Alloy4PV : un Framework pour la Vérification de Procédés Métiers

Yoann Laurent

Thèse de Doctorat
de l'Université Pierre & Marie Curie

Spécialité :
Sciences pour l'ingénieur

Présentée par
M. Laurent Yoann

pour obtenir le grade de
Docteur de l'Université Pierre et Marie Curie

Sujet de la thèse :
Alloy4PV : Un Framework pour la Vérification de Procédés Métiers

Nom	Qualité	Titre
Reda Bendraou	Encadrant	Maître de conférences, Université Paris VI
Souheib Baarir	Encadrant	Maître de conférences, Université Paris X
Marie-Pierre Gervais	Directrice	Professeur, Université Paris X
Mireille Blay-Fornarino	Rapporteur	Professeur, Université de Nice
Kais Klai	Rapporteur	Maître de conférences, HDR, Université Paris XIII
Pierre-Alain Muller	Examinateur	Professeur, Université de Haute-Alsace
Bernard Coulette	Examinateur	Professeur, Université de Toulouse II
Fabrice Kordon	Examinateur	Professeur, Université Paris VI

I hated every minute of training, but I said, "Don't quit. Suffer now and live the rest of your life as a champion."

MUHAMMAD ALI

Remerciements

Je souhaite tout d'abord adresser ma profonde gratitude envers mes encadrants, M. Reda Bendraou, maître de conferences à l'Université Paris VI, pour l'appui constant qu'il m'a apporté, l'autonommie et l'apprentissage qu'il m'a permis. Je le remercie pour la disponibilité, la patience, et l'ouverture d'esprit dont il a fait preuve, ses qualités humaines et son sens de l'écoute, ainsi que pour le support financier qu'il m'a accordé durant la réalisation de cette recherche. Je remercie également chaleureusement M. Souheib Baarir, maître de conferences à l'Université Paris X et recemment papa, pour sa rigueur intellectuelle et scientifique, sa convivialité, la qualité des informations qu'il m'a transmises et son expertise sur toute la partie formelle de mes travaux. Enfin, je remercie Mme. Marie-Pierre Gervais, Professeure à l'Université Paris X et directrice de ma thèse pour sa disponibilité et sa supervision tout au long de ce travail. Je tiens à tous vous remercier sincèrement de m'avoir offert la chance de travailler dans ces conditions.

Je tiens a remercier aussi chaleuseusement Véronique Varenne pour son aide et sa patience dans toutes les différentes et nombreuses tâches administratives.

Je voudrais remercier Mme Mireille Blay-Fornarino, Professeure à l'Université de Nice, et M. Kais Klai, maître de conferences à l'Université Paris 13, pour le temps qu'ils ont accordé à la relecture de ce manuscrit. De la même manière, merci à M. Pierre-Alain Muller, Professeur à l'Université de Haute-Alsace, et M. Bernard Coulette, Professeur à l'Université de Toulouse II Le Mirail pour avoir tous les deux accepté d'être les examinateurs de ma thèse. Je tiens aussi à remercier M. Fabrice Kordon, Professeur à l'Université Paris VI, et responsable de l'équipe MoVe du LIP6, pour me faire l'honneur de participer au jury de ma soutenance de thèse.

Merci à Vincent Marchal et Michel Knoertzer, étudiants en Master 2 à l'Université Paris VI pour avoir grandement contribué au developpement des outils présents dans cette thèse.

Merci à tous les membres de l'équipe MoVe du LIP6. Merci à Denis Poitrenaud, Cedric Besse, Tewfik Ziadi, Yann Thierry-Mieg, Jean-Luc Mounier et ceux que j'oublie pour leur gentillesse depuis le début. Merci à Jacques Robin pour tous les encouragements sur les derniers kilomètres de la thèse.

Je remercie tous les anciens deja partis, Marcos Almeida Da Silva pour son rôle de modèle et tous ses conseils, Yann Ben Maissa pour toutes les petites conversations, Ibrahima Fall pour sa gentillesse et sa convivialité, Laurent Woutters, sans qui une formule sur la fin aurait sans doute été fausse ; spécial dédicace à Dr. Selma Kchir pour tout son soutien.

Merci à Etienne Renault et Florian David, les anciens du M2, pour toutes les petites discussions durant ces trois ans, Djamel-Eddine Khelladi pour les ouvertures et fermetures du bureau chaque jour, sa bonne humeur constante et ses nombreuses contributions sur le developement des outils présents dans cette thèse lors de son stage ; Loïc Girault, l'étoile montante du LIP6, aka le solutionneur, l'homme à qui s'adresser quand on a une question, Regina Hebig pour tout le travail accompli sur mon accent anglais et surtout, ma capacité à le comprendre ; Fahad Golra le sage, pour son aide, ses conseils, sa bonne humeur, *"ça va"*, *"on va voir"*, et ses techniques couch surfing.

Merci à ma famille, à Brico, à Tantine, à Jason Statham, à Marcellus, et plus spécifiquement à ma mère pour ses nombreuses relectures et ses talents de correctrice hors pair.

Resumé

En capturant l'ordonnancement des activités et le flux d'informations des données requises et produites, les modèles de procédés décrivent précisément les interactions entre les différents participants du procédé. Cependant, comme toute activité intellectuelle impliquant l'homme, un modélisateur peut introduire certaines erreurs ou incohérences lors de la modélisation. La modélisation de procédés est une tâche complexe. Un modèle de procédé peut contenir un flux de contrôle complexe impliquant boucles, synchronisation, parallélisme, contraintes temporelles et dépendances entre les données et les ressources. Pour ces raisons, et comme cela a été souligné par de récentes études, les modélisateurs tendent à faire beaucoup d'erreurs. La vérification de procédé avant sa mise en application est donc une étape critique afin d'assurer le bon fonctionnement de l'entreprise ainsi que la qualité des produits et services délivrés.

Afin de vérifier un modèle de procédé, l'approche la plus communément utilisée consiste à recourir à des techniques de *model-checking* : une technique de vérification automatique et exhaustive de systèmes réactifs. Bien que de nombreuses approches aient été proposées par la communauté scientifique pour vérifier des procédés, aucune approche n'a réussi à s'imposer. Une des principales raisons provient du fait que la littérature s'adresse principalement à la perspective du flux de contrôle. Certaines approches proposent de vérifier une partie de la perspective des données et d'autres quelques contraintes temporelles. Cependant, aucune d'entre elles ne propose de vérifier toutes les perspectives des procédés de façon unifiée. En conséquence, une grande gamme de propriétés impliquant plusieurs perspectives n'est pas exprimable en utilisant les approches existantes. De plus, la complexité des outils proposés, leur nature pas toujours complètement automatique et intégrée dans un environnement de modélisation et d'exécution, ainsi que l'impossibilité, pour un simple modélisateur, d'exprimer des propriétés métiers, a empêché leur adoption.

Dans cette thèse, nous avons tout d'abord fait une étude de l'état de l'art dans les différents domaines des procédés (métier, logiciel, militaire, médical, etc) afin d'identifier et de catégoriser les principales propriétés à garantir. À partir de cette étude, nous avons défini une bibliothèque de propriétés génériques et paramétrables pour tout modèle de procédé. Ensuite, nous avons défini un framework pour la vérification de procédés appelé Alloy4PV. Il utilise un sous-ensemble des diagrammes d'activités UML 2 comme langage de modélisation. Afin d'effectuer la vérification de procédés, nous avons (1) défini un modèle formel des diagrammes d'activités basé sur la sémantique fUML (le standard de l'OMG donnant une sémantique à un sous-ensemble de UML) en utilisant la logique de premier ordre, (2) implémenté cette formalisation en utilisant le langage Alloy afin d'effectuer du *model-checking* borné, et (3) automatisé, dans un outil graphique intégré à Eclipse, la possibilité d'exprimer et de vérifier des propriétés sur toutes les perspectives du procédé.

Abstract

By capturing the ordering of activities and the flow of required and produced artifacts, process models describe precisely all the interactions amongst the different process stakeholders. However, like any activity involving intellectual human tasks, a modeler can introduce errors or inconsistencies during the modeling phase. Indeed, a process model can contain a sophisticated control-flow involving loops, synchronization, parallelism, timing constraints, and dependencies between data and resources. Consequently, and as highlighted by some recent studies, process modelers tend to do a lot of mistakes. The verification of the process model before its deployment in an effective production context becomes more than critical in order to ensure a good quality of the company's delivered products and services.

In order to verify a process model, the most common approach consists in using model-checking, an automatic and exhaustive technique to verify reactive systems. Although a lot of approaches have been proposed in the literature to verify process models, no approach has succeeded to be largely adopted. One of the main reasons comes from the fact that these approaches addressed principally the control-flow perspective. Some approaches propose to verify a part of the data perspective, and others offer the possibility to check some temporal constraints. However, none of them verify all the perspectives of process models in a unified way. Consequently, a wide range of properties implying multiple perspectives is not expressible using these approaches. Moreover, the complexity of the associated tools, their lack of a fully automatic support of the entire verification chain, the fact that they are not integrated in a process environment and the impossibility for the average process modeler to express business properties prevent their adoption.

In this thesis, we realized a study of the start-of-the-art on different process domains (business, software, military, medical, etc.). The aim was to identify and categorize critical properties that can be verified on any process model. This study resulted in a library of generic and configurable properties. As a second step, we have defined a framework for process verification called Alloy4PV. This framework uses a subset of UML 2 Activity Diagram as a process modeling language. For process verification, (1) we defined a formal model of UML 2 Activity Diagram based on the fUML semantics, the OMG standard that gives a semantic to a subset of UML 2. This was achieved using first-order logic, (2) we implemented this formalization using the Alloy language in order to perform bounded model-checking, and (3) we automatized in a graphical tool integrated to Eclipse, the possibility to express and verify properties on all the perspectives of a process model. This contribution resulted in a tool which is under evaluation by our MerGE project's partners and to five publications in conferences proceedings.

Table des matières

Table des figures

Chapitre 1

Introduction

Le travail réalisé lors de cette thèse se situe à la croisée de plusieurs disciplines : celle des procédés métiers, de la vérification formelle ainsi que de l'ingénierie dirigée par les modèles (IDM). Le bénéfice principal de l'IDM peut être atteint en exploitant son potentiel pour l'abstraction et l'automatisation. La vérification et la validation de modèles est l'un de ses meilleurs exemples. Cependant, l'utilisation de méthodes formelles de vérification est loin d'être couramment répandue dans la pratique du génie logiciel. L'une des raisons qui explique cette situation est que les modélisateurs n'ont pas l'expérience et l'expertise nécessaire en ce qui concerne les méthodes formelles. Une approche où le modélisateur se concentre sur son domaine de compétence, quand la vérification est effectuée sur demande, automatiquement et de façon transparente serait souhaitable. L'objectif principal de cette thèse est de proposer une approche tirant parti des avancées des méthodes formelles, et de construite au-dessus de technologies matures et standards, afin de vérifier automatiquement des modèles de procédés.

1.1 Business Process Management

Il a été largement reconnu que l'utilisation de procédés métiers au sein d'une entreprise permet de fournir des produits et des services de manière plus efficace [277]. En capturant l'ordonnancement des activités, le flux d'informations des données requises et produites, les *modèles de procédé* décrivent de manière détaillée les interactions entre les différents participants du procédé [265, 277]. La gestion de procédés métiers (BPM, pour Business Process Management) a été définie par Van der Aalst comme une façon de *"... supporter les procédés métiers en utilisant des méthodes, techniques, et logiciels pour modéliser, exécuter, contrôler, et analyser les procédés opérationnels impliquant des humains, organisations, applications, documents ou toute autre source d'informations"* [260]. Les modèles de procédés représentent donc une part essentielle du patrimoine des entreprises et constituent leur plus grand facteur de réussite.

Les systèmes BPM (BPMSs) sont des outils logiciels destinés à automatiser la mise en application de ces procédés. Lorsque l'un de ces systèmes est déployé au sein d'une entreprise, cela a un impact significatif sur la productivité de l'entreprise. La figure 1.1 montre le cycle de vie classique de la gestion des procédés métiers. Ce cycle commence par la *modélisation* du procédé, où celui-ci est identifié, examiné, validé, et finalement représenté sous forme de modèle de procédé [277]. Généralement, les modèles de procédés sont développés en utilisant un langage de modélisation (PML, pour Process Modeling Language), p. ex., Business Process Modeling Notation (BPMN) [192], Business Process Execution Language (BPEL) [133], Unified Modeling Language (UML) [190], Software Process Engineering Metamodel (SPEM) [188], etc. Ensuite, le modèle de procédé est utilisé pour *configurer* les informations au sein du système, ex., assigner

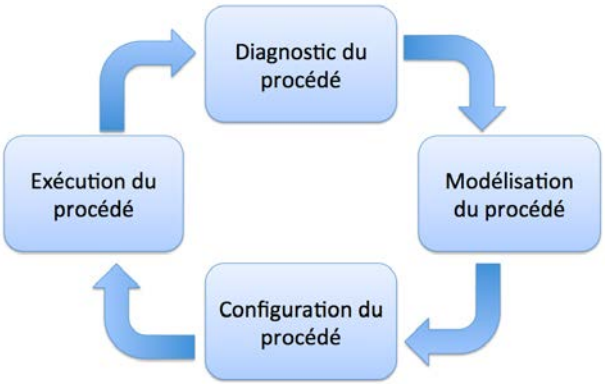


FIGURE 1.1 – Cycle de vie de la gestion des processus d'affaires [65]

les activités aux employés. Dans la phase d'*exécution*, le procédé doit être exécuté au sein de l'organisation en suivant la voie prescrite par la modélisation du procédé. La dernière phase de *diagnostic* utilise les informations résultant de l'exécution afin de l'évaluer. Ce résultat est utilisé pour fermer le cycle BPM dans le but de continuellement améliorer le procédé [36], selon le diagnostic, certaines parties du procédé peuvent être remodélisées. L'objectif principal des BPMSs est donc de supporter complètement ce cycle de vie.

1.1.1 Complexité et contraintes de modélisation

L'adoption croissante des BPMSs dans les entreprises encourage une automatisation toujours plus poussée des procédés métiers. Certains modèles de procédés peuvent atteindre plus de 250 activités [34] contenant un flux de contrôle complexe impliquant boucles, synchronisation et dépendances entre les données et ressources [103]. En effet, la figure 1.2 montre que les modèles de procédés sont régis principalement par 3 perspectives : le flux de contrôle, les ressources et les données. Sur cette figure, le modèle est un exemple simple contenant 3 activités pour *enregistrer un album* dans la base de données. La *perspective de flux de contrôle* définit l'ordre d'exécution des activités [268, 218]. La *perspective des ressources* détermine quelles ressources (humaines ou non) sont autorisées à exécuter chacune des activités ainsi que leurs allocations par rapport à celles-ci [268, 84, 219, 217]. La *perspective des données* définit quelles données sont disponibles dans le procédé et comment les utilisateurs peuvent y accéder lors de l'exécution [268, 216]. De plus, les modèles de procédés possèdent souvent une liste de contraintes spécifiques au projet ou à l'entreprise. Par exemple, certains modèles contiennent des informations concernant le temps affecté à chacune des activités ainsi que la disponibilité des ressources [69, 149]. Pour toutes ces raisons, la modélisation de procédé est une tâche complexe et est facilement sujette aux erreurs, même pour un expert dans le domaine [268].

D'autre part, le cycle de vie des modèles de procédés est loin d'être trivial. Après avoir été modélisés dans la phase de *modélisation*, ces modèles peuvent être modifiés à la volée pendant la phase d'*exécution* [208, 199, 50] afin de prévenir le besoin de futures déviations du modèle [46, 135, 49]. En effet, dans certains domaines de procédés métiers, il n'est pas toujours possible de modéliser un modèle de procédé anticipant toutes les situations possibles qui pourraient arriver pendant l'execution. Par exemple, dans le cas d'utilisation de procédé de développement *logiciel*, Lanubile et Vissagio ont montrés que 98% des agents dévient, peu importe le procédé utilisé (ex. scum, rup ...) [148]. De plus, ces modèles sont une nouvelle fois sujets à des raffinements

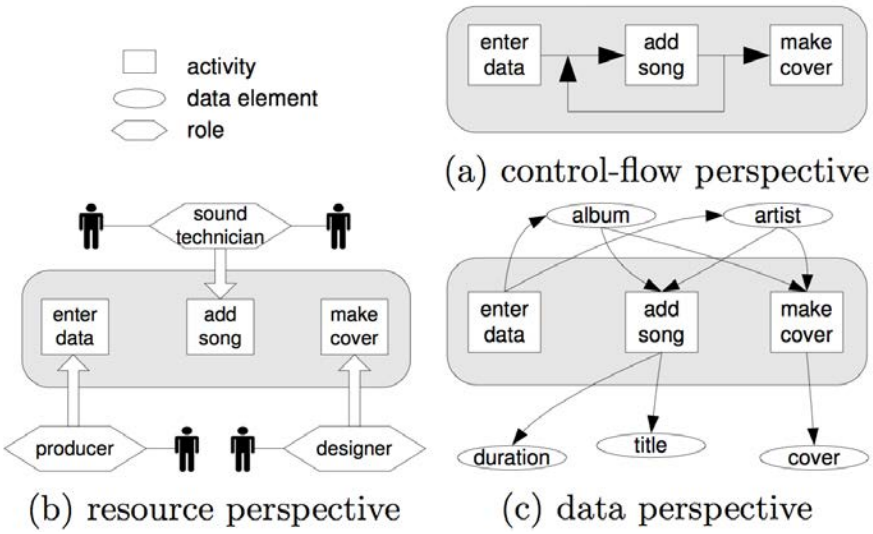


FIGURE 1.2 – Les différentes perspectives des modèles de procédés [197]

dans la phase de *diagnostic* par souci d'amélioration et de correction du modèle pour la prochaine exécution [231, 52, 36].

Il est de notoriété publique qu'il est préférable de détecter les problèmes logiciels le plus tôt possible, de préférence avant leurs mises en production [182]. Dans le cas des procédés métiers, ceci est particulièrement important du fait qu'ils impactent directement le cœur du fonctionnement de l'entreprise. Du fait que tout le cycle de vie BPM est dirigé par ces modèles de procédés, leur exactitude est cruciale. *"Le procédé est-il correct ?"*. Comme toute autre activité intellectuelle impliquant l'homme, un modélisateur peut introduire certaines erreurs ou incohérences lors de la modélisation. Une erreur de modélisation peut, en plus d'entraîner une perte de temps et d'argent, entraver le bon fonctionnement de l'entreprise et mener à un déclin dans la qualité des produits et services délivrés. De toute évidence, plus une erreur est détectée tard, plus une partie du travail à refaire sera importante.

