token\_owner(s) \mapsto s)$
THEN
$act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FL\_DT\_4\})$
$act2$: $suspct\_peers := suspct\_peers \Leftarrow ((prop \times \{s\}) \times \{susp\})$
$act3$: $rect\_inst(token\_owner(s) \mapsto s) := \varnothing$
$act4$: $rctt\_inst(token\_owner(s) \mapsto s) := \varnothing$
END

— Evénement IS_OK : l'instance « token owner » d'un service s a réussi à recontacter toutes les instances de s suspectées d'être dans un état illégal, et par conséquent la liste des instances suspectes maintenue par l'instance « token owner » est vide ($grd4$). Il s'agit donc juste d'une fausse alerte et le service s peut revenir à un état légal RUN_4.

EVENT IS_OK REFINES IS_OK $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $prop \subseteq PEERS$
$grd3$: $inst\_state(token\_owner(s) \mapsto s) = FL\_DT\_4$
$grd4$: $suspct\_peers(token\_owner(s) \mapsto s) = \varnothing$
$grd5$: $prop = run\_peers(s) \setminus unav\_peers$
THEN
$act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{RUN\_4\})$
END

— Evénement FAIL_ACTIV : l'instance « token owner » d'un service s n'a pas réussi à recontacter toutes les instances de s suspectées d'être dans un état illégal, et par conséquent la liste des instances suspectes maintenue par l'instance « token owner » n'est pas vide ($grd4$). Les instances suspectes sont alors considérées comme étant défaillantes, dans un état illégal ($act3$), mais ne sont plus considérées comme suspectes ($act4$). Cette information est propagée à un ensemble $prop$ composé des instances fonctionnelles d'un service s, sans celle indisponibles et celles anciennement suspectées ($grd5$, $act4$). Cet événement est à cheval sur deux phases : il marque la fin de la phase de **self-detection** tout en introduisant un appel à la phase de **self-activation** ($act1$).

EVENT FAIL_ACTIV REFINES FAIL_ACTIV $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $prop \subseteq PEERS$
$grd3$: $inst\_state(token\_owner(s) \mapsto s) = FL\_DT\_4$
$grd4$: $suspct\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
$grd5$: $prop = run\_peers(s) \setminus (unav\_peers \cup suspct\_peers(token\_owner(s) \mapsto s))$
THEN
$act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FL\_ACT\_4\})$
$act2$: $run\_peers(s) := run\_peers(s) \setminus suspct\_peers(token\_owner(s) \mapsto s)$
$act3$: $failr\_peers(s) := failr\_peers(s) \cup suspct\_peers(token\_owner(s) \mapsto s)$
$act4$: $suspct\_peers := suspct\_peers \Leftarrow ((prop \times \{s\}) \times \{\varnothing\})$
END

Ce dix-septième raffinement est résumé par le diagramme suivant :

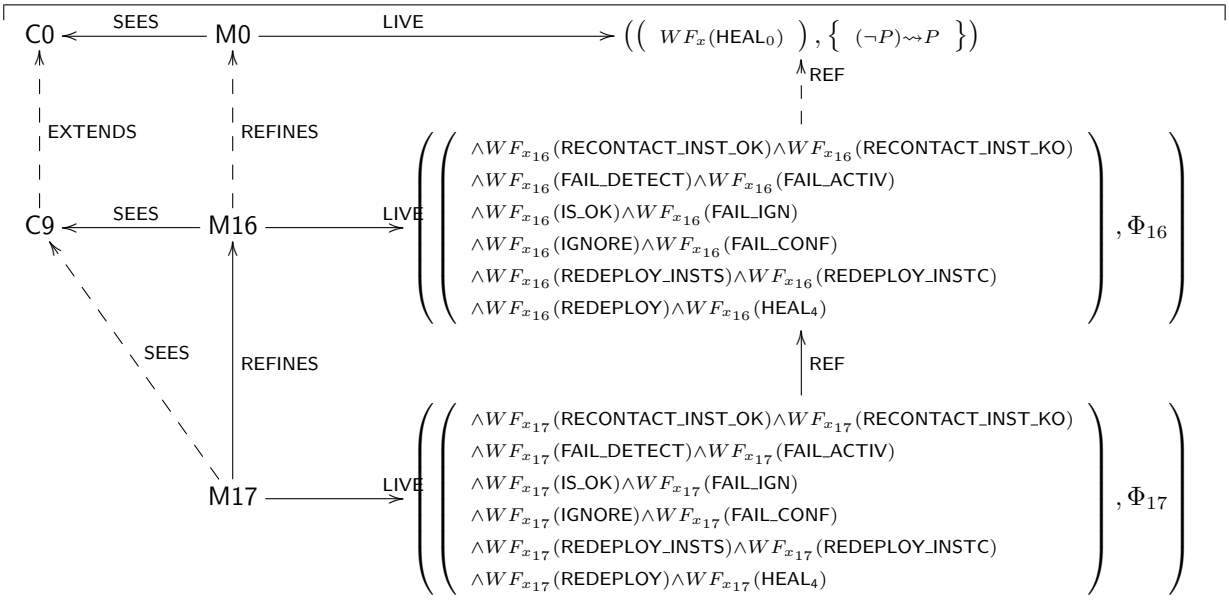


FIGURE 8.22 – Dix-septième raffinement du self-healing

Les prochains raffinements mettent encore plus l'accent sur les instances « token owner » des services coordonnant et guidant la procédure de *self-healing* et propageant les états, informations sur les instances, etc à toutes les autres instances fournissant les mêmes services que les instances « token owner ».

8.3.5.3 Propagation d'informations par le « token owner » : instances actives

Ce raffinement introduit le fait que, similairement à ce qui a été fait lors du modèle précédent, par rapport aux instances suspectes, les instances « token owner » des services maintiennent aussi des listes des instances fonctionnelles (dans un état légal) des services qu'elles fournissent, et en cas de mises à jour, de modifications de ces listes, les instances « token owner » les propagent aux autres instances des services. Pour mettre en place ce mécanisme de monitorat d'instances et de propagation d'informations sur ces dernières dans le système P2P, nous utilisons, dans ce raffinement, une constante définie dans le contexte C8 précédent, à savoir, $InitRunPeers$: cette constante associe à chaque pair instanciant un service, la liste de tous les pairs fournissant initialement le service fourni par le pair.

Une machine M18, raffinant la machine M17 précédente, introduit une variable *run_inst* :

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus act2$: $run\_peers := InitSrvcPeers$
$\oplus act2$: $run\_inst := InitRunPeers$
...
END

— La variable *run_inst* associe chaque pair instanciant un service à un ensemble contenant tous les pairs instanciant le même service ($inv1$).

Nous notons x_{18} l'ensemble des variables de la machine M18.

Un invariant que nous notons $I_{18}(x_{18})$ caractérise ces variables et est défini comme suit :

— Nous avons du typage :

$$inv1 \ : \ run\_inst \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS)$$

— La variable *run_inst* remplace la variable *run_peers*. Les deux propriétés suivantes nous permettent ce remplacement :

$$inv2 \ : \ \forall s \cdot s \in SERVICES \Rightarrow token\_owner(s) \mapsto s \in dom(run\_inst)$$
$$inv3 \ : \ \forall s \cdot s \in SERVICES \Rightarrow run\_inst(token\_owner(s) \mapsto s) = run\_peers(s)$$

— $inv2$ exprime que l'instance « token owner » d'un service s maintient une liste de tous les pairs instanciant le même service s.
— $inv3$ exprime que l'instance « token owner » d'un service s, au niveau concret, maintient la même liste d'instances que le service s, au service s, au niveau abstrait. C'est cet invariant en particulier qui nous permet d'utiliser *run_inst* à la place de *run_peers*.

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	Variable concrète
run_peers	*run_inst*

Les axiomes du contexte C9, ainsi que l'invariant de collage I_{18} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M17 en M18, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M18, ainsi que pour justifier la préservation des propriétés Φ_{17} lors du raffinement.

Une liste Φ_{18} de propriétés de vivacité, détaillée en annexe, permet de guider le raffinement de M17 en M18, en présentant les nouveaux événements ou ceux ayant changé par raffinement :

— Evénement MAKE_PEER_UNAVAIL : Cet événement modélise le fait que des pairs contenus dans un ensemble prs deviennent indisponibles (ne fournissent plus les services demandés). Nous requérons ici, pour que cet événement soit observé, que si l'instance « token owner » d'un service srv fait partie de prs, alors un remplaçant $E(srv)$ avec les même caractéristiques, principalement que le replaçant maintienne pour le service srv la même liste d'instances fonctionnelles (dans un état légal) que l'instance « token owner » ($grd6$). Nous interdisons ensuite aux instances membres de prs de maintenir des listes des instances fonctionnelles des services qu'ils fournissent ($act9$).

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd6$: ...
$\oplus grd6$: $\forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge \left(\begin{array}{c} token\_owner(srv) \\ \in \\ prs \end{array} \right) \end{array} \right)$
$\Rightarrow \left(\begin{array}{l} \wedge E(srv) \in \left(\begin{array}{c} run\_inst(token\_owner(srv) \mapsto srv) \\ \setminus \\ \left(\begin{array}{l} \cup unav\_peers \\ \cup prs \\ \cup failr\_peers(srv) \\ \cup suspct\_peers \left(\begin{array}{c} token\_owner(srv) \\ \mapsto \\ srv \end{array} \right) \end{array} \right) \end{array} \right) \\ \wedge E(srv) \mapsto srv \in \left(\begin{array}{l} \cap dom(suspct\_peers) \\ \cap dom(inst\_state) \\ \cap dom(run\_inst) \end{array} \right) \\ \wedge \left(\begin{array}{c} inst\_state(E(srv) \mapsto srv) \\ = \\ inst\_state(token\_owner(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{c} suspct\_peers(E(srv) \mapsto srv) \\ = \\ suspct\_peers(token\_owner(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{c} run\_inst(E(srv) \mapsto srv) \\ = \\ run\_inst(token\_owner(srv) \mapsto srv) \end{array} \right) \end{array} \right)$
THEN
...
$\oplus act9$: $run\_inst := (prs \times SERVICES) \lhd run\_inst$
END

— Evénement SUSPECT_INST : les instances d'un service s, membres d'un ensemble $prop$, auxquelles l'instance « token owner » du service s propage la liste les instances qu'elle a suspecté d'être dans un état illégal sont les instances fonctionnelles de s, sans les instances indisponibles ($grd3$).

EVENT SUSPECT_INST REFINES SUSPECT_INST $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd3$: $prop = run\_peers(s) \setminus unav\_peers$
$\oplus grd3$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
...
END

— Evénement FAILURE : les instances d'un service s, membres d'un ensemble $prop$, auxquelles l'instance « token owner » du service s propage le fait que le service est entré dans un état illégal, sont les instances fonctionnelles de s, sans les instances suspectes et indisponibles ($grd5$).

EVENT FAILURE REFINES FAILURE $\widehat{=}$
ANY
...
WHERE
...
$\ominus grd5$: $prop = run\_peers(s) \setminus (suspc\_inst(token\_owner(s) \mapsto s) \cup unav\_peers)$
$\oplus grd5$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus (suspc\_inst(token\_owner(s) \mapsto s) \cup unav\_peers)$
THEN
...
END

— Evénement FAIL_DETECT : les instances d'un service s, membres d'un ensemble *prop*, auxquelles l'instance « token owner » du service s propage le fait que l'état illégal du service est détecté, sont les instances fonctionnelles de s, avec les instances suspectes qu'elle a pu recontacter, mais sans les instances suspectes qu'elle n'a pas réussi à joindre et celles indisponibles ($grd5$).

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus grd6$: $prop = \left(\begin{array}{l} \cup((run\_peers(s) \setminus suspct\_peers(token\_owner(s) \mapsto s)) \\ \cup rctt\_inst(token\_owner(s) \mapsto s)) \setminus unav\_peers \end{array} \right)$
    $\oplus grd6$: $prop = \left(\begin{array}{l} \cup((run\_inst(token\_owner(s) \mapsto s) \setminus suspct\_peers(token\_owner(s) \mapsto s)) \\ \cup rctt\_inst(token\_owner(s) \mapsto s)) \setminus unav\_peers \end{array} \right)$
    ...
  THEN
    ...
  END

— Evénement IS_OK : les instances d'un service s, membres d'un ensemble *prop*, auxquelles l'instance « token owner » du service s propage le fait que l'état illégal détecté du service est une fausse alerte, sont les instances fonctionnelles de s, sans celles indisponibles ($grd5$).

EVENT IS_OK REFINES IS_OK $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus grd5$: $prop = run\_peers(s) \setminus unav\_peers$
    $\oplus grd5$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
  THEN
    ...
  END

— Evénement FAIL_ACTIV : les instances d'un service s, membres d'un ensemble *prop*, auxquelles l'instance « token owner » du service s propage le fait que le service entre dans la phase de **self-activation**, sont les instances fonctionnelles de s, sans celles indisponibles et celles suspectes ($grd5$). L'instance « token owner » procède ensuite à une mise à jour des instances fonctionnelles du service s : elle y retire les instances suspectes et propage cette modification aux instances membres de l'ensemble *prop* ($act2$).

EVENT FAIL_ACTIV REFINES FAIL_ACTIV $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus grd5$: $prop = run\_peers(s) \setminus (unav\_peers \cup suspct\_peers(token\_owner(s) \mapsto s))$
    $\oplus grd5$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus (unav\_peers \cup suspct\_peers(token\_owner(s) \mapsto s))$
  THEN
    ...
    $\ominus act2$: $run\_peers(s) := run\_peers(s) \setminus suspct\_peers(token\_owner(s) \mapsto s)$
    $\oplus act2$: $run\_inst := run\_inst \Leftarrow ((prop \times \{s\}) \times \left\{ \begin{array}{c} run\_inst(token\_owner(s) \mapsto s) \\ \setminus \\ suspct\_peers(token\_owner(s) \mapsto s) \end{array} \right\})$
    ...
  END

— Evénement FAIL_IGNORE : Cet événement est observé lorsque l'instance « token owner » d'un service s constate que l'état illégal du service n'est pas critique et peut être ignoré car le nombre d'instances fonctionnelles du service s est supérieur ou égal au minimum requis pour fournir s ($grd3$). Les instances d'un service s, membres d'un ensemble *prop*, auxquelles l'instance « token owner » du service s propage ces informations, sont les instances fonctionnelles de s, sans celles indisponibles ($grd5$).

EVENT FAIL_IGNORE REFINES FAIL_IGNORE $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\oplus grd3$: $card(run\_peers(s)) \geq min\_inst(s)$
    $\oplus grd3$: $card(run\_inst(token\_owner(s) \mapsto s)) \geq min\_inst(s)$
    ...
    $\ominus grd5$: $prop = run\_peers(s) \setminus unav\_peers$
    $\oplus grd5$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
  THEN
    ...
  END

— Evénement IGNORE : les instances d'un service s, membres d'un ensemble *prop*, auxquelles l'instance « token owner » du service s propage le fait que l'état illégal du service est ignoré et que le service revient dans un état légal, sont les instances fonctionnelles de s, sans celles indisponibles ($grd4$).

EVENT IGNORE REFINES IGNORE $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus grd4$: $prop = run\_peers(s) \setminus unav\_peers$
    $\oplus grd4$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
  THEN
    ...
  END

— Evénement FAIL_CONFIGURE : Cet événement est observé lorsque l'instance « token owner » d'un service s constate que le nombre d'instances fonctionnelles du service s est inférieur au minimum requis pour fournir s ($grd3$). Le service entre alors dans la phase de **self-configuration**. Les instances d'un service s, membres d'un ensemble *prop*, auxquelles l'instance « token owner » du service s propage ces informations, sont les instances fonctionnelles de s, sans celles indisponibles ($grd5$).

EVENT FAIL_CONFIGURE REFINES FAIL_CONFIGURE $\widehat{=}$
  ANY
    ...
  WHERE
    ...
    $\ominus grd3$: $card(run\_peers(s)) < min\_inst(s)$
    $\oplus grd3$: $card(run\_inst(token\_owner(s) \mapsto s)) < min\_inst(s)$
    ...
    $\ominus grd5$: $prop = run\_peers(s) \setminus unav\_peers$
    $\oplus grd5$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
  THEN
    ...
  END

— Evénement REDEPLOY_INSTC : Cet événement est observé lorsque l'instance « token owner » d'un service s doit activer une nouvelle instance i du service s ; la condition pour que cette instance i soit activée est la suivante : elle n'a pas encore fourni le service s précédemment, ne fait pas partie des pairs défaillants et/ou indisponibles et n'a pas encore été déployée pour s ($grd3$). Une autre condition nécessaire à l'observation de cet événement est que le nombre d'instances du service s nouvellement déployées, ajouté à celui du nombre d'instances fonctionnelles du service s est inférieur au minimum requis pour fournir s ($grd7$).

EVENT REDEPLOY_INSTC REFINES REDEPLOY_INSTC ≙
ANY
...
WHERE
...
$\ominus grd3$: $i \notin run\_peers(s) \cup failr\_peers(s) \cup unav\_peers \cup dep\_instc(s)$
$\oplus grd3$: $i \notin run\_inst(token\_owner(s) \mapsto s) \cup failr\_peers(s) \cup unav\_peers \cup dep\_instc(s)$
...
$\ominus grd7$: $card(dep\_instc(s)) + card(run\_peers(s)) < min\_inst(s)$
$\oplus grd7$: $card(dep\_instc(s)) + card(run\_inst(token\_owner(s) \mapsto s)) < min\_inst(s)$
THEN
...
END

— Evénement REDEPLOY_INSTS : les instances activées par l'instance « token owner » d'un service s sont déployées. Une condition nécessaire à l'observation de cet événement est que le nombre d'instances du service s nouvellement déployées, ajouté à celui du nombre d'instances fonctionnelles du service s est inférieur au minimum requis pour fournir s ($grd3$).

EVENT REDEPLOY_INSTS REFINES REDEPLOY_INSTS ≙
ANY
...
WHERE
...
$\ominus grd3$: $card(dep\_instc(s)) + card(run\_peers(s)) < min\_inst(s)$
$\oplus grd3$: $card(dep\_instc(s)) + card(run\_inst(token\_owner(s) \mapsto s)) < min\_inst(s)$
...
THEN
...
END

— Evénement REDEPLOY : Cet événement marque la fin de la phase de redéploiement. Il est observé lorsque le nombre d'instances du service s nouvellement déployées, ajouté à celui du nombre d'instances fonctionnelles du service s atteint ou dépasse le minimum requis pour fournir s ($grd5$). Ces nouvelles instances sont alors considérées par l'instance « token owner » du service s comme de nouvelles instances fonctionnelles de s ($act2$). Les instances d'un service s, membres d'un ensemble $prop$, auxquelles l'instance « token owner » du service s propage ces informations, sont les instances fonctionnelles de s, sans celles indisponibles ($grd7$, $act2$).

EVENT REDEPLOY REFINES REDEPLOY ≙
ANY
...
WHERE
...
$\ominus grd5$: $card(run\_peers(s)) + card(dep\_instc(s)) \geq min\_inst(s)$
$\oplus grd5$: $card(run\_inst(token\_owner(s) \mapsto s)) + card(dep\_instc(s)) \geq min\_inst(s)$
...
$\ominus grd7$: $prop = run\_peers(s) \setminus unav\_peers$
$\oplus grd7$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
...
$\ominus act2$: $run\_peers(s) := run\_peers(s) \cup dep\_instc(s)$
$\oplus act2$: $run\_inst := run\_inst \Leftarrow ((prop \times \{s\}) \times \{run\_inst(token\_owner(s) \mapsto s) \cup dep\_instc(s)\})$
...
END

— Evénement HEAL_4 : les instances d'un service s, membres d'un ensemble $prop$, auxquelles l'instance « token owner » du service s propage le fait que le redéploiement du service est terminé et que ce dernier est revenu dans un état légal RUN_4, sont les instances fonctionnelles de s, sans celles indisponibles ($grd4$).

EVENT HEAL_4 REFINES HEAL_4 ≙
ANY
...
WHERE
...
$\ominus grd4$: $prop = run\_peers(s) \setminus unav\_peers$
$\oplus grd4$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
...
END

Ce dix-huitième raffinement est résumé par le diagramme suivant :

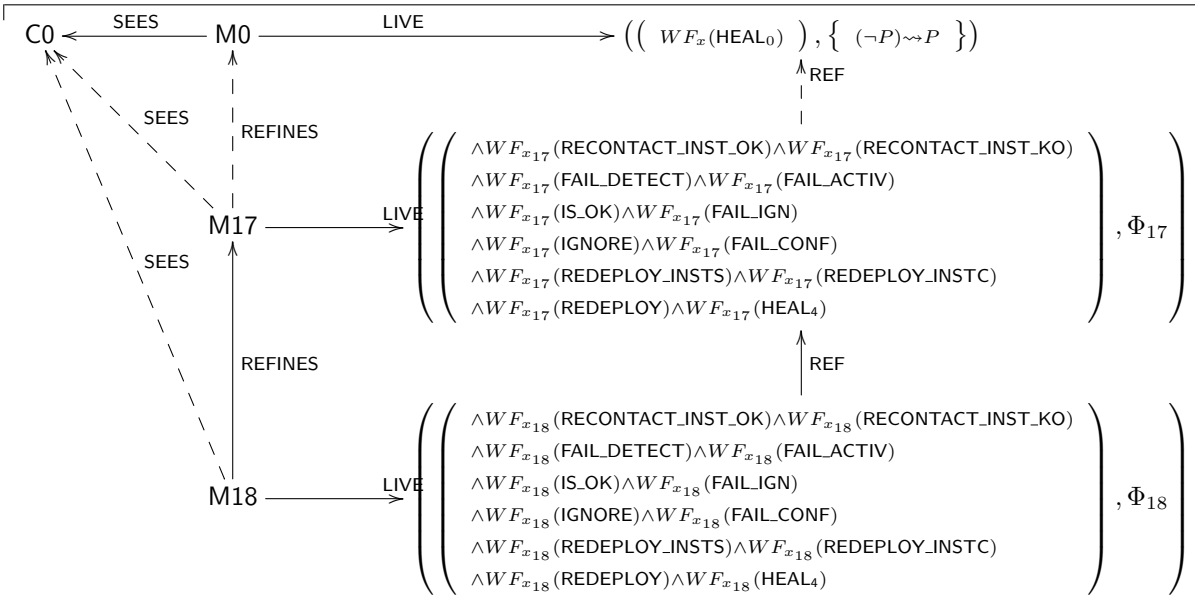


FIGURE 8.23 – Dix-huitième raffinement du self-healing

Ce raffinement nous montre encore plus l'importance des instances « token owner » des services fournis par le système P2P dans la procédure de *self-healing* de service : ces dernières ne monitorent pas seulement les états des services, elles gère aussi les instances fonctionnelles, les instances suspectes, à réactiver, etc. Nous montrons dans le raffinement qui suit la gestion des instances entrées dans un état illégal (défaillantes) par ces instances « token owner ».

8.3.5.4 Propagation d'informations par le « token owner » : défaillances

Ce raffinement détaille la gestion des instances entrées dans un état illégal par un service : nous avions considéré durant les raffinements précédents que ces instances défaillantes étaient gérées directement par le service lui-même ; maintenant, nous considérons que les instances défaillantes sont détectées par l'instance « token owner » du service et cette instance se charge ensuite de propager les informations à propos de ces instances défaillantes, aux instances demeurées fonctionnelles (dans un état légal) du service. Nous rappelons ici que nous utilisons dans ce raffinement, des élements du contexte C8 défini dans les sections précédentes, à savoir la constante $InitSuspPeers$, qui associe à chaque pair instanciant un service, un ensemble vide ($\varnothing$) d'instances.

Nous introduisons dans une machine M19, raffinement de la machine M18 précédente, une variable $failr\_inst$:

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus act3$: $failr\_peers := InitFail$
$\oplus act3$: $failr\_inst := InitSuspPeers$
...
END

— La variable $failr\_inst$ associe à chaque pair instanciant un service, un ensemble contenant les instances de ce service défaillantes (entrées dans un état illégal).

Nous notons x_{19} l'ensemble des variables de la machine M19.

Un invariant noté $I_{19}(x_{19})$ caractérisant ces variables est défini comme suit :

— Nous avons du typage :

$$inv1 \ : \ failr\_inst \in (PEERS \times SERVICES) \pfun \mathbb{P}(PEERS)$$

— La variable $failr\_inst$ est une version plus concrète de la variable $failr\_peers$. Deux propriétés établissent les relations entre ces deux variables :

$$\begin{array}{l} inv2 \ : \ \forall s \cdot s \in SERVICES \Rightarrow token\_owner(s) \mapsto s \in dom(failr\_inst) \\ inv3 \ : \ \forall s \cdot s \in SERVICES \Rightarrow failr\_inst(token\_owner(s) \mapsto s) = failr\_peers(s) \end{array}$$

— $inv2$ exprime que l'instance « token owner » d'un service s fait partie des instances de s maintenant une liste d'instances défaillantes.
— $inv3$ exprime que les instances défaillantes détectées par l'instance « token owner » d'un service s au niveau concret sont les mêmes que celle détectées par le service s au niveau abstrait. C'est cet invariant $inv3$ qui nous permet notamment le remplacement de la variable $failr\_peers$ par $failr\_inst$.

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
$failr\_peers$	$failr\_inst$

Les axiomes du contexte C9, ainsi que l'invariant de collage I_{19} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement de M18 en M19, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M19, ainsi que pour justifier la préservation des propriétés Φ_{18} lors du raffinement.

Une liste Φ_{19} de propriétés de vivacité, détaillée en annexe, guide le raffinement de M18 en M19, en présentant les nouveaux événements ou ceux ayant changé par raffinement :

— Evénement MAKE_PEER_UNAVAIL : si une instance « token owner » d'un service srv venait à devenir indisponible, un remplaçant $E(srv)$ est choisi par les instances fonctionnelles de srv, sans celles qui sont déjà ou vont devenir indisponibles, celles que l'instance « token owner » aura déclaré défaillantes et celles suspectées de l'être ($grd6$). Ce remplaçant doit aussi posséder les mêmes caractéristiques que l'ancienne instance « token owner », parmi lesquelles, celle qui nous intéresse ici : il doit gérer la même liste d'instances défaillantes que le « token owner ». Nous interdisons aussi aux instances devenues nouvellement indisponible de gérer des listes d'instances défaillantes ($act10$).
— Evénement FAIL_ACTIV : les instances d'un service s suspectes détectées durant la phase de **self-detection** ($grd4$), ne sont plus considérées comme suspectes, mais sont maintenant considérées comme étant défaillantes par l'instance « token owner » du service s ($act2$) et cette dernière propage cette information à un ensemble $prop$ d'instances ($act3$) contenant celles fonctionnelles du service s sans celles indisponibles et celles anciennement suspectes ($grd5$).

EVENT FAIL_ACTIV REFINES FAIL_ACTIV $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1 \ : \ s \in SERVICES$
$grd2 \ : \ prop \subseteq PEERS$
$grd3 \ : \ inst\_state(token\_owner(s) \mapsto s) = FL\_DT\_4$
$grd4 \ : \ suspct\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
$grd5 \ : \ prop = run\_inst(token\_owner(s) \mapsto s) \setminus (unav\_peers \cup suspct\_peers(token\_owner(s) \mapsto s))$
THEN
$act1 \ : \ inst\_state := inst\_state \ovr ((prop \times \{s\}) \times \{FL\_ACT\_4\})$
$act2 \ : \ run\_inst := run\_inst \ovr ((prop \times \{s\}) \times \left\{ \begin{array}{c} run\_inst(token\_owner(s) \mapsto s) \\ \setminus \\ suspct\_peers(token\_owner(s) \mapsto s) \end{array} \right\})$
$act3 \ : \ failr\_inst := failr\_inst \ovr ((prop \times \{s\}) \times \left\{ \begin{array}{l} \cup failr\_inst(token\_owner(s) \mapsto s) \\ \cup suspct\_peers(token\_owner(s) \mapsto s) \end{array} \right\})$
$act4 \ : \ suspct\_peers := suspct\_peers \ovr ((prop \times \{s\}) \times \{\varnothing\})$
END

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL $\widehat{=}$

ANY

prs, E

WHERE

$grd1 : prs \subseteq PEERS$

$grd2 : prs \not\subseteq unav\_peers$

$grd3 : \forall srv \cdot srv \in SERVICES \Rightarrow dom(dom(inst\_state) \rhd \{srv\}) \setminus prs \neq \varnothing$

$grd4 : E \in SERVICES \to PEERS$

$$grd5 : \forall srv \cdot \left(\begin{array}{c} \wedge srv \in SERVICES \\ \wedge (token\_owner(srv) \notin prs) \end{array} \right) \Rightarrow E(srv) = token\_owner(srv)$$

$$grd6 : \forall srv \cdot \left(\begin{array}{c} \wedge srv \in SERVICES \\ \wedge \left(\begin{array}{c} token\_owner(srv) \\ \in \\ prs \end{array} \right) \end{array} \right) \Rightarrow \left(\begin{array}{l} \wedge E(srv) \in \left(\begin{array}{c} run\_peers(srv) \\ \setminus \\ \left(\begin{array}{l} \cup unav\_peers \\ \cup prs \\ \cup failr\_inst(token\_owner(srv) \mapsto srv) \\ \cup suspct\_peers \left(\begin{array}{c} token\_owner(srv) \\ \mapsto \\ srv \end{array} \right) \end{array} \right) \end{array} \right) \\ \wedge E(srv) \mapsto srv \in \left(\begin{array}{l} \cap dom(run\_inst) \\ \cap dom(suspct\_peers) \\ \cap dom(failr\_inst) \\ \cap dom(inst\_state) \end{array} \right) \\ \wedge \left(\begin{array}{c} inst\_state(E(srv) \mapsto srv) \\ = \\ inst\_state(token\_owner(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{c} run\_inst(E(srv) \mapsto srv) \\ = \\ run\_inst(token\_owner(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{c} failr\_inst(E(srv) \mapsto srv) \\ = \\ failr\_inst(token\_owner(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{c} suspct\_peers(E(srv) \mapsto srv) \\ = \\ suspct\_peers(token\_owner(srv) \mapsto srv) \end{array} \right) \end{array} \right)$$

THEN

$act1 : unav\_peers := unav\_peers \cup prs$

$act2 : token\_owner := token\_owner \lhd\!\!+ E$

$act3 : rect\_inst := ((prs \times SERVICES) \lhd rect\_inst) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$

$act4 : rctt\_inst := ((prs \times SERVICES) \lhd rctt\_inst) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$

$act5 : actv\_instc := ((prs \times SERVICES) \lhd actv\_instc) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$

$act6 : suspct\_peers := (prs \times SERVICES) \lhd suspct\_peers$

$act7 : suspc\_inst := ((prs \times SERVICES) \lhd suspc\_inst) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$

$act8 : inst\_state := (prs \times SERVICES) \lhd inst\_state$

$act9 : run\_inst := (prs \times SERVICES) \lhd run\_inst$

$act10 : failr\_inst := (prs \times SERVICES) \lhd failr\_inst$

END

— La garde $grd3$ est modifiée au niveau de l'événement REDEPLOY_INSTC : la nouvelle instance i que l'instance « token owner » d'un service s va activer, ne fait pas partie des instances considérées comme défaillantes par l'instance « token owner » ($grd3$).

EVENT REDEPLOY_INSTC REFINES REDEPLOY_INSTC $\widehat{=}$
ANY
s, i
WHERE
$grd1 : s \in SERVICES$
$grd2 : i \in PEERS$
$grd3 : i \notin \left(\begin{array}{l} \cup run\_inst(token\_owner(s) \mapsto s) \\ \cup failr\_inst(token\_owner(s) \mapsto s) \\ \cup unav\_peers \\ \cup dep\_instc(s) \end{array} \right)$
$grd4 : i \notin actv\_instc(token\_owner(s) \mapsto s)$
$grd5 : inst\_state(token\_owner(s) \mapsto s) = FAIL\_CONFIG\_4$
$grd6 : card(actv\_instc(token\_owner(s) \mapsto s)) < deplo\_inst(s)$
$grd7 : card(dep\_instc(s)) + card(run\_inst(token\_owner(s) \mapsto s)) < min\_inst(s)$
THEN
$act1 : actv\_instc(token\_owner(s) \mapsto s) := actv\_instc(token\_owner(s) \mapsto s) \cup \{i\}$
END

— Evénement UNFAIL_PEER : un pair p ayant instancié un service s et étant défaillant est retiré par l'instance « token owner » du service s de sa liste de pairs défaillants (dans un état illégal). L'instance « token owner » du service s propage ensuite cette information à un ensemble $prop$ d'instances, composées de celles fonctionnelles du service s et de p, sans celles qui sont restées défaillantes.

EVENT UNFAIL_PEER REFINES UNFAIL_PEER $\widehat{=}$
ANY
$s, p, prop$
WHERE
$grd1 : s \in SERVICES$
$grd2 : prop \subseteq PEERS$
$grd3 : p \in PEERS$
$grd4 : p \in failr\_inst(token\_owner(s) \mapsto s)$
$grd5 : prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
$act1 : failr\_inst := failr\_inst \Leftarrow ((prop \times \{s\}) \times \{failr\_inst(token\_owner(s) \mapsto s) \setminus \{p\}\})$
END

Ce dix-neuvième raffinement est résumé par le diagramme suivant :

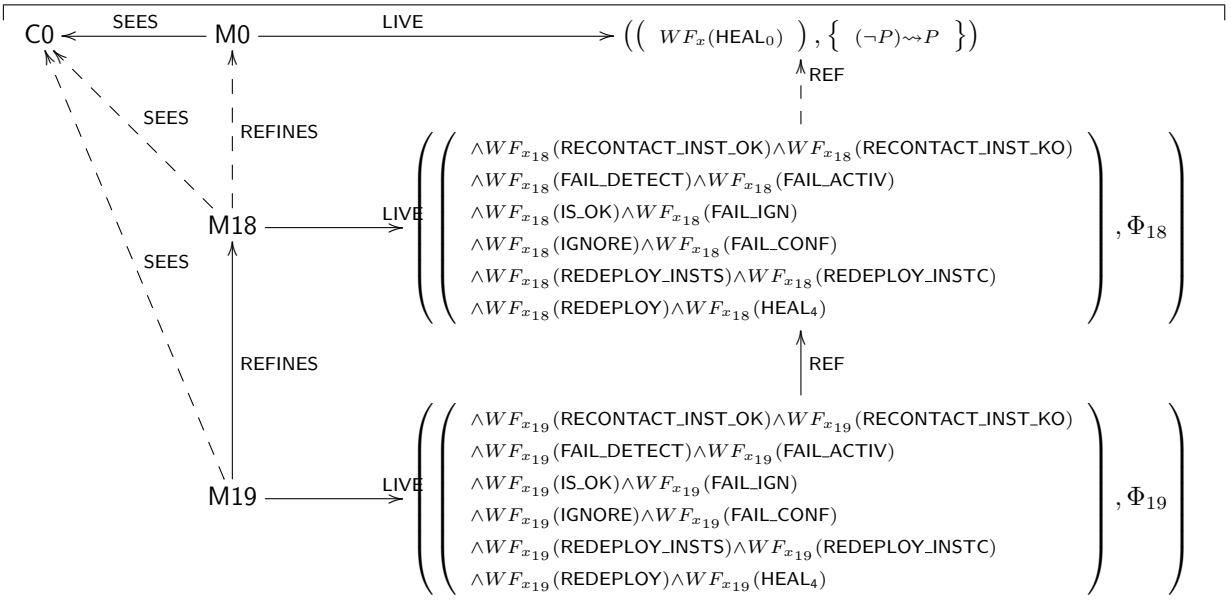


FIGURE 8.24 – Dix-neuvième raffinement du self-healing

Ce raffinement nous a présenté la gestion des instances défaillantes d'un services par son instance « token owner ». Le raffinement final suivant nous présentera la localisation de la phase de redéploiement, à l'aide de son instance « token owner », ainsi que les versions finales détaillées de tous les événements.

8.3.5.5 Modèle local

Ce raffinement localise le redéploiement d'un service, mais comme il s'agit du dernier raffinement, nous présentons ausso tous les événements de manière détaillées. Nous rappelons ici que ce raffinement utilise une partie du contexte C8, défini dans les sections précédentes : il s'agit de la constante $InitSuspPeers$, qui associe à chaque pair instanciant un service, un ensemble vide ($\varnothing$) d'instances.

La machine M20, raffinant la machine M19 vue dans la sous-section précédente, introduit une nouvelle variable *dep_instcs* :

INITIALISATION $\widehat{=}$
BEGIN
...
$\ominus act4 \ : \ dep\_instc := InitFail$
$\oplus act4 \ : \ dep\_instcs := InitSuspPeers$
...
END

— La variable *dep_instcs* associe à chaque pair instanciant un service, un sous-ensemble de nouvelles instances déployées pour le service. Cette variable est initialisée à l'aide de la constante $InitSuspPeers$, car à l'état initial, aucune nouvelle instance n'a encore été déployée pour aucun service.

Nous notons x_{20} l'ensemble des variables de la machine M20.

Un invariant noté $I_{20}(x_{20})$ caractérise ces variables et est défini comme suit :

— Nous avons du typage :

$$inv1 \ : \ dep\_instcs \in (PEERS \times SERVICES) \nrightarrow \mathbb{P}(PEERS)$$

— La variable *dep_instcs* remplace une variable abstraite *dep_instc*. Les invariants $inv2$ et $inv3$ suivants, aussi appelés *invariants de collage*, nous permettent de faire le lien entre la variable abstraite et la variable concrète :

$$inv2 \ : \ \forall s \cdot s \in SERVICES \Rightarrow token\_owner(s) \mapsto s \in dom(dep\_instcs)$$
$$inv3 \ : \ \forall s \cdot s \in SERVICES \Rightarrow dep\_instcs(token\_owner(s) \mapsto s) = dep\_instc(s)$$

— $inv2$ exprime que l'instance « token owner » d'un service s fait partie des instances de s qui maintiennent une liste des instances déployées pour le service s.

— $inv3$ exprime que les listes des instances déployées pour un service s, maintenue par l'instance « token owner » du service s au niveau concret et par le service s au niveau abstrait sont les mêmes. C'est principalement cet invariant qui nous permet de remplacer *dep_instc* par *dep_instcs* : au lieu de récupérer globalement, au niveau du service s, les nouvelles instances déployées, nous les récupérons localement, au niveau d'une instance, qui est l'instance « token owner » du service s.

— La propriété $inv4$ exprime que chaque pair fonctionnel, instanciant un service est un pair qui monitore les états des services qu'il instancie.

$$inv4 \ : \ dom(run\_inst) \subseteq dom(inst\_state)$$

Les remplacements de variables obtenus par raffinement de données sont résumés par le tableau suivant :

Variable abstraite	**Variable concrète**
dep_instc	*dep_instcs*

Les axiomes du contexte C9, ainsi que l'invariant de collage I_{20} nous permettent de définir la relation bijective r entre les variables abstraites et concrètes. Nous pouvons alors appliquer, pour ce raffinement

de M19 en M20, la deuxième règle de raffinement automatique (cf. page 55) pour établir les propriétés satisfaites par M20, ainsi que pour justifier la préservation des propriétés Φ_{19} lors du raffinement.

Nous nous focalisons maintenant sur les propriétés de vivacité satisfaites par la machine M20. Pour cela, nous définissons quelques propriétés :

— $(\neg P_{18}) \ \widehat{=} \ \left(\begin{array}{l} \wedge prop = \left(\begin{array}{c} run\_inst(token\_owner(s) \mapsto s) \\ \setminus \\ (suspc\_inst(token\_owner(s) \mapsto s) \cup unav\_peers) \end{array} \right) \\ \wedge (prop \times \{s\} \times \{FAIL\_4\}) \subseteq inst\_state \\ \wedge fp \subseteq PEERS \\ \wedge fp = suspc\_inst(token\_owner(s) \mapsto s) \\ \wedge fp \neq \varnothing \\ \wedge fp \subset run\_peers(s) \\ \wedge (prop \times \{s\} \times fp) \subseteq suspct\_peers \\ \wedge rect\_inst(token\_owner(s) \mapsto s) = \varnothing \end{array} \right)$

Cette propriété exprime que le service s se trouve dans un état illégal $FAIL\_4$ et aucune des instances suspectées d'être dans un état illégal, dans un ensemble fp, n'ont fait l'objet d'un contact (avec ou sans succès) par l'instance « token owner » du service s ($token\_owner(s)$). Ces informations sont propagées aux instances courantes de s, sans celles demeurées indisponibles et celles suspectes.

— $(\neg P_{181}) \ \widehat{=} \ \left(\begin{array}{l} \wedge prop = \left(\begin{array}{c} run\_inst(token\_owner(s) \mapsto s) \\ \setminus \\ (suspc\_inst(token\_owner(s) \mapsto s) \cup unav\_peers) \end{array} \right) \\ \wedge (prop \times \{s\} \times \{FAIL\_4\}) \subseteq inst\_state \\ \wedge fp \subseteq PEERS \\ \wedge fp = suspc\_inst(token\_owner(s) \mapsto s) \\ \wedge fp \neq \varnothing \\ \wedge fp \subset run\_inst(token\_owner(s) \mapsto s) \\ \wedge (prop \times \{s\} \times fp) \subseteq suspct\_peers \\ \wedge suspc\_peers(token\_owner(s) \mapsto s) = rect\_inst(token\_owner(s) \mapsto s) \\ \wedge sf \subseteq PEERS \\ \wedge sf \subseteq suspct\_peers(token\_owner(s) \mapsto s) \\ \wedge sf = rctt\_inst(token\_owner(s) \mapsto s) \end{array} \right)$

Cette propriété exprime que le service s se trouve dans un état illégal $FAIL\_4$ et toutes les instances suspectées d'être dans un état illégal, dans un ensemble fp, ont fait l'objet d'un contact (avec ou sans succès) par l'instance « token owner » du service s ($token\_owner(s)$). Des instances suspectes, regroupées dans un ensemble sf, s'avèrent être dans un état légal, car elles ont été contactées avec succès par l'instance « token owner » du service s ($token\_owner(s)$). Ces informations sont propagées aux instances courantes de s, sans celles demeurées indisponibles et celles suspectes.

— $O_{17} \ \widehat{=} \ \left(\begin{array}{l} \wedge inst\_state(token\_owner(s) \mapsto s) = FAIL\_4 \\ \wedge i \in suspct\_peers(token\_owner(s) \mapsto s) \\ \wedge i \notin rect\_inst(token\_owner(s) \mapsto s) \end{array} \right)$

Cette propriété exprime qu'une instance i suspecte d'un service s (dans un état illégal) n'a pas encore été contactée par l'instance « token owner » du service s ($token\_owner(s)$).

— $O_{171} \ \widehat{=} \ \left(\begin{array}{l} \wedge inst\_state(token\_owner(s) \mapsto s) = FAIL\_4 \\ \wedge i \in suspct\_peers(token\_owner(s) \mapsto s) \\ \wedge i \notin rect\_inst(token\_owner(s) \mapsto s) \\ \wedge i \notin unav\_peers \end{array} \right)$

Cette propriété exprime qu'une instance i suspecte d'un service s (dans un état illégal), redevenue disponible, n'a pas encore été contactée par l'instance « token owner » du service s ($token\_owner(s)$).