1.1.2 Modèles erronées

Comme indiqué dans de nombreuses études, les modélisateurs de procédés tendent à faire beaucoup d'erreurs. Par exemple, l'étude [173] montre que sur plus de 2000 modèles de procédés, 10% des modèles sont imparfaits et comportent des erreurs comportementales. Les modèles SAP, un ensemble public de modèles contenant plus de 600 procédés non triviaux exprimés en terme d'EPC (Event-Driven Process Chains), a montré un taux d'erreurs de plus de 20% [172, 171]. Gruhn *et al.* [104] ont collecté 285 EPCs à partir de différentes sources (thèses, master, papiers scientifiques, thèses de doctorat...) et sont arrivés à la conclusion que plus de 38% des procédés contenaient des erreurs comportementales. Vanhatalo *et al.* [269] ont analysé le flux de contrôle de plus de 340 procédés métiers modélisés avec IBM WebSphere Business Modeler [121] pour découvrir que plus de la moitié d'entre eux n'étaient pas sûrs. Des erreurs classiques peuvent consister en des inters-blocages, des activités jamais exécutées, une donnée ou une ressource non disponible au démarrage d'une activité, un manque de temps pour effectuer les activités, etc. Bien que les procédés métiers soient utilisés depuis plus d'une vingtaine d'années [145], force est de constater que ces récentes études soulignent l'incapacité des outils de modélisation à capturer certaines de ces erreurs.

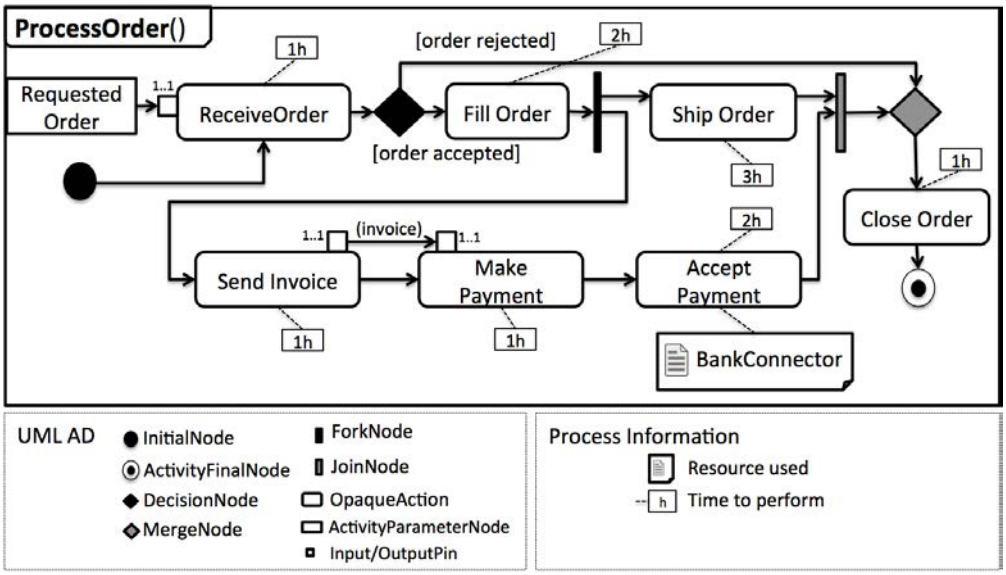


FIGURE 1.3 – Procédé `ProcessOrder` de la spécification UML [190]

1.1.3 Besoin de vérification

Idéalement, après chacune des modifications apportées aux modèles, le procédé est analysé et vérifié afin de garantir son exactitude [270, 261]. Cette vérification doit être non seulement capable de relever les erreurs classiques communes à tous les procédés tels que les inter-blocages, mais aussi de s'assurer que les contraintes relatives à l'entreprise, au projet et au domaine (p. ex. médical, militaire, entreprise, développement logiciel) [19] soient respectées et préservées pendant tout le cycle de vie du procédé. Des modèles de procédés efficaces et *fiables* font partie des points clés pour le succès des entreprises modernes. Cependant, sans outils proposant une vérification automatique et exhaustive du procédé, il est impossible pour un expert du domaine de se faire une représentation mentale de tous les chemins possibles d'exécutions et de garantir que certaines propriétés essentielles sont respectées par le modèle.

Une façon d'effectuer cette analyse consiste à recourir à des techniques de vérification formelle, telles que le *model-checking* : une technique pour la vérification automatique de systèmes réactifs introduite par Clarke *et al.* [40, 42]. L'approche consiste à définir formellement le modèle de procédé et sa sémantique, ainsi que la propriété à vérifier. Ensuite, ces deux entités sont envoyées à ce que l'on appelle un *model-checker*, qui va répondre à la question de satisfiabilité de la propriété par le modèle en explorant exhaustivement l'espace d'état. Dans le cas où la propriété n'est pas vérifiée, un contre-exemple est généré par le *model-checker*.

Bien que de nombreuses approches aient été proposées pour vérifier des procédés, aucune technique de vérification n'a réussi à s'imposer et à détrôner les autres [179]. Leurs complexités [103], l'utilisation de formalisme mathématique et l'impossibilité pour un simple modélisateur de procédés d'utiliser et de comprendre ces outils ont été parmi les obstacles majeurs à leurs adoptions [171].

Afin de définir plus précisemment les principaux problèmes que nous avons identifiés, prenons comme exemple un procédé de la spécification UML. La figure 1.3 montre ce procédé modélisé en utilisant les diagrammes d'activités UML. Le but de ce procédé est de gérer la réception d'une commande au sein d'une entreprise. Ce procédé comporte 7 actions (`ReceiveOrder`, `FillOrder`...) reliées par différents noeuds de contrôle (`DecisionNode`, `ForkNode` ...) pour représenter la perspective du flux de controle. Les données sont représentées à l'aide d'épingles (pins) attachés

aux actions (`InputPin`/`OutputPin`). Certaines informations organisationnelles sont attachées aux actions telles que les resources utilisées par les actions (ex. `BankConnector`), ainsi que le temps nécessaire pour les effectuer. Dans la suite, nous catégorisons en deux sous-ensembles les types de problèmes liés aux approaches de la littératures : (a) la formalisation utilisée et (b) leurs mises en application.

Formalisation

(a1) **- Inadéquation du formalisme.** Un des problèmes avec les approches de la littérature concerne les formalismes et outils utilisés pour effectuer la vérification. Quel que soit le PML utilisé, une sémantique est donnée au langage en associant ses constructions soit à une variante des automates [118], des réseaux de Petri [200] ou encore des algèbres de processus [16]. Cette étape est nécessaire du fait que tous les *model-checkers* supportent un formalisme qui leur est propre. Cependant, cela implique que la formalisation s'appuie sur la sémantique et les concepts du langage formel utilisé en terme d'expressivité, au lieu de se baser directement sur la sémantique du PML lui-même. Par exemple, pour vérifier la propriété "est-ce que `CloseOrder` est toujours executé ?", le modèle peut être exprimé en utilisant des réseaux de Petri classiques [200]. Cependant, la propriété "est-ce que la facture (`invoice`) est toujours produite apres l'execution de `FillOrder` ?" implique des données du système et impose l'utilisation de réseaux de Petri colorés (CPN) [131]. Bien que les réseaux de Petri (en incluant leurs différentes extensions) soient *turing-complet*, c'est-à-dire capable de représenter n'importe quel type d'algorithme, *"cela n'implique pas que l'effort de modélisation soit acceptable"* [263]. En effet, Van Der Aalst [263] montre quelques exemples de *workflow patterns* [259] modélisés avec différentes extensions de réseaux de Petri de manière complexe, quand un langage de modélisation tel que les diagrammes d'activités UML ou BPMN peut l'exprimer de façon naturelle.

(a2) **- Support limité de type de propriétés.** Concernant les types de propriétés supportés par ces approches, c'est principalement la perspective du *flux de contrôle* qui a attiré l'attention [261]. Certaines approches proposent de vérifier une partie de la *perspective des données* [14, 247, 144], de s'assurer de certaines propriétés temporelles [78, 105, 274], de vérifier certaines propriétés métiers [13, 14, 162], mais aucune approche ne propose de vérifier toutes ces propriétés de façon *unifiée*. Par exemple, la propriété "est-il possible de terminer le procédé `ProcessOrder` en moins de 8 heures, en produisant la facture (`invoice`), sans utiliser la resource `BankConnector` ?" nécessite d'être au courant de toutes les perspectives de manière combinée. Cependant, ce type de propriétés n'est pas actuellement supportée par l'état de l'art.

Mise en application

(b1) **- Difficulté d'expression des propriétés.** Un autre obstacle provient de la difficulté d'exprimer des propriétés à vérifier sur un modèle de procédé. Généralement, les modélisateurs ont tendance à écrire leurs besoins et spécifications en langage naturel. Cependant, afin d'être automatiquement vérifiée, une propriété doit être exprimée en utilisant un formalisme mathématique [76], le plus souvent en utilisant une logique temporelle telle que LTL (Linear Temporal Logic) [204] ou CTL (Computation Tree Logic) [107]. Toutefois, l'utilisation de ce type de logique demande un bagage mathématique et formel poussé [41] pas toujours compatible avec les compétences techniques d'un modélisateur lambda [12, 13, 235]. Parfois, la propriété exprimée est "presque" correcte, mais échoue à capturer des concepts importants et subtils du comportement attendu par le système [235].

(b2) - **Manque d'outillage.** Finalement, un dernier problème provient des outils actuels pour la vérification de procédés. La plupart du temps, ces outils ne sont pas directement intégrés aux BPMSs et/ou le processus de vérification n'est pas automatisé. Certaines étapes concernant la traduction vers le formalisme [253] ou l'expression de la propriété doivent être faites manuellement [57]. De plus, le résultat de la vérification, le plus souvent sous forme de contre-exemple représentant une exécution invalidant la propriété, n'est pas représenté graphiquement à l'utilisateur [279, 280, 281]. Seule une trace d'état représentant l'exécution est textuellement présentée à l'utilisateur, sans explication sur le "*pourquoi*" ce contre-exemple se produit. Par exemple, en cas de présence d'inter-blocage, un message affichant "le procédé est sujet aux inter-blocages du fait que la transition de l'action `MakePayment` vers l'action `AcceptPayment` est impossible" est infiniment plus utile qu'une simple liste d'état représentant l'exécution. En somme, les approches actuelles ne sont pas toujours complètement automatisées et se cantonnent à retourner le résultat renvoyé par le *model-checker* sans l'analyser.

1.2 Objectifs de la thèse

Cette thèse est financée par le projet européen MERgE [1] [123]. L'objectif de ce projet est de fournir des outils et solutions combinant à la fois *sûreté* et *sécurité* pour le développement de systèmes. Le rôle du LIP6 dans ce projet consiste principalement à aider au développement de systèmes par l'intermédiaire d'exécution de procédés logiciels. La *vérification* de ces procédés est donc un des points importants du projet afin de s'assurer de la *sûreté* de développement, une des thématiques principales de ce projet.

Dans cette section, nous détaillons notre choix du langage de modélisation étudié ainsi que les objectifs et étapes nécessaire afin de répondre aux principaux problèmes des approches de vérification de la littérature.

1.2.1 Choix du langage de modélisation et sémantique

Une des contraintes du projet MERgE concerne l'utilisation de diagrammes d'activités UML (AD, pour Activity Diagram) pour la modélisation de procédés. UML est un standard reposant sur différents langages graphiques pour modéliser des systèmes logiciels. Parmi ces langages, les diagrammes d'activités UML sont connus pour décrire le comportement dynamique d'un système et ont été extensivement utilisés comme PML [64, 20, 21, 75, 220]. Un des avantages de UML est que c'est un standard très largement utilisé dans l'industrie et supporté par un grand nombre d'outils commerciaux. De plus, il a été montré que AD supporte un niveau d'expression suffisant pour modéliser la plupart des *workflow patterns* [259, 220, 218].

Néanmoins, un des problèmes majeurs de la spécification de UML [190] vient du fait que la sémantique opérationnelle est imprécise et ambiguë. La sémantique est donnée seulement en *langage naturel* et dispersée dans la spécification. Ce problème a été réglé avec la récente sortie de fUML 1.0 (Semantics of a Foundational Subset for Executable UML Models) [189], un nouveau standard définissant précisément la sémantique d'exécution d'un sous-ensemble de UML. Dorénavant, la sémantique opérationnelle de UML est donnée en pseudo Java-code et n'est plus sujette aux interprétations humaines. Cependant, cette sémantique n'est pas basée sur un formalisme mathématique, ce qui empêche l'application de méthodes formelles tels que le *model-checking*. Il n'existe pour le moment aucune approche permettant de faire de la vérification de procédés modélisés avec des diagrammes d'activités UML basés sur la sémantique de fUML. En

1. http://www.merge-project.eu/

effet, ce standard étant relativement récent, la littérature est pauvre en recherche et cas d'études l'utilisant.

Il convient de noter que l'approche proposée dans cette thèse n'est pas exclusive aux diagrammes d'activités UML, mais aurait pu être adaptée pour la vérification de modèles de procédés modélisés avec BPMN [192].

1.2.2 Revue de l'état de l'art

Un des premiers objectifs consiste à faire une étude de la littérature dans les différents domaines de procédé afin d'*identifier* et de *catégoriser* les différents types de propriétés intéressantes à vérifier. Il existe principalement deux types de propriétés : *syntaxique* et *comportementale*. Le premier type permet d'imposer des contraintes structurelles, vues comme des invariants, qui ne peuvent être exprimées avec le langage de modélisation lui-même. Le second type permet de déterminer *en avance* si le modèle de procédé expose certains comportements désirables ou indésirables *pendant* l'exécution. Pour ce faire, il est nécessaire d'effectuer une exploration complète de l'espace d'états du procédé afin de garantir que la propriété soit vraie quelle que soit l'exécution.

Dans cette thèse, nous nous intéressons principalement aux propriétés comportementales. Ce choix s'est imposé du fait que les erreurs syntaxiques, bien que toujours présentes [172], peuvent être déjà vérifiées de manière efficace [55] ou imposées par l'outil de modélisation [119]. Nous catégorisons les propriétés comportementales en cinq sous-groupes :

Propriétés de flux de contrôle (soundness). Introduite dans [261], l'idée derrière le concept de *sûreté* est de détecter toute anomalie sur le flux de contrôle sans connaissance du domaine, par exemple, le fait qu'un procédé qui commence doit toujours pouvoir se terminer, ou que le procédé ne doit pas contenir d'activités qui ne peuvent être exécutées.

Propriétés de flux de données. De la même façon, ces propriétés ont pour but de détecter toute anomalie sur le flux de données, tel qu'une donnée manquante pour démarrer une activité.

Propriétés de ressources. Ces propriétés permettent de vérifier l'allocation des ressources sur le procédé. Par exemple, de répondre à des questions telles que "est-ce que les ressources seront toujours disponibles pour commencer à executer une activité ?".

Propriétés de temps. Ces propriétés permettent de vérifier les contraintes temporelles du procédé. Par exemple, de répondre à "est-il possible terminer le procédé *à temps* quelle que soit l'exécution du procédé ?".

Propriétés métiers. Elles représentent des propriétés spécifiques adaptées à un procédé donné. Elles jouent un rôle important du fait qu'un procédé peut être syntaxiquement correct et valide par rapport aux propriétés comportementales définies précédemment, mais violent des contraintes métiers. Ces propriétés correspondent donc à des questions comme "est-ce que l'activité `ImportantAction` sera exécutée, peu importe les choix faits pendant l'exécution du procédé ?", "est-ce que l'artéfact `Documentation` sera forcément disponible à la fin de l'exécution ?", etc.

1.2.3 Modèle formel basé sur UML AD et fUML

Afin de pouvoir vérifier automatiquement des procédés en utilisant des techniques de *model-checking*, il est nécessaire de représenter de façon formelle le modèle de procédé ainsi que sa sémantique. Le second objectif consiste à formaliser les diagrammes d'activités UML ainsi que leur sémantique basée sur fUML. L'intérêt principal de cette formalisation est de donner une sémantique mathématique *claire* et *non ambiguë* basée directement sur les concepts et la sémantique de UML AD plutôt que de se reposer sur un formalisme tel que les réseaux de Petri (cf. (a1)).

Afin de pouvoir raisonner sur chacune des perspectives du procédé, la formalisation couvre à la fois la perspective de *flux de contrôle* et *des données* à travers l'utilisation de la notation des diagrammes d'activités, mais prend aussi en compte les informations organisationnelles associées telles que la perspective des ressources et les contraintes temporelles. Ces informations sont essentielles pour supporter l'ensemble des propriétés identifiées lors de l'étude de la littérature. De cette façon, il est possible d'exprimer une grande gamme de propriétés sur toutes les perspectives du procédé de façon unifiée (cf. (a2)).

Nous utilisons Alloy [126] comme langage afin d'implémenter cette formalisation. Alloy est un langage de modélisation déclaratif basé sur la logique de premier ordre (FOL, pour first-order logic) associé aux calculs relationnels afin d'exprimer des contraintes complexes, aussi bien structurelles que comportementales. La logique d'Alloy est très générique et n'engage pas à l'utilisation d'un style particulier de spécification [47]. Nous pensons que cela est plus naturel et permet de préserver la puissance d'expression du PML choisi (UML AD). Alloy est associé à un outil, appelé `Alloy Analyzer`, un solveur de contraintes fournissant une *simulation* et *vérification* automatiques basées sur une méthode de *model-finding* à travers un SAT solveur [124]. Bien que Alloy n'ait pas été encore utilisé pour la vérification de procédés, il a été extensivement utilisé dans le domaine de l'ingénierie dirigée par les modèles [141], mais aussi employé avec succès dans une large gamme d'applications telles que la modélisation et l'analyse de protocoles dans les systèmes distribués [240], les réseaux [97], ou dans des systèmes critiques [56].

1.2.4 Outillage

Dans le cadre du projet MERgE, il est nécessaire de développer un outil intégré à sa plate-forme et basé sur Eclipse permettant la *vérification* (A) de procédés logiciels modélisés avec UML AD. De plus, nous avons proposé un *générateur de procédés aléatoires* (B) à des fins de validation de l'approche de vérification.

(A) Outil de vérification. Afin de vérifier simplement un procédé, il est nécessaire de construire un outil graphique au-dessus de l'implémentation Alloy afin d'automatiser tout le processus de vérification. La figure 1.4 donne un aperçu global de l'approche proposée pour vérifier un procédé. Les modélisateurs créent le procédé (1), puis peuvent, quand ils le souhaitent, configurer l'outil de vérification en annotant le modèle et en sélectionnant les propriétés désirées (2). Le but est d'intégrer directement un ensemble de propriétés prêtes à être utilisées et configurables afin de favoriser l'adoption de l'outil par les experts du domaine (cf. (b1)). De plus, pour simplifier l'expression de propriétés métiers sur le procédé, nous proposons un langage d'annotation à utiliser directement sur le modèle (cf. (b1)). Par exemple, pour vérifier que l'`ActivitéA` est toujours exécutée avant l'`ActivitéE`, une simple annotation *before(`ActivitéE`)* ajoutée sur l'`ActivitéA` permet d'exprimer cette contrainte. Ainsi, l'outil est capable de traduire automatiquement le modèle de procédé ainsi que les propriétés désirées en spécification Alloy (3) afin d'être automatiquement vérifiées par l'`Alloy Analyzer`. Le résultat de la vérification est ainsi directement analysé et affiché visuellement à l'utilisateur (4) (cf. (b2)). Cette approche réduit le travail de l'expert à la spécification de son procédé et de ce qu'il souhaite vérifier dessus, sans qu'il ne se préoccupe de la manipulation de la représentation formelle. Ainsi, l'approche ne nécessite aucune connaissance particulière en méthode formelle afin d'être utilisée.