— $O_{172} \ \widehat{=} \ \left(\begin{array}{l} \wedge inst\_state(token\_owner(s) \mapsto s) = FAIL\_4 \\ \wedge i \in suspct\_peers(token\_owner(s) \mapsto s) \\ \wedge i \notin rect\_inst(token\_owner(s) \mapsto s) \\ \wedge i \in unav\_peers \end{array} \right)$

Cette propriété exprime qu'une instance i suspecte d'un service s (dans un état illégal), redevenue disponible, n'a pas encore été contactée par l'instance « token owner » du service s ($token\_owner(s)$).

— $N_{17} \ \widehat{=} \ \begin{pmatrix} \wedge inst\_state(token\_owner(s) \mapsto s) = FAIL\_4 \\ \wedge i \in suspct\_peers(token\_owner(s) \mapsto s) \\ \wedge i \in rect\_inst(token\_owner(s) \mapsto s) \\ \wedge i \in rctt\_inst(token\_owner(s) \mapsto s) \end{pmatrix}$

Cette propriété exprime qu'une instance i suspecte d'un service s (dans un état illégal) a été contactée avec succès par l'instance « token owner » du service s ($token\_owner(s)$).

— $F(s) \ \widehat{=} \ (susp\_peers(s) \setminus rec\_inst[\{token\_owner(s)\}][\{s\}])$ exprime l'ensemble des instances d'un service s, suspectées d'être dans un état illégal et non encore contactées par l'instance « token owner » du service s ($token\_owner(s)$). Nous posons $F(s,n) \ \widehat{=} \ (card(F(s)) = n)$, avec $n \in \mathbb{N}$.

— $g_{90} \ \widehat{=} \ (F(s) \neq \varnothing)$

— $g_{91} \ \widehat{=} \ (i \notin unav\_peers(s))$

— $g_{92} \ \widehat{=} \ (i \in unav\_peers(s))$

— $g_{93} \ \widehat{=} \ (i \in rec\_inst[\{token\_owner(s)\}][\{s\}])$

— $g_{94} \ \widehat{=} \ (F(s) = \varnothing)$

— $R_{18} \ \widehat{=} \ \begin{pmatrix} \wedge prop = \begin{pmatrix} \cup \begin{pmatrix} run\_inst(token\_owner(s) \mapsto s \\ \setminus \\ suspc\_peers(token\_owner(s) \mapsto s) \end{pmatrix} \\ \cup rctt\_inst(token\_owner(s) \mapsto s)) \setminus unav\_peers \end{pmatrix} \\ \wedge (prop \times \{s\} \times \{FL\_DT\_4\}) \subseteq inst\_state \\ \wedge (prop \times \{s\} \times (fp \setminus sf)) \subseteq suspct\_peers \end{pmatrix}$

Cette propriété exprime que l'état illégal du service s a été détecté. Les instances dans un état légal, de l'ensemble sf, sont retirées des instances suspectées d'être dans un état illégal. Ces informations sont propagées aux instances courantes de s, avec celles recontactées avec succès, mais sans celles demeurées indisponibles et celles suspectes.

— $R_{171} \ \widehat{=} \ \begin{pmatrix} \wedge inst\_state(token\_owner(s) \mapsto s) = FL\_DT\_4 \\ \wedge suspct\_peers(token\_owner(s) \mapsto s) = \varnothing \end{pmatrix}$

Cette propriété exprime que l'état illégal du service s a été détecté. Les instances dans un état légal sont retirées des instances suspectées d'être dans un état illégal et le nombre des instances suspectes est passé à 0 : il n'y a plus d'instances suspectes.

— $R_{182} \ \widehat{=} \ \begin{pmatrix} \wedge inst\_state(token\_owner(s) \mapsto s) = FL\_DT\_4 \\ \wedge suspct\_peers(token\_owner(s) \mapsto s) \neq \varnothing \\ \wedge run\_p = run\_inst(token\_owner(s) \mapsto s) \end{pmatrix}$

Cette propriété exprime que l'état illégal du service s a été détecté. Les instances dans un état légal sont retirées des instances suspectées d'être dans un état illégal et il reste des instances suspectes. Un ensemble run_p contient les instances courantes du service s.

— $S_{19} \ \widehat{=} \ \begin{pmatrix} \wedge prop = \begin{pmatrix} run\_inst(token\_owner(s) \mapsto s) \\ \setminus \\ (suspct\_peers(token\_owner(s) \mapsto s) \cup unav\_peers) \end{pmatrix} \\ \wedge (prop \times \{s\} \times \{FL\_ACT\_4\}) \subseteq inst\_state \\ \wedge (prop \times \{s\} \times (run\_p \setminus (fp \setminus sf))) \subseteq run\_inst \\ \wedge (prop \times \{s\} \times \varnothing) \subseteq suspct\_peers \\ \wedge (prop \times \{s\} \times (fp \setminus sf)) \subseteq failr\_inst \end{pmatrix}$

Cette propriété exprime que l'état illégal du service s a été évalué. Les instances demeurées dans un état suspect sont retirées de la liste des instances courantes du service s, car elles sont considérées comme étant dans un état illégal. Ces informations sont propagées aux instances courantes de s, sans celles demeurées indisponibles et celles suspectes.

— $S_{181} \ \widehat{=} \ \begin{pmatrix} \wedge inst\_state(token\_owner(s) \mapsto s) = FL\_ACT\_4 \\ \wedge card(run\_inst(token\_owner(s) \mapsto s)) \geq min\_inst(s) \\ \wedge suspct\_peers(token\_owner(s) \mapsto s) = \varnothing \end{pmatrix}$

Cette propriété exprime que l'état illégal du service s a été évalué. Après évaluation, le nombre restant des instances courantes du service s est supérieur ou égal au nombre minimal d'instances

requises pour le fonctionnement correct du service s.

— $S_{182} \ \widehat{=} \ \left(\begin{array}{l} \wedge inst\_state(token\_owner(s) \mapsto s) = FL\_ACT\_4 \\ \wedge card(run\_inst(token\_owner(s) \mapsto s)) < min\_inst(s) \\ \wedge suspct\_peers(token\_owner(s) \mapsto s) = \varnothing \end{array} \right)$
Cette propriété exprime que l'état illégal du service s a été évalué. Après évaluation, le nombre restant des instances courantes du service s est inférieur au nombre minimal d'instances requises pour le fonctionnement correct du service s.

— $T_{611} \ \widehat{=} \ \left(\begin{array}{l} \wedge prop = run\_peers(s) \setminus unav\_peers \\ \wedge (prop \times \{s\} \times \{FL\_IGN\_4\}) \subseteq inst\_state \end{array} \right)$
Cette propriété exprime que l'état illégal du service s est ignoré. Cette information est propagée aux instances courantes de s, sans celles demeurées indisponibles.

— $T_{621} \ \widehat{=} \ \left(\begin{array}{l} \wedge prop = run\_peers(s) \setminus unav\_peers \\ \wedge (prop \times \{s\} \times \{FL\_CONF\_4\}) \subseteq inst\_state \end{array} \right)$
Cette propriété exprime que le service s est configuré. Cette information est propagée aux instances courantes de s, sans celles demeurées indisponibles.

— $A(s) \ \widehat{=} \ deplo\_inst(s) - card(actv\_instc(token\_owner(s) \mapsto s))$, où $A(s) \in 0..deplo\_inst(s)$; $A(s)$ représente le nombre d'instances à activer pour un service s lors des étapes intermédiaires de redéploiement, pour atteindre le nombre $deplo\_inst(s)$ (instances que le service s peut déployer en un coup). Nous notons : $A(s, n) \ \widehat{=} \ A(s) = n$, où $n \in 0..deplo\_inst(s)$.

— $B_{16} \ \widehat{=} \ \left(\begin{array}{l} \wedge inst\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4 \\ \wedge actv\_instc(token\_owner(s) \mapsto s) = \varnothing \end{array} \right)$
Cette propriété exprime qu'un service s se trouve dans la phase de **self-configuration** et qu'aucune nouvelle instance, pour atteindre le nombre d'instances $deplo\_inst(s)$ que le service s peut déployer en un coup, n'a encore été déployée.

— $C_{16} \ \widehat{=} \ \left(\begin{array}{l} \wedge inst\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4 \\ \wedge actv\_instc(token\_owner(s) \mapsto s) \neq \varnothing \\ \wedge o = deplo\_inst(s) \\ \wedge card(actv\_instc(token\_owner(s) \mapsto s)) = o \\ \wedge act = actv\_instc(token\_owner(s) \mapsto s) \end{array} \right)$
Cette propriété exprime qu'un service s se trouve dans la phase de **self-configuration** et que de nouvelles instances au nombre de $deplo\_inst(s)$ (instances que le service s peut déployer en un coup) ont été déployée. Un ensemble act contient ces nouvelles instances déployées.

— $g_{183} \ \widehat{=} \ \left(\begin{array}{l} \wedge suspct\_peers(token\_owner(s) \mapsto s) \neq \varnothing \\ \wedge run\_p = run\_inst(token\_owner(s) \mapsto s) \end{array} \right)$

— $g_{184} \ \widehat{=} \ \left(\ card(run\_inst(token\_owner(s) \mapsto s)) \geq min\_inst(s) \ \right)$

— $g_{185} \ \widehat{=} \ \left(\ card(run\_inst(token\_owner(s) \mapsto s)) < min\_inst(s) \ \right)$

— $Dpl(s) \ \widehat{=} \ min\_inst(s) - \left(\begin{array}{c} card(dep\_instcs(token\_owner(s) \mapsto s)) \\ + \\ card(run\_inst(token\_owner(s) \mapsto s)) \end{array} \right)$. $Dpl(s)$ donne pour un service s dans la phase de reconfiguration ($inst\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$) le nombre d'instances manquantes à déployer pour atteindre le minimum requis pour le fonctionnement correct du service s. $Dpl(s) \in m..k$, où m est le premier entier négatif ou nul atteint en faisant la différence entre $min\_inst(s)$ et ($card(dep\_instcs(token\_owner(s) \mapsto s)) + card(run\_inst(token\_owner(s) \mapsto s))$) et k le nombre original d'instances manquantes pour atteindre le minimum requis. Nous posons $Dpl(s, n) \ \widehat{=} \ (Dpl(s) = n)$, où $n \in m..k$.

— $g_{76} \ \widehat{=} \ (Dpl(s, k))$

— $g_{77} \ \widehat{=} \ (Dpl(s, m))$

— $g_{100} \ \widehat{=} \ Dpl(s) > 0$. Cette garde exprime que le nombre d'instances d'un service s (incluant celles courantes et celles nouvellement déployées) n'est pas encore supérieur ou égal au minimum requis pour le fonctionnement de s.

— $g_{151} \ \widehat{=} \ actv\_instc(token\_owner(s) \mapsto s) = \varnothing$

— $g_{102} \ \widehat{=} \ A(s, deplo\_inst(s))$. Cette garde exprime que le nombre d'instances à activer lors des étapes intermédiaires de redéploiement est $deplo\_inst(s)$, ce qui signifie que g_{101} est vraie.

— $g_{103} \ \widehat{=} \ A(s, 0)$. Cette garde exprime que le nombre d'instances activées lors des étapes intermé-

diaires de redéploiement est $deplo\_inst(s)$.

— $T_{2021} \;\widehat{=}\; \begin{pmatrix} \wedge inst\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4 \\ \wedge dep\_instc(token\_owner(s) \mapsto s) \neq \varnothing \\ \wedge Dpl(s, m) \end{pmatrix}$

Cette propriété exprime que le service s est en train d'être reconfiguré et que le nombre d'instances déployées est maintenant supérieur au minimum $min\_inst(s)$ requis pour le fonctionnement correct de s : $Dpl(s, m) \Leftrightarrow (card(dep\_instcs(token\_owner(s) \mapsto s)) + card(run\_inst(token\_owner(s) \mapsto s)) \geq min\_inst(s))$.

— $D_{20} \;\widehat{=}\; \begin{pmatrix} \wedge inst\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4 \\ \wedge card(act) = deplo\_inst(s) \\ \wedge prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers \\ \wedge (prop \times \{s\} \times act) \subseteq dep\_instcs \\ \wedge Dpl(s, n - deplo\_inst(s)) \\ \wedge actv\_instc(token\_owner(s) \mapsto s) = \varnothing \end{pmatrix}$

Cette propriété exprime qu'un service s se trouve dans la phase de **self-configuration** et que de nouvelles instances au nombre de $deplo\_inst(s)$ (instances que le service s peut déployer en un coup) ont été déployées. Ces nouvelles instances sont ajoutées à l'ensemble dep_instcs qui contient les instances déployées pour chaque service.

— $U_{20} \;\widehat{=}\; \begin{pmatrix} \wedge prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers \\ \wedge (prop \times \{s\} \times \{DPL\_4\}) \subseteq inst\_state \\ \wedge dep\_instcs(token\_owner(s) \mapsto s) \subseteq run\_inst(token\_owner(s) \mapsto s) \\ \wedge (prop \times \{s\} \times (run\_p \cup dep\_instc(s))) \subseteq run\_inst \end{pmatrix}$

Cette propriété exprime que le service s est déployé : l'ensemble $dep\_inst[\{s\}]$ de pairs, dont l'union avec les instances courantes du service s donne un nombre supérieur ou égal au nombre minimal des instances requises pour le fonctionnement correct du service s, fait maintenant partie des instances courantes fournissant le service s. Ces informations sont propagées aux instances courantes de s, sans celles demeurées indisponibles.

Nous utilisons ces propriétés pour construire le diagramme 8.25, qui représente visuellement la liste Φ_{20} des propriétés de vivacité caractérisant la machine M20. Nous détaillons maintenant le raffinement de M19 en M20. Mais auparavant, nous donnons, dans cette machine M20, les éléments relatifs aux services offerts par le système P2P [4] :

— Un service offert par le système P2P est appelé « management service », et il est fourni par des pairs membres du système P2P.

— Nous rappelons que chaque phase de la procédure de *self-healing* d'un service s est guidée par une instance spéciale, appelée « token owner ». Dans notre modélisation, l'instance « token owner » d'un service s est donnée par : $token\_owner(s)$.

— Un ensemble de pairs instanciant un même service est appelée « management peer group », raccourci en MPG. L'ensemble MPG d'un service s est donné dans nos modèles par l'expression $run\_inst(token\_owner(s) \mapsto s)$: il s'agit de la liste maintenue par l'instance « token owner »du service s, des pairs fonctionnels fournissant ce service.

— Chaque service s dispose de deux caractéristiques : le nombre minimal requis d'instances pour fournir le service s, donné dans nos modèles par l'expression $min\_inst(s)$ et le nombre de nouvelles instances qu'un service peut activer/déployer en un coup, donné dans nos modèles par l'expression $deplo\_inst(s)$. Ces caractéristiques sont généralement définies par les administrateurs du système P2P et le service s y a accès, par l'intermédiaire de son instance « token owner ».

Nous utilisons ces éléments pour détailler les événements du modèle M20. Nous commençons par présenter des événements événements relatifs à des actions de l'environnement, à savoir, MAKE_PEER_UNAVAIL (des pairs instanciant des services sont rendus indisponibles), MAKE_PEER_AVAIL (des pairs indisponibles sont à nouveau disponibles) et UNFAIL_PEER (une instance dans un état illégal d'un service revient dans un état légal).

— L'événement MAKE_PEER_UNAVAIL modélise le fait qu'un ensemble prs ($grd1$, $grd2$) de pairs fournissant des services soient rendus indisponibles (incapables de fournir des services et ne pouvant être recontactés par les instances « token owner » des services qu'ils fournissaient) ($act1$). Pour que cette événement soit observé, nous requérons que pour chaque service srv (qui peuvent être

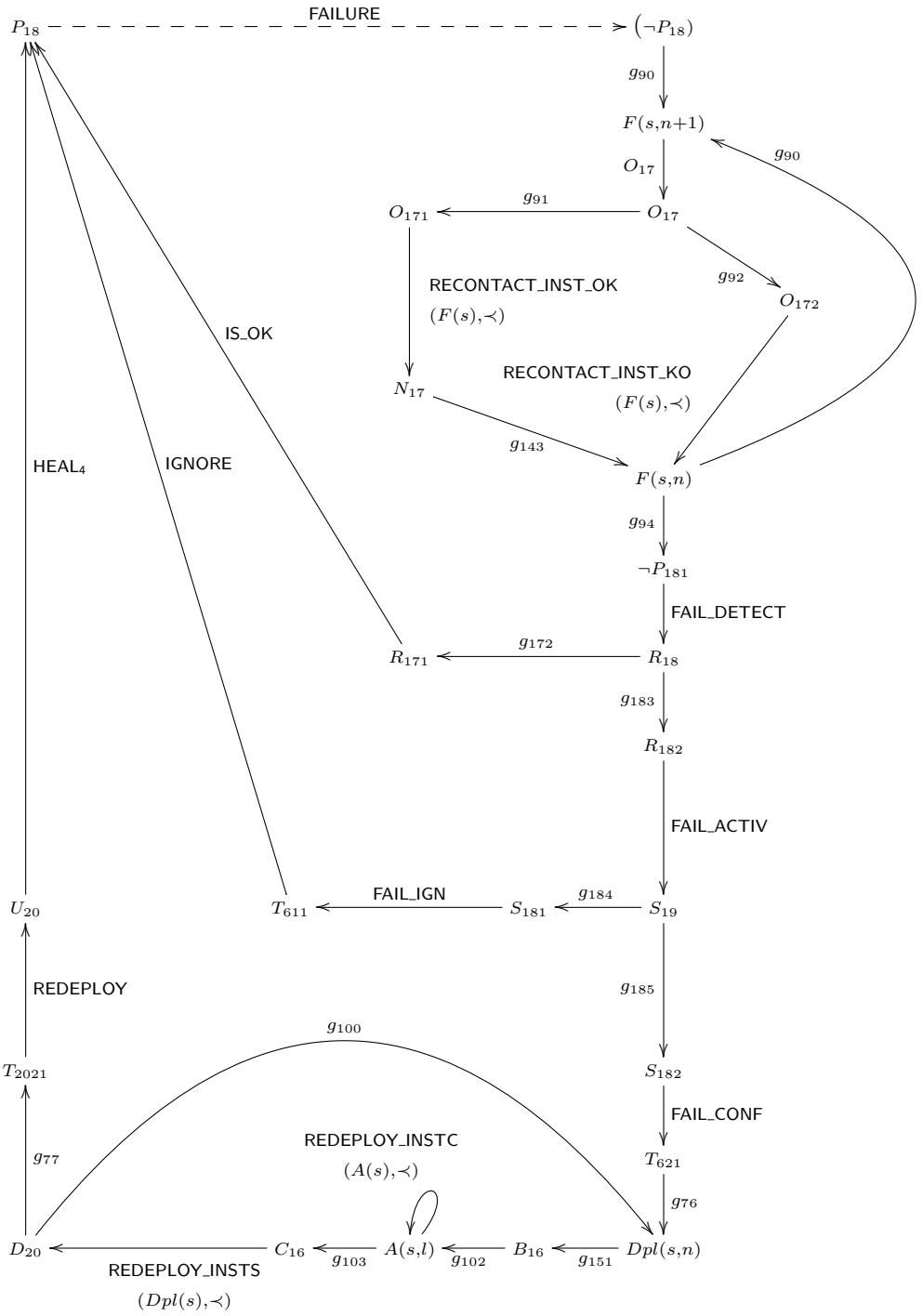


FIGURE 8.25 – Localisation

tous impactés par l'indisponibilité des membres de l'ensemble prs), au moins une instance de ce service, monitorant les états de ce dernier reste disponible et fonctionnel ($grd3$) ; nous requérons aussi qu'une instance « token owner » soit toujours disponible pour chaque service :

— Si l'instance « token owner » d'un service srv ne fait pas partie de l'ensemble prs, alors l'instance « token owner » de ce service demeure inchangée ($grd5$).

— Sinon, si l'instance « token owner » d'un service srv fait partie de l'ensemble prs, alors l'instance $E(srv)$ qui remplacera l'instance « token owner » du service srv doit posséder les caractéristiques suivantes ($grd6$) : le remplaçant doit faire partie des pairs fonctionnels, instanciant le service srv et ne pas être inclus dans les pairs indisponibles, l'ensemble prs, les instances défaillantes (dans un état illégal) ou celles suspectées d'être dans un état illégal ; en outre, le remplaçant $E(srv)$ doit maintenir une liste des instances de srv fonctionnelles, une liste des instances suspectes, une liste des instances défaillantes et une liste des instances déployées ; ces listes que le remplaçant maintient doivent être les mêmes que celles de l'instance « token owner » à remplacer et les états que monitorés par les deux instances doivent être identiques.

Si ces conditions sont satisfaites, les pairs contenus dans prs deviennent indisponibles ($act1$), les instances « token owner » des services sont mises à jour ($act2$), et les droits de contacter des instances ($act3$, $act4$), d'en déployer/activer ($act5$, $act11$), d'en suspecter ($act6$, $act7$), de maintenir une liste de celles fonctionnelles ($act9$), défaillantes ($act10$) ou encore le droit de monitorer des états de services ($act8$) sont retirés aux membres de l'ensemble prs.

EVENT MAKE_PEER_UNAVAIL REFINES MAKE_PEER_UNAVAIL $\widehat{=}$

ANY

prs, E

WHERE

$grd1 : prs \subseteq PEERS$

$grd2 : prs \nsubseteq unav\_peers$

$grd3 : \forall srv \cdot srv \in SERVICES \Rightarrow dom(dom(inst\_state) \rhd \{srv\}) \setminus prs \neq \varnothing$

$grd4 : E \in SERVICES \rightarrow PEERS$

$$grd5 : \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge \left(\begin{array}{c} token\_owner(srv) \\ \notin \\ prs \end{array} \right) \end{array} \right) \Rightarrow E(srv) = token\_owner(srv)$$

$$grd6 : \forall srv \cdot \left(\begin{array}{l} \wedge srv \in SERVICES \\ \wedge \left(\begin{array}{c} token\_owner(srv) \\ \in \\ prs \end{array} \right) \end{array} \right)$$

$$\Rightarrow \left(\begin{array}{l} \wedge E(srv) \in \left(\begin{array}{l} run\_peers(srv) \\ \setminus \\ \left(\begin{array}{l} \cup unav\_peers \\ \cup prs \\ \cup failr\_inst(token\_owner(srv) \mapsto srv) \\ \cup suspct\_peers \left(\begin{array}{c} token\_owner(srv) \\ \mapsto \\ srv \end{array} \right) \end{array} \right) \end{array} \right) \\ \wedge E(srv) \mapsto srv \in \left(\begin{array}{l} \cap dom(run\_inst) \cap dom(suspct\_peers) \\ \cap dom(failr\_inst) \cap dom(dep\_instcs) \end{array} \right) \\ \wedge \left(\begin{array}{c} inst\_state(E(srv) \mapsto srv) \\ = \\ inst\_state(token\_owner(srv) \mapsto srv) \end{array} \right) \wedge \left(\begin{array}{c} run\_inst(E(srv) \mapsto srv) \\ = \\ run\_inst(token\_owner(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{c} failr\_inst(E(srv) \mapsto srv) \\ = \\ failr\_inst(token\_owner(srv) \mapsto srv) \end{array} \right) \wedge \left(\begin{array}{c} dep\_instcs(E(srv) \mapsto srv) \\ = \\ dep\_instcs(token\_owner(srv) \mapsto srv) \end{array} \right) \\ \wedge \left(\begin{array}{l} suspct\_peers(E(srv) \mapsto srv) = suspct\_peers(token\_owner(srv) \mapsto srv) \end{array} \right) \end{array} \right)$$

THEN

$act1 : unav\_peers := unav\_peers \cup prs$

$act2 : token\_owner := token\_owner \lhd\!\!+ E$

$act3 : rect\_inst := ((prs \times SERVICES) \lhd rect\_inst) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$

$act4 : rctt\_inst := ((prs \times SERVICES) \lhd rctt\_inst) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$

$act5 : actv\_instc := ((prs \times SERVICES) \lhd actv\_instc) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$

$act6 : suspct\_peers := (prs \times SERVICES) \lhd suspct\_peers$

$act7 : suspc\_inst := ((prs \times SERVICES) \lhd suspc\_inst) \lhd\!\!+ (((E \setminus token\_owner)^{-1}) \times \{\varnothing\})$

$act8 : inst\_state := (prs \times SERVICES) \lhd inst\_state$

$act9 : run\_inst := (prs \times SERVICES) \lhd run\_inst$

$act10 : failr\_inst := (prs \times SERVICES) \lhd failr\_inst$

$act11 : dep\_instcs := (prs \times SERVICES) \lhd dep\_instcs$

END

— L'événement MAKE_PEER_AVAIL modélise le fait qu'un pair p indisponible (incapable de fournir un

service et ne pouvant être contacté par les instances « token owner » des services qu'il fournissait) ($grd2$) redevienne à nouveau disponible ($act1$) : le pair p est retiré de la liste des pairs/instances indisponibles.

EVENT MAKE_PEER_AVAIL REFINES MAKE_PEER_AVAIL $\widehat{=}$
ANY
p
WHERE
$grd1 : p \in PEERS$
$grd2 : p \in unav\_peers$
THEN
$act1 : unav\_peers := unav\_peers \setminus \{p\}$
END

— Evénement UNFAIL_PEER : un pair p instanciant un service s fait partie de la liste maintenue par l'instance « token owner » du service s des instances de s défaillantes (dans un état illégal) ($grd4$). Le pair retourne à un état fonctionnel (légal) ($act1$) et cette information est propagée à un ensemble $prop$, constitué des instances fonctionnelles de s, sans celles indisponibles ($grd5$, $act1$).

EVENT UNFAIL_PEER REFINES UNFAIL_PEER $\widehat{=}$
ANY
$s, p, prop$
WHERE
$grd1 : s \in SERVICES$
$grd2 : prop \subseteq PEERS$
$grd3 : p \in PEERS$
$grd4 : p \in failr\_inst(token\_owner(s) \mapsto s)$
$grd5 : prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
$act1 : failr\_inst := failr\_inst \Leftarrow ((prop \times \{s\}) \times \left\{ \begin{array}{c} failr\_inst(token\_owner(s) \mapsto s) \\ \setminus \\ \{p\} \end{array} \right\})$
END

Nous détaillons maintenant l'entrée d'un service dans un état illégal. Avec l'événement MAKE_PEER_-UNAVAIL, les deux événements SUSPECT_INST et FAILURE suivants décrivent comment un service entre dans un état illégal.

— Evénement SUSPECT_INST : un sous-ensemble $susp$ des instances d'un service s dans un état légal ($grd5$) sont devenues indisponibles ($grd3$, $grd6$). L'instance « token owner » du service s suspecte alors ces dernières d'être dans un état illégal ($grd4$, $act1$).

EVENT SUSPECT_INST REFINES SUSPECT_INST
ANY
$s, susp$
WHERE
$grd1 : s \in SERVICES$
$grd2 : susp \subseteq PEERS$
$grd3 : susp = run\_inst(token\_owner(s) \mapsto s) \cap unav\_peers$
$grd4 : suspc\_inst(token\_owner(s) \mapsto s) = \varnothing$
$grd5 : inst\_state(token\_owner(s) \mapsto s) = RUN\_4$
$grd6 : susp \neq \varnothing$
THEN
$act1 : suspc\_inst(token\_owner(s) \mapsto s) := susp$
END

— Evénement FAILURE : un service s se trouve dans un état légal RUN_4 et son instance « token owner » suspecte des instances d'être défaillantes, dans des états illégaux. L'instance « token owner » déclare alors le service s dans un état illégal $FAIL\_4$ et propage cette information ($act1$), ainsi que la liste des instances suspectées d'être défaillantes ($act2$) à un ensemble $prop$ contenant les instances fonctionnelles du service s, sans celles suspectées d'être défaillantes et celles indisponibles ($grd5$). La liste intermédiaire des instances suspectes maintenue par l'instance « token owner » est remise à zéro ($act3$).

EVENT FAILURE REFINES FAILURE $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1 : s \in SERVICES$
$grd2 : prop \subseteq PEERS$
$grd3 : inst\_state(token\_owner(s) \mapsto s) = RUN\_4$
$grd4 : suspc\_inst(token\_owner(s) \mapsto s) \neq \varnothing$
$grd5 : prop = run\_inst(token\_owner(s) \mapsto s) \setminus (suspc\_inst(token\_owner(s) \mapsto s) \cup unav\_peers)$
THEN
$act1 : inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FAIL\_4\})$
$act2 : suspct\_peers := suspct\_peers \Leftarrow ((prop \times \{s\}) \times \{suspc\_inst(token\_owner(s) \mapsto s)\})$
$act3 : suspc\_inst(token\_owner(s) \mapsto s) := \varnothing$
END

Nous décrivons maintenant la partie du modèle décrivant la phase de **self-detection**, modélisée par les événements RECONTACT_INST_OK, RECONTACT_INST_KO, FAIL_DETECT, IS_OK.

— Evénements RECONTACT_INST_OK et RECONTACT_INST_KO : ces deux événements modélisent l'opération de recontact des instances d'un service s devenues suspectes par l'instance « token owner » de ce dernier. Ces événements sont observés tant que l'instance « token owner » du service s n'a pas essayé de recontacter toutes les instances suspectes de ce dernier ($grd7$). L'instance « token owner » essaie de recontacter une instance i, parce qu'elle est suspecte ($grd4$, $grd5$), parce qu'il n'a pas encore essayé de la recontacter ($grd6$). Nous distinguons deux cas : **(1)** soit l'instance i est redevenue disponible ($grd5$ de RECONTACT_INST_OK), auquel cas l'opération de recontact de l'instance i est réussie ($act1$ et $act2$ de RECONTACT_INST_OK), **(2)** soit l'instance i est restée indisponible ($grd5$ de RECONTACT_INST_KO), auquel cas l'opération de recontact de l'instance i échoue ($act1$ de RECONTACT_INST_KO).

EVENT RECONTACT_INST_OK REFINES RECONTACT_INST_OK $\widehat{=}$
ANY
s, i
WHERE
$grd1 : s \in SERVICES$
$grd2 : i \in PEERS$
$grd3 : inst\_state(token\_owner(s) \mapsto s) = FAIL\_4$
$grd4 : suspct\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
$grd5 : i \in suspct\_peers(token\_owner(s) \mapsto s) \setminus unav\_peers$
$grd6 : i \notin rect\_inst(token\_owner(s) \mapsto s)$
$grd7 : rect\_inst(token\_owner(s) \mapsto s) \subset suspct\_peers(token\_owner(s) \mapsto s)$
THEN
$act1 : rect\_inst(token\_owner(s) \mapsto s) := rect\_inst(token\_owner(s) \mapsto s) \cup \{i\}$
$act2 : rctt\_inst(token\_owner(s) \mapsto s) := rctt\_inst(token\_owner(s) \mapsto s) \cup \{i\}$
END

EVENT RECONTACT_INST_KO REFINES RECONTACT_INST_KO $\widehat{=}$
ANY
s, i
WHERE
$grd1 : s \in SERVICES$
$grd2 : i \in PEERS$
$grd3 : inst\_state(token\_owner(s) \mapsto s) = FAIL\_4$
$grd4 : suspct\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
$grd5 : i \in suspct\_peers(token\_owner(s) \mapsto s) \cap unav\_peers$
$grd6 : i \notin rect\_inst(token\_owner(s) \mapsto s)$
$grd7 : rect\_inst(token\_owner(s) \mapsto s) \subset suspct\_peers(token\_owner(s) \mapsto s)$
THEN
$act1 : rect\_inst(token\_owner(s) \mapsto s) := rect\_inst(token\_owner(s) \mapsto s) \cup \{i\}$
END

— Evénement FAIL_DETECT : cet événement est observé après l'opération de recontact de toutes les instances suspectes d'un service s par son instance « token owner » ($grd5$). Des instances de s sont suspectées ($grd4$) d'être dans un état illégal ($grd4$) : l'instance « token owner » détecte donc un possible état illégal du service s. Elle constitue une liste d'instance suspectes $susp$ ($grd8$), composée des instances suspectes, mais sans celles qu'elle a réussi à recontacter ($grd8$). L'instance « token owner » propage cette liste d'instance suspectes, ainsi que le fait qu'un état illégal ait été détecté à un ensemble $prop$ ($act1$, $act2$), constitué des instances fonctionnelles de s, sans celles suspectes et celles que l'instance « token owner » n'a pas réussi à recontacter ($grd6$). Les listes des instances suspectes recontactées par l'instance « token owner » sont remises à zéro ($act3$, $act4$).

EVENT FAIL_DETECT REFINES FAIL_DETECT $\widehat{=}$
ANY
$s, prop, susp$
WHERE
$grd1 : s \in SERVICES$
$grd2 : prop \subseteq PEERS$
$grd7 : susp \subseteq PEERS$
$grd3 : inst\_state(token\_owner(s) \mapsto s) = FAIL\_4$
$grd4 : suspct\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
$grd5 : rect\_inst(token\_owner(s) \mapsto s) = suspct\_peers(token\_owner(s) \mapsto s)$
$grd6 : prop = ((run\_inst(token\_owner(s) \mapsto s) \setminus \begin{pmatrix} \cup suspct\_peers(token\_owner(s) \mapsto s)) \\ \cup rctt\_inst(token\_owner(s) \mapsto s)) \setminus unav\_peers \end{pmatrix}$
$grd8 : susp = suspct\_peers(token\_owner(s) \mapsto s) \setminus rctt\_inst(token\_owner(s) \mapsto s)$
THEN
$act1 : inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FL\_DT\_4\})$
$act2 : suspct\_peers := suspct\_peers \Leftarrow ((prop \times \{s\}) \times \{susp\})$
$act3 : rect\_inst(token\_owner(s) \mapsto s) := \varnothing$
$act4 : rctt\_inst(token\_owner(s) \mapsto s) := \varnothing$
END

— Evénement IS_OK : un service s se trouve dans la phase de **self-detection** ($grd3$) et l'instance « token owner » du service a réussi à contacter toutes les instances suspectes : par conséquent, la liste des instances suspectes maintenue par l'instance « token owner » est vide. L'état illégal du service s était donc une fausse alerte et le service s peut retourner à un état légal RUN_4 ; l'instance « token owner » du service s propage ces informations à un ensemble $prop$ constitué des instances fonctionnelles du service s, sans celles indisponibles.

EVENT IS_OK REFINES IS_OK $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1 : s \in SERVICES$
$grd2 : prop \subseteq PEERS$
$grd3 : inst\_state(token\_owner(s) \mapsto s) = FL\_DT\_4$
$grd4 : suspct\_peers(token\_owner(s) \mapsto s) = \varnothing$
$grd5 : prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
$act1 : inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{RUN\_4\})$
END

Nous nous intéressons maintenant à la phase de **self-activation**, durant laquelle, un service qui a détecté un état illégal réel évalue si cet état illégal est critique ou non. Cette phase est modélisée par les événements suivants :

— L'événement FAIL_ACTIV modélise l'évaluation de la gravité de l'état illégal d'un service s. L'instance « token owner » du service s n'a pas réussi à recontacter toutes les instances suspectes : la liste des instances suspectes maintenue par l'instance « token owner » n'est pas vide ($grd4$). Le service passe ainsi de la phase de **self-detection** ($grd3$) à la phase de **self-activation** ($act1$). L'instance « token owner » propage alors à un ensemble $prop$ constitué des instances fonctionnelles du service s, mais sans les instances indisponibles ou suspectes, cette information ($act1$), ainsi que les mises à jour concernant : **(1)** les instances défaillantes (dans un état illégal), c'est-à-dire que les instances suspectes deviennent des instances défaillantes ($act3$, $act4$), **(2)** les instances fonctionnelles du service s, à savoir que l'on retire de cet ensemble d'instances, les instances suspectes devenues défaillantes ($act2$).

EVENT FAIL_ACTIV REFINES FAIL_ACTIV $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $prop \subseteq PEERS$
$grd3$: $inst\_state(token\_owner(s) \mapsto s) = FL\_DT\_4$
$grd4$: $suspct\_peers(token\_owner(s) \mapsto s) \neq \varnothing$
$grd5$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus (unav\_peers \cup suspct\_peers(token\_owner(s) \mapsto s))$
THEN
$act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FL\_ACT\_4\})$
$act2$: $run\_inst := run\_inst \Leftarrow ((prop \times \{s\}) \times \left\{ \begin{array}{c} run\_inst(token\_owner(s) \mapsto s) \\ \setminus \\ suspct\_peers(token\_owner(s) \mapsto s) \end{array} \right\})$
$act3$: $failr\_inst := failr\_inst \Leftarrow ((prop \times \{s\}) \times \left\{ \begin{array}{l} \cup failr\_inst(token\_owner(s) \mapsto s) \\ \cup suspct\_peers(token\_owner(s) \mapsto s) \end{array} \right\})$
$act4$: $suspct\_peers := suspct\_peers \Leftarrow ((prop \times \{s\}) \times \{\varnothing\})$
END

— Evénement FAIL_IGNORE : cet événement modélise un des résultats de la phase de **self-activation**. L'instance « token owner » d'un service s a détecté que le nombre d'instances encore fonctionnelles du service s est supérieur ou égal au nombre d'instances minimal requis pour fournir s ($grd4$). L'état illégal du service s peut donc être ignoré ($act1$) et l'instance « token owner » du service s propage cette information à un ensemble $prop$ ($act1$) constitué des instances fonctionnelles de s, sans celles qui sont restées indisponibles ($grd5$).

EVENT FAIL_IGNORE REFINES FAIL_IGNORE $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $prop \subseteq PEERS$
$grd3$: $inst\_state(token\_owner(s) \mapsto s) = FL\_ACT\_4$
$grd4$: $card(run\_inst(token\_owner(s) \mapsto s)) \geq min\_inst(s)$
$grd5$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
$act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FAIL\_IGN\_4\})$
END

— Evénement IGNORE : l'instance « token owner » d'un service s considère que l'état illégal de ce dernier peut être ignoré ($grd3$). L'état illégal du service est donc ignoré et ce dernier revient dans un état légal RUN_4. Ces informations sont propagées par l'instance « token owner » du service à un ensemble $prop$ ($act1$) constitué des instances fonctionnelles de s, sans celles qui sont restées indisponibles ($grd4$).

EVENT IGNORE REFINES IGNORE $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $prop \subseteq PEERS$
$grd3$: $inst\_state(token\_owner(s) \mapsto s) = FAIL\_IGN\_4$
$grd4$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
$act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{RUN\_4\})$
END

Nous nous focalisons sur la phase de **self-configuration** : l'état illégal détecté d'un service est réel et critique. Des événements modélisant cette phase sont alors observés.

— Evénement FAIL_CONFIGURE : l'état illégal détecté par l'instance « token owner » d'un service s est critique ($grd3$), parce que le nombre d'instances fonctionnelles courantes du service s est inférieur au nombre d'instances minimal requis pour fournir le service s ($grd4$). Le service passe alors de la phase de **self-activation** ($grd3$) à celle de **self-configuration** ($act1$). L'instance « token owner » du service s propage ces information à un ensemble $prop$ ($act1$) composé des instances fonctionnelles du service s, sans celles indisponibles ($grd5$).

EVENT FAIL_CONFIGURE REFINES FAIL_CONFIGURE $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1 : s \in SERVICES$
$grd2 : prop \subseteq PEERS$
$grd3 : inst\_state(token\_owner(s) \mapsto s) = FL\_ACT\_4$
$grd4 : card(run\_inst(token\_owner(s) \mapsto s)) < min\_inst(s)$
$grd5 : prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
$act1 : inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{FL\_CONF\_4\})$
END

— Evénement REDEPLOY_INSTC : l'instance « token owner » d'un service s déploie/active une nouvelle instance i ($act1$) du service s, parmi des pairs n'ayant pas encore instancié le service s, et aussi n'étant ni défaillants, ni indisponibles, ni déjà activés pour le service s ($grd3$). Cet événement est observé tant que le nombre d'instances activées est inférieur au nombre d'instances que le « token owner » du service peut déployer en un seul coup ($grd6$), et si le nombre d'instances courantes du service s, auquel on ajoute le nombre d'instances nouvellement déployées est inférieur au minimum requis pour instancier le service s ($grd7$).

EVENT REDEPLOY_INSTC REFINES REDEPLOY_INSTC $\widehat{=}$
ANY
s, i
WHERE
$grd1 : s \in SERVICES$
$grd2 : i \in PEERS$
$grd3 : i \notin \left(\begin{array}{l} \cup run\_inst(token\_owner(s) \mapsto s) \\ \cup failr\_inst(token\_owner(s) \mapsto s) \\ \cup unav\_peers \\ \cup dep\_instcs(token\_owner(s) \mapsto s) \end{array} \right)$
$grd4 : i \notin actv\_instc(token\_owner(s) \mapsto s)$
$grd5 : inst\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
$grd6 : card(actv\_instc(token\_owner(s) \mapsto s)) < deplo\_inst(s)$
$grd7 : card(dep\_instcs(token\_owner(s) \mapsto s)) + card(run\_inst(token\_owner(s) \mapsto s)) < min\_inst(s)$
THEN
$act1 : actv\_instc(token\_owner(s) \mapsto s) := actv\_instc(token\_owner(s) \mapsto s) \cup \{i\}$
END

— Evénement REDEPLOY_INSTS : le nombre d'instances nouvelles d'un service s que l'instance « token owner » de ce dernier peut activer en un coup est atteint ($grd3$). Ces instances activées sont alors déployées ($act1$, $act2$) officiellement par l'instance « token owner » : cette information est propagée à un ensemble $prop$ ($act1$) constitué des instances fonctionnelles du service s, sans celles indisponibles ($grd6$). Cet événement est observé tant que le nombre d'instances courantes du service s, auquel on ajoute le nombre d'instances nouvellement déployées est inférieur au minimum requis pour instancier le service s ($grd6$).

EVENT REDEPLOY_INSTS REFINES REDEPLOY_INSTS $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1 : s \in SERVICES$
$grd2 : prop \subseteq PEERS$
$grd3 : card(actv\_instc(token\_owner(s) \mapsto s)) = deplo\_inst(s)$
$grd4 : card(dep\_instcs(token\_owner(s) \mapsto s)) + card(run\_inst(token\_owner(s) \mapsto s)) < min\_inst(s)$
$grd5 : inst\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
$grd6 : prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
$act1 : dep\_instcs := dep\_instcs \Leftarrow ((prop \times \{s\}) \times \left\{ \begin{array}{l} \cup dep\_instcs(token\_owner(s) \mapsto s) \\ \cup actv\_instc(token\_owner(s) \mapsto s) \end{array} \right\})$
$act2 : actv\_instc(token\_owner(s) \mapsto s) := \varnothing$
END

— Evénement REDEPLOY : Cet événement est observé lorsque le redéploiement d'un service s se termine ($grd4$) : le nombre d'instances courantes du service s, auquel on ajoute le nombre d'instances nouvellement déployées ($grd5$) est supérieur ou égal au minimum requis pour instancier le service s ($grd6$). La terminaison de l'opération de redéploiement, ainsi que la mise à jour du nombre courant d'instances fonctionnelles du services, c'est-à-dire que les instances déployées sont

maintenant considérées comme des instances fonctionnelles, son propagées à un ensemble *prop* ($act1$) constitué des instances fonctionnelles du service s, sans celles indisponibles ($grd7$).