(B) Outil de génération de procédés. Afin de valider précisément notre approche de vérification de procédés, il est nécessaire d'avoir un grand nombre de procédés réalistes. Cependant, le petit ensemble de modèles de procédés disponibles publiquement dans la littérature est insuffisant pour conduire une étude empirique sérieuse, et ainsi valider complètement les travaux réalisés

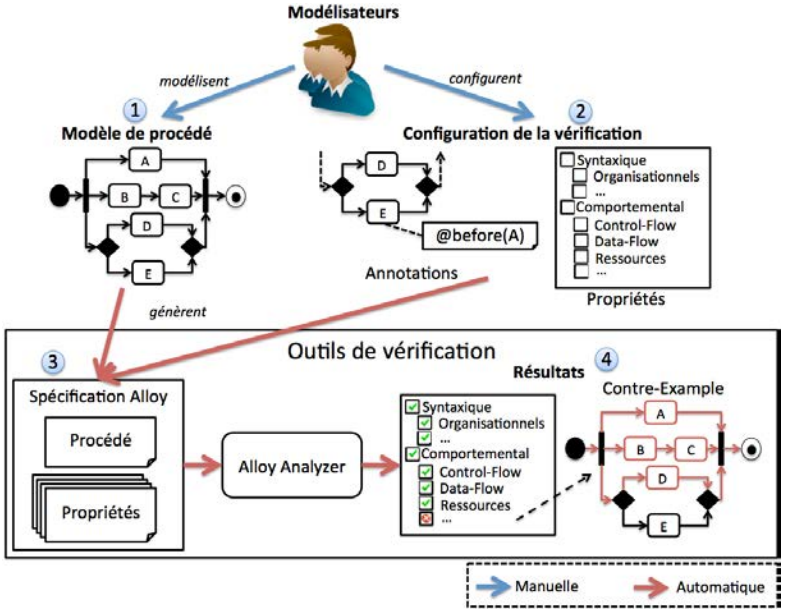


FIGURE 1.4 – Vue globale de l'approche pour vérifier un procédé

autour de l'analyse et la vérification de procédés. De plus, pour des raisons de confidentialité, les entreprises partenaires sont souvent réticentes à partager leurs modèles représentant le fruit d'un long travail de modélisation, dévoilant l'expertise et le savoir-faire de leur coeur de métier.

Une façon de remédier à ce problème consiste à *générer* un ensemble de modèles de tests. Dans la littérature, les approches proposant la génération de modèles sont limitées à la génération de modèles *structurels* [31, 180, 74, 202], c'est à dire des modèles *non comportementaux* tels que les diagrammes de classes UML. Nous avons donc proposé un générateur de procédés basé sur un algorithme génétique multi objectifs [156]. Les procédés sont générés automatiquement à travers une séquence de modifications de haut niveau (*change pattern* [275]) inspirée par la façon dont un modélisateur pourrait avoir réellement modélisé le procédé. Le point clé de l'approche réside dans le réalisme du procédé généré. Nous utilisons cette approche afin de générer des modèles de procédés pour tester nos approches de vérification de procédés.

1.3 Structure du document

Le plan de cette thèse est structuré de la manière suivante :

- **Chapitre 2 : Contexte et État de l'art.** Dans ce chapitre, nous établissons les définitions et concepts de base utilisés par la suite. Nous identifions et catégorisons les propriétés importantes à vérifier sur un procédé. Finalement, nous comparons, selon les différents formalismes et propriétés supportés, les approches existantes permettant la vérification de procédés et l'expression de propriétés métiers.

- **Chapitre 3 : Formalisation de UML AD avec la sémantique fUML pour la vérification de procédés.** Ce chapitre commence par présenter le standard fUML. Ensuite, nous présentons notre formalisation de UML AD et fUML en définissant la syntaxe du langage, puis sa

sémantique. Finalement, nous présentons la formalisation des propriétés intéressantes à vérifier sur un procédé en utilisant la PLTL.

– **Chapitre 4 : Alloy4PV : Un framework basé sur Alloy pour la vérification de procédés métiers.** Ce chapitre présente ALLOY4PV, notre framework basé sur la logique Alloy et implémentant la formalisation du chapitre précédent, permettant de vérifier un large éventail de propriétés sur un modèle de procédé. Nous commençons par présenter le langage Alloy ainsi que son outil associé pour analyser et vérifier une spécification Alloy. Ensuite, nous présentons les différents modules Alloy afin d'implémenter la formalisation, après avoir motivé notre choix concernant l'utilisation d'Alloy. Finalement, nous présentons l'outil graphique associé à ALLOY4PV et intégré à eclipse, afin d'automatiser tout le processus de vérification.

– **Chapitre 5 : Generation de procédés basée sur un algorithme génétique multi objectif.** Dans ce chapitre, nous proposons un générateur aléatoire de procédés basé sur un algorithme génétique multi objectif. Nous présentons l'algorithme, son implémentation, discutons de l'état de l'art concernant la génération de modèles, et finalement, évaluons notre approche.

– **Chapitre 6 : Evaluation.** Ce chapitre présente l'évaluation de notre approche. En plus d'évaluer si nos objectifs de recherche ont été accomplis, ce chapitre évalue les performances de notre approche en analysant trois cas d'études : (1) un procédé de la spécification UML, (2) un procédé fourni par un partenaire du projet MERgE, et (3) un ensemble de procédés générés.

– **Chapitre 7 : Conclusion.** Ce chapitre présente la conclusion générale de notre travail. Il met en regard les objectifs de l'introduction générale avec les contributions détaillées dans le contenu de cette thèse. Finalement, ce chapitre aborde les perspectives à court, moyen et long terme, de ce travail.

– **Appendice A : Alloy for Process Verification (Alloy4PV).** Cet appendice montre l'ensemble des modules Alloy constituant ALLOY4PV afin d'effectuer la vérification d'un des cas d'étude de l'évaluation.

1.4 Publications

Ce travail a donné lieu à des publications dans les conférences internationales suivantes :

1. Laurent Yoann, Reda Bendraou, and Marie-Pierre Gervais. *"Executing and debugging UML models : an fUML extension."* In Proceedings of the 28th Annual ACM Symposium on Applied Computing, **SAC**, pp. 1095-1102. ACM, 2013. [155]
2. Laurent Yoann, Reda Bendraou, and Marie-Pierre Gervais. *"Generation of process using multi-objective genetic algorithm."* In Proceedings of the 2013 International Conference on Software and System Process, **ICSSP**, pp. 161-165. ACM, 2013. [156]
3. Laurent Yoann, Reda Bendraou, Souheib Baarir, and Marie-Pierre Gervais. *"Planning for Declarative Processes."* In Proceedings of the 29th Annual ACM Symposium on Applied Computing, **SAC**, pp. 1126-1133. ACM, 2014. [154]
4. Laurent Yoann, Reda Bendraou, Souheib Baarir, and Marie-Pierre Gervais. *"Formalization of fUML : an Application to Process Verification."* In Advanced Information Systems Engineering - 26th International Conference, **CAISE**, pp. 347-363. Springer, 2014. [153]

5. Laurent Yoann, Reda Bendraou, Souheib Baarir, and Marie-Pierre Gervais. *"Alloy4SPV : a Formal Framework for Software Process Verification."*, In Modelling Foundations and Applications - 10th European Conference, **ECMFA**, pp 83-100. Springer, 2014. [152]

Chapitre 2

Contexte et État de l'art

Dans ce chapitre, nous présentons les définitions et les concepts de base concernant la vérification de procédés métiers. La section 2.1 présente les notions utilisées tout au long de ce travail. La section 2.2 présente l'ensemble des propriétés à vérifier afin de garantir la validité d'un procédé métier basé sur les diagrammes d'activités UML. La section 2.3 présente un état de l'art concernant la vérification de procédés métiers. La section 2.4 présente les principales approches de la littérature afin d'exprimer des propriétés métiers sur un modèle de procédé. Finalement, la section 2.5 présente nos contributions et justifie leur pertinence par rapport aux approches détaillées précédemment.

2.1 Concepts et définitions

Dans cette section, nous présentons les définitions des concepts de base par rapport à la gestion de procédé métiers et la modélisation de procédés. Nous présentons le standard UML et mettons l'accent sur les diagrammes d'activités. Puis, nous présentons les différentes approches de la littérature pour représenter un procédé en se basant sur les diagrammes d'activités. Enfin, nous présentons les techniques de vérification formelles.

2.1.1 Gestion de procédé métiers (BPM)

Dans les années 90, le terme "procédé" est devenu un nouveau paradigme de productivité [212]. Les entreprises étaient encouragées à penser en terme de *procédés* au lieu de *fonctions* ou *procédures*. Au regard d'une intensification toujours plus grande de la mondialisation, une gestion efficace de ces procédés d'entreprise est devenue primordiale. En effet, beaucoup de facteurs tels que l'augmentation de la fréquence des commandes, le besoin d'un transfert rapide de l'information, le besoin de s'adapter aux changements et aux nouveaux besoins, une compétition internationale accrue, et la demande pour des cycles de développement toujours plus courts, ont rendu plus difficiles la rentabilité et la survie des petites et grandes entreprises [232].

Pour répondre à ces challenges, les systèmes d'informations ont été exploités pour gérer les procédés d'entreprises [51, 98]. Au cours de ces vingt dernières années, les formulaires précédemment remplis à la main ont été graduellement remplacés par leurs homologues électroniques. L'influence de ces technologies pour gérer des procédés métiers peut être retracée grâce aux travaux de Hammer et Champy sur le paradigme BPR (Business Process Re-engineering) [108] ainsi que l'ouvrage de Davenport sur le "comment" l'innovation des procédés peut faciliter le paradigme BPR [51]. Cependant, BPM et BPR ne sont pas la même chose : quand BPR appelle

une oblitération radicale des procédés d'affaires existants, son descendant BPM (Business Process Management) est plus concret, itératif, et incrémental afin d'affiner un procédé d'affaires.

Deux autres terminologies souvent utilisées de manière vague sont WfM (Workflow Management) et BPM. Hill *et al.* voient BPM comme une discipline de management avec WfM la supportant comme technologie [109]. Un autre point de vue en provenance du monde académique est que, selon Georgakopoulos *et al.* [98], WfM est un sous-ensemble de BPM défini par van der Aalst *et al.* [260]. La principale différence est que la phase de diagnostic du cycle BPM n'est pas supportée par le WfM. La WfM coalition (WfMC) définit un workflow comme *"l'automatisation des procédés d'affaires, en partie ou dans son ensemble, durant laquelle, des documents, informations, ou tâches sont passés d'un participant à un autre, selon un ensemble de règles de procédure"* [157]. Un système WfM (WfMS) est défini comme un *"système qui définit, crée, et gère l'exécution des workflows à travers l'utilisation d'un logiciel, fonctionnant sur un ou plusieurs moteurs de workflow, lequel est en mesure d'interpréter la définition du procédé, d'interagir avec les participants du workflow et, si besoin, d'invoquer l'utilisation d'outils et applications logiciels"* [157]. Les deux définitions mettent en avant l'exécution, c'est-à-dire, l'utilisation d'un logiciel pour supporter l'exécution de procédé opérationnel. Ces dernières années, les praticiens ont commencé à réaliser que se concentrer sur l'exécution est trop restrictif. De ce fait, le nouveau terme BPM a commencé à faire son apparition.

Il existe beaucoup de définitions de BPM mais la plupart d'entre elles incluent le concept WfM. Van der Aalst définit BPM comme suit [260] :

Definition 1 (Business Process Management)**.** *Supporting business processes using methods, techniques, and software to design, enact, control, and analyze operational processes involving humans, organizations, applications, documents and other sources of information.*

Les systèmes BPM (BPMSs) sont des outils logiciels destinés à automatiser la mise en application de ces procédés. Van der Aalst définit un *système BPM* comme suit [260] :

Definition 2 (BPMS)**.** *A generic software system that is driven by explicit process designs to enact and manage operational business processes.*

Cependant, il semblerait, en réalité, que la plupart des systèmes BPM (BPMS) soient simplement des systèmes WfM (WfMS) et n'aient pas maturé suffisamment pour supporter la phase de diagnostic [145]. En effet, il apparaîtrait que depuis ces dernières années, beaucoup de vendeurs de solutions ont mis à jour le nom de leur produit de "WfM" vers le plus contemporain "BPM" [145].

2.1.2 Modélisation de procédés

La modélisation de procédés métiers est l'activité de représenter le procédé d'une entreprise, de manière à ce que ce procédé puisse être, entre autres, utilisé à des fins de simulation, d'analyse, de vérification, et d'exécution. Traditionnellement, les modélisateurs obtiennent des informations à propos de leurs procédés à partir d'interviews et de documentation existante. L'information obtenue donne aux analystes et managers une idée sur les préoccupations commerciales qui régissent leurs procédés opérationnels, afin de pouvoir améliorer l'efficacité, la qualité, et réduire les délais. À partir des informations obtenues, des *modèles de procédés* sont construits. Un *modèle de procédé métier* a été défini par Davenport comme suit [51]

Definition 3 (Business Process)**.** *A specific ordering of work activities across time and space, with a beginning and an end, and clearly defined inputs and outputs. Processes are the structure by which an organization does what is necessary to produce value for its customers.*

Afin de construire ces modèles, il est nécessaire d'utiliser un *langage de modélisation de procédés*. La modélisation d'un procédé est un processus compliqué et il est évident que différentes

techniques de modélisation ont leurs avantages et inconvénients dans différents aspects dus à la variété des formalismes sous-jacents. Il y a beaucoup de problèmes connus concernant les langages de modélisation, tels que le compromis entre l'expressivité du langage de modélisation et sa complexité d'analyse. Certains langages offrent une syntaxe riche permettant d'exprimer la plupart des activités métiers et leurs relations dans le modèle de procédé, tandis que d'autres fournissent des constructions de modélisation plus génériques facilitant une analyse efficace du modèle de procédés aux moments de la modélisation. Pour définir un langage de modélisation de procédé (PML pour Process Modeling Language), nous adoptons la définition de Lu *et al.* [163] :

Definition 4 (Process Modeling Language)**.** *A process modeling language provides appropriate syntax and semantics to precisely specify business process requirements, in order to support automated process verification, validation, simulation and process automation. The syntax of the language provides grammar to specify objects and their dependencies of the business process, often represented as a language-specific process model, while the semantics defines consistent interpretation for the process model to reflect the underlying process logic.*

Il y a principalement deux formalismes sur lesquels les PMLs sont développés : orientés sur un graphe ou des règles. Les PMLs basés sur un graphe ont leurs racines dans la théorie des graphes [29] et leurs variantes, tandis que les PMLs basés sur les règles sont basés sur la logique mathématique [142].

Les PMLs basés sur un graphe comme BPMN [192], les diagrammes d'activités UML [190], ou des langages plus formels tels que les réseaux de Petri [181] définissent un modèle de procédé comme une spécification détaillée étape par étape d'une procédure qui doit être suivie pendant l'exécution du procédé. Ces PMLs se basent sur les langages de programmation *impératifs*, en adoptant des concepts tels que le branchement conditionnel (`if`, `then`, `else`) et boucle (`for`, `while`, `repeat`) afin de représenter le flux d'exécution du procédé. Ces approches définissent avec exactitude *comment* le procédé doit être exécuté.

Les PMLs basés sur les règles abstraient la logique du procédé en un ensemble de règles, où chacune d'entre elles est associée à une ou plusieurs activités métiers, spécifiant certaines propriétés de l'activité comme les *pre* et *post* conditions de l'exécution. Le moteur d'exécution est généralement un moteur d'inférence de règles. Pendant l'exécution, ce moteur examine les conditions de données et de contrôle et détermine l'ordre des activités à exécuter selon l'ensemble des règles définies. Un exemple de ce type de procédé est DECLARE proposé par Pesic *et al.* [198].

Dans cette thèse, nous vérifions des procédés modélisés avec les diagrammes d'activités UML (UML AD pour Activity Diagram), un PML basé sur la théorie des graphes. Comme expliqué dans l'introduction, ce choix est une des contraintes du projet MERgE. Dans la prochaine section, nous présentons rapidement le standard UML.

2.1.3 Unified Modeling Language (UML)

UML [190] est un langage de modélisation généraliste pour spécifier, visualiser, construire et documenter les artefacts d'un système logiciels. UML est né de la fusion des méthodes objet dominantes (OMT [213], Booch [30] et OOSE [128]), puis normalisé par l'OMG depuis 1997 [186]. L'OMG a pour but principal d'améliorer les pratiques actuelles des entreprises en permettant l'utilisation d'outils de modélisation graphiques, interopérables et standardisés. UML a réussi à s'imposer comme un langage dominant aussi bien dans le monde académique qu'industriel [88].

La figure 2.1 montre la catégorisation hiérarchique des différents diagrammes de UML. UML fournit différents types de diagrammes (`Diagram`) pour modéliser à la fois le côté structurel et comportemental d'un système :

– Les *diagrammes structurels* (`Structure Diagram`) mettent l'accent sur les éléments qui doivent être présents sur le système modélisé. Ils représentent une structure qui est souvent

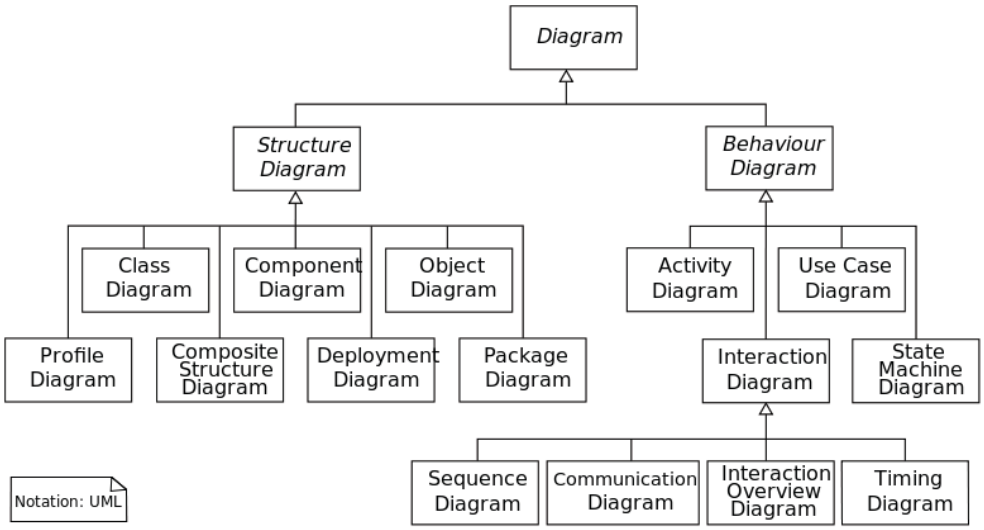


FIGURE 2.1 – Catégorisation hiérarchique des différents diagrammes de UML

utilisée pour documenter l'architecture des systèmes logiciels. Par exemple, le diagramme de composant (`Component Diagram`) permet de décrire comment un système logiciel est divisé en sous-composant tout en montrant les dépendances entre eux.

– Les *diagrammes comportementaux* (`Behaviour Diagram`) mettent l'accent sur ce qui doit se passer dans le système. Ils représentent un comportement qui est souvent utilisé pour décrire les fonctionnalités d'un système. Par exemple, les diagrammes d'états (`State Machine Diagram`) permettent de décrire les changements d'états d'un objet ou d'un composant, en réponse aux interactions avec d'autres objets/composants ou avec des acteurs.
– Finalement, les *diagrammes d'interactions* (`Interaction Diagram`), un sous-ensemble des diagrammes comportementaux, mettent l'accent sur le flux de contrôle et de données parmi les éléments du système modélisé. Par exemple, le diagramme de séquence (`Sequence Diagram`) montre comment les objets communiquent entre eux en terme de séquence de message.

Une explication plus détaillée de tous les différents types de diagrammes UML ainsi que sur leurs cas d'utilisation est disponible dans [214]. Dans la prochaine section, nous mettons l'accent sur les diagrammes d'activités UML.

2.1.4 Les diagrammes d'activités UML

Le diagramme d'activités UML est un diagramme comportemental permettant de représenter le séquencement d'actions au sein d'un système. Quand la version 1 de la spécification UML définissait UML AD comme une forme spécialisée de diagramme à états, la version 2 de la spécification UML à complètement redéfini ces diagrammes en les basant sur une sémantique de règles de circulation de jetons entre les noeuds. Cette sémantique est inspirée des réseaux de Petri colorés. Beaucoup de nouvelles structures comme les itérations, branchements conditionnels (*if-then-else*) et gestion d'exceptions ont été introduites afin d'améliorer l'expressivité et la concision de ces diagrammes. Ainsi, les diagrammes d'activités UML 2 sont basés sur un meta-modèle complètement nouveau comportant les packages `Actions` et `Activities`. *"Une activité est la spécification d'un comportement paramétré comme une séquence coordonnée d'un sous-ensemble*

d'unités dont chacun de ses éléments est une actions. " [190], quand une `Action` est définie comme *"une unité fondamentale de la spécification d'un comportement"*. Ainsi, les diagrammes d'activités sont utilisés pour modéliser le séquencement et les conditions pour coordonner le comportement d'un système. Ces diagrammes reposent sur la définition d'un flux de contrôle et de données pour modéliser ce comportement. UML fournit une taxonomie très détaillée d'actions, comportant plus de 40 types d'actions. En revanche, une discussion détaillée de chacune d'entre elles dépasserait le cadre de cette thèse.

Dans la suite, nous présentons seulement les éléments nécessaires afin de modéliser un procédé [20]. L'ensemble de ces éléments est visible sur la figure 2.2. Une `Activity` est ainsi représentée comme un graphe orienté d'`ActivityNode`s et d'`ActivityEdge`s. Une `ActivityNode` *"peut être l'exécution d'un comportement, telle qu'une opération arithmétique, l'appel d'une opération, ou la manipulation du contenu d'un objet"*, mais aussi *"inclut des constructions de flux de contrôle, telles que les synchronisations, les décisions, et des contrôles concurrents"* [190]. Une `ActivityNode` peut faire partie de trois types de noeuds : `ObjectNode`, `ControlNode` et `ExecutableNode`. Un noeud `ObjectNode` représente les données dans un procédé, un noeud `ControlNode` coordonne le flux d'exécution et un noeud `ExecutableNode` représente un noeud qui peut être exécuté, c'est-à-dire, une action du procédé. Il y a deux types d'`ActivityEdge` pour lier les noeuds : `ObjectFlow` et `ControlFlow`. Les `ObjectFlow` relient les `ObjectNode`s et permettent d'orienter le flux de données entre eux. Les `ControlFlow` contraignent l'ordre d'exécution désiré des `ActivityNode`s. Les `ControlNode` peuvent être utilisés pour le routage parallèle (`ForkJoin`), le routage conditionnel (`DecisionNode`), la synchronisation (`JoinNode`) et la fusion de flux alternatifs (`MergeNode`). Les `InitialNode`s et `AcitivityFinalNode`s représentent respectivement le début et la fin d'une `Activity`, tandis que les `FlowFinalNode` terminent un flux. Les `InputPin` et `OutputPin` sont ancrés aux `Action`s afin de représenter les données consommées et produites par l'action. Similairement, une `Activity` peut avoir de multiples `ActivityParameterNode` pour représenter ses données entrantes et sortantes.

Dans la prochaine section, nous présentons les moyens de mettre à profit les diagrammes d'activités UML pour modéliser un procédé métier.

2.1.5 Modélisation de procédés avec UML AD

Reprenons le procédé de la figure 1.3 présenté dans l'introduction. Ce procédé est modélisé en utilisant les symboles présentés dans la section précédente. En utilisant strictement les éléments définis dans le diagramme d'activité UML, il est principalement possible de représenter la perspective du flux de contrôle et de données. En effet, dans [220], Russell *et al.* ont vérifiés l'adéquation de UML AD pour la modélisation de procédés métiers, en évaluant leurs capacités à capturer l'ensemble des *workflow patterns* définis dans [259]. Ils concluent que, bien que UML AD supporte les principaux *workflow patterns*, UML AD tend à être *"principalement concentré sur le flux de contrôle et de données pour capturer l'acheminement statique associé aux actions"*. UML AD n'inclut pas les éléments nécessaires pour représenter les ressources et autres informations organisationnelles nécessaires lors de la modélisation d'un procédé, tel que le temps alloué pour effectuer une activité.

Dans la littérature, les PMLs basés sur UML AD étendent en général le meta-modèle des diagrammes d'activités UML avec les méta-classes nécessaires pour stocker et représenter ces informations. Par exemple, UML4SPM [20], une extension du meta-modèle UML 2 pour la modélisation de procédé logiciel, inclut ce genre d'informations organisationnelles, telles que les agents, rôles et outils utilisés par les actions. Il permet aussi de spécifier les temps de début et de fin des activités, permettant ainsi de modéliser toutes les dimensions d'un procédé. De la même manière, Ellnet *et al.* [75] propose eSPEM, une extension de SPEM [188] se basant à la fois sur les diagrammes d'activités UML pour modéliser la partie comportementale du modèle

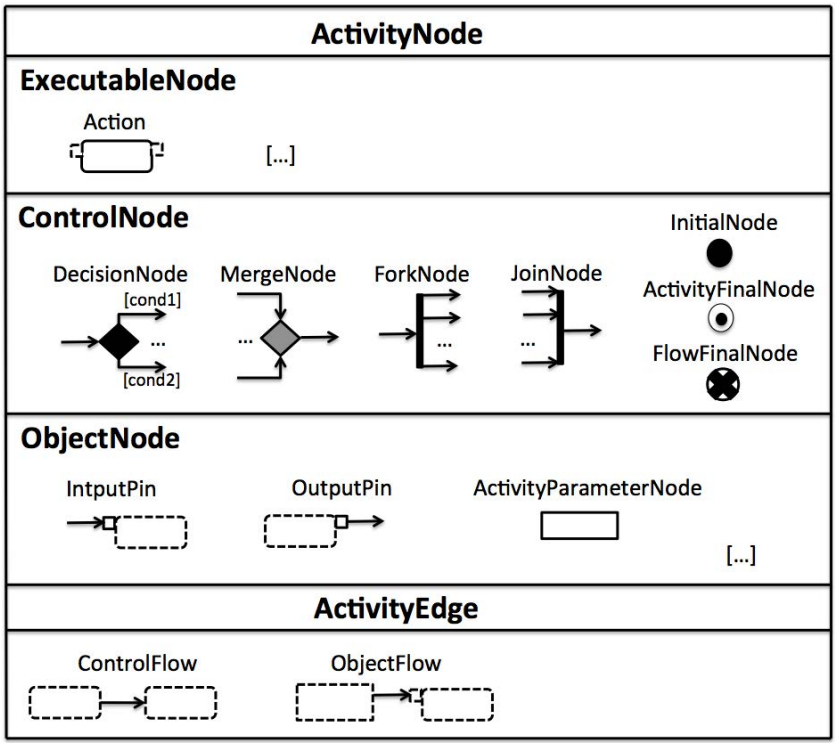


FIGURE 2.2 – Quelques éléments de la notation graphique pour représenter un UML AD

de procédé, et sur les diagrammes de machine à états UML pour capturer le cycle de vie des artefacts. Strembeck *et al.* [239] propose un meta-modèle étendant celui de UML AD 2 afin de pouvoir spécifier des procédés métiers supportant les rôles et les contrôles d'accès. Une discussion plus détaillée de différents PMLs basés sur les diagrammes d'activité UML est présentée par Bendraou *et al.* dans [21].

D'autres approches étendent le meta-modèle UML à travers la définition de profile UML [190]. Ils fournissent un mécanisme pour étendre les méta-classes UML existantes à travers l'utilisation de *stéréotype*, afin de permettre l'utilisation d'une terminologie spécifique adaptée au domaine désiré. Les propriétés ajoutées par un *stéréotype* sont appelées *tagged values*. Jager *et al.* [129] utilise ce mécanisme d'UML pour définir leurs modèles de procédés. L'avantage principal est de pouvoir directement utiliser les outils standard de modélisation d'UML.

Pour résumer, il existe principalement deux approches afin de caractériser et représenter un procédé *avec* des informations organisationnelles : (1) en utilisant des profiles UML et (2) en définissant un méta-modèle étendant le celui d'UML.

Dans cette thèse, nous n'avons pas adopté un profil ou un méta-modèle particulier de la littérature afin de capturer les informations organisationnelles d'un procédé basé sur UML AD. Nous avons simplement défini un simple méta-modèle étendant le méta-modèle UML, afin de stocker sur le modèle les informations organisationnelles nécessaires pour analyser l'ensemble des propriétés comportementales présentées dans la section suivante. Ces concepts ont été formalisés dans le chapitre 3. En redéfinissant la procédure de génération de code définie dans le chapitre 4, l'approche proposée dans cette thèse n'est pas limitée à l'utilisation de notre méta-modèle UML, mais peut être adaptée pour tout PMLs basé sur UML AD supportant la définition de ces informations.

La prochaine section présente les deux principales approches des méthodes de vérification formelle.

2.1.6 Vérification formelle

La vérification formelle représente l'acte de prouver ou réfuter l'exactitude d'un système par rapport à certains cahiers des charges ou spécifications formelles, à l'aide d'outils mathématiques des *méthodes formelles*. La vérification formelle peut être utile pour vérifier l'exactitude de systèmes tels que : protocole cryptographique, matériel, ou bien code source de logiciel.

Il existe deux approches pour la vérification formelle : la *preuve de théorèmes* et le *model-checking*. Dans la suite, nous présentons ces deux approches en mettant l'accent sur leurs avantages et inconvénients.

Theorem proving

Le *theorem proving* consiste à exprimer une description du système à partir d'un ensemble d'axiomes et de règles d'inférences, ainsi que les propriétés désirées afin de définir un théorème. Les propriétés sont ensuite vérifiées sur le système à l'aide d'un *theorem prover* [195]. Les calculis utilisés pour ces preuves ont été introduits pour la première fois par Hoare en 1969 dans [112] où il raisonne sur la justesse du programme en terme de *pre* et *post conditions*.

Le principal avantage des *theorem provers* provient du fait qu'ils sont capables d'être utilisés dans le cas de systèmes infinis. Cependant, ils ont pour désavantages de n'être pas complètement automatiques. En effet, les assistants de preuve nécessitent un utilisateur avec une *forte expertise* du système sous-jacent pour "*donner*" le chemin amenant à la solution du système. Ce problème représente le frein le plus important entravant une plus large adoption par l'industrie. Le *theorem proving* est principalement utilisé pour la vérification de système matériel [48]. Par exemple, la méthode B [5] a été utilisée pour vérifier les composants critiques du système *Météor* équipant la ligne 14 du métro parisien [18].

Model checking

L'autre technique importante de vérification formelle est le *model-checking* , une technique de vérification automatique de systèmes réactifs à états finis introduit par Clarke *et al.* [40, 42]. Le *model-checking* consiste à construire un modèle formel du système sous la forme d'un automate à états finis. Généralement, un langage de spécification tel que les réseaux de Petri [201] ou les processus séquentiels communicants [113] est utilisé. Il faut ensuite définir les propriétés importantes à vérifier sur le modèle. Généralement, ces propriétés sont exprimées en utilisant un langage logique de spécification de propriétés tels que LTL (Linear Temporal Logic) [204] ou CTL (Computation Tree Logic) [107]. À partir de ces deux entités, un *model-checker* sera capable de répondre à la question de satisfiabilité de la propriété par rapport au modèle en explorant exhaustivement l'espace d'états. Dans le cas où la propriété n'est pas vérifiée, un exemple de chemins invalidant la propriété est généré par le *model-checker*. La figure 2.3 résume le principe du *model-checking*

L'avantage du *model-checking* est non seulement le caractère automatique de la procédure de vérification, mais aussi, dans certains cas, la génération de contre-exemples. Le travail de l'utilisateur est réduit à la modélisation formelle de son système et à la définition de la propriété. Dans l'industrie, le caractère automatique du *model-checking* en fait une approche de vérification formelle populaire. Un exemple récent concerne la sonde *Curiosity* envoyée par la NASA sur la planète Mars. Après vérification, plusieurs parties clefs du code multi-threadé ont montré des situations de compétitions (*race condition*) ou d'inter-blocages [116]. Cependant, un des inconvénients du *model-checking* concerne le problème d'explosion combinatoire de l'espace d'états

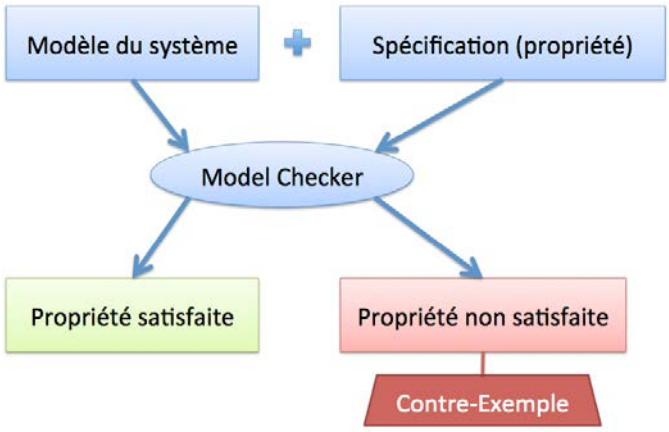


FIGURE 2.3 – Principe du *model-checking*

du système. Ce problème est dû à la croissance *exponentielle* de la taille de l'espace d'état d'un système concurrent en fonction du nombre de processus et du nombre de composants par processus [43].

La prochaine section présente l'ensemble des propriétés importantes à vérifier pour garantir la conformité d'un procédé métier.

2.2 Propriétés à vérifier pour garantir la conformité d'un procédé

La vérification de propriétés sur un modèle de procédé est un moyen de valider la cohérence d'un procédé. Il existe principalement deux types de propriétés qui peuvent être analysées sur un procédé : des propriétés syntaxiques et des propriétés comportementales. Dans cette section, nous présentons ces deux types de propriétés et mettons l'accent sur la présentation des propriétés comportementales. Nous basons nos exemples sur les procédés modélisés avec les diagrammes d'activités UML.

2.2.1 Propriétés syntaxiques

La vérification des propriétés *syntaxiques* correspond à examiner si la syntaxe du modèle respecte son méta-modèle ainsi que ses contraintes associées. En effet, afin de garantir la conformité syntaxique d'un modèle de procédé, il doit de toute évidence respecter la spécification de son méta-modèle (c-à-d. ne pas utiliser de méta-classes ou attributs non définis dans celui-ci). Cependant, les meta-modèles ne sont généralement pas assez expressifs pour fournir tous les aspects pertinents de leurs spécifications. Il y a un besoin pour décrire des contraintes additionnelles aux méta-modèles [141]. Ces contraintes sont souvent décrites en langage naturel. La pratique a montré que cela était source d'ambiguïtés, et que cela empêchait leur vérification automatique.

En conséquence, l'OMG a défini un langage formel nommé Object Constraint Language (OCL) pour spécifier des contraintes sur n'importe quel modèle ou méta-modèle conforme au Meta-Object Facility (MOF) [187]. OCL peut être utilisé pour spécifier des contraintes concernant la structure statique du système, c-à-d., des invariants qui doivent être vrais à tout moment pour toutes les instances du méta-modèle [210]. Selon Atkinson *et al.* [11], beaucoup d'erreurs qui affectent la cohérence d'un procédé sont généralement liées aux erreurs typographiques ou syntaxiques.

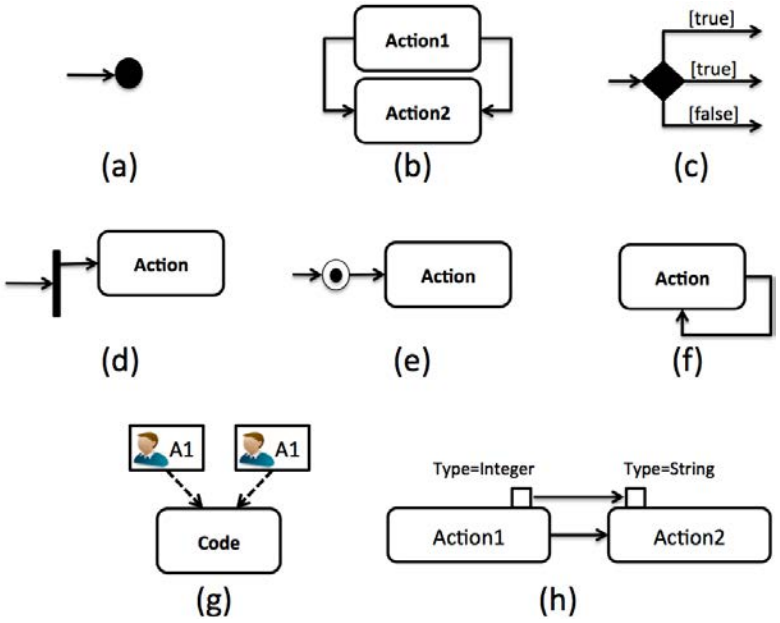


FIGURE 2.4 – Quelques exemples d'erreurs syntaxiques sur des UML AD

La figure 2.4 montre quelques exemples d'erreurs syntaxiques qui peuvent être évitées en vérifiant le modèle avec des contraintes OCL :

(a) Le noeud `InitialNode` représente le point d'entrée de l'activité, ainsi il ne doit pas avoir de `ControlFlow` en entrée.