EVENT REDEPLOY REFINES REDEPLOY $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $prop \subseteq PEERS$
$grd3$: $inst\_state(token\_owner(s) \mapsto s) = FL\_CONF\_4$
$grd4$: $actv\_instc(token\_owner(s) \mapsto s) = \varnothing$
$grd5$: $dep\_instcs(token\_owner(s) \mapsto s) \neq \varnothing$
$grd6$: $card(run\_inst(token\_owner(s) \mapsto s)) + card(dep\_instcs(token\_owner(s) \mapsto s)) \geq min\_inst(s)$
$grd7$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
$act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{DPL\_4\})$
$act2$: $run\_inst := run\_inst \Leftarrow ((prop \times \{s\}) \times \left\{ \begin{array}{l} \cup run\_inst(token\_owner(s) \mapsto s) \\ \cup dep\_instcs(token\_owner(s) \mapsto s) \end{array} \right\})$
$act3$: $dep\_instcs := dep\_instcs \Leftarrow ((prop \times \{s\}) \times \{\varnothing\})$
END

La guérison effective d'un service, après la phase de **self-configuration**, est modélisée par un événement HEAL_4.

— Evénement HEAL_4 : l'instance « token owner » d'un service s considère que l'étape de redéploiement du service est terminée ($grd3$). Le service revient dans un état légal RUN_4. Ces informations sont propagées par l'instance « token owner » du service à un ensemble *prop* ($act1$) constitué des instances fonctionnelles de s, sans celles qui sont restées indisponibles ($grd4$).

EVENT HEAL_4 REFINES HEAL_4 $\widehat{=}$
ANY
$s, prop$
WHERE
$grd1$: $s \in SERVICES$
$grd2$: $prop \subseteq PEERS$
$grd3$: $inst\_state(token\_owner(s) \mapsto s) = DPL\_4$
$grd4$: $prop = run\_inst(token\_owner(s) \mapsto s) \setminus unav\_peers$
THEN
$act1$: $inst\_state := inst\_state \Leftarrow ((prop \times \{s\}) \times \{RUN\_4\})$
END

Ce vingtième raffinement est résumé par le diagramme suivant :

Ce raffinement nous a présenté le modèle local de la procédure de self-healing. La section suivante présente nos conclusions et discussions pour ce chapitre.

8.4 Conclusions

Nous avons présenté dans ce chapitre une méthodologie pour l'analyse et l'étude des systèmes auto-stabilisants [1] (voir figure 8.1) : nous caractérisons ces systèmes par trois types d'états, qui sont les états légitimes, les états illégitimes et les états de récupération. Le paradigme *service-as-event*, ainsi que les propriétés de vivacité nous permettent d'établir les relations entre ces états : l'idée fondamentale que nous appliquons ici est l'identification de ces états, ainsi que des étapes abstraites permettant de naviguer d'un type d'état à un autre. Le raffinement nous permet ensuite de détailler ces étapes abstraites, pour arriver à une solution proche d'un algorithme. Nous avons illustré cette méthodologie à travers l'analyse formelle d'un système P2P doté d'une procédure d'« auto-guérison » : nous avons débuté d'un modèle abstrait M0 pour aboutir à un modèle concret M20.

Le modèle concret M20 ainsi obtenu nous permet de distinguer clairement les trois phases de la procédure de *self-healing* d'un service entré dans un état illégitime :

1. La phase de **self-detection** commence après l'observation de l'événement FAILURE : un sous-ensemble des pairs instanciant un service s est suspecté d'être dans un état illégitime, parce que l'instance « token owner » du service s ne reçoit aucun signal de ces pairs et de ce fait les considère comme étant indisponibles. L'objectif de cette phase est de confirmer l'état illégitime du service en recontactant chacun des pairs suspects, ce qui est modélisé par deux événements : un

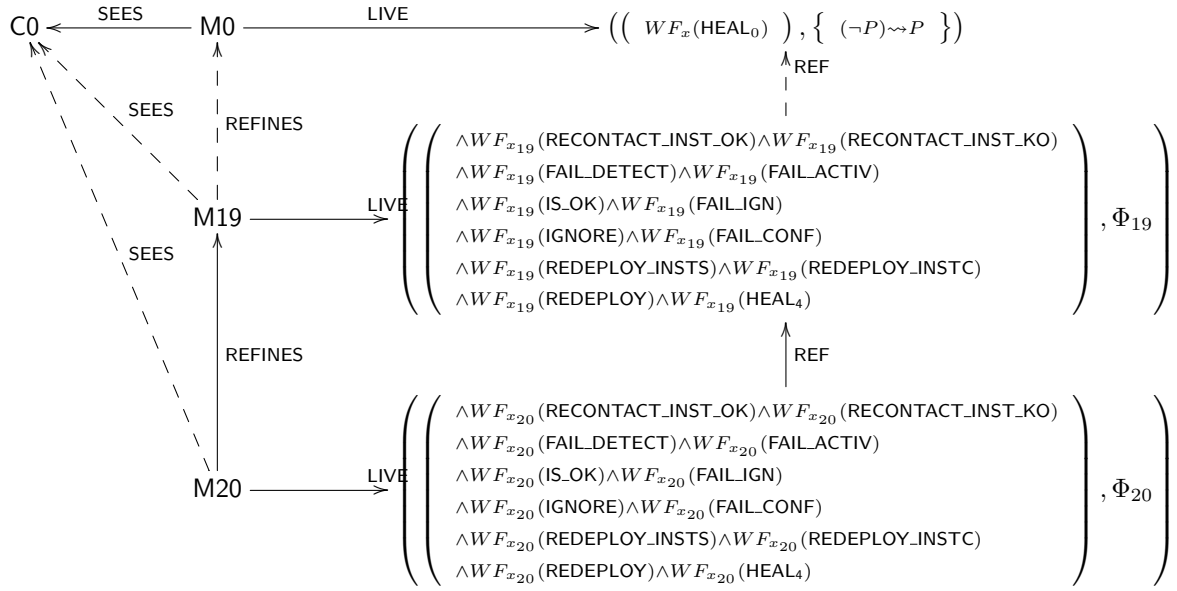


FIGURE 8.26 – Vingtième raffinement du self-healing

événement RECONTACT_INST_OK modélisant un contact d'un pair suspect réussi et RECONTACT_INST_KO modélisant l'échec d'un contact d'un pair suspect. L'observation ensuite d'un événement FAIL_DETECT permet de déterminer si l'appel à la procédure de **self-activation** est nécessaire (tous les pairs suspects n'ont pas pu être recontactés) ou si l'état illégitime détecté est une fausse alerte, auquel cas le service s converge vers un état légitime (événement IS_OK).

2. La phase de **self-activation** débute par l'observation de l'événement FAIL_ -ACTIV, qui modélise l'appel, depuis la phase de **self-detection**, à celle de **self-activation**. L'état illégitime du service y est évalué : si le nombre courant d'instances fonctionnelles dans un état légitime du service est supérieur ou égal au minimum requis pour fournir le service s, l'état illégitime est ignoré et le service converge vers un état légitime (événements FAIL_IGNORE et IGNORE), sinon la phase de **self-configuration** est activé, par l'observation de l'événement FAIL_CONFIG, à cheval sur les phases de **self-activation** et de **self-configuration**.
3. La phase de **self-configuration** : l'entrée du service s dans cette phase est modélisée par l'événement FAIL_CONFIG. Cette phase se focalise surtout sur le remplacement des instances défaillantes du service : tant que le nombre minimal d'instances fonctionnelles requises pour fournir le service s n'est pas atteint, de nouvelles instances de s sont déployées sur des pairs sains et n'instanciant pas encore s (événements REDEPLOY_INSTC et REDEPLOY_INSTS). L'événement REDEPLOY marque la fin du redéploiement du service s, car le nombre minimal d'instances fonctionnelles requises pour s a été atteint ou dépassé. L'événement HEAL modélise la convergence du service s vers un état légitime.

Nous remarquons que la décomposition en trois phases proposées par Marquezan et al [4] se retrouve dans nos modèles, ainsi que les étapes décrites par les algorithmes 9, 10, 11 et 12 détaillant ces phases. Nous avons en outre formulé des hypothèses pour nous assurer du fonctionnement correct de la procédure de self-healing :

— Evénement MAKE_PEER_UNAVAIL : *si l'instance « token owner » d'un service s devient indisponible, il existe au moins un pair instanciant le service s, et possédant les mêmes caractéristiques que le « token owner » (monitorant le même état du service s, maintenant les mêmes listes d'instances fonctionnelles, suspectes, défaillantes, etc), fonctionnel et disponible, et étant en mesure de remplacer l'instance « token owner » du service s.*

— Evénement REDEPLOY_INSTC : *il existe toujours un nombre suffisant de pairs sains, disponibles,*

n'instanciant pas encore le service s, sur lesquelles s peut être déployé, permettant ainsi d'atteindre ou de dépasser le nombre minimal d'instances requises pour fournir le service s.

La seconde hypothèse nous démarque de la démarche de Marquezan et al [4] pour la reconfiguration d'un service : dans notre cas, le déploiement de remplaçantes pour les instances défaillantes d'un service dans un état illégitiment peut toujours être effectué, alors que Marquezan et al limitent ce déploiement en y ajoutant une limite (*num_retries*) sur le nombre d'essais de redéploiements effectués (voir algorithme 12).

La complexité du développement formel du système P2P doté d'une procédure d'« auto-guérison » est donnée par le nombre d'obligations de preuve déchargées automatiquement ou interactivement :

Modèles	Total	Automatiques		Interactives	
C0	0	0	100%	0	0%
C1	0	0	100%	0	0%
C2	0	0	100%	0	0%
C3	0	0	100%	0	0%
C4	0	0	100%	0	0%
C5	4	4	100%	0	0%
C6	4	4	75%	1	25%
C7	4	4	75%	1	25%
C8	6	6	100%	0	0%
C9	6	6	100%	0	0%
M0	3	3	100%	0	0%
M1	21	18	85.71%	3	14.29%
M2	46	43	93.48%	3	6.52%
M3	68	24	35.30%	44	64.70%
M4	142	13	9.15%	129	90.85%
M5	46	18	39.13%	28	60.87%
M6	83	21	22.58%	62	77.42%
M7	35	12	34.29%	23	65.71%
M8	56	24	42.86%	32	57.14%
M9	60	26	43.33%	34	56.67%
M10	60	20	33.33%	40	66.67%
M11	124	31	25%	93	75%
M12	63	23	36.51%	40	63.49%
M13	33	10	30.30%	23	69.70%
M14	113	39	34.51%	74	65.49%
M15	48	10	20.83%	38	79.17%
M16	201	21	10.45%	180	89.55%
M17	69	22	31.88%	47	68.12%
M18	60	11	18.33%	49	81.67%
M19	29	6	20.69%	23	79.31%
M20	50	8	16%	29	84%
Total	1434	425	29.64%	1009	70.36%

TABLE 8.1 – Systèmes auto-stabilisants : Bilan des POs

Nous précisons ici qu'il s'agit des statistiques obtenues suite à l'utilisation exclusive des prouveurs de l'Atelier B. Nous remarquons une augmentation du nombre de preuves interactives surtout à partir de la machine M3 : cela est dû à l'introduction par raffinement de la phase de **self-configuration**, ainsi qu'à l'introduction des groupes de pairs instanciant un service (MPG). Un exemple d'obligation de preuve difficile à décharger manuellement est de démontrer que ces MPGs sont finis et le demeurent, durant la phase de **self-configuration**.

Notre développement formel, ainsi que les paradigmes de *correction-par-construction* et *service-as-event*, le raffinement et les propriétés de vivacité guidant le raffinement permettent une compréhension des mécanismes de la procédure d'« auto-guérison », car les étapes nécessaires à la réalisation de cette procédure sont ajoutées et détaillées progressivement.

Le chapitre suivant les expérimentations que nous avons réalisées à l'aide de la plateforme RODIN et ses greffons, à partir de nos cas d'études et des observations que nous avons pu en retirer.

Bibliographie

[1] M. B. Andriamiarina, D. Méry, and N. K. Singh. Analysis of self-* and p2p systems using refinement. In Y. Aït-Ameur and K.-D. Schewe, editors, *ABZ*, volume 8477 of *Lecture Notes in Computer Science*,

pages 117–123. Springer, 2014.

[2] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11) :643–644, Nov. 1974.

[3] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[4] C. C. Marquezan and L. Z. Granville. *Self-* and P2P for Network Management - Design Principles and Case Studies*. Springer Briefs in Computer Science. Springer, 2012.

Quatrième partie

Analyses des études de cas et expérimentations, bilan conclusif

9

Analyses et expérimentations

Sommaire

9.1 Introduction

Ce chapitre présente les expérimentations que nous avons réalisées à l'aide de la plateforme Rodin et certains de ses greffons, notamment dans le domaine de la réutilisation de modèles et de preuves et de l'enrichissement des outils de preuves de la méthode Event-B par les prouveurs SMT. Nous nous intéressons aux résultats de nos expérimentations et de nos développements formels, que nous analysons par l'intermédiaire des bilans des obligations de preuve des développements formels réalisés durant cette thèse : nous commentons et expliquons ces bilans, notamment au niveau des obligations de preuves déchargées interactivement, pour l'analyse et la quantification de l'efficacité de la réutilisations de modèles et de preuves et celle des prouveurs SMT.

Ce chapitre est structuré comme suit. La section 9.2 se focalise sur des techniques de réutilisations de modèles et de preuves, notamment la conception et l'application de patrons de conception en Event-B, présente les outils et greffons utilisés à cette fin, ainsi qu'une discussion sur le sujet. La section 9.3 contient une analyse des bilans des obligations de preuve pour chaque cas d'étude développé durant cette thèse. Finalement, la section 9.4 présente nos conclusions pour ce chapitre.

9.2 Réutilisation de développements

Le développement [6] d'un cas d'étude, le protocole Anycast RP [12, 19, 20], nous a introduit à une problématique, qui est la réutilisation d'une solution générale à un problème commun répétitif. En effet, Anycast RP est constitué de trois phases : **(1)** une transmission d'un paquet d'une source à un premier intermédiaire, appelé Routeur Désigné (DR), **(2)** puis du Routeur Désigné à un autre intermédiaire appelé Point de Rendez-vous (RP), **(3)** puis de ce dernier aux destinataires du paquet (voir figure 9.1).

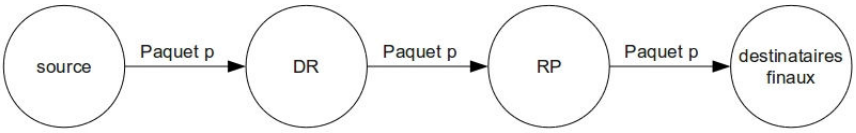


Figure 9.1 – Anycast RP : les trois phases

Nous nous retrouvons donc dans trois cas pouvant se résumer de la manière suivante : il s'agit du routage d'un paquet entre un point source (respectivement la source originale du paquet, un DR, un RP) et une ou plusieurs destinations (respectivement un DR, un RP, les destinataires finaux du paquet) (voir figure 9.2).

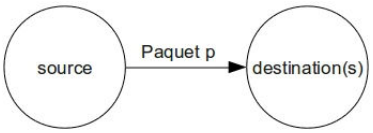


Figure 9.2 – Anycast RP : le template reproduit 3 fois

Ce développement qui est finalement basé sur la reproduction des mêmes éléments de modèles, c'est-à-dire constantes, variables, invariants, événements, raffinements modulo quelques adaptations et renommages, pour trois phases nous a emmené à nous interroger sur la notion de patron de conception dans les méthodes formelles, plus précisément en Event-B.

9.2.1 Patrons de conception

Un patron de conception [3, 11, 17, 14] peut être vu comme une solution générique à un problème récurrent (e.g. les actions/réactions dans les systèmes réactifs [2], ou encore la transmission d'un paquet). Le but d'un patron de conception est de rendre plus facile le développement d'une solution à un problème posé : par exemple, Alexander donne dans [4] des solutions générales par rapport aux problèmes rencontrés dans l'architecture des bâtiments, et dans le domaine de la programmation orientée objets, Gamma et al

[10] proposent des solutions génériques de conception et de codage. Un exemple, proche de la méthode Event-B, de l'utilisation des patrons de conception pour l'aide au développement formel par raffinement dans la méthode B [1], est l'outil BART (B Automatic Refinement Tool) [15] intégré dans la plateforme Atelier B. BART permet le raffinement automatique de modèles B, soit en générant automatiquement une implémentation pour une machine B, ou un raffinement B suffisamment détaillé, à l'aide de règles de raffinement (règles de raffinement de variables, d'opérations, etc).

Dans le cadre de Event-B, nous définissons, à la manière d'Abrial et al [3, 11], un patron de conception comme un développement Event-B formalisant des problèmes typiques récurrent : un patron de conception est donc composé d'un modèle de spécification abstrait et de modèles raffinant cette spécification abstraite. La problématique réside ici dans la réutilisation de développements formels existants (des patrons de conception) dans un développement plus large dont certains éléments correspondent aux problèmes auxquel s'adresse les patrons. Cette problématique nous pousse à compléter notre définition d'un patron de conception, rejoignant ainsi le point de vue présenté par d'Abrial et al [3, 11] : un patron de conception Event-B n'est pas seulement constitué de modèles (contextes, machines, constantes, variables, invariants, événements, raffinements, etc), mais aussi de la correction de ces derniers et des preuves de cette correction.

Illustration par l'exemple : définition d'un patron. Nous avons ainsi formalisé le problème de routage (voir figure 9.2) d'un paquet entre une source et une ou plusieurs destinations, en nous inspirant du développement du protocole Anycast RP et nous avons obtenu le patron suivant (pour les détails du développement, voir le chapitre 5, partie III) :

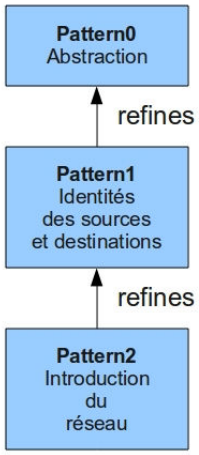


Figure 9.3 – Patron pour le routage

— Pattern0 introduit l'idée générale du routage : ce modèle présente la transmission (avec pertes et renvois possibles) d'un paquet entre la source et les destinataires du paquet. Nous ne modélisons ici que les flux (de paquets/données) entre les sources et les destinataires, nous n'identifions pas encore précisément les différents acteurs du routage.
— Pattern1 nous permet d'identifier les processus acteurs du routage : une source précise est associée à chaque paquet et chaque destinataire est maintenant clairement identifié.
— Pattern2 introduit le réseau reliant les processus entre eux. Un paquet envoyé par une source peut traverser plusieurs routeurs adjacents les uns aux autres, avant d'être reçu par son (ses) destinataire(s). Nous modélisons aussi la mobilité du réseau : les liens existant entre deux processus voisins peuvent disparaître, tandis que des liens qui n'existaient pas dans le réseau peuvent apparaître.

En résumé, nous présentons un patron de conception, finalement assez abstrait, répondant au problème du routage, plus précisément, à l'acheminement d'un paquet dans un réseau entre sa source et son (ses) destinataire(s). Nous ne modélisons pas les tables de routages, ou les techniques de mises à jour de ces dernières, qui, selon notre expérience, sont spécifiques aux types de systèmes étudiés (e.g. routage XY

pour un réseau sur puce bidimensionnel, routage XYZ pour un réseau sur puce tridimensionnel, routage utilisant des routeurs spéciaux DR et RP pour le protocole Anycast RP, etc).

Illustration par l'exemple : modélisation d'un réseau sur puce (NoC). Un patron de conception est généralement instancié dans un développement plus large, présentant le problème auquel le patron s'adresse. L'utilisation d'un patron se fait en deux étapes : **(1)** une étape d'adaptation (*matching*) durant laquelle un utilisateur fait correspondre les éléments du patron avec certains éléments du développement plus large, puis **(2)** une étape d'incorporation du patron de conception dans le développement.

Nous nous sommes intéressés durant cette thèse aux réseaux sur puce ou NoCs (voir chapitre 6, partie III) : il s'agit d'un paradigme permettant de considérer une puce comme un réseau de blocs (voir figure 9.4), intégrant chacun routeur permettant ainsi aux blocs de communiquer entre eux.

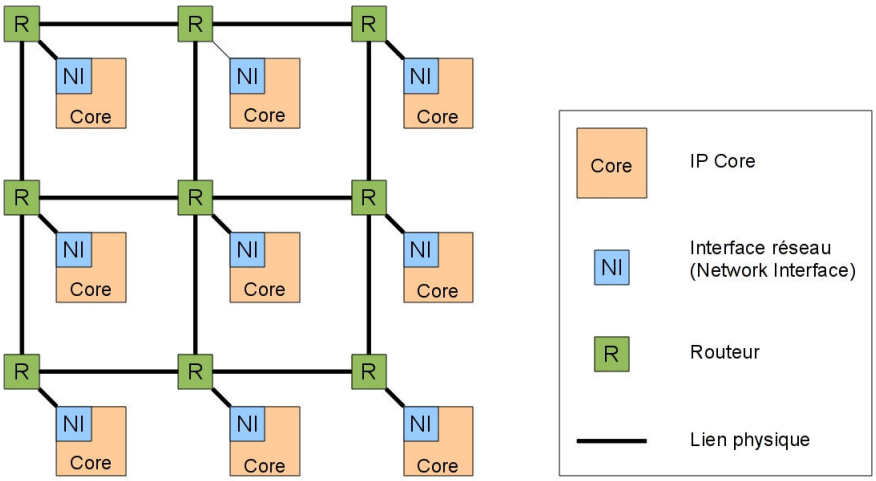


Figure 9.4 – Problème à résoudre : modélisation d'un NoC

Nous avons assimilé le paradigme du NoC à un problème de routage, d'acheminement d'un message entre deux blocs différents, permettant ainsi l'instanciation du patron de conception pour le routage défini précédemment.

Avant de détailler cette instanciation, nous présentons une comparaison entre le problème de routage général résolu par le patron et le problème du NoC, pour justifier l'application du patron de conception.