(b) Un des deux `ControlFlow` est inutile du fait qu'il est identique à l'autre.

(c) Après exécution, le noeud `DecisionNode` doit choisir seulement *un* de ses `ControlFlow` sortants, cependant dans ce cas, si l'évaluation de la décision est *vraie*, deux des `ControlFlows` peuvent être choisies (la décision est non-déterministe, dépendant des *points de variation sémantique* de UML utilisés).

(d) Un noeud `ForkNode` est utilisé pour démarrer des actions en parallèle, cependant celui-là a seulement un `ControlFlow` sortant ce qui rend le noeud `ForkNode` inutile.

(e) Le noeud `ActivityFinalNode` représente la fin de l'activité, ainsi le noeud `Action` ne sera jamais exécuté.

(f) Le noeud `Action` ne pourra jamais démarrer du fait que le `ControlFlow` a la même *source* et *destination*.

(g) La ressource `A1` est assignée deux fois à la même action `Code`.

(h) Le type de donnée créée par l'`Action1` est de type *integer*, quand l'entrée de l'`Action2` reçoit normalement une donnée de type *string*.

Certains de ces exemples peuvent être vus juste comme des avertissements (ex. (b), (d) et (g)) ou des erreurs (ex. (a), (c), (e), (f) et (h)). Les avertissements n'ont aucun impact sur l'exécution du procédé. Les erreurs doivent être réglées avant l'exécution du procédé du fait que le procédé pourrait ne pas être exécuté comme attendu par le modélisateur.

Chacun de ces problèmes peut être vérifié en spécifiant une règle OCL, de manière à ce que le résultat rendu par l'évaluation soit clair et non ambigü. Par exemple, l'évaluation de la contrainte visible sur la figure 2.5 permet de vérifier le problème de la sous figure (f), c-à-d. pour chacun des

`ControlFlow`, si leurs *source* et *destination* sont différentes. En cas d'évaluation négative de la règle, le `ControlFlow` incriminé peut être directement detecté.

```
1 context ControlFlow inv NoControlFlowSameSourceTarget:
2         self.source != self.target
```

FIGURE 2.5 – Contrainte OCL pour vérifier le problème de la sous figure (f)

Ce type de propriétés est bien supporté par de nombreux outils et approches [119, 7, 196], et est vérifié *presque* instantanément [73]. Par exemple, Akehurst [7] propose un ensemble de règles OCL pour valider une spécification BPEL. De la même manière, Pereira *et al.* [196] vérifient la cohérence des procédés logiciels exprimés avec SPEM 2.0 en utilisant un ensemble de règles syntaxiques exprimées en logique de premier ordre.

Cependant, la vérification de ces propriétés syntaxiques sur le modèle de procédés n'est pas suffisante pour s'assurer de sa justesse. Beaucoup de problèmes *comportementaux* peuvent rester sur le modèle, même après avoir résolu les problèmes syntaxiques. La prochaine section présente des propriétés temporelles qui ne peuvent être exprimées en utilisant un langage tel que OCL.

2.2.2 Propriétés comportementales

Les propriétés *comportementales* expriment des contraintes qui doivent être garanties pour toutes les exécutions possibles d'un modèle de procédés. En effet, contrairement à certains modèles statiques tels que les diagrammes de classes UML, les modèles de procédés définissent un *comportement* s'établissant au cours du temps.

Dans la suite, nous présentons et catégorisons différentes propriétés comportementales qui peuvent être exprimées sur un procédé modélisé avec UML AD. Comme expliqué dans l'introduction en section 1.2.1, cette thèse est basée sur la sémantique de UML AD définie par le standard fUML [189], un nouveau standard définissant précisément la sémantique d'exécution d'un sous-ensemble d'UML. Ainsi, nous nous limitons au sous-ensemble défini par fUML pour exprimer un AD. Par exemple, certains problèmes de données peuvent être introduits sur un procédé avec l'utilisation d'un noeud `DataStoreNode` (un noeud pour stocker des informations non transitoires) tel que l'*incohérence des données* du fait que ce noeud de données peut être écrit par plusieurs activités en parallèle. Cependant, le noeud `DataStoreNode` n'étant pas disponible dans fUML, nous ne catégorisons pas ce type de propriété dans la suite. Il est important de noter que les exemples donnés dans cette section ne nécessitent pas de connaître la sémantique fUML pour être assimilés. En revanche, le lecteur intéressé peut voir la section 3.1 pour une introduction à fUML.

(1) Flux de contrôle. La littérature adresse principalement une sous-catégorie de ces propriétés appelée *propriété de sûreté* (soundness). Introduite par Van Der Aalst *et al.* [261], cette notion tend à étudier 3 cas :

(1.1) Possibilité de complétion (interblocage). Un procédé qui commence doit toujours pouvoir se terminer. Sur la figure 2.6, cette propriété se traduit par l'obligation d'atteindre le noeud `ActivityFinalNode` sur chaque exécution du procédé. Un contre-exemple est exposé : $\{Initial, A, Decision, B, Merge\}$. En effet, quand le noeud `DecisionNode` décide de l'exécution de l'action `B`, l'action `D` ne peut jamais être exécutée (du fait que `C` ne l'est pas), impliquant une situation d'interblocage du fait qu'il n'est plus possible de continuer l'exécution du procédé.

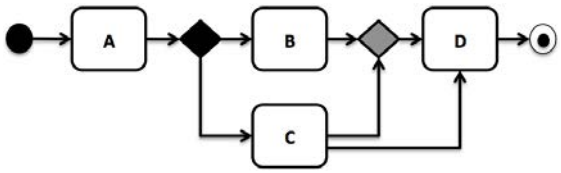


FIGURE 2.6 – Exemple de problème d'*impossibilité de complétion*

(1.2) Complétion non propre. Il ne doit y avoir aucune autre tâche en cours d'exécution quand le procédé se termine. Sur la figure 2.7, une situation de compétition (*race condition*) entre B et C implique que le noeud `ActivityFinalNode` peut être atteint, alors que l'action `C` est toujours en cours d'exécution.

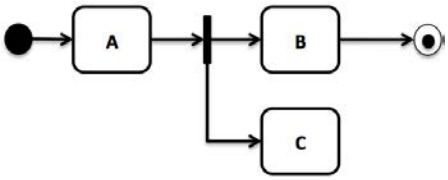


FIGURE 2.7 – Exemple de problème de *complétion non propre*

(1.3) Transition morte. Un procédé ne doit pas contenir des activités qui ne peuvent jamais s'exécuter. Sur l'exemple de la figure 2.8, l'action `A` est morte, du fait que pour exécuter `A`, il faut aussi au préalable exécuter `B`, et inversement.

FIGURE 2.8 – Exemple de problème de *transition morte*

(2) Flux de données. Dans la littérature, il est aussi possible de trouver des propriétés qui se rapportent à la notion de *sûreté*, mais qui se concentrent sur l'analyse du flux de données plutôt que sur le flux de contrôle [221, 246, 247]. Certaines des propriétés référencées par ces travaux ne demandent pas l'exploration de l'espace d'état du procédé, mais peuvent être simplement exprimées à l'aide de propriétés syntaxiques (ex. l'incompatibilité des données [221] de la figure 2.4 (h)). L'idée ici est de vérifier le procédé contre différents problèmes de données :

(2.1) Données manquantes. Quand certaines données doivent être accédées, mais elles n'ont jamais été créées ou ont déjà été consommées. Sur l'exemple de la figure 2.9, ce problème entraîne un inter-blocage. En effet, l'action `D` a besoin de la donnée produite par l'action `B` pour s'exécuter, cependant, dans le cas où le noeud `DecisionNode` continue l'exécution vers l'action `C`, cette donnée n'est jamais produite.

(2.2) Données redondantes. Quand certaines données sont créées, mais jamais utilisées par la suite. Ce cas se présente sur la figure 2.10, lorsque le noeud `DecisionNode` choisit le chemin vers l'action `C`, la donnée produite par l'action `A` n'est jamais utilisée.

La plupart des approches de vérification de procédés se concentrent sur la vérification du flux de contrôle, et quelques une s'interessent aussi aux flux de données [246]. Cependant, il

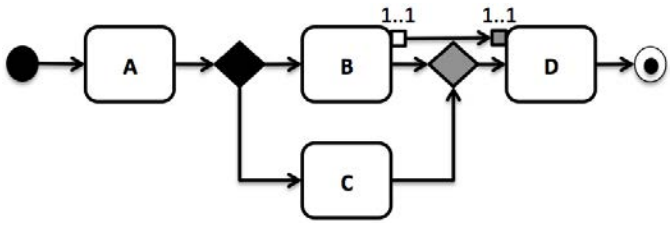


FIGURE 2.9 – Exemple de problème de *données manquantes*

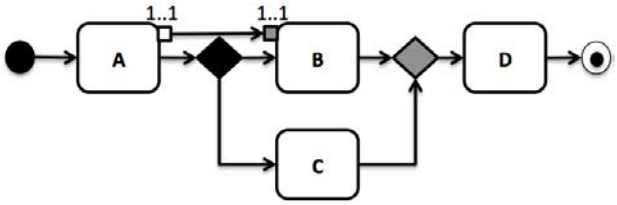


FIGURE 2.10 – Exemple de problème de *données redondantes*

est aussi important de prendre en compte **la partie organisationnelle du procédé**. Ainsi, dans la suite nous introduisons les propriétés liés aux *ressources* et aux *contraintes temporelles*. A notre connaissance, peu d'approches proposent de vérifier certains problèmes temporels [33], mais aucunes ne s'interessent aux problèmes liées aux ressources.

(3) Ressources. Généralement, les activités d'un procédé peuvent avoir besoin de ressources comme des humains ou des ordinateurs. Ces ressources sont souvent finies et ne peuvent être partagées avec d'autres activités. Dans ce cas, des conditions de compétitions peuvent poser problème : une activité qui a besoin de certaines ressources ne peut commencer à s'exécuter à temps à cause d'une autre activité utilisant la même ressource.

(3.1) Ressource manquante. Quand une activité ne peut commencer à cause d'un manque de disponibilité des ressources nécessaires. Sur l'exemple de la figure 2.11, si l'action `B` est en cours d'exécution, l'action `C` ne peut pas commencer à s'exécuter (et inversement) du fait que la ressource `R1` ne sera pas disponible.

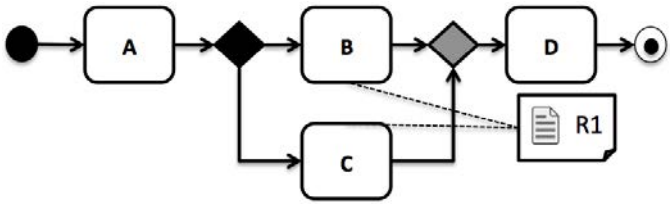


FIGURE 2.11 – Exemple de problème de *ressource manquante*

Il est important de noter qu'une ressource peut avoir différentes capacités d'utilisation.

Par exemple, si `R1` est une ressource correspondant à une voiture, il n'est pas possible de l'utiliser par deux personnes en même temps pour aller à des destinations différentes. Si `R1` est une ressource humaine, il est fort probable qu'elle ne puisse pas travailler sur deux activités en même temps. En revanche, ces informations dépendent grandement du contexte du procédé, dans le cas des procédés de développement logiciel, il est très courant qu'une

ressource humaine travaille sur deux activités en même temps. Il est aussi important de noter qu'un potentiel inter-blocage peut aussi survenir si une activité a besoin d'un type particulier de ressources, non disponible sur le procédé. Dans ce cas-là, il est possible de vérifier *structurellement* (syntaxiquement) si le procédé possède ce type-là.

(3.2) Utilisation inefficace des ressources. Quand une ressource est utilisée de manière inefficace. Sur cet exemple (figure 2.12), la ressource `R1` est affectée à l'action `B`. Cependant, si le noeud `DecisionNode` choisit de continuer l'exécution vers l'action `C`, la ressource `R1`, bien qu' allouée pour l'exécution de l'action `B`, ne sera jamais utilisée. Il est important de noter

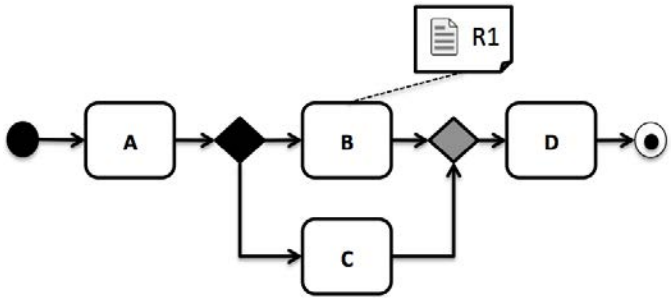


FIGURE 2.12 – Exemple de problème d'*utilisation inefficace des ressources*

qu'une utilisation inefficace des ressources n'est pas une *erreur*, mais plus une *indication* sur le fait que la modélisation autorise des exécutions sur lesquelles certaines ressources ne sont pas utilisées.

(4) Temps. Peu d'approches dans la littérature ont enquêté sur la modélisation du temps dans un procédé. Selon Marjanovic *et al.* [168], trois contraintes temporelles peuvent être spécifiées : (1) une contrainte de durée qui modélise la durée attendue d'une activité dans un procédé (un temps relatif simple ou bien un intervalle de deux valeurs) ; (2) une contrainte de date limite (deadline) qui peut être spécifiée en terme de limite de temps absolu quand une activité doit débuter ou finir durant l'exécution du procédé ; (3) une contrainte temporelle interdépendante qui détermine quand une activité doit commencer ou finir relativement au commencement ou fin d'une autre activité. La cohérence temporelle joue un rôle crucial dans la modélisation des contraintes de temps du procédé. Elle doit non seulement être vérifiée durant la modélisation du procédé, mais aussi durant l'exécution du procédé (ex. à chaque début ou fin d'activité) pour s'assurer que les activités sont en train de s'exécuter dans les temps comme prévu.

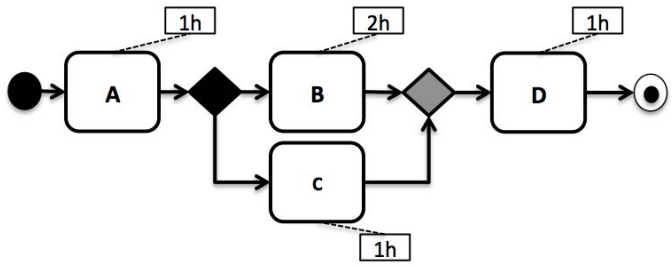


FIGURE 2.13 – Exemple de procédé annoté avec le temps nécessaire pour effectuer les actions

Dans la figure 2.13, ces contraintes correspondent à :

(4.1) Durée du procédé. Si le procédé est prévu pour se terminer en *3h*, le chemin d'exécution $\{A, B, D\}$ montre que c'est impossible et que le procédé terminera en *4h*. En revanche, le chemin d'exécution $\{A, C, D\}$ termine le procédé dans les temps.

(4.2) Contrainte de début/fin d'activité. Si l'action `D` doit commencer à s'exécuter *2h* après le début du procédé, de la même manière que l'exemple précédent, le chemin d'exécution $\{A, B, ...\}$ montre que le début d'exécution de `D` commencera *3h* après le début du procédé.

(4.3) Contrainte relative de début/fin d'activité. Si l'action `D` doit commencer à s'exécuter *1h* après la fin d'exécution de l'action `A`, encore une fois, le chemin d'exécution $\{A, B, ...\}$ montre que `D` commencera *1h* trop tard.

(5) Métiers. Quand les autres propriétés précisent des contraintes qui doivent être valides pour tous les procédés, les propriétés métiers représentent des propriétés adaptées à un procédé donné. Elles jouent un rôle important du fait qu'un procédé peut être syntaxiquement correct et valide par rapport aux propriétés présentées précédemment, mais invalide par rapport à des contraintes métiers.

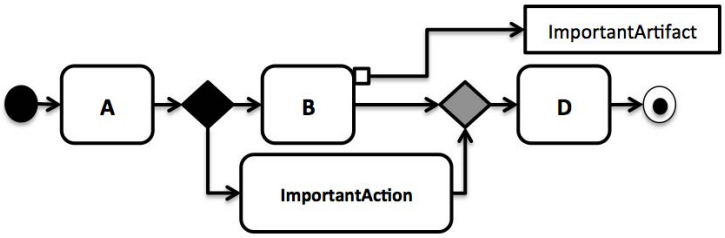


FIGURE 2.14 – Exemple d'un procédé valide

La figure 2.14 montre un exemple de procédé valide. Cependant, l'action `ImportantAction` est considérée critique dans le sens où le modélisateur souhaite que cette action soit toujours exécutée au moins une fois durant l'exécution du procédé. "Est-ce que `ImportantAction` est exécutée, peu importe les choix faits durant l'exécution du procédé ?". Sur cet exemple, ce n'est pas toujours le cas du fait que le chemin d'exécution $\{A, B, D\}$ termine le procédé correctement sans exécuter `ImportantAction`. Une autre question importante pourrait être : "Est-ce que `ImportantArtefact` (le but du procédé) est toujours disponible à la fin du procédé ?". L'autre chemin d'exécution $\{A, ImportantAction, D\}$ montre un chemin où cet artefact n'est jamais créé. Les propriétés métiers correspondent donc à des contraintes supplémentaires que la modélisation du procédé doit respecter.

Le tableau 2.1 référence les principales propriétés métiers exprimables sur un procédé. Il représente le résultat d'une étude de la littérature concernant à la fois la vérification de procédés impératifs [179], mais aussi de contraintes de procédés déclaratifs [154]. Bien que non représenté sur ce tableau, chacune de ces propriétés peut être représentée de façon négative. Par exemple, "`A` **n'**est **pas** exécuté avant `B`". Cependant, il est important de noter qu'il est très difficile de faire un listing exhaustif de toutes les propriétés métiers exprimables sur un procédé. En effet, une propriété métier peut être une combinaison logique de plusieurs autres propriétés métiers basiques. En partant du principe que $\mathsf{existence[a]}$ est la propriété métier définissant que l'activité "a" doit être exécutée pendant l'exécution du procédé, il serait possible d'imaginer une propriété telle que :

$$\mathsf{existence[a, b, c]} \stackrel{def}{=} \mathsf{existence[a]} \implies (\mathsf{existence[b]} \wedge \mathsf{existence[c]})$$

impliquant que si l'activité a est exécutée, alors b et c doivent être exécutées aussi pendant l'exécution (sans aucune contrainte d'ordre d'exécution, mais simplement d'existence). Un grand éventail de propriétés métiers exprimables sur le *flux de contrôle* du procédé est disponible dans le langage déclaratif DecSerFlow [257].

Les propriétés métiers peuvent être divisées en 2 catégories : (1) les propriétés spécifiées par le modélisateur ; (2) les propriétés automatiquement déduites et générées à partir du modèle de procédé. Selon le méta-modèle utilisé par le PML, certains proposent des informations permettant de générer automatiquement ce genre de contraintes. Par exemple, UML4SPM [20] permet de définir si les données produites sont des délivrables logiciels. Un délivrable étant généralement un des buts principaux de l'exécution de procédés logiciels, une propriété métier spécifiant pour chacun d'entre eux que cette donnée doit toujours être présente à la fin de l'exécution est souhaitable pour s'assurer d'une modélisation correcte.

2.2.3 Propriétés faibles et fortes

Chacune de ces propriétés (flux de contrôle, données, ressources, et temps) peut être distinguée en une version *faible* et *forte*. La version *faible* s'assure que la propriété est vraie pour au moins *une* exécution possible. La version *forte* s'assure que la propriété est vraie pour toutes les exécutions possibles du procédé. Prenons en exemple la propriété d'*impossibilité de complétion* par rapport à la figure 2.6. La version *faible* de cette propriété correspond à "existe-t-il au moins une exécution telle que le procédé se termine ?", quand la version *forte* correspond à "quelque soit l'exécution, le procédé se termine-t-il toujours ?". Sur cet exemple, la version *faible* est valide grâce au chemin d'exécution $\{A, C, D\}$. Cependant, la version *forte* ne l'est pas due au contre-exemple $\{A, B\}$. La version *faible* est de toute évidence une version moins restrictive que la version *forte*. Généralement, pour les propriétés sur le flux de contrôle et de données, il est important que la version *forte* soit satisfaite. En effet, une possibilité d'interblocage n'est jamais une "bonne" situation, et est toujours due à un problème d'erreurs de modélisation. En outre, un problème de *ressource manquante* n'est pas forcément un problème critique. Cela peut simplement retarder l'exécution de l'activité.

Bien que les exemples utilisés ici ont l'air simples, il est important de comprendre la complexité associée à l'interconnexion des éléments du procédé. Un modèle de procédé peut connecter plus d'une centaine d'éléments, avec un flux de contrôle très complexe, ayant différentes dépendances, boucles, synchronisation, contraintes temporelles, et avoir une vision globale du comportement lors de l'exécution est impossible même pour un modélisateur expérimenté.

La prochaine section présente et compare les différentes approches de la littérature concernant la vérification de procédés métiers.

2.3 Comparaison des approches de vérification de procédés métiers

Il existe une abondante littérature sur la vérification de modèles de procédés. Beaucoup de formalismes et de techniques ont été utilisés pour s'assurer de la validité d'un procédé. La technique la plus couramment utilisée pour parvenir à ce but est le *model-checking*. Ceci est principalement dû à son caractère complètement automatique, exhaustif, et son habilité à générer des contre-exemples en cas de non-satisfaction.

Pour appliquer des techniques de *model-checking*, il est nécessaire de représenter la sémantique du PML de manière formelle. Généralement, cette sémantique est représentée dans un des formalismes (ex. réseaux de Petri) pris en entrée par le *model-checker*. Il est important de noter que beaucoup de chercheurs s'accordent à dire qu'un modèle formel doit être directement utilisé

Propriété	Definition
Existence	`A` est exécuté plus de x fois. `A` est exécuté moins de x fois. `A` est exécuté entre x et y fois.
Relation	`A` est exécuté avant `B`. `A` est exécuté après `B`. `A` est exécuté en parallèle de `B`. `A` est exécuté en exclusion de `B`. `A` est exécuté entre `B` et `C`.
ExistenceData	`DataA` est disponible.
RelationData	`DataA` est disponible avant `DataB`. `DataA` est disponible après `DataB`. `DataA` est disponible en parallèle de `DataB`. `DataA` est disponible en exclusion de `DataB`.
RelationDataActivity	`DataA` est disponible avant l'exécution de `A`. `DataA` est disponible après l'exécution de `A`. `DataA` est disponible pendant l'exécution de `A`. `DataA` est disponible en exclusion de l'exécution de `A`. `DataA` est disponible entre l'exécution de `A` et `B`.
ExistenceTimeActivity	`A` est executé avant x unité de temps. `A` est exécuté après x unité de temps. `A` est exécuté entre x et y unité de temps.
ExistenceTimeData	`DataA` est disponible avant x unité de temps. `DataA` est disponible après x unité de temps. `DataA` est disponible entre x et y unité de temps.
ExistenceTimeResource	`ResourceA` est utilisé avant x unité de temps. `ResourceA` est utilisé après x unité de temps. `ResourceA` est utilisé entre x et y unité de temps.
LogicBased	$pred_a \wedge pred_b$ (ex. $Existence[\mathsf{A}] \wedge Existence[\mathsf{B}]$) $pred_a \vee pred_b$ $\neg pred$

Table 2.1 – Propriétés métiers essentielles

comme une base pour un langage de modélisation de procédé [260, 255]. Les raisons les plus courantes sont que (1) les modèles formels ne laissent aucune marge pour l'ambiguïté [251], et (2) ils augmentent les possibilités d'analyse [278]. Selon Sid Askary, *"cela nous permettrait de non seulement raisonner sur la spécification courante et ses problèmes, mais aussi de découvrir des problèmes qui auraient été, dans le cas contraire, inaperçus"* [82]. Dans ce contexte, un grand effort a été concentré sur l'utilisation des réseaux de Petri et des algèbres de processus tels que π-calculus. Cependant, cela a aussi conduit à un long débat pour décider quelle est la fondation formelle la plus appropriée pour les modèles de procédés métiers. Smith soutien le π-calculus et argumente que *"les workflows sont justes π-calculus"* [234]. Par exemple, le langage d'orchestrations des services web BPEL est basé sur π-calculus [164]. D'un autre coté, Van der Aalst a appelé à faire des travaux plus rigoureux pour prouver l'efficacité de π-calculus pour la modélisation de procédés métiers [255], et prône l'utilisation de réseaux de Petri pour la modélisation de procédés métiers [252].

Dans cette section, nous commençons par présenter l'ensemble des critères sur lesquels nous allons comparer les différentes approches de vérification. Puis, nous donnons un aperçu global de l'état de l'art concernant la vérification de modèles de procédés en se concentrant sur les approches formelles utilisant le *model-checking*. Nous présentons aussi un état de l'art concernant la vérification de modèles UML et certaines approches alternatives au *model-checking* pour la vérification de procédés. Enfin, nous présentons et comparons des approches permettant l'expression de propriétés *métiers* sur un procédé. A la fin de chacune de ces sections, un tableau de synthèse permet de *résumer* et de *comparer* les différentes approches entre elles. Finalement, nous présentons une synthèse générale de l'état de l'art à l'égard des problématiques soulevées dans l'introduction de la thèse.

2.3.1 Critères de comparaisons des approches de la littérature

Cette section présente l'ensemble des critères sur lesquels nous allons comparer les approches de vérification de la littérature. Ces critères ont été déduits des problématiques soulevées dans la section 1.1.3 de l'introduction :

(1) Langage de modélisation La première caractéristique sur laquelle il est important de comparer les approches concerne le langage de modélisation utilisé pour définir le procédé. Il existe beaucoup de PMLs tels que UML AD [190], BPMN [192], SPEM [188], EPC [185], ou des langages plus formels tels que les réseaux de Petri [200] et algèbre de processus [16]. Lu *et al.* proposent un survey comparant l'ensemble des PMLs les plus couramment utilisé en terme d'expressivité, flexibilité, adaptabilité, dynamisme et complexité [163] . Il n'est pas question ici de débattre quel est le meilleur langage pour modéliser les procédés selon ces précédents critères, mais de connaitre sur quel langage la technique de vérification a été appliquée.

(2) Formalisme Une autre caractéristique des approches concerne le formalisme utilisé pour effectuer la vérification. Afin d'appliquer des techniques de *model-checking*, il est d'abord nécessaire de donner une sémantique formelle aux langages de modélisation du procédé afin de pouvoir l'analyser par un *model-checker*. Généralement, le modèle (ex. UML AD, BPMN, BPEL ...) est traduit en un formalisme mathématique tel que les automates, réseaux de Petri ou algèbre de processus afin de donner une *sémantique formelle* au langage. Ainsi, cette sémantique peut être associée à des techniques d'*analyse* et de *vérification* qui peuvent être utilisées pour examiner des propriétés d'une spécification. Ce point de comparaison est tout particulièrement important du fait que chacun des formalismes expose différents avantages et inconvénients pour représenter la sémantique du PML [17].

(3) Expressions des propriétés Ce critère détermine la façon dont sont exprimées les propriétés sur le système. Selon les *model-checkers*, les propriétés sont généralement exprimées en utilisant des logiques temporelles (CTL et LTL), la logique de premier-ordre, *failure-divergence* (FDR2) [102], etc. Par exemple, certains *model-checkers* pour les réseaux de Petri proposent seulement la vérification de propriétés CTL [224, 244], quand d'autres proposent aussi LTL [226]. Il n'est pas question ici de déterminer l'expressivité des différentes logiques telle que dans [39], mais de catégoriser pour chaque approche la logique utilisée pour exprimer les propriétés.

(4) Model-Checker Il existe beaucoup de *model-checkers* pour vérifier les différents formalismes existants. Certains formalismes tel que les réseaux de Petri possèdent une surabondance d'outils permettant de les vérifier, quand d'autres proposent un formalisme directement adapté pour un *model-checker* donné (ex. le formalisme Promela et le *model-checkers* SPIN [115]). Comme souligné dans l'étude comparant 6 *model-checker* (Alloy, CADP, FDR2, NuSMV, ProB, SPIN) pour la vérification de systèmes d'informations [89], il est très difficile de comparer des *model-checkers* en terme de facilité d'utilisation pour spécifier le système et les propriétés, ainsi que le nombre d'instances vérifiables. Un cas d'étude peut donner l'avantage à l'un, quand un autre cas donne des résultats contraires. Le point le plus important souligné par les auteurs concerne le fait *"qu'un bon* model-checker *doit être polyvalent"*. Ce critère correspond donc à catégoriser pour chaque approche le *model-checker* utilisé.

(5) Automatisation Ce critère distingue les approches selon les outils proposés pour automatiser le processus de vérification :

(5.1) Traduction automatique La traduction vers le formalisme est-elle faite de manière automatique ou manuelle ? Ce point est essentiel pour automatiser le processus de vérification. Une traduction manuelle, en plus d'être facilement sujet aux erreurs, est particulièrement longue et fastidieuse à effectuer. Ceci est particulièrement vrai pour des modèles de tailles importantes.

(5.2) Intégration dans un environnent Le processus de vérification est-il directement intégré et automatisé dans un environnement de vérification ? Tout se passe-t'il dans l'interface du *model-checker* ? Beaucoup d'approches n'automatisent pas le processus de vérification et se contentent de proposer la vérification *dans* l'interface du *model-checker*. Les *model-checkers* ne sont pas adaptés à la vérification de procédés métiers, mais de systèmes dynamiques au sens général. Beaucoup d'entre eux ne possèdent pas d'interface graphique et s'utilisent seulement en ligne de commande. Les contre-exemples générés par ses outils sont compliqués à comprendre et à analyser du fait qu'ils dépendent grandement de la manière dont la sémantique du PML a été mappée vers le langage formel. Les différents éléments du PML ne sont jamais mappés un-à-un vers un concept du formalisme utilisé par le *model-checker*. Ainsi, en plus de la difficulté d'utiliser le *model-checker* pour un modélisateur lambda, il est difficile d'analyser et de comprendre les contre-exemples générés. Afin de garantir l'adoption des approches de vérification formelle, il est important de "cacher" à l'utilisateur l'utilisation d'un *model-checker* et de lui présenter les contre-exemples de manière graphique, de préférence directement sur le modèle lui-même (ex. en colorant le chemin représentant le contre-exemple). L'utilisateur ne devrait pas pouvoir se rendre compte qu'un *model-checker* est utilisé pour effectuer la vérification.

(5.3) Expressions de propriétés métiers Est-il possible d'exprimer des propriétés métiers ? La plupart des approches proposent de vérifier certaines propriétés génériques (ex. si le procédé est sujet aux interblocages). Cependant, comme expliqué dans la section 2.2.2, les propriétés métiers sont importantes pour s'assurer de certaine contraintes métiers

relatives au projet ou à l'organisation. Certaines approches où tout le processus de vérification se passe dans le *model-checker* revendique la possibilité d'exprimer des propriétés métiers en utilisant, par exemple, la logique LTL ou CTL. Cependant, il est non seulement nécessaire d'avoir une forte connaissance du système sous-jacent, mais aussi un bagage formel poussé afin d'exprimer ce genre de propriétés [41]. Ainsi, ce critère détermine si l'approche propose une technique pour exprimer des propriétés métiers de manière simplifiée. Il est important de noter que nous détaillerons ce critère en lui consacrant une section entière. En effet, la section 2.4 compare les différentes techniques existantes pour exprimer des propriétés métiers sur un procédé.

(6) Perspectives supportées Quelles sont les perspectives du procédé supportées par la formalisation ?

(6.1) Contrôle Est-ce que l'approche supporte les propriétés exprimées sur le flux de contrôle (ex. *impossibilité de complétion*) ?

(6.2) Données Est-ce que l'approche supporte les propriétés exprimées sur le flux de données (ex. *données manquantes*) ?

(6.3) Ressources Est-ce que l'approche supporte les propriétés liées aux ressources (ex. *ressource manquante*) ?

(6.4) Temps Est-ce que l'approche supporte les propriétés liées aux contraintes de temps (ex. *durée du procédé*) ?

Il est important de noter que sur ces critères, nous partons du principe que si les auteurs proposent de vérifier au moins *une* propriété liée à ladite perspective, nous considérons la perspective comme *supportée*. Par exemple, une approche revendiquant la vérification des inter-blocages (*impossibilité de complétion*) sera considérée comme supportant la perspective de contrôle, même si les auteurs ne mentionnent pas forcément la vérification de "*complétion non propre*" ou "*transition morte*". Nous ne rentrons pas dans les détails de chacune des propriétés supportées du fait que nous comparons différents PMLs. En effet, certaines propriétés présentées dans la section 2.2 ne s'appliquent pas forcément sur tous les PMLs.

Malheureusement, certains critères importants tels que le temps nécessaire pour effectuer la vérification ne peuvent être inclus. Ceci est dû à plusieurs facteurs :

- Presque aucune des publications ne donne des informations concernant le temps de vérification.
- La sémantique des PMLs sous-jacents n'étant pas la même, vérifier, par exemple, que le procédé n'est pas sujet aux inter-blocages avantagera de toute évidence les langages possédant seulement les structures de flux de contrôle basique.
- Les formalisations plus simplistes telles que celles supportant seulement le flux de contrôle et faisant abstraction des autres perspectives gagneront un avantage certain sur le temps de vérification.

Dans la suite, nous présentons les principales approches basées sur les réseaux de Petri, les algèbres de processus et la théorie des automates. Un tableau de synthèse reprenant chacun des critères présentés ci-dessus est disponible à la fin de la prochaine section.

2.3.2 Approches basées sur les réseaux de Petri

Un réseau de Petri est un modèle mathématique servant pour la description et l'analyse de systèmes dynamiques tels qu'un système distribué. Un réseau de Petri est un graphe connecté, bipartite et orienté, dans lequel chaque noeud est soit une place soit une transition. Des jetons marquent les places. Quand il y a au moins un jeton dans chacune des places connectées à une

transition (et si les arcs sont marqués par un 1), la transition est dite active. N'importe quelle transition active peut se déclencher en retirant un de ses jetons à partir de chacune de ses places d'entrée, et déposer un jeton dans chacune des places de sortie. Une introduction complète aux réseaux de Petri est disponible dans [209].

Du fait que les réseaux de Petri offrent une notation graphique facilement compréhensible, ils ont été très largement appliqués pour la modélisation et la vérification de procédés, principalement pour trois raisons [251, 262] :

1. Une sémantique formelle en dépit de sa nature graphique.
2. Basé sur des états au lieu d'uniquement des évènements.
3. Abondance de techniques d'analyse.

En effet, beaucoup d'approches de vérification ont leurs racines dans ce formalisme, soit parce que le langage de modélisation du procédé est basé dessus (ex., FunSoft Nets [77], Workflow Nets [252] ou YAWL [258]) soit à travers un mapping vers celui-ci.

Verbeek *et al.* présentent Woflan [271], un outil pour analyser le flux de contrôle de workflows modélisés avec COSA [100], Staffware [203], METEOR [229] et Protos [10]. Le modèle est traduit en réseau de Petri puis est analysé par l'outil qui guide l'utilisateur pour résoudre les erreurs de flux de contrôle de manière itérative en présentant les contre-exemples. Dans [272], Verbeek et Van der Aalst se concentrent sur l'analyse de procédés basés sur BPEL (Business Process Execution Language). Ils présentent un mapping vers la classe de réseau de Petri appellée Workflow Nets [252]. L'analyse est basée sur l'outil Woflan, afin de vérifier des propriétés de terminaison et détecter les noeuds qui ne peuvent jamais être exécutés. Seul un sous-ensemble de BPEL est pris en compte. Dans [194], Ouyang *et al.* proposent un mapping de toutes les constructions basées sur le flux de contrôle de BPEL vers les réseaux de Petri. En conséquence, ces auteurs proposent une sémantique formelle pour BPEL basée sur la sémantique des réseaux de Petri. Dans [193, 194], les auteurs décrivent deux outils qui, s'ils sont utilisés en combinaison, permettent la vérification automatique des procédés basés sur BPEL. L'outil BPEL2PNML est utilisé pour effectuer une traduction de BPEL vers le Petri Net Markup Language (PNML) [27], un format standard pour spécifier des modèles de réseaux de Petri. Le modèle résultant peut ensuite être utilisé par l'outil WofBPEL [193], basé lui aussi sur Woflan. Ainsi, WofBPEL est capable d'analyser les activités inaccessibles, de détecter l'activation simultanée d'activités qui consomme le même type de message, et de déterminer, pour chacun des états possibles de l'exécution du procédé, quel type de message peut-être consommé dans le reste de l'exécution.

Dijkman *et al.* [57] proposent une traduction de BPMN vers les réseaux de Petri. Les auteurs proposent un outil permettant de traduire automatiquement un modèle BPMN dans le format XMI en un PNML. Les auteurs utilisent ensuite le *model-checker* ProM [266] pour vérifier l'absence de tâches mortes ainsi que la complétion propre du procédé.

Awad *et al.* [13] proposent de vérifier des règles de conformités sur un modèle BPMN en utilisant BPMN-Q, un langage visuel basé sur BPMN. Les règles de conformités sont transformées en logique temporelle (LTL) et le modèle est traduit en réseau de Petri en suivant la traduction proposée par Dijkman *et al.* dans [57]. Ensuite, ces deux entités sont données en entrée au *model-checker* LoLa [224] pour vérifier si le procédé les satisfait. Le concept de règles de conformité correspond aux propriétés de type *métiers*. Le langage permet de définir des propriétés métiers sur le flux de contrôle, et les données dans [14]. Afin de vérifier efficacement les propriétés, les auteurs définissent des règles de réduction de graphe [267, 170] afin de combattre le problème d'explosion combinatoire.

Dans [253], Van der Aalst propose une traduction de modèle Event-driven Process Chains (EPCs) vers les réseaux de Petri. Ainsi, l'approche présentée permet de donner une sémantique formelle basée sur les réseaux de Petri aux EPCs. Le but principal est l'analyse de ces modèles en

utilisant les techniques de vérification classiques des réseaux de Petri. En revanche, seul le flux de contrôle est formalisé.

Dans [225], Schmidt *et al.* discutent un mapping à partir de BPEL vers les réseaux de Petri en donnant quelques exemples. Chacune des constructions BPEL est mappée en certains patrons de réseaux de Petri. La transformation complète de BPEL vers les réseaux de Petri est donnée par Stahl dans [237]. Hinze, Schmidt et Stahl [111] décrivent l'outil BPEL2PN qui implémente la transformation automatique en supportant les exceptions et les compensations, mais en faisant abstraction des données. Seul le flux de contrôle est pris en compte. Les détails de cet outil sont présentés par Hinz dans [110]. Le *model-checker* LoLa [224] est ensuite utilisé pour vérifier le réseau de Petri et générer l'espace d'états, c'est-à-dire le graphe d'accessibilité du réseau de Petri. Les propriétés autres que les propriétés classiques liées aux réseaux de Petri sont générées manuellement par les auteurs.

Fahland *et al.* [80] traduisent des diagrammes d'activités UML 2 vers des réseaux de Petri. La sémantique adoptée n'est pas celle de la spécification UML 2, mais celle utilisée par l'outil IBM WebSphere Business Modeler [121], un langage combinant des éléments des diagrammes d'activités UML ainsi que certains concepts de BPMN. Pour chacun des concepts du langage, les auteurs fournissent un réseau de Petri paramétré. Le modèle est traduit en instanciant le réseau de Petri équivalent de chaque élément du modèle, puis en fusionnant les différents fragments pour former un réseau de Petri en accord avec la structure du modèle original spécifiant sa sémantique comportementale. Le réseau de Petri généré peut ensuite être vérifié pour des erreurs de flux de contrôle en utilisant le *model-checker* LoLa [224]. Dans un travail plus récent [81], de nombreux procédés d'affaires industriels ont été testés avec succès sur les propriétés de flux de contrôle en utilisant le *model-checker* LoLa [224], ainsi qu'en utilisant l'outil de diagnostics Woflan [271]. Afin d'améliorer la vitesse de vérification, ces auteurs découpent les procédés en sous-fragments, en utilisant la décomposition SESE (Single Entry Single Exit) [269], afin d'analyser de multiples "petits" procédés au lieu de l'analyser dans sa globalité. En effet, la vérification à l'aide de technique de *model-checking* étant un problème NP-Complet, cette stratégie montre qu'il est possible de vérifier des propriétés structurelles sur le flux de contrôle de manière très efficace en ayant recours à cette décomposition.

Dans [134], le procédé modélisé en diagramme d'activités UML 2 est mappé vers un réseau de Petri coloré pour permettre sa vérification automatique. Les auteurs proposent des règles de traduction et un algorithme de transformation pour traduire le modèle. En revanche, aucune chaîne d'outil n'est présentée ; la vérification se passe dans le *model-checker* CPN Tools [244], et le type de propriétés supporté n'est pas mentionné. Le cas d'étude présente un diagramme d'activités représentant le retrait d'argent dans un ATM. Cependant, aucune vérification n'est effectuée, seuls plusieurs cas sont simulés (retrait normal, code pin invalide...).

Dans [238], les auteurs traduisent un diagramme d'activités UML 2 dans le framework Fundamental Modeling Concepts (FMC) afin d'avoir une notation compacte pour éviter la complexité engendrée par la traduction, et faciliter la visualisation par les participants [143]. Ensuite, cette représentation intermédiaire est transformée en un réseau de Petri coloré à des fins d'analyse et d'exécution à travers CPN Tools [244]. Les étapes de transformations font abstraction complète des noeuds objets du diagramme d'activités. Les auteurs ne présentent pas les types de propriétés vérifiés, mais se limitent aux possibilités offertes par l'outil CPN Tools (ex. invariant place/transition, accessibilité, etc).

Thierry-Mieg *et al.* [243] propose de vérifier un modèle UML en le traduisant en Instantiable Petri Nets (IPN), un formalisme basé sur les réseaux de Petri incluant le concept de *type*, d'*instance*, de *hiérarchie* en utilisant la notion de *synchronisation de transition* pour composer le comportement. La traduction se concentre sur le flux de contrôle des diagrammes d'activités UML et supporte la composition hiérarchique (c'est-à-dire l'utilisation de `CallBehaviorAction` pour

exécuter des sous-activités). Ainsi, il est possible de vérifier les erreurs classiques des réseaux de Petri ainsi que d'analyser des erreurs de flux de contrôle (interblocage et possibilité de complétion).

Huai *et al.* [120] présentent une méthode pour vérifier des modèles BPMN basés sur les réseaux de Petri temporels [205]. Premièrement, les auteurs traduisent le modèle BPMN en réseau de Petri temporel. Ensuite, ils construisent le graphe d'accessibilité du réseau de Petri afin de vérifier des propriétés de flux de contrôle ainsi que les conflits de temps. Les auteurs n'utilisent pas un *model-checker* mais implémentent en Java un module de vérification construisant le graphe d'accessibilité pour vérifier les propriétés. La vérification des propriétés est donc "encodée" dans l'algorithme. Aucune métrique n'est donnée par les auteurs permettant de juger de l'efficacité de leur approche, et il semble difficile d'ajouter de nouvelles propriétés à vérifier sans modifier le code Java.

L'utilisation directe des réseaux de Petri classiques [201] pour la *modélisation* de procédés a été depuis longtemps plébiscitée [264]. L'avantage de l'utilisation de langages de modélisation basés sur les réseaux de Petri est la possibilité intrinsèque d'être analysés en utilisant les techniques classiques de vérification de réseaux de Petri. En revanche, leurs utilisations n'ont pas donné le résultat désiré. En effet, la description de procédés basés sur les réseaux de Petri tend à devenir très complexe et extrêmement large. De plus, les réseaux de Petri classiques ne permettent pas de modéliser les données et le temps. De ce fait, l'utilisation de réseaux de Petri de haut niveau, c'est-à-dire des réseaux de Petri augmentés avec la notion de données, temps et hiérarchie [268] a été investiguée [263]. Cependant, l'effort de modélisation engendré par l'utilisation de ces formalismes n'est pas adapté à la modélisation de procédés [263]. De ce fait, Van der Aalst *et al.* introduit dans [252] les Workflow Nets, une classe particulière de réseaux de Petri dédiée à la conception de workflow avec une notation graphique augmentée. Dans [261], l'auteur définit la notion de "*soundness*", catégorisée dans cette thèse comme des propriétés de flux de contrôle, et montre comment analyser les Workflow Nets vis-à-vis de ces propriétés en utilisant les techniques classiques des réseaux de Petri. Plus particulièrement, les auteurs utilisent des techniques classiques telles que le graphe de couverture [181], de l'analyse structurelle d'invariants [254, 271], ainsi que des règles de réductions [23]. Ling *et al.* [161] propose des "Time Workflow Nets", un type spécial de Workflow Nets étendu avec des intervalles de temps sur les activités. Ces auteurs ont adopté la sémantique des Time Petri Net (TPN) [24] pour modéliser le temps dans le système. En revanche, aucune implémentation n'est proposée démontrant leurs analyses. Dans [246, 247], des "Workflow Nets with Data", un type spécial de Workflow Nets avec des annotations concernant la manipulation des données sont vérifiés par rapport à différents problèmes de données exprimées en LTL et CTL. Les auteurs ne fournissent pas d'implémentation, mais proposent de simplement choisir parmi les nombreux *model-checker* de réseaux de Petri disponibles, tels que Kit [226] qui supportent les logiques LTL et CTL, ou CPN Tools [244], un puissant framework pour modéliser et analyser les réseaux de Petri colorés avec la possibilité d'analyser des propriétés CTL.

Dans [258], Van der Aalst *et al.* ont proposé YAWL (Yet Another Workflow Language) basé sur les premiers travaux sur les Workflow Nets, pour supporter toujours plus de workflow patterns [259]. Selon eux, YAWL est plus adapté que les réseaux de Petri de haut niveau dans le sens qu'il supporte directement l'utilisation de plusieurs *workflow patterns* qui sont difficiles à exprimer directement avec les réseaux de Petri. Les auteurs prétendent que YAWL est suffisamment expressif pour mapper la plupart des langages de modélisation de procédés (ex. BPMN, UML AD, EPCs, BPEL) vers YAWL sans perte d'informations concernant les détails du flux de contrôle. En revanche, YAWL est pour le moment limité à la perspective du flux de contrôle. Dans [283], Wynn *et al.* montrent comment vérifier le langage YAWL. La vérification se concentre sur les propriétés de flux de contrôle, en incluant des structures de contrôle avancés comme les régions d'annulation (ex. pour gérer les exceptions), et prend en compte certaines optimisations telles que des règles de réductions pour améliorer la vitesse de vérification.

Du fait que les réseaux de Petri jouissent d'une notation graphique facilement compréhensible ainsi que d'une surabondance d'outils matures permettant leur analyse de façon efficace, ils ont été très largement appliqués dans le domaine de la modélisation et de l'analyse de procédés. Cependant, même si vérifier les procédés basés sur le formalisme des réseaux de Petri est efficace pour examiner des propriétés comme l'accessibilité et la vivacité, les réseaux de Petri échouent quand le système a besoin de gérer une grande variété de données. En effet, l'utilisation de données sur le système multiplie le nombre de places et introduit des problèmes d'explosion de l'espace d'états rendant leur analyse impossible. De plus, les réseaux de Petri classiques n'ont pas la capacité d'intégrer les tâches, les agents, les ressources et les contraintes entre ces éléments de façon naturelle. Les réseaux de Petri de hauts niveaux ont la capacité d'intégrer ces éléments, mais leur complexités pour représenter les différents *workflow pattern* rencontrés dans les procédés rend leurs utilisations complexes. Aussi, la majorité des approches reposant sur les réseaux de Petri se concentrent sur la vérification des propriétés de flux de contrôle.

Dans la suite, nous présentons les principales approches basées sur les algèbres de processus.

2.3.3 Approches basées sur les algèbres de processus

D'autres approches utilisent les algèbres de processus, une théorie stricte et bien établie prenant en charge la vérification automatique de propriétés comportementales des systèmes au même titre que les réseaux de Petri. Beaucoup d'algèbre de processus ont été introduites incluant, par exemple, les Calculus of Communicating Systems (CCS) [175], Communicating Sequential Processes (CSP) [114], LOTOS [28], et π-calculus [176]. Les algèbres de processus sont généralement modélisées par le moyen d'un Labelled Transition Systems (LTS). La relation de transition du LTS est généralement définie par une collection d'axiomes et de règles. Une introduction complète aux algèbres de processus est disponible dans [22]. Dans la suite, nous discutons des principales approches basées sur les algèbres de processus pour vérifier un modèle de procédés métier.

Dans [279], les auteurs montrent comment l'algèbre CSP [114] peut être appliqué pour modéliser des systèmes de workflow complexes. Ils utilisent le *model-checker* FDR (Failures-Divergences Refinement) [102] pour vérifier automatiquement certaines propriétés comportementales sur un procédé modélisé avec BPMN. Les propriétés sont exprimées en utilisant des *affirmations* CSP et peuvent être exprimées sur le flux de contrôle seulement. Par la suite, ces auteurs généralisent l'approche en proposant une sémantique formelle d'un sous-ensemble BPMN en utilisant la syntaxe abstraite de Z [282] et la sémantique du langage CSP [280]. Enfin, ils augmentent leur modèle formel avec la notion du temps relatif pour gérer les délais et durées du procédé [281]. Ainsi, le *model-checker* FDR [102] est utilisé par ces auteurs pour vérifier si le procédé est libre d'interblocages et si aucune incompatibilité de temps n'est présente. Cependant, aucune chaîne d'outils n'est proposée pour vérifier un modèle de façon automatique.

Dans [162], Liu *et al.* transforment les modèles exprimés en BPEL en π-calculus. Puis, le modèle π-calculs est transformé en Finite State Machine (FSM) afin de pouvoir être utilisé en entrée par un *model-checker*. Selon ces auteurs, l'avantage de la transformation intermédiaire est de pouvoir avoir l'opportunité d'appliquer d'autres techniques de vérification comportementale proposées par π-calculus comme les interblocages et la bisimulation. Ils capturent aussi des règles de conformité dans la notation graphique Business Property Specification Language (BPSL, présenté en section 2.4) et les transforment automatiquement en logique temporelle. Cette approche permet de vérifier les propriétés de type flux de contrôle et des propriétés *métier* exprimées avec leur langage.

Xu *et al.* [284] proposent une sémantique formelle à UML AD 2 en utilisant le langage CSP. Les auteurs ont implémenté un outil pour permettre la transformation automatique de AD 2 vers CSP. Ainsi, le modèle peut être vérifié à travers le *model-checker* FDR. Les auteurs ne précisent pas quel type de propriétés il est possible de vérifier par rapport à leur formalisation,

mais simplement l'analyse liée au *model-checker* FDR (Traces refinement, Failures-divergence refinement et Failures refinement).

Abdelhalim *et al.* [4, 1] présentent une approche pour transformer un modèle UML/fUML (un modèle contenant certains éléments non compris dans le sous-ensemble fUML tels que les diagrammes d'états) dans le langage CSP et utilisent le *model-checker* FDR pour vérifier si le modèle n'est pas sujet aux interblocages. Quand un interblocage est trouvé, une trace représentant ce contre-exemple est générée. Leurs formalisations se concentrent principalement sur la communication asynchrone entre les objets fUML, qui a été guidée par leur cas d'étude. Dans [1], les auteurs présentent la partie outil intégrée a MagicDraw pour vérifier leurs modèles. Les auteurs proposent de "rejouer" le contre-exemple de la même façon que l'on débogue un modèle afin d'avoir un retour visuel en animant le modèle. Dans [2], ils tentent de combattre les problèmes d'explosion combinatoire lors de la vérification en générant un modèle CSP "optimisé", c'est-à-dire en suivant certaines règles de formalisation et en s'interdisant d'utiliser certains éléments du langage CSP qui sont plus couteux en terme de temps de vérification. Finalement, dans [3], les auteurs présentent leurs chaînes d'outils intégrées à MagicDraw en présentant les règles de transformation du modèle UML vers CSP. Une des particularités de leur approche provient de la façon dont les contre-exemples sont présentés dans cet outil : pour chacun d'entre eux, un diagramme de séquence UML est généré représentant l'exécution menant au contre-exemple. Leur approche se concentre principalement sur la vérification d'interblocage.

La principale différence entre les réseaux de Petri et les algèbres de processus est que les réseaux de Petri sont basés sur un graphe biparti quand les algèbres de processus sont basés sur une description textuelle. Les deux domaines jouissent d'une impressionnante accumulation de connaissances. Beaucoup de notions développées pour les réseaux de Petri ont été transmises aux algèbres de processus et vice versa. Cependant, quelques différences fondamentales persistent. Par exemple, la notion d'invariants développée pour les réseaux de Petri n'existe pas dans les algèbres de processus. Une description plus détaillée des différences entre ces deux modèles mathématiques est disponible dans la thèse de Basten [17].

En outre, les automates et les réseaux de Petri sont souvent utilisés pour modéliser un système *fermé*, dont le comportement est complètement déterminé et contrôlé *seulement* par l'état du système. Cependant, les algèbres de processus sont conçus pour modéliser des systèmes *ouverts* de communication dont le comportement est défini par l'état du système *et* les interactions avec l'environnement [8]. En effet, Liu *et al.* [162] argumentent que bien souvent un procédé a besoin de feedback interactif de l'environnement du procédé, et prône l'utilisation de π-calculus. Un de ses avantages proviendrait de sa mobilité et compositionnalité, c'est-à-dire, qu'il est possible de modéliser un système à partir de ses sous-composants en utilisant l'opérateur naturel de composition proposé par π-calculus. Cet opérateur n'est pas disponible dans les réseaux de Petri, ce qui implique l'utilisation d'opérations et calculs additionnels pour gérer la composition. π-calculus est théoriquement sûr, supporte l'analyse de bisimulation ainsi que le *model-checking*. De plus, il bénéficie d'une popularité croissante en industrie et possède de nombreux outils le supportant. En revanche, Van der Aalst a publié dans une courte note [255], une comparaison entre les réseaux de Petri et π-calculus et montre un exemple simpliste de réseau de Petri demandant une expression complexe de π-calculus. Ainsi, π-calculus est considéré comme *"un langage pour experts dans lequel les choses simples deviennent rapidement très compliquées"* [255]. En effet, il semblerait que les algèbres de processus comme π-calculus sont limités dans la capacité à supporter directement la plupart des workflows patterns [259] utilisés dans les procédés [255].

Dans la suite, nous présentons les principales approches basées sur la théorie des automates.

2.3.4 Approches basées sur la théorie des automates

Il existe aussi des approches basées sur la théorie des automates. Généralement, le *model-checker* prend en entrée la représentation d'une structure de Kripke [42], c'est-à-dire un automate non déterministe composé par un ensemble d'états, d'actions, de transitions entre les états, et un état initial pour représenter la dynamique d'un système. Une introduction complète sur la théorie des automates est disponible dans [117].

Eshuis [78] vérifie des diagrammes d'activités UML dans le contexte de la modélisation de workflow en transformant le modèle dans le langage d'entrée de NuSMV [37], un *model-checker* symbolique. Le travail a été réalisé avant la finalisation de la spécification UML 2.0, ainsi beaucoup d'hypothèses et de choix ont été faits pour définir la sémantique donnée aux diagrammes d'activités UML. Eshuis utilise un Clocked Transition System (CTS) [167, 138] pour définir la sémantique. Les propriétés de flux de contrôle et de temps sont exprimables, en revanche l'auteur fait l'hypothèse que les ressources sont toujours disponibles pour exécuter les activités, ainsi il n'est pas possible d'exprimer les propriétés liées aux ressources. La sémantique prend en compte les données sur le diagramme, mais contrairement aux diagrammes d'activités UML 2 où le flux de données est modélisé *explicitement* sur le diagramme (à travers l'utilisation d'`ObjectNode` et d'`ObjectFlow`), Eshuis permet simplement aux actions du diagramme d'écrire et de lire des variables globales (de type entier).