Caractéristiques	**Réseau sur puce (NoC)**	**Patron de conception**
Sources par paquet	1	1
Destinataires par paquet	1	1 ou plusieurs
Pertes de paquets	non	oui
Renvois de paquets	non	oui
Topologie du réseau	fixe et prédéfinie	pas de contraintes
Mobilité du réseau	oui, mais limitée par la topologie fixe du réseau, qui doit être préservée	oui (pas de contraintes)

Table 9.1 – Comparaisons : patron et NoC

Le tableau 9.1 permet de dire que le patron de conception sur le routage peut être instancié pour résoudre (en partie) le problème du NoC : nous pouvons constater que le NoC est une spécialisation du patron, qui est restreinte, notamment par la topologie du réseau qui est fixe et par l'absence de perte de paquets pour le NoC [5]. Nous adaptons en conséquence notre patron de conception pour prendre en compte ces aspects du NoC.

Nous détaillons maintenant l'instanciation du patron de conception pour le routage, dont la première étape consiste en une **adaptation** (*matching*) : il s'agit de comparer le modèle du NoC, à partir duquel le problème résolu par le patron a été identifié, au modèle abstrait du patron de conception, et d'y détecter les correspondances (voir figure 9.5).

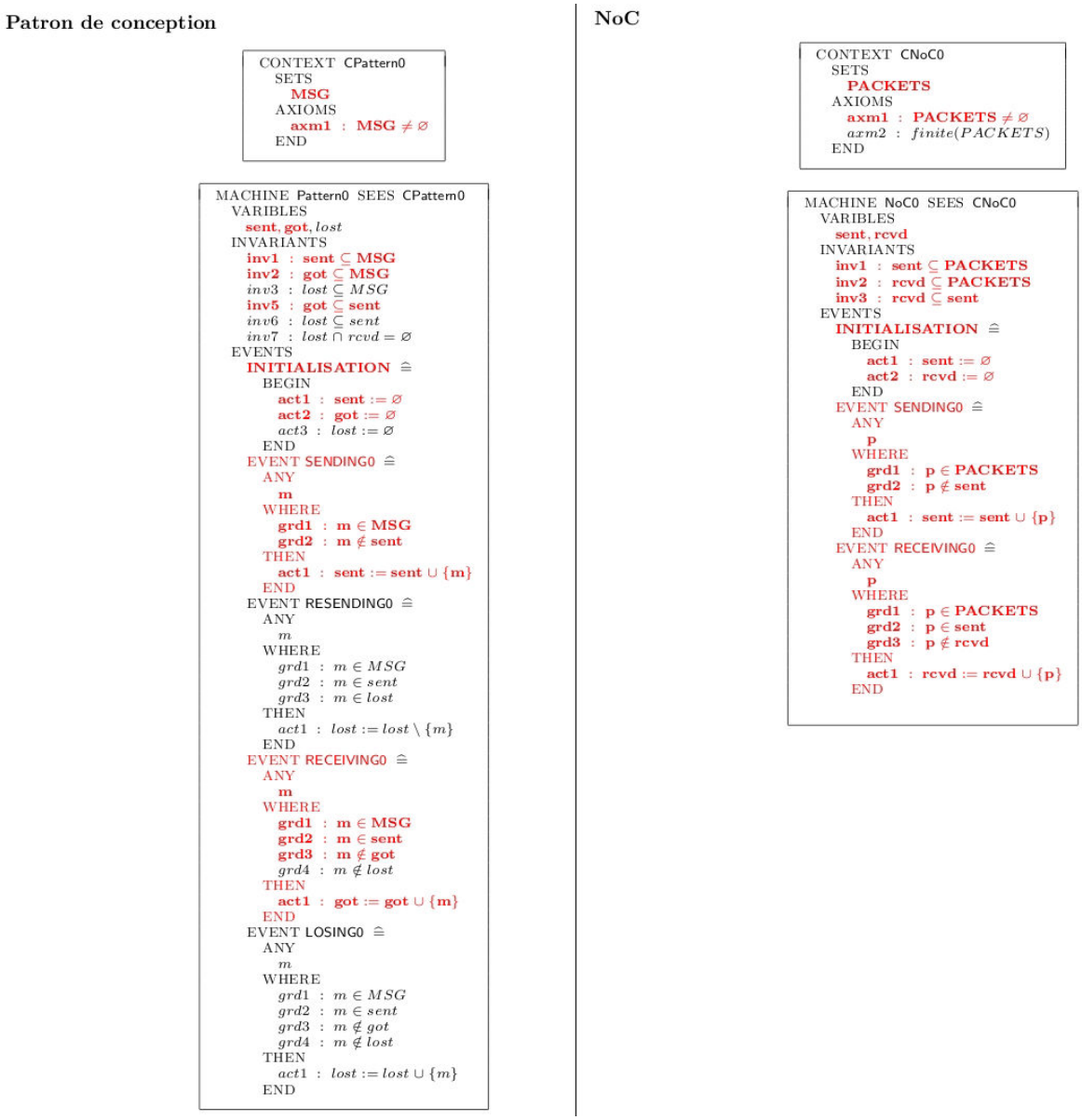


FIGURE 9.5 – Adaptation : *mapping* et correspondances

Le patron de conception pouvant être instancié à partir de la machine abstraite NoC0, les correspondances ont été établies, entre cette machine et celle Pattern0, ainsi qu'entre les contextes utilisés par ces machines, respectivement CNoC0 et CPattern0. Les correspondances entre les deux modèles abstraits sont mises en évidence par la couleur rouge, dans la figure 9.5, dont notamment :

- L'ensemble MSG du patron correspondant à $PACKETS$ du contexte CNoC0.
- Les variables *sent*, *got* du patron de conception correspondant respectivement à *sent* et *rcvd* de la machine NoC0.
- Les invariants $inv1$, $inv2$, $inv5$ du patron correspondant respectivement aux invariants $inv1$, $inv2$, $inv3$ de la machine NoC0.
- Les événements SENDING0, RECEIVING0 du patron de conception correspondant aux événements SENDING0, RECEIVING0 de la machine NoC0.

Nous passons ensuite à l'étape **d'incorporation** : il s'agit ici de réutiliser et d'intégrer dans le développement du NoC, le modèle abstrait du patron de conception, ainsi que les raffinements Pattern1 et Pattern2, en procédant à divers renommages, tels que MSG en $PACKETS$, *got* en *rcvd*, les paramètres m des événements en p, et d'y retirer toutes les références aux pertes et renvois de messages, comme les événements RESENDING0 et LOSING0, leurs raffinements, ainsi que la variable *lost* et les invariants et éléments des raffinement y faisant référence. Nous obtenons de cette façon les trois premiers niveaux de raffinement NoC0, NoC1 et NoC2 du développement formel du NoC, tel que montré par la figure 9.6.

La figure 9.6 illustre l'utilisation du patron de conception pour le routage lors du développement formel du NoC : les trois premiers niveaux de raffinement NoC0, NoC1, NoC2, ainsi que leurs propriétés, les preuves de leurs corrections ont été obtenus grâce au patron de conception. Les autres niveaux du développement de NoC3 à NoC81 et NoC82 correspondent à la prise en compte des spécificités du problème du NoC (structure des routeurs, des liens entre les routeurs, de l'architecture du réseau).

Nous notons que lors de l'utilisation d'un patron, nous pouvons être amenés à adapter (modifier) des éléments fournis par le patrons, aussi bien au niveau abstrait que dans les raffinements : par exemple, dans le développement décrit par la figure 9.6, lors de l'introduction du réseau (raffinement NoC2), nous avons eu à modifier les événements (fournis par le patron) modélisant l'évolution du réseau (ajouts/retraits de liens) pour nous conformer aux exigences induites par les particularités du NoC (réseau à structure fixe, maillé, désactivation d'un routeur et de ses voisins, réactivation/reconfiguration, etc).

9.2.2 Outils

Event-B, par l'intermédiaire de la plateforme Rodin [16], dispose des outils nécessaires à la mise en œuvre des patrons de conception. Nous citons ici deux greffons de la plateforme Rodin, nous ayant permis d'expérimenter sur les patrons de conception :

- Le greffon « Refactoring Framework » [18]. Lors de l'instanciation du patron de conception dédié au routage, pour la résolution du problème posé par le NoC, nous nous sommes aperçus, que des éléments des modèles du patron, principalement, ceux relatifs aux contextes (ensembles porteurs, constantes), devaient être renommés [11], pour correspondre à ceux proposés dans le problème à résoudre. Ce renommage est une pré-condition nécessaire à l'utilisation du greffon d'instanciation des patrons de conception [11]. Un exemple est le renommage de l'ensemble porteur MSG du patron de conception en $PACKETS$ (voir figure 9.5). Pour cela, nous avons choisi d'utiliser le greffon « Refactoring Framework » [18] : ce greffon permet d'éviter des effets indésirables du renommage d'éléments en Event-B tels que la perte de preuves, car il met aussi à jour les preuves, les conserve, et il permet en outre, en une seule opération, de propager le renommage dans tout le développement.
- Le greffon « Pattern » [11] automatise la démarche d'instanciation d'un patron de conception : il permet de choisir le patron de conception à instancier et le problème à résoudre correspondant. Il donne la possibilité au niveau des machines, d'associer des variables du patron à celles du problème, de faire correspondre des étapes d'initialisation, des événements, des paramètres d'événements, des gardes, des actions, etc. Il permet aussi le renommage des variables et des événements (et paramètres) si nécessaire. Dans notre cas, il nous a servi (voir figure 9.5) par exemple à faire correspondre les variables *got* (Pattern0) et *rcvd* (NoC0), les variables *sent* (Pattern0) et *sent* (NoC0), les événements SENDING0 et RECEIVING0 de Pattern0 et NoC0, à renommer les paramètres

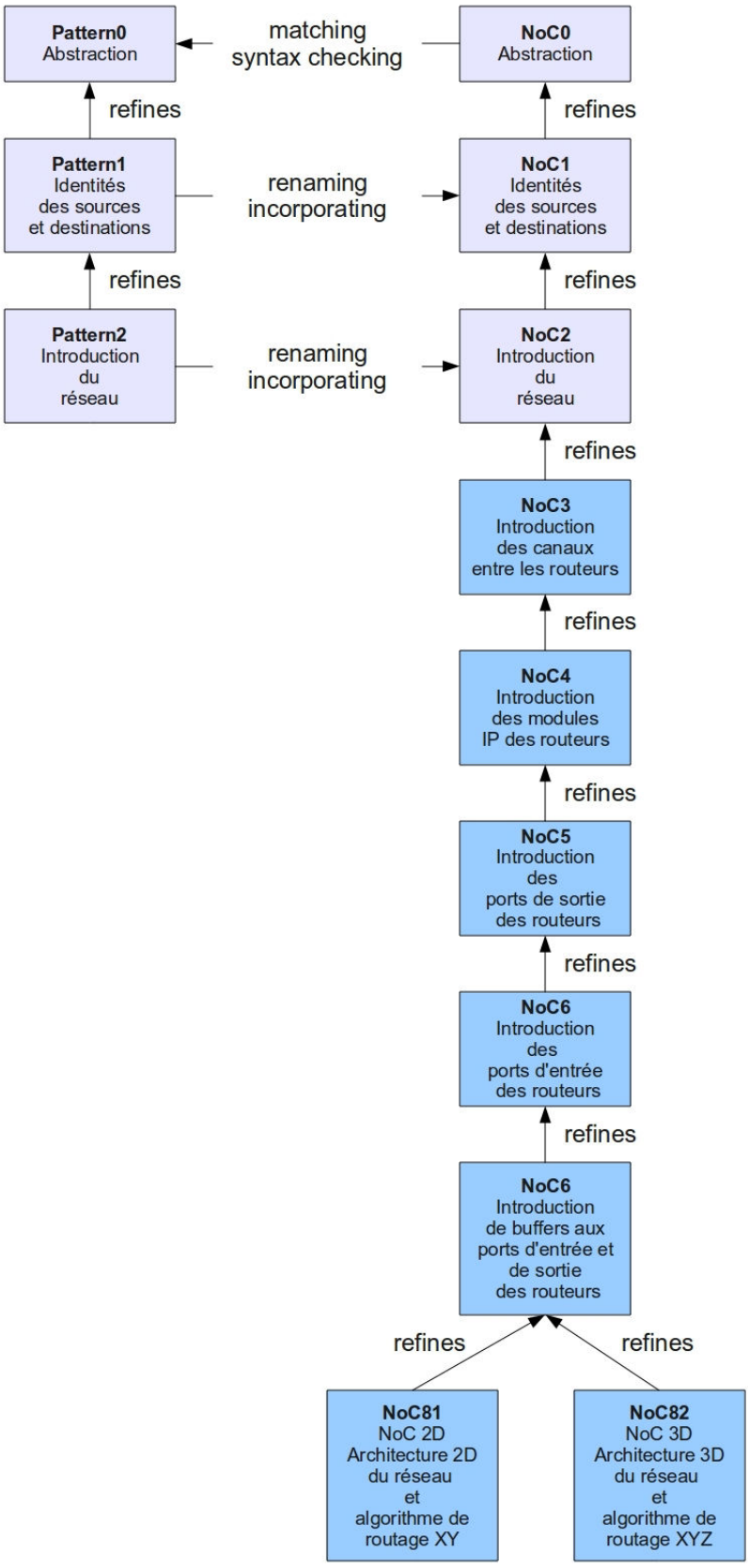


FIGURE 9.6 – Utilisation du patron pour le routage

m des événements à instancier en p et à faire correspondre les gardes et actions de ces événements avec ceux de NoC0 correspondants. Ce greffon permet aussi de d'instancier les raffinements proposés par le patron de conception, générant ainsi des automatiquement des raffinements (NoC1, NoC2) pour le problème à résoudre.

Ces deux outils nous ont permis d'éviter de réutiliser des modèles existants, et d'éviter principalement de reprouver des obligations de preuves déjà prouvées. Les obligations de preuves à décharger concernent surtout les événements du patron modifiés pour prendre en compte les spécificités du problèmes ou les éléments non fournis par le patron : lors du développement formel du NoC, les preuves à refaire pendant l'instanciation du patron de conception pour le routage concernaient surtout les événements (fournis par le patron) du modèle NoC2, modélisant l'évolution du réseau (ajouts/retraits de liens), et modifiés pour nous conformer aux exigences induites par les particularités du NoC (réseau à structure fixe, maillé, désactivation d'un routeur et de ses voisins, réactivation/reconfiguration, etc).

9.2.3 Discussions

Le greffon « Pattern » nous a permis d'instancier un patron de conception lors du développement du NoC. Il nous a donné la possibilité de réutiliser les modèles, ainsi que les preuves et la correction fournis par le patron de conception, notamment au niveau abstrait, et de générer, de manière automatique, des raffinements au développement formel du NoC. Nous donnons dans cette section, les remarques et observations que nous avons pu constater lors de l'instanciation des patrons de conception à l'aide de ce greffon.

Nous avons établi durant cette thèse des conventions de développement concernant l'utilisation des mécanismes de raffinement offerts par Event-B : l'une de celles-ci est l'enrichissement progressif, lorsque cela est nécessaire, d'un contexte par une extension de ce dernier, comme illustré par le développement du patron de conception pour le routage, présenté par la sous-figure **(a)** de la figure 9.7 ; le nouveau contexte ainsi produit est utilisé (vu - directive sees) par les raffinements ayant nécessité cette extension. L'utilisation du greffon « Pattern » nous a contraint à modifier cette convention : le greffon ne gérant pas encore l'extension [11] des contextes, nous avons dû fusionner les contextes fournis par le patron de conception du routage en un seul contexte, utilisé (vu - directive sees) par la spécification abstraite du patron et vu par transition par les autres niveaux de raffinement (voir la sous-figure **(b)** de la figure 9.7).

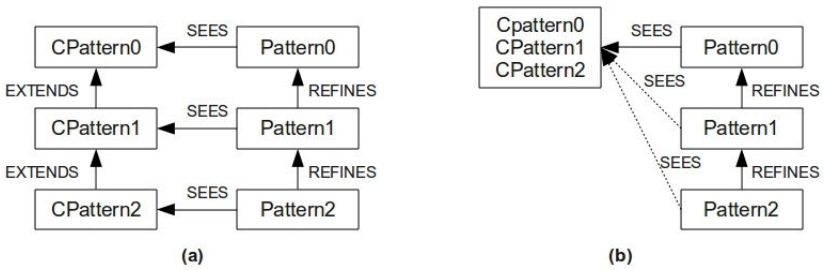


Figure 9.7 – Restrictions lors du développement d'un patron

Une possible solution pour cet aspect, forçant à ré-architecturer un développement, est de permettre, pour le greffon « Pattern », de générer à partir des contextes fournis par un patron de conception, des extensions d'un contexte d'un problème, de la même manière que le greffon génère des niveaux de raffinement aux machines du problème, à partir de ceux définis dans le patron.

Une autre observation que nous avons pu faire est que le contexte, vu par la machine modélisant le problème et correspondant à la spécification abstraite du patron, doit contenir des ensembles porteurs, des constantes ayant les mêmes noms que ceux présents dans le contexte défini par le patron, et soit contenir aussi les propriétés (axiomes et/ou théorèmes) définies dans le contexte du patron, soit permettre de déduire ces dernières (qui pourront alors être exprimés sous forme de théorèmes) (voir figure 9.8).

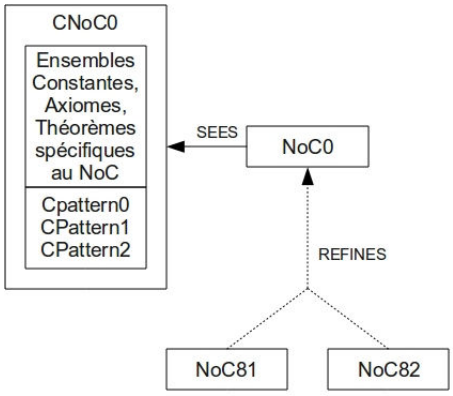


FIGURE 9.8 – Patron et problème : contextes

La principale difficulté consiste ici à renommer (pour permettre la correspondance entre le patron et le problème à résoudre) soit les ensembles et constantes du patron de conception, soit ceux du développement formel du problème, ce qui est réalisable à l'aide du greffon « Refactoring Framework ». Une possible solution est de donner au greffon « Pattern » la possibilité d'effectuer des « *matchings* » (correspondances) au niveau des contextes des patron et problème, comme réalisé au niveau des machines et de permettre, le cas échéant, de renommer des éléments de ces contextes.

Nous avons aussi remarqué que certains invariants et propriétés de sûreté, notamment ceux définis dans les raffinements du patron de conception pour le routage (lors de l'introduction du réseau), n'étaient pas recopiés dans les raffinements des modèles du NoC, générés à partir du patron. Nous avons donc dû les incorporer manuellement aux raffinements générés et décharger ainsi des obligations de preuves, opérations qui selon nous, auraient pu être évitées. Nous proposons comme solution d'intégrer toutes les propriétés de sûreté/d'invariance définies par le patron et de les corriger et/ou supprimer, si cela s'avérait nécessaire.

En résumé, nous avons pu expérimenter la réutilisation de modèles formels EVENT-B et des preuves associées. Un support à l'aide d'outils de cette technique de développement formel existe, notamment à travers le greffon « Pattern ». Des problèmes (décrits précédemment), probablement liées à la jeunesse de l'outil (en version 1.0.3), existent, mais peuvent être contournés et/ou résolus en complétant l'utilisation de « Pattern » à l'aide d'autres greffons, tels que « Refactoring Framework ». Nous tenons aussi à remarquer que lors de nos expérimentations, le greffon « Pattern » n'était pas compatible avec la dernière version de la plateforme RODIN (version 3.0). Nous avons donc mené nos expérimentations sur la version 2.7 de RODIN.

9.3 Analyse des obligations de preuves

Nous présentons aussi, dans ce chapitre, une réflexion sur la complexité du développement formel d'un problème (un algorithme réparti) à travers l'analyse des obligations de preuves engendrées pour les modèles du dit problème et déchargées soit automatiquement, soit interactivement par l'utilisateur. Notre objectif est ici d'identifier les aspects d'un développement formel d'algorithme et de système répartis pouvant produire et/ou faire augmenter le nombre d'obligations de preuves à décharger manuellement par l'utilisateur.

Nous donnons pour chacun des cas d'études, deux analyses des obligations de preuves engendrées par la plateforme RODIN, et déchargées automatiquement ou manuellement :

— La première est l'analyse des obligations de preuves, quand seulement les prouveurs de l'Atelier B sont utilisés.
— La seconde est l'analyse des obligations de preuves, quand des solveurs SMT (ici veriT [8] et CVC3 [7]) sont utilisés de concert avec les prouveurs de l'Atelier B, par l'intermédiaire du greffon « SMT Plugin » [9].

Nous définissons le calcul du gain (en pourcentages) en obligations de preuves (POs) automatiques (auto) comme suit :

$$gain = \frac{nombre\ de\ POs\ auto\ sans\ SMT - nombre\ de\ POs\ auto\ avec\ SMT}{nombre\ de\ POs\ auto\ sans\ SMT}$$

La section suivante présente l'analyse des obligations de preuves pour les études de cas abordés durant cette thèse.

9.3.1 Anycast RP

Le tableau 9.2 suivant présente les obligations de preuves engendrées lors de la modélisation du protocole de routage Anycast RP et déchargées soit automatiquement, soit interactivement.

Modèles	Total	Automatiques				Interactives			
		Sans SMT		Avec SMT		Sans SMT		Avec SMT	
C0	0	0	100%	0	100%	0	0%	0	0%
C1	1	1	100%	1	100%	0	0%	0	0%
C2	0	0	100%	0	100%	0	0%	0	0%
C3	0	0	100%	0	100%	0	0%	0	0%
C4	0	0	100%	0	100%	0	0%	0	0%
C5	3	3	100%	3	100%	0	0%	0	0%
C6	3	3	100%	3	100%	0	0%	0	0%
C7	11	11	100%	11	100%	0	0%	0	0%
C8	15	15	100%	15	100%	0	0%	0	0%
C9	2	2	100%	2	100%	0	0%	0	0%
M0	9	9	100%	9	100%	0	0%	0	0%
M1	14	14	100%	14	100%	0	0%	0	0%
M2	16	16	100%	16	100%	0	0%	0	0%
M3	100	76	76%	100	100%	24	24%	0	0%
M4	48	33	68.75%	48	100%	15	31.25%	0	0%
M5	149	110	73.83%	149	100%	39	26.17%	0	0%
M6	93	50	53.76%	93	100%	43	46.24%	0	0%
M7	14	5	35.72%	14	100%	9	64.28%	0	0%
M8	53	33	62.26%	49	92.45%	20	37.74%	4	7.55%
M9_P1	34	18	59.94%	28	82.35%	16	47.06%	6	17.65%
M9_P2	106	60	56.60%	77	72.64%	46	43.40%	29	27.36%
M9_P3	140	88	62.86%	97	69.29%	52	37.14%	43	30.71%
M9_R	40	35	87.5%	34	85%	5	12.5%	6	15%
Total	851	582	68.39%	763	89.66%	269	31.61%	88	10.34%

Table 9.2 – Anycast RP : Bilan des POs

Avant d'analyser ce tableau, nous rappelons ici que le principe général du protocole textscAnycast RP, composé de trois phases : **(1)** une transmission d'un paquet de sa source à un Routeur Désigné (DR), **(2)** une autre transmission du même paquet du DR à un routeur Point de Rendez-vous (RP), **(3)** et finalement la transmission du paquet par le RP aux destinataires finaux.

Analyse du tableau 9.2. Nous remarquons que le nombre des obligations de preuves manuelles commence à devenir assez conséquent à partir de la machine M3 : ces obligations sont dues, dans un premier temps, aux faits que nous identifions les routeurs impliqués dans le protocole, que nous définissons des propriétés sur ces routeurs (unicité d'une source, d'un DR, d'un RP récepteurs et émetteurs d'un paquet, etc) et que nous exprimons à l'aide d'invariants la séquentialité des trois phases du protocole. Les obligations de preuves manuelles introduites par les machines de M4 à M7 concernent des propriétés relatives aux particularités du protocole Anycast RP : encapsulation des paquets reçus des sources par le DR ; la troisième phase entre le RP et le destinataires caractérisée par plusieurs phases de relais de messages et de désencapsulation de ces derniers ; l'envoi de messages d'acquittement des paquets reçus et l'entrée des routeurs dans des états spéciaux lors de la réception de paquets. Les obligations de preuves engendrées pour M8, M9_P1, M9_P2 et M9_P3 sont dues notamment à l'introduction du système réparti et à la localisation de chaque phase du protocole (transformation par raffinement de données de relations en fonctions).

Nous constatons que l'utilisation combinée des solveurs SMT et des prouveurs de l'Atelier B permet un gain de 67,29% de preuves automatiques. Les gains les plus conséquents sont visibles notamment lors de l'introduction de l'identification des routeurs, du détail des phases de l'algorithme, des acquittements, du mécanisme d'encapsulation/désencapsulation des messages (passage d'un taux d'obligations de preuves

manuelles compris entre 24% et 64,28% à un taux de 0% pour les machines de M3 à M7). Les autres gains sont constatés lors de l'introduction du réseau et de étapes de localisation du protocole.

9.3.2 Patron de conception pour le routage

Nous rappelons que notre patron de conception est inspiré des trois phases identifiées dans ANYCAST RP et se décompose en deux sous-patrons :

1. un patron pour les communications *one-to-one* (unicast - une seule source et une seule destination par paquet), modélisé par les contextes et machines C00, C01, C02, M00, M01, M02.

2. un patron pour les communications *one-to-many* (multicast - une seule source et plusieurs destinations par paquet), modélisé par les contextes et machines C00, C11, C12, M00, M11, M12.

Le tableau 9.3 présente les obligations de preuves engendrées pour ce patron de conception.

Modèles	Total	Automatiques				Interactives			
		Sans SMT		Avec SMT		Sans SMT		Avec SMT	
C00	0	0	100%	0	100%	0	0%	0	0%
C01	1	1	100%	0	100%	0	0%	0	0%
C02	0	0	100%	0	100%	0	0%	0	0%
C11	1	1	100%	0	100%	0	0%	0	0%
C12	0	0	100%	0	100%	0	0%	0	0%
M00	9	9	100%	9	100%	0	0%	0	0%
M01	34	27	79.41%	34	100%	7	20.59%	0	0%
M02	19	15	78.95%	19	100%	4	21.05%	0	0%
M11	30	27	90%	30	100%	3	10%	0	0%
M12	21	17	80.95%	21	100%	4	19.05%	0	0%
Total	115	97	84.35%	115	100%	18	15.65%	0	0%

TABLE 9.3 – Patron : Bilan des POs

Analyse du tableau 9.3. Les obligations de preuves interactives constatées dans ce tableau proviennent essentiellement des aspects suivants :

— Nous identifions, dans les contextes et machines C01, M01, C11 et M11, les sources et destinataires de chaque paquet et nous exprimons des propriétés telles que les différences entre ces sources et ces destinataires, l'unicité des sources, et selon les cas, l'unicité ou la multiplicité des destinataires.
— Les machines M02 et M12 introduisent le réseau, ainsi que des propriétés sur ce réseau et les modifications possibles sur ce dernier.

Nous avons obtenu un gain de 100% de preuves automatiques suite à l'utilisation combinée des solveurs SMT et des prouveurs de l'Atelier B : toutes les preuves interactives ont pu être déchargées automatiquement.

9.3.3 Réseaux sur puce (NoC)

Une partie de ce développement reprend celui du patron de conception pour le routage : les trois premiers niveaux de raffinement (C0, C1, C2, M0, M1, M2) sont des instanciations du patron.

Modèles	Total	Automatiques				Interactives			
		Sans SMT		Avec SMT		Sans SMT		Avec SMT	
C0	0	0	100%	0	100%	0	0%	0	0%
C1	1	1	100%	1	100%	0	0%	0	0%
C2	0	0	100%	0	100%	0	0%	0	0%
C3	3	2	66.67%	3	100%	1	33.33%	0	0%
C4	8	7	87.5%	8	100%	1	12.5%	0	0%
C5	0	0	100%	0	100%	0	0%	0	0%
C6	0	0	100%	0	100%	0	0%	0	0%
C7	8	0	0%	0	0%	8	100%	8	100%
C8	23	21	91.3%	18	78.26%	2	8.7%	5	21.74%
M0	3	3	100%	3	100%	0	0%	0	0%
M1	15	12	80%	15	100%	3	20%	0	0%
M2	28	19	67.86%	28	100%	9	32.14%	0	0%
M3	53	40	75.47%	50	94.34%	13	24.53%	3	5.66%
M4	57	41	71.93%	57	100%	16	28.07%	0	0%
M5	72	36	50%	68	94.44%	36	50%	4	5.56%
M6	65	28	43.08%	57	87.70%	37	56.92%	8	12.30%
M7	45	17	37.78%	21	46.67%	28	62.22%	24	53.33%
M8	43	19	44.19%	23	53.48%	24	55.81%	20	46.52%
Total	424	245	57.78%	352	83.02%	179	42.22%	72	16.98%

TABLE 9.4 – NoC : Bilan des POs

Analyse du tableau 9.4. Les trois premiers niveaux de raffinements sont des instanciations du patron de conception pour le routage, plus spécifiquement de la partie du patron modélisant les communications de type *unicast* : nous y retrouvons les éléments apportés par ce dernier (contextes C00, C01, C02, machines M00, M01, M02), hormis la perte et le renvoi de paquets, et avec quelques modifications apportées lors de l'introduction du réseau et de ses possibles modifications. Nous constatons ainsi moins d'obligations de preuves en général pour les contextes C0, C1 et les machines M0, M1, par rapport aux contextes et machines correspondants (C00, C01, M00, M01) car les éléments ayant trait aux pertes et renvois de paquets y ont été supprimés. Par contre pour M2, nous rencontrons une augmentation du nombre d'obligations de preuves, celle-ci étant surtout dues à l'adaptation aux spécificités du NoC des éléments des modèles ayant trait à la gestion et aux modifications du réseau (désactivation d'un routeur, reconfiguration du système réparti). Les obligations de preuves engendrées pour les machines de M3 à M6 sont principalement dues au fait que nous détaillons à chaque pas de raffinement les structures des routeurs (modules IP, ports d'entrées/sorties, etc) et celles des autres composantes du réseau (canaux, etc). Nous remarquons que l'augmentation drastique du nombre d'obligations de preuves manuelles (∼60%) commence à partir de la machine M6 : nous commençons dans cette machine (et continuons dans les suivantes) à transformer des canaux, des ports d'entrées/sorties modélisés par des ensembles en des files FIFO, caractérisées par un nombre de places finies ; nous introduisons finalement dans le huitième niveau de raffinement (C8, M8) l'architecture et la topologie du réseau : il s'agit d'un réseau maillé, en deux dimensions, de forme carrée. Les obligations de preuves pour ce dernier niveau incluent donc aussi la preuve de propriétés liées à cette architecture, ainsi qu'à l'algorithme de routage (XY) utilisé pour la communication entre les routeurs.

L'utilisation des solveurs SMT combinés aux prouveurs de l'Atelier B pour décharger les obligations de preuves, nous ont permis de passer d'un taux d'obligations de preuves déchargées automatiquement de 57,78% à 83,02%, soit un gain de 59,78%. Les diminutions notables du nombre de preuves manuelles (nous arrivons à des pourcentages allant de 0 à 12,30%) se remarquent à partir de la machine M3, jusqu'à la machine M6 : les solveurs SMT ont donc été efficaces pour décharger les obligations de preuves liées aux étapes dans lesquelles nous détaillons et localisons les composants du système réparti. Les solveurs SMT ont aussi permis de réduire le nombre d'obligations de preuves manuelles relatives à l'introduction de files FIFO et la structure particulière (maillée et carrée) du système réparti.

9.3.4 Algorithmes de snapshot

Avant de détailler le bilan des obligations de preuves présenté par le tableau 9.5, nous rappelons l'architecture du développement formel du problème du snapshot :

1. Le contexte NETWORK et la machine SYSTEM modélisent le système réparti pour lequel un snapshot (photographie d'un état global du système) sera calculé.

2. La machine OBSERVATION introduit de manière abstraite le calcul du snapshot en un coup.
3. Cette machine OBSERVATION est ensuite raffinée pour obtenir différents algorithmes de snapshot : Lai et Yang (ASYNC-PROCESS, LAI-PROCESS), Chandy et Lamport (FIFO-PROCESS, LOC-FIFO-PROCESS), Morgan (SYNC-PROCESS).

Modèles	Total	Automatiques				Interactives			
		Sans SMT		Avec SMT		Sans SMT		Avec SMT	
NETWORK	6	6	100%	6	100%	0	0%	0	0%
FIFO-NETWORK	5	4	80%	3	100%	1	20%	2	0%
SYSTEM	55	50	90.91%	37	67.27%	5	9.09%	18	32.73%
OBSERVATION	41	37	90.24%	37	90.24%	4	9.76%	4	9.76%
SYNC-PROCESS	55	51	92.73%	43	78.18%	4	7.27%	12	21.82%
ASYNC-PROCESS	96	66	68.75%	63	65.625%	30	31.25%	33	34.375%
LAI-PROCESS	85	46	54.12%	40	47.06%	39	45.88%	45	52.94%
FIFO-PROCESS	229	12	5.24%	129	56.33%	217	94.76%	100	43.67%
LOC-FIFO-PROCESS	5	4	80%	4	80%	1	20%	1	20%
Total	577	276	47.83%	362	62.74%	301	51.17%	215	37.26%

Table 9.5 – Snapshot : Bilan des POs

Analyse du tableau 9.5. Nous remarquons que le nombre d'obligations de preuves manuelles augmente à partir des modèles ASYNC-PROCESS et LAI-PROCESS : cela est dû notamment au fait que nous avons eu à démontrer un certain nombre de propriétés relatives à la consistance et à la correction de la coupure associée à un snapshot, et dans une certaine mesure à la localisation du modèle de l'algorithme de Lai et Yang. Pour le modèle FIFO-PROCESS, le nombre très important d'obligations de preuves interactives est principalement dû à la transformation des canaux de communications modélisés par des ensembles en files FIFO et à l'écriture des propriétés relatives à ces files.

Nous remarquons ici que les solveurs SMT permettent de rendre automatiques, notamment, les obligations de preuves relatives à la transformation des canaux de communication (ensemble de paquets) en files FIFO (ensemble de paquets numérotés et ordonnés) et les propriétés concernant ces files FIFO. Leur utilisation, conjuguée à celle des prouveurs de l'Atelier B, nous permet de gagner un pourcentage supplémentaire de 27,18% de preuves automatiques.

9.3.5 Systèmes auto-stabilisants

Nous rappelons, avant d'analyser les obligations de preuves présentées par le tableau 9.6, que le système modélisé ici est un système de type « self-healing » : le système auquel nous nous intéressons est un système P2P, fournissant des services, lesquels peuvent tomber en panne/devenir défectueux [13] ; si un tel cas se produit, un protocole de récupération décomposé en trois phases a lieu : **(1)** une phase de self-identification durant laquelle la défaillance d'un service est détectée, **(2)** une phase de self-activation lors de laquelle la gravité de la défaillance est évaluée, **(3)** une phase de self- configuration qui permet au service défectueux (dans un état illégal) de revenir dans un état légal fonctionnel.

Analyse du tableau 9.6. Nous pouvons voir que le nombre d'obligations de preuves déchargées manuellement devient drastiquement supérieur (atteignant des pourcentages entre 56,67 et 90,85%) à celles déchargées automatiquement, de la machine M3 à la machine M20. Pour les machines abstraites M3 et M4, cette augmentation est due au fait que nous détaillons dans ces deux modèles toutes les étapes abstraites de l'algorithme : Il s'agit essentiellement de preuves d'invariants de collage reliant des états abstraits du système P2P à des états plus concrets les raffinant. Les machines de M5 à M15 introduisent les pairs instanciant les services, les groupes (ensemble) finis de pairs instanciant un même service, un pair spécial par service appelé « token owner » guidant les étapes de l'algorithme : les obligations de preuves sont ici surtout générées par des preuves sur les cardinalités des groupes d'instances, sur la finitude de ces derniers et sur des propriétés concernant le « token owner » (ce dernier est toujours disponible et fonctionnel, quelque soit l'état du service qu'il instancie). De M16 à M20, nous localisons (sur le « token owner ») chacune des étapes de l'algorithme de « self-healing » : des propriétés sur cette localisation (transformations de relations en fonctions) et sur le fait que le « token owner » d'un service propage des

Modèles	Total	Automatiques				Interactives			
		Sans SMT		Avec SMT		Sans SMT		Avec SMT	
C0	0	0	100%	0	100%	0	0%	0	0%
C1	0	0	100%	0	100%	0	0%	0	0%
C2	0	0	100%	0	100%	0	0%	0	0%
C3	0	0	100%	0	100%	0	0%	0	0%
C4	0	0	100%	0	100%	0	0%	0	0%
C5	4	4	100%	4	100%	0	0%	0	0%
C6	4	4	75%	3	100%	1	25%	0	0%
C7	4	4	75%	3	100%	1	25%	0	0%
C8	6	6	100%	6	100%	0	0%	0	0%
C9	6	6	100%	6	100%	0	0%	0	0%
M0	3	3	100%	3	100%	0	0%	0	0%
M1	21	18	85.71%	21	100%	3	14.29%	0	0%
M2	46	43	93.48%	46	100%	3	6.52%	0	0%
M3	68	24	35.30%	68	100%	44	64.70%	0	0%
M4	142	13	9.15%	142	100%	129	90.85%	0	0%
M5	46	18	39.13%	41	89.13%	28	60.87%	5	10.87%
M6	83	21	22.58%	59	71.08%	62	77.42%	24	28.92%
M7	35	12	34.29%	31	88.57%	23	65.71%	4	11.45%
M8	56	24	42.86%	55	98.21%	32	57.14%	1	1.79%
M9	60	26	43.33%	59	98.33%	34	56.67%	1	1.67%
M10	60	20	33.33%	50	83.33%	40	66.67%	10	16.67%
M11	124	31	25%	101	81.45%	93	75%	23	18.55%
M12	63	23	36.51%	51	80.95%	40	63.49%	12	19.05%
M13	33	10	30.30%	27	81.82%	23	69.70%	6	18.18%
M14	113	39	34.51%	80	70.8%	74	65.49%	33	29.2%
M15	48	10	20.83%	34	70.83%	38	79.17%	14	29.17%
M16	201	21	10.45%	36	17.91%	180	89.55%	165	82.09%
M17	69	22	31.88%	44	63.77%	47	68.12%	25	36.23%
M18	60	11	18.33%	29	48.33%	49	81.67%	31	51.67%
M19	29	6	20.69%	17	58.62%	23	79.31%	12	41.38%
M20	50	8	16%	32	64%	29	84%	18	36%
Total	1434	425	29.64%	1050	73.22%	1009	70.36%	384	26.78%

TABLE 9.6 – Systèmes auto-stabilisants : Bilan des POs

informations (état du service, nombre de pairs défectueux, etc) à toutes les instances de ce dernier sont à l'origine des obligations de preuves manuelles de ces derniers niveaux de raffinement.

L'utilisation des solveurs SMT combinés aux prouveurs de l'Atelier B pour décharger les obligations de preuves, nous ont permis de passer d'un taux d'obligations de preuves déchargées automatiquement de 29,64% à 73,22%, soit un gain de 59,52%. Nous remarquons que cette technique a été particulièrement efficace pour les obligations de preuves engendrées de M4 à M15 et de M17 à M20 : c'est-à-dire sur celles relatives à l'introduction de détails sur les étapes de l'algorithme, sur l'introduction des pairs instanciant les services, de groupes de pairs finis et dénombrables, de la notion de « token owner » par service, de la localisation de chaque étape de l'algorithme de « self-healing ». Par contre, même s'il y a une légère diminution, nous remarquons que le nombre de preuves interactives reste toujours élevé pour la machine M16 : c'est dans celle-ci que nous commençons à centraliser les modèles, ainsi que les hypothèses posées sur le système (disponibilité permanente des « token owners », de ressources pour pallier aux défaillances), sur les « token owners ». Les propriétés exprimant cette centralisation sur les « token owners » des modèles et des hypothèses sur le système sont à l'origine du nombre élevé d'obligations de preuves manuelles.

9.3.6 Synthèse

Nous constatons que le nombre d'obligations de preuves à décharger manuellement augmente lors du développement d'un algorithme réparti et de son système support, principalement, lorsque nous faisons apparaître les processus du système dans les modèles, c'est-à-dire, lorsque nous commençons à localiser : en effet, nous devons tenir compte des propriétés de chaque processus, prendre en compte les interactions de chacun d'entre eux avec leur environnement, etc. Les raffinements de données effectués lors de la localisation, tels que la transformation de relations en fonctions, engendrent aussi un certain nombre d'obligations de preuves interactives. Une autre source d'augmentation du nombre de preuves interactives est la transformation d'un ensemble en une structure de données de type file FIFO : les propriétés des files FIFO que nous exprimons dans nos modèles, le fait d'ordonner l'ensemble en la transformant en une suite, etc, engendrent un nombre conséquent de preuves manuelles.

Nous avons expérimenté, dans le cadre de la réduction du nombre d'obligations de preuves à décharger manuellement, l'utilisation de solveurs SMT pour compléter les prouveurs offerts par la plateforme Rodin : les études de cas vues durant cette thèse nous permettent d'affirmer que cette utilisation conjointe des solveurs SMT et des prouveurs procure des gains de preuves automatiques, allant d'environ 27% à 100%. Nous pouvons par conséquent dire que dans notre cas, l'utilisation des solveurs SMT aide au développement formel de systèmes et algorithmes répartis en Event-B et ces solveurs arrivent généralement à automatiser les déchargements des obligations de preuves liées à la localisation, aux raffinements de données (e.g. transformation d'un ensemble en fonction), ainsi qu'à l'introduction d'ordre dans les ensembles (transformations en files FIFO). Par contre, pour pouvoir généraliser cette affirmation, nous pensons que plus d'expériences avec les solveurs SMT doivent être menées sur plus de développements formels.

9.4 Conclusion

Nous nous sommes intéressés à des expérimentations sur des modèles Event-B, à l'aide de la plateforme Rodin et de ses greffons, dont « Pattern », « Refactoring Framework » et « SMT Plugin ». Notre objectif principal via l'utilisation de ces outils est la réduction des efforts de modélisation et du nombre des obligations de preuves, à décharger manuellement, lors de la modélisation d'algorithmes et systèmes répartis.

Notre première expérimentation a été axée sur la réutilisation d'anciens développements formels, principalement les modèles (contextes, machines et raffinements), leurs corrections et les preuves liées à ces dernières. Nous avons basé notre approche sur les *patrons de conception* en Event-B : il s'agit de développements formels (modèles, raffinement et preuves), résolvant, totalement ou en partie, des problèmes récurrents/communs, et qui sont ensuite instanciés (réutilisations des modèles, raffinements et preuves, après adaptations) dans des développements plus larges, à l'aide du greffon « Pattern » complété par le greffon « Refactoring Framework ». Nous avons ainsi notamment développé un patron de conception, inspiré d'un cas d'étude : Anycast RP, et instancié pour modéliser autre cas d'étude : le NoC (réseau sur puce).

Nous nous sommes ensuite penchés sur la réduction du nombre des obligations de preuves à décharger interactivement lors de développements formels. En effet, nous nous sommes aperçus que ce nombre augmentait de manière assez conséquente, principalement lors de l'introduction des processus composant le système support de l'algorithme réparti étudié, lors de la prise en compte des particularités de l'algorithme (utilisation de canaux FIFO, topologie particulière du système réparti, etc). La piste que nous avons explorée est l'utilisation, grâce au greffon « SMT Plugin », des solveurs SMT, pour compléter les prouveurs de l'Atelier B utilisés avec la plateforme Rodin, ce qui s'est avérée être une solution très satisfaisante, car elle permet un gain en obligations de preuves déchargées automatiquement de l'ordre de 27% à 100% et faisant ainsi baisser de manière drastique le nombre d'obligations de preuves à décharger manuellement : sur certains développement ce nombre est descendu à 0 (cas du patron de conception) ou est passé de 1009 à 384 (cas du système auto-stabilisant).