Dans [105], Guelfi *et al.* proposent une transformation des diagrammes d'activités UML dans le langage Promela (Process ou Protocol Metal Language) afin de vérifier des propriétés comportementales avec le *model-checker* SPIN [115]. Ces auteurs ont aussi adopté un Clocked Transition System pour formaliser la sémantique de UML AD. Le type de propriétés supportées n'est pas présenté par les auteurs, bien qu'il semblerait qu'il soit possible de vérifier des propriétés de type flux de contrôle et de temps. Ni les ressources ni les données ne sont prises en compte. De plus, les auteurs ne proposent pas d'implémentation ni d'évaluation.

Dans [169], Mauw *et al.* vérifient une spécification Architectural Modelling Box for Enterprise Redesign (AMBER) [72] en la transformant automatiquement en une machine à état dans le langage Promela, le langage d'entrée du *model-checker* SPIN. Des propriétés comportementales sur le flux de contrôle et les données sont exprimés en LTL.

Dans [144], les auteurs se concentrent sur l'aspect des données des procédés BPMN en faisant une abstraction de certaines plages de données (jugées redondantes et non nécessaires) afin de combattre le problème d'explosion combinatoire. Leur implémentation transforme automatiquement le modèle et le transmet au *model-checker* SAL [54] qui retourne un contre-exemple en cas de problèmes.

Watahiki *et al.* [274] proposent une spécification formelle de BPMN avec des automates temporisés. Premièrement, les auteurs étendent BPMN pour gérer des contraintes temporelles (le temps minimal et maximal pour exécuter une activité) ainsi que des contraintes de concurrence (ex. le nombre maximal d'instances exécutables en parallèle). Puis, ils fournissent un mapping automatique de leur extension de BPMN vers les automates temporisés. Ils utilisent des formules CTL pour vérifier les différentes propriétés avec le *model-checker* UPPAAL [150]. UPPAAL permet d'exprimer seulement un sous-ensemble de CTL. L'ensemble des propriétés supportées n'est pas défini par les auteurs, seul un exemple de vérification d'interblocage et d'une contrainte temps réel est proposée. De plus, seul un petit sous-ensemble des éléments BPMN est pris en compte.

Chen *et al.* [34] decrivent comment la vérification à états finis peut être utilisée pour trouver des erreurs dans des procédés médicaux. Les procédés sont modélisés avec Little-JIL [32], un langage exécutable de haut niveau avec une syntaxe et une sémantique opérationnelle formelle pour définir la coordination d'un procédé. Afin de spécifier les propriétés, les auteurs utilisent Propel [235] (détaillé en section 2.4). Afin d'effectuer la vérification, le modèle est d'abord

traduit dans la représentation BIR (Bandera Intermediate Representation) [122], un langage pour décrire des systèmes à états finis avec pour but d'être facilement traductible dans le langage d'entrée de nombreux *model-checker*. Ainsi, cette représentation est traduite vers le langage d'entrée de Flavers FLAVERS (FLow Analysis for VERification of Systems) [67] et de SPIN [115] afin de pouvoir analyser la satisfiabilité des propriétés exprimées sur le flux de contrôle.

Dans la suite, nous présentons les autres approches de vérification alternative au *model-checking*.

2.3.5 Autres approches de vérification

Sadiq *et al.* [222] argumentent que la fondation d'un modèle de procédé repose sur sa spécification *structurelle*. Les auteurs proposent un ensemble de règles de réduction pour vérifier des problèmes comportementales tels que des interblocages et manques de synchronisation (un comportement non voulu où une activité peut s'exécuter plusieurs fois). L'idée de base de l'approche est de supprimer toutes les parties correctes du graphe représentant le procédé. Dans le cas où le procédé est correct, la réduction entraîne un graphe vide. Cependant, Van der Aalst [256] démontre plus tard que l'ensemble des règles de réduction proposées par les précédents auteurs n'est pas complet. Il présente un exemple de modèle de procédé correct, mais non réductible avec l'ensemble des règles proposées par Sadiq *et al.*. L'alternative proposée par Van der Aalst consiste à traduire le procédé en Workflow Nets, afin de pouvoir l'analyser en utilisant les outils de vérification des réseaux de Petri.

D'autres approches proposent de détecter des erreurs à travers l'utilisation de scénarios de tests [35, 147] et simulation [137, 130, 119, 45]. Cependant, un procédé peut avoir une infinité de scénarios due aux boucles et aux données ; il n'est donc pas possible de tout tester. En effet, le principal désavantage des tests est que, comme souligné par Diskstra dans [58], ils montrent la présence d'erreurs, mais pas leurs *absences*. En d'autres termes, si le procédé passe tous les tests, il est toujours incertain si le procédé contient toujours des erreurs ou non. Ce problème est une des principales raisons de l'utilisation de techniques de *model-checking* : effectuer une vérification automatique et surtout exhaustive du modèle.

Dans la prochaine section, nous présentons un état de l'art concernant la vérification de modèle UML en utilisant les méthodes formelles.

2.3.6 UML et les méthodes formelles

Beaucoup d'approches ont été proposées pour vérifier formellement des modèles UML. La plupart des approches de la littérature se concentrent sur la vérification des diagrammes d'états, et peu d'approches ont été proposées pour vérifier les diagrammes d'activités. Dans la suite, nous présentons l'état de l'art concernant la vérification de ces deux types de diagrammes.

Diagramme d'états

Beaucoup d'approches ont été proposées ces 15 dernières années pour vérifier les diagrammes d'états UML [151, 159, 160, 60, 223, 227, 228, 184, 286, 287, 285].

Latella *et al.* vérifient ces diagrammes en les encodant en automates hiérarchiques (HA, pour Hierarchical Automata) [174] à partir desquels ils génèrent une spécification pour le *model-checker* SPIN [151]. Lilius *et al.* proposent un outil appellé vUML, pour la vérification automatique de modèle UML, en se concentrant sur les diagrammes d'états [159, 160]. De la même manière, ces auteurs transforment le modèle dans le langage d'entrée du *model-checker* SPIN. VeriUML [227] est un ensemble d'outils développés par l'université du Michigan permettant de vérifier la syntaxe et le comportement d'un diagramme d'états UML en utilisant un *model-checker* basé sur le langage

SMV [228]. Zhao *et al.* [287] proposent de vérifier la cohérence entre les diagrammes de séquences et les diagrammes d'états en utilisant le *model-checker* SPIN. Ng *et al.* [183] traduisent un modèle UML (diagramme de classe et d'états) dans le langage CSP pour permettre sa vérification formelle en utilisant le *model-checker* FDR.

Une comparaison et une catégorisation plus complète des approches de vérification des diagrammes d'états sont disponibles dans [44].

Diagramme d'activités

Depuis la refonte des diagrammes d'activités UML dans la version 2.0 de la spécification, la syntaxe abstraite et la sémantique des diagrammes d'activités ont radicalement changé. Beaucoup de nouvelles structures ont été introduites pour améliorer l'expressivité et la concision de ces diagrammes. Contrairement à UML AD 1, les diagrammes d'activités UML 2 ne sont plus basés sur un type spécial de diagramme de machine à états, mais comme un graphe orienté basé sur une sémantique de règles de circulation de jetons entre les nœuds, inspirés par les réseaux de Petri colorés.

Sémantique opérationnelle. Certaines approches proposent, de la même façon que le standard fUML, des sémantiques opérationnelles pour AD 2.0 [273, 146, 45, 140]. Par sémantique opérationnelle, nous entendons d'une sémantique d'*exécution*, qui est souvent définie en utilisant du code. Ces sémantiques n'étant généralement pas basées sur un langage ayant des fondations formelles, il n'est pas possible d'utiliser directement ces sémantiques opérationnelles pour faire du *model-checking*. Vitolins *et al.* [273] proposent une sémantique opérationnelle à UML AD 2.0 en utilisant un mix entre pseudocode et des pré- et post-conditions OCL. La sémantique proposée est basée sur des règles selon lesquelles les jetons peuvent transiter entre les noeuds. L'intérêt principal de leur approche est la définition d'une machine virtuelle afin d'avoir un moteur d'exécution et de simulation de UML AD. Korherr *et al.* [146] proposent une sémantique d'exécution pour UML AD 2.0 en le traduisant vers BPEL. Crane *et al.* [45] presentent un interpréteur sous la forme d'une machine virtuelle pour les actions et activités de UML 2.0. L'interpréteur supporte seulement un sous-ensemble de UML AD (séquence, fork, join, merge, decision) et peut être utilisé pour simuler et exécuter un UML AD. L'exécution peut être faite de manière aléatoire ou guidée par l'utilisateur. En générant de nombreux chemins d'exécution, les auteurs proposent d'analyser certaines erreurs par rapport au chemin généré. Du fait que la génération des chemins n'est pas forcément exhaustive, les analyses effectuées par l'outil ne sont pas garanties d'être *sûres*. Kirshin *et al.* [140] décrivent une architecture pour implémenter un moteur d'exécution de modèles génériques pour permettre la simulation de modèle. Ils implémentent cette architecture pour UML AD. En revanche, très peu de détails sont donnés sur la sémantique opérationnelle adoptée.

Sémantique formelle. D'autres approches proposent des sémantiques basées sur un formalisme mathématique (ex. réseaux de Petri), permettant ainsi d'appliquer des techniques de *model-checking*. Concernant la vérification des diagrammes d'activités UML 2.0, peu d'approches ont été proposées, mais tous les précédents formalismes ont été utilisés : (1) les algèbres de processus en utilisant π-calculus [61] et CSP [284, 4], (2) les automates en utilisant les formalismes de NuSMV [79, 87] et de Promela [105], et (3) les réseaux de Petri à travers une transformation [80, 134, 238, 243]. Ces approches ont été détaillées dans les sections précédentes.

Il est important de noter que seuls les travaux de Abdelhalim *et al.* [4, 1, 2, 3] sont basés sur la sémantique standard fUML. En revanche, leur formalisation a été guidée par le besoin de leurs cas d'étude. Ainsi, elle se concentre sur la communication asynchrone entre les objets fUML et la

vérification des interblocages. Il n'est pas possible de vérifier l'ensemble des propriétés présentées en section 2.2 sur l'ensemble des perspectives d'un procédé.

2.3.7 Synthèse comparative des approches de vérification

Dans cette section, nous synthétisons les caractéristiques principales des approches que nous avons présentées et discutons leurs forces et faiblesses. La comparaison des différentes approches se fait sur la base de l'ensemble des critères définis dans la section 2.3.1 :

1. Qui sont les auteurs ?
2. Quel est le langage de modélisation ?
3. Quel est le formalisme utilisé pour faire la vérification ?
4. Comment sont exprimées les propriétés ?
5. Quel est le *model-checker* utilisé ?
6. Concernant l'automatisation de la vérification :
   (a) La traduction vers le formalisme est-elle automatique ?
   (b) La vérification est-elle intégrée et automatisée dans un environnement de vérification (tout ne se passe pas simplement dans l'interface du *model-checker*) ?
   (c) Est-il possible d'exprimer des propriétés métiers ?
7. Quelles sont les perspectives du procédé (flux de contrôle, données, ressources et temps) supportées par la formalisation ?

Le tableau 2.2 présente la synthèse des approches de la littérature. Chacune des colonnes correspond respectivement aux questions présentées dans le paragraphe précédent. La majorité des approches se basent sur les réseaux de Petri. Ce choix est souvent vu comme naturel du fait que la sémantique de la plupart des langages de modélisation de procédés sont basés sur un flux d'exécution d'activités, similairement à la sémantique des réseaux de Petri. En revanche, il est surprenant de voir que l'état de l'art actuel se concentre principalement sur la vérification de propriétés liées à la perspective du flux de contrôle et néglige les autres perspectives. En dépit du grand nombre d'approches pour vérifier des propriétés comportementales sur un procédé, aucune d'entre elles ne propose de vérifier les propriétés liées aux ressources. Comme expliqué en section 1.1.3, certaines propriétés métiers nécessitent la prise en compte de toutes les perspectives du procédé de manière unifiée. Par exemple, la propriété "est-il possible de terminer le procédé `ProcessOrder` en moins de 8 heures, en produisant la facture (`invoice`), sans utiliser la ressource `BankConnector` ?" par rapport au procédé de la figure 1.3 nécessite prise en compte de toutes les perspectives de manière combinée. Cependant, aucune des approches de la littérature ne supporte ce type de propriétés.

Si une comparaison doit être faite avec la plupart des approches de la littérature, dans cette thèse, notre approche ne repose pas sur la sémantique et les concepts d'un langage formel donné en terme d'expressivité (ex. les réseaux de Petri), mais directement sur la sémantique du langage de modélisation. Dans le chapitre 3, nous donnons cette sémantique en logique de premier ordre en nous basant sur les notions d'*états*, *activation*, et *tir* de transitions. Dans les approches mentionnées précédemment, les auteurs partent de l'hypothèse que les choix sémantiques faits par ces langages formels sont aussi valides pour le langage de modélisation. Bien que cela puisse être le cas, cette problématique n'est jamais discutée ouvertement par ces auteurs. La seule exception étant le travail d'Eshuis [78] dans lequel cette problématique est longuement discutée par rapport aux choix de la sémantique de UML AD.

Concernant la vérification de diagrammes d'activités UML, il n'existe pas pour le moment d'approches permettant de vérifier un procédé en se basant sur la sémantique standard fUML.

Approche	Langage	Formalisme	Propriétés	Model-Checker	Outil			Perspectives				Notes
					Traduction automatique	Integration	Expr. prop.	Contrôle	Données	Ressources	Temps	
Réseaux de Petri (RdP)												
Verbeek *et al.* [271]	COSA, Staffware, METEOR, Protos	RdP	Propriétés classiques RdP	Woflan, erreur affiché dedans	✓	✓	-	✓	-	-	-	
Verbeek *et al.* [272, 194, 193]	BPEL	RdP	Propriétés classiques RdP	Woflan	✓	✓	-	✓	-	-	-	
Dijkman *et al.* [57]	BPMN	RdP	Propriétés classiques RdP	ProM	✓	-	-	✓	-	-	-	
Awad *et al.* [13, 14]	BPMN	RdP	Propriétés classiques RdP	LoLa	✓	-	✓	✓	✓	-	-	
Van der Aalst *et al.* [253]	EPC	RdP	Propriétés classiques RdP	Woflan	-	-	-	✓	-	-	-	
Schmidt *et al.* [225, 111]	BPEL	RdP	Propriétés classiques RdP, CTL	LoLa	✓	-	-	✓	-	-	-	
Fahland *et al.* [80]	UML AD	RdP	Propriétés classiques RdP, CTL	LoLa	✓	-	-	✓	✓	-	-	Sémantique de IBM WebSphere pour UML AD, flux de données limité à une multiplicité de [0,1]
Jung *et al.* [134]	UML AD	RdP coloré	Propriétés classiques RdP	CPN Tools	✓	-	-	✓	✓	-	-	aucune vérification proposé par les auteurs, juste de la simulation
Staines *et al.* [238]	UML AD	RdP colorés	Propriétés classiques RdP	CPN Tools	✓	-	-	✓	-	-	-	
Thierry-mieg *et al.* [243]	UML AD	Instantiable PN	Propriétés classiques RdP	CPN-AMI	✓	✓	-	✓	-	-	-	
Huai *et al.* [120]	BPMN	Time PN	Algorithme pour (1) Tache morte (2) interblocages (3) boucles et (4) conflits de temps	Java implémentation	✓	✓	-	✓	-	-	✓	
Trcka *et al.* [246, 247]	Workflow Nets with Data	RdP	LTL, CTL	Kit, CPN Tools	-	-	-	✓	✓	-	-	
Wynn *et al.* [283]	YAWL	RdP	Propriétés classiques RdP	Woflan	✓	✓	-	✓	-	-	-	

Approche	**Langage**	**Formalisme**	**Propriétés**	**Model-Checker**	**Outil**			**Perspectives**				**Notes**
					Traduction automatique	Integration	Expr. prop.	Controle	Données	Ressources	Temps	
Algèbres de processus												
Wong *et al.* [279, 280, 281]	BPMN	CSP	FDR affirmations	FDR	-	-	-	✓	-	-	✓	
Liu *et al.* [162]	BPEL	π-calculus puis FSM	LTL	NuSMV 2	✓	✓	✓	✓	-	-	-	
Xu *et al.* [284]	UML AD	CSP	FDR affirmations	FDR	✓	-	-	✓	-	-	-	
Abdelhalim *et al.* [4, 1, 2, 3]	UML AD	CSP	FDR affirmations	FDR	✓	✓	-	✓	-	-	-	
Automates												
Eshuis *et al.* [78]	UML AD	SMV	LTL	NuSMV$_{fair}$	✓	✓	$\sim$	✓	✓	-	✓	Les ressources sont toujours disponible, beaucoup d'hypothèses sur la sémantique de UML AD (entre la version 1.4 et 2 de la spécification UML)
Guelfi *et al.* [105]	UML AD	Promela	LTL	SPIN	-	-	-	✓	-	-	✓	
Mauw *et al.* [169]	AMBER	Promela	LTL	SPIN	-	-	-	✓	✓	-	-	
Knuplesch *et al.* [144]	AristaFlow model	SAL language	LTL	SAL	✓	✓	✓	✓	✓	-	-	
Watahiki *et al.* [274]	BPMN	automates temporisés	CTL	UPPAAL	✓	-	-	✓	-	-	✓	
Chen *et al.* [34]	Little-JIL	Flavers language, Promela	LTL	Flavers, SPIN	✓	✓	✓	✓	-	-	-	
Approche de cette thèse												
Laurent *et al.*	UML AD	First-Order Logic	LTL	Alloy	✓	✓	✓	✓	✓	✓	✓	Sémantique de UML basé sur fUML

TABLE 2.2 – Synthèses des approches de *model-checking* pour vérifier un procédé

2.4 Expression de propriétés métiers sur un modèle de procédé

Cette section présente les principales approches de la littérature permettant d'exprimer, de manière simplifiée, des propriétés métiers sur un modèle de procédé. En effet, l'expression de propriétés temporelles demande l'utilisation de formalismes qui ne sont pas adaptés aux compétences d'un modélisateur de procédé. Ainsi, une approche où la complexité du formalisme sous-jacent est cachée et simplifiée par une méthode d'expression plus haut niveau est souhaitable.

2.4.1 Revues de la littérature

Ly *et al.* [166] introduisent des règles de conformité exprimables sur un procédé. Ces règles correspondent à ce que l'on appelle dans cette thèse les propriétés métiers. Les règles sont exprimées sous forme de graphes pour exprimer des contraintes de relation entre les activités (ex. `A` doit être exécuté avant `B`). La