Pour résumer, nous pouvons dire que nos expériences concernant la réutilisation de développements formels et l'utilisation de solveurs SMT pour décharger les obligations de preuves engendrées par la plateforme Rodin pour des modèles d'algorithmes et systèmes répartis, se sont avérées concluantes pour notre objectif, qui est la réduction des efforts de modélisation et de preuve.

Bibliographie

[1] J.-R. Abrial. *The B-book : Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*, chapter A Mechanical Press Controller. Cambridge University Press, 2009.

[3] J.-R. Abrial and T. S. Hoang. Using design patterns in formal methods : An event-b approach. In

J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigün, editors, *ICTAC*, volume 5160 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2008.

[4] C. Alexander, S. Ishikawa, and M. Silverstein. *A pattern language : towns, buildings, construction.* Oxford University Press, 1977.

[5] M. B. Andriamiarina, H. Daoud, M. Belarbi, D. Méry, and C. Tanougast. Formal Verification of Fault Tolerant NoC-based Architecture. In *First International Workshop on Mathematics and Computer Science (IWMCS2012)*, Tiaret, Algérie, Dec. 2012.

[6] M. B. Andriamiarina, D. Méry, and N. K. Singh. Integrating proved state-based models for constructing correct distributed algorithms. In E. B. Johnsen and L. Petre, editors, *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 268–284. Springer, 2013.

[7] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[8] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. verit : an open, trustable and efficient smt-solver. In R. A. Schmidt, editor, *Proc. Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science. Springer-Verlag, 2009. To appear.

[9] D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. Smt solvers for rodin. In J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 194–207. Springer Berlin Heidelberg, 2012.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[11] T. S. Hoang, A. Furst, and J.-R. Abrial. Event-b patterns and their tool support. *Software Engineering and Formal Methods, International Conference on*, 0 :210–219, 2009.

[12] J. Kang, J. Sucec, V. Kaul, S. Samtani, and M. A. Fecko. Robust pim-sm multicasting using anycast rp in wireless ad hoc networks. In *Proceedings of the 2009 IEEE international conference on Communications*, ICC'09, pages 5139–5144, Piscataway, NJ, USA, 2009. IEEE Press.

[13] C. C. Marquezan and L. Z. Granville. *Self-* and P2P for Network Management - Design Principles and Case Studies.* Springer Briefs in Computer Science. Springer, 2012.

[14] D. Méry. Refinement-based guidelines for algorithmic systems. *International Journal of Software and Informatics*, 3(2-3) :197–239, June/September 2009.

[15] Project RIMEL and Clearsy. Projet BART. `http://www.methode-b.com/recherche-developpement/bart/`.

[16] Project RODIN. Rigorous open development environment for complex systems. http ://www.eventb.org/, 2004-2010.

[17] J. Rehm. *Gestion du temps par le raffinement.* Thèse, Université Henri Poincaré - Nancy I, Dec. 2009.

[18] R. Silva. Refactoring Framework. `http://rodin-b-sharp.sourceforge.net/`, 2009.

[19] C. Systems. Anycast RP. `http://www.cisco.com/en/US/docs/ios/solutions_docs/ip_multicast/White_papers`.

[20] C. Systems. Anycast RP using PIM. `http://tools.ietf.org/html/draft-ietf-pim-anycast-rp-07`.

10

Conclusion

Nous nous sommes intéressés durant cette thèse à la modélisation et à la vérification des algorithmes répartis, en utilisant une méthodologie fondée sur le paradigme de *correction-par-construction* [15] et le raffinement de modèles [1]. La méthodologie que nous proposons est appelée *service-as-event* [7, 8, 5, 6, 3] et est basée sur la logique temporelle TLA [14] et la méthode Event-B [1]. Elle consiste dans un premier temps à caractériser un algorithme réparti $\mathcal{A}$ par une liste des services (fonctionnalités) s_i (avec $i \in \mathbb{N}$) qu'il fournit, soit :

$$\mathcal{A} \ \widehat{=} \ \{s_0, s_1, s_2, \ldots, s_n\}$$

Chacun des services est ensuite exprimé par un ensemble de propriétés de vivacité de type *leadsto* ($\rightsquigarrow$), e.g. :

$$\begin{array}{ll} s_0 \ \widehat{=} & \{P_0 \rightsquigarrow Q_0\} \\ s_1 \ \widehat{=} & \{P_1 \rightsquigarrow R_1, R_1 \rightsquigarrow Q_1\} \\ \ldots & \\ s_n \ \widehat{=} & \{P_n \rightsquigarrow R_n, R_n \rightsquigarrow S_n, \ldots, X_n \rightsquigarrow Q_n\} \end{array}$$

Nous rassemblons ces propriétés de vivacité pour caractériser l'algorithme réparti $\mathcal{A}$ étudié, par un ensemble Φ_0 de propriétés de vivacité :

$$\Phi_0 \ \widehat{=} \ \left\{ \begin{array}{l} P_0 \rightsquigarrow Q_0, \\ P_1 \rightsquigarrow R_1, R_1 \rightsquigarrow Q_1, \\ \ldots, \\ P_n \rightsquigarrow R_n, R_n \rightsquigarrow S_n, \ldots, Z_n \rightsquigarrow Q_n \end{array} \right\}$$

Cet ensemble Φ_0 nous permet ensuite de construire une machine Event-B $\mathsf{M0}$ (avec des variables x et un invariant I), modélisant l'algorithme $\mathcal{A}$: chaque propriété de vivacité de type $P \rightsquigarrow Q$ est modélisée par un événement e_j ($j \in \mathbb{N}$), où P exprime la condition d'observation de l'événement e_j et Q, la condition satisfaite par l'observation de e_j. Nous notons qu'une hypothèse d'équité est posée pour chaque événement e_j, elle est soit faible ($WF_x(\mathsf{e}_j)$), soit forte ($SF_x(\mathsf{e}_j)$). Cela nous conduit à une définition de la machine $\mathsf{M0}$ selon la logique TLA, c'est-à-dire, que nous définissons pour $\mathsf{M0}$, une spécification TLA $\mathcal{S}pec(\mathsf{M0})$, telle que :

(1) $\mathcal{S}pec(\mathsf{M0}) \ \widehat{=} \ \mathsf{INIT_0}(x) \wedge [\mathsf{NEXT_0}]_x \wedge L_0$, où $\mathsf{INIT_0}(x)$ est le prédicat définissant les états initiaux des variables x de $\mathsf{M0}$, $[\mathsf{NEXT_0}]_x \ \widehat{=} \ (\bigvee_{j=0}^{j=k} BA(\mathsf{e}_j)(x, x')) \vee (x = x')$ est la relation *avant-après* (transitions entre les états de $\mathsf{M0}$, définies par $k+1$ événements e_j et *skip*) et L_0, la conjonction des hypothèses d'équité posée sur les événements e_j.

(2) L_0 est choisie de telle manière que pour toutes les propriétés $P \rightsquigarrow Q$ de Φ_0, nous avons $\mathcal{S}pec(\mathsf{M0}) \vdash (P \rightsquigarrow Q)$.

Nous voyons déjà apparaître quelques une des principales difficultés de notre méthode, notamment dans les choix d'une abstraction $\mathsf{M0}$ et des propriétés de vivacité, d'équité abstraites la caractérisant : il faut en effet que l'abstraction, les propriétés abstraites de vivacité, d'équité la caractérisant soient les plus simples possibles.

Le diagramme suivant explique notre méthodologie *service-as-event*, ainsi que le raffinement du modèle $\mathsf{M_0}$ par un modèle $\mathsf{M_1}$:

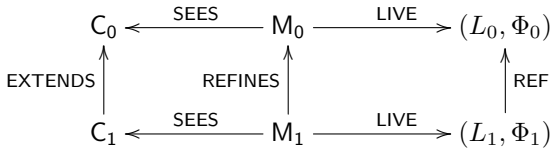


Figure 10.1 – *Service-as-event* et raffinement de modèle

La relation LIVE est expliquée par les conditions **(1)** et **(2)** vues précédemment et nous pouvons expliquer la relation REF par les conditions suivantes :

— Pour tout P, Q, tels que $(P \rightsquigarrow Q) \in \Phi_1$, nous avons $\mathcal{S}pec(\mathsf{M_1}) \vdash (P \rightsquigarrow Q)$.

— $\mathsf{M_1}$ raffine $\mathsf{M_0}$ (au sens Event-B).

— Pour tout P, Q, tels que $(P \rightsquigarrow Q) \in \Phi_0$, nous avons $(\mathcal{S}pec(\mathsf{M}_1), \Phi_1) \vdash (P \rightsquigarrow Q)$.

La relation REF exprime un processus similaire au raffinement, dans lequel des propriétés de Φ_0 sont détaillées à l'aide des règles d'inférence relatives à l'opérateur *leadsto* ($\rightsquigarrow$). Il s'agit d'une relation basée sur la déduction des propriétés de vivacité de Φ_0 à partir de celles de Φ_1 : toute propriété de vivacité $P \rightsquigarrow Q$ de Φ_0 peut être dérivée de Φ_1 et de la machine M_1 (avec des variables y, un prédicat d'initialisation $\mathsf{INIT}_1(y)$, un invariant de collage $J(x, y)$). Les événements de M_1 sont définis par rapport aux propriétés de vivacité de Φ_1. Il s'agit ensuite de déterminer une hypothèse d'équité L_1 pour la machine M_1, de telle manière que les propriétés de Φ_1 soient satisfaites et que nous puissions dériver toutes les propriétés de vivacité $P \rightsquigarrow Q$ de Φ_0, à partir de M_1 et Φ_1.

Notre méthodologie permet la préservation de deux types de propriétés, lors du raffinement de M_0 en M_1 :

— les propriétés de sûreté : le raffinement Event-B permet la préservation de ces propriétés.

— les propriétés de vivacité : les relations REF et REFINES garantissent que les propriétés de vivacité caractérisant un niveau abstrait sont dérivables à partir d'un niveau plus concret.

Par rapport à d'autres méthodes de raffinement plus classiques (au sens TLA) [14, 10] de spécifications, nous ne garantissons pas l'inclusion des traces équitables engendrées par le modèle concret M1 dans celles engendrées par le modèle abstrait M1 : pour cela, il faudrait démontrer $L_1 \Rightarrow L_0$. La figure ci-dessous illustre cette différence :

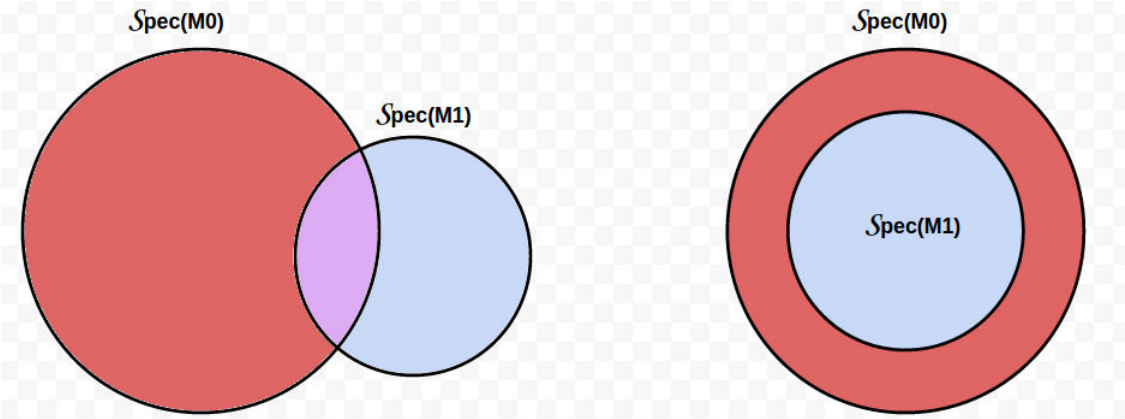


Figure 10.2 – Différences entre méthodes de raffinement

A gauche, nous avons une représentation graphique des possibles interactions entre les traces équitables engendrées par M0 et M1 dans notre méthodologie, et à droite, nous avons l'inclusion des traces engendrées par M1 dans celles de M0.

Notre méthodologie se base aussi sur l'utilisation de diagrammes d'assertions [5], inspirés des diagrammes de prédicats [10], de preuves [17] et des treillis de preuves [20], pour la représentation graphique des propriétés de vivacité, en tenant compte des hypothèses d'équité. Ces diagrammes permettent aux utilisateurs de comprendre les développements formels, ainsi que les mécanismes et fonctionnements des algorithmes répartis étudiés. Ils ont également pour vocation à expliquer clairement les phases d'un algorithme réparti aux utilisateurs et à guider ces dernier lors des différentes étapes du développement par raffinement. Mais les rôles de ces diagrammes ne se limitent pas seulement à expliquer les mécanismes et phases des algorithmes répartis étudiés et leur modélisation ; ils permettent aussi (principalement) de justifier et de comprendre les hypothèses d'équité que nous posons sur les modèles de ces algorithmes. Notre méthodologie permet en effet une séparation claire des événements d'un modèle d'un algorithme réparti : nous distinguons clairement ceux ayant trait à des étapes/phases algorithmiques et ceux relatifs aux interactions entre les acteurs de l'algorithme (les processus) et leur environnement. Les diagrammes d'assertions représentent les étapes d'un algorithme et les possibles interactions entre ces étapes et l'environnement de l'algorithme ; et selon les possibles effets de ces interactions (constatés sur les diagrammes), nous posons, sur les événements modélisant les étapes, des hypothèses d'équité faible ou forte :

— si les interactions ne perturbent pas des phases/étapes algorithmiques, nous ne nous intéressons dans ce cas qu'à la progression du système : nous posons sur les événements modélisant ces étapes, des hypothèses d'équité faible.
— sinon, si les interactions induisent des perturbations de phases/étapes algorithmiques, nous posons sur les événements modélisant ces étapes, des hypothèses d'équité forte, pour assurer ainsi la progression du système étudié.

Nous avons pu appliquer notre méthodologie pour l'analyse différents algorithmes répartis :

— un algorithme de routage : Anycast RP [6], proposé et mis au point par CISCO Systems. Il s'agit d'un algorithme complexe, dont le développement complet a nécessité une douzaine d'étapes de raffinement, de l'abstraction à la version locale de l'algorithme. Nous avons pu en outre extraire du développement formel de cet algorithme un patron de conception pour le routage.
— les réseaux-sur-puces (NoC) [3] : nos travaux sur Anycast RP nous ont permis d'appliquer le patron de conception [12] pour le routage, extrait des modèles du protocole Anycast RP, pour le développement des réseaux-sur-puces. Nous avons pu réutiliser ainsi les modèles, ainsi que les preuves de corrections fournis par le patron de conception pour le routage.
— des algorithmes auto-stabilisants : nous avons dégagé une expression générale d'un système auto-stabilisant, qui s'appuie sur la propriété de vivacité $(\neg P) \rightsquigarrow P$, où, à partir d'un état illégal du système décrit par $(\neg P)$, et exprimant qu'une propriété P d'un système n'est plus satisfaite, nous arrivons fatalement à un état légal du système satisfaisant P. Nous l'avons appliquée, principalement, à la modélisation et à la vérification d'un algorithme d'auto-guérison (self-healing) [7].
— des algorithmes de snapshot [5, 8] : par rapport aux algorithmes précédents, nous nous sommes intéressés ici à un type d'algorithme réparti particulier, c'est-à-dire, ceux fonctionnant en concurrence avec un ou plusieurs algorithmes répartis. Notre méthodologie *service-as-event* a permis notamment d'expliquer ce fonctionnement en concurrence/parallèle et de détailler aux utilisateurs les mécanismes et principes des algorithmes de snapshot ; elle nous a aussi donné la possibilité d'établir et de montrer l'existence de relations sémantiques entre les algorithmes de snapshot étudiés, grâce au raffinement de modèles.

Nous avons exploré durant ces travaux d'autres domaines, tels que l'utilisation d'outils de preuves différents (solveurs SMT) des outils traditionnels pour Event-B (prouveurs fournis par l'Atelier B), qui nous ont permis de réduire le nombre de preuves manuelles lors de nos développement. Nous nous sommes aussi particulièrement concentrés sur la réutilisation de modèles formels, de raffinement et de preuves [3] : nous nous sommes focalisés sur la détection de développements formels entiers ou en partie réutilisables, injectables et instanciables dans des développements plus larges.

En résumé, nous avons proposé une méthodologie de modélisation et de vérification par raffinement de systèmes réactifs (algorithmes répartis), guidée par des spécifications temporelles dans l'esprit des travaux de Méry et al [10, 17, 18] et de STeP de Manna et Pnueli [9, 16], et dont les principales briques sont la logique TLA [14] et la méthode Event-B [1]. Un de nos objectifs principaux, pour la poursuite de nos travaux, est l'intégration de notre méthodologie au sein de la plateforme Rodin : il s'agit d'étendre la sémantique de la méthode Event-B aux propriétés de vivacité et d'intégrer nos diagrammes d'assertions dans le processus de développement formel. Nous réfléchissons actuellement à une méthode de vérification de ces diagrammes au sein de la plateforme Rodin et à une génération de modèles, et des propriétés de vivacité les caractérisant à partir de ces diagrammes. Un autre de nos buts est aussi l'utilisation conjointe de l'outil Rodin et du prouveur « TLAPS » [11] de TLA, notamment pour la preuve de satisfaction des propriétés de vivacité par nos modèles. Nous souhaitons aussi, à partir de nos modèles Event-B concrets, être capables de générer du code source et/ou des applications répartis, en utilisant des outils tels que [19]. Notre expérience sur les patrons de conception nous pousse aussi pour nos travaux futurs à en produire davantage, à approfondir des patrons existants, e.g. sur le temps réel [21], et ainsi à enrichir un catalogue de patrons et d'algorithmes répartis (`http://rimel.loria.fr/`). Une de nos pistes est ainsi la mise en valeur des patrons dans le domaine du développement formel, à travers l'établissement d'une « banque » de patrons, le développement et l'enrichissement des outils formels [12] permettant de les réutiliser, c'est-à-dire, les modèles formels qu'ils fournissent, leurs propriétés et les preuves de ces propriétés, de manière automatique.

Bibliographie

[1] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, 2010.

[2] M. B. Andriamiarina. Stepwise development of distributed algorithms. In *FM 2011 : Doctoral Symposium*, 2011.

[3] M. B. Andriamiarina, H. Daoud, M. Belarbi, D. Méry, and C. Tanougast. Formal Verification of Fault Tolerant NoC-based Architecture. In *First International Workshop on Mathematics and Computer Science (IWMCS2012)*, Tiaret, Algérie, Dec. 2012.

[4] M. B. Andriamiarina and D. Méry. Stepwise development of distributed vertex colouring algorithms (abstract). In *2011 Grande Region Security and Reliability Day*, 2011.

[5] M. B. Andriamiarina, D. Méry, and N. K. Singh. Revisiting snapshot algorithms by refinement-based techniques. In H. Shen, Y. Sang, Y. Li, D. Qian, and A. Y. Zomaya, editors, *13th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2012, Beijing, China, December 14-16, 2012*, pages 343–349. IEEE, 2012.

[6] M. B. Andriamiarina, D. Méry, and N. K. Singh. Integrating proved state-based models for constructing correct distributed algorithms. In Johnsen and Petre [13], pages 268–284.

[7] M. B. Andriamiarina, D. Méry, and N. K. Singh. Analysis of self-* and p2p systems using refinement. In Y. Aït-Ameur and K.-D. Schewe, editors, *ABZ*, volume 8477 of *Lecture Notes in Computer Science*, pages 117–123. Springer, 2014.

[8] M. B. Andriamiarina, D. Méry, and N. K. Singh. Revisiting Snapshot Algorithms by Refinement-based Techniques (Extended Version). *Computer Science and Information Systems*, 11(1) :251–270, Jan. 2014.

[9] N. Bjørner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H. Sipma, and T. E. Uribe. Verifying temporal properties of reactive systems : A step tutorial. *Formal Methods in System Design*, 16(3) :227–270, 2000.

[10] D. Cansell, D. Méry, and S. Merz. Diagram refinements for the design of reactive systems. *J. UCS*, 7(2) :159–174, 2001.

[11] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the tla+ proof system. In J. Giesl and R. Hähnle, editors, *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 142–148. Springer Berlin Heidelberg, 2010.

[12] T. S. Hoang, A. Furst, and J.-R. Abrial. Event-b patterns and their tool support. *Software Engineering and Formal Methods, International Conference on*, 0 :210–219, 2009.

[13] E. B. Johnsen and L. Petre, editors. *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, volume 7940 of *Lecture Notes in Computer Science*. Springer, 2013.

[14] L. Lamport. A temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3) :872–923, May 1994.

[15] G. T. Leavens, J.-R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In *Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006)*, pages 221–235. ACM, Oct. 2006.

[16] Z. Manna and A. Pnueli. Temporal verification diagrams. In M. Hagiya and J. C. Mitchell, editors, *TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 726–765. Springer, 1994.

[17] D. Méry. Refinement-based guidelines for algorithmic systems. *International Journal of Software and Informatics*, 3(2-3) :197–239, June/September 2009.

[18] D. Méry and M. Poppleton. Formal modelling and verification of population protocols. In Johnsen and Petre [13], pages 208–222.

[19] D. Méry and N. K. Singh. Automatic code generation from event-b models. In *Proceedings of the Second Symposium on Information and Communication Technology*, SoICT '11, pages 179–188, New York, NY, USA, 2011. ACM.

[20] S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3) :455–495, 1982.

[21] J. Rehm. Proved development of the real-time properties of the ieee 1394 root contention protocol with the event b method. *International Journal on Software Tools for Technology Transfer (STTT)*, 2009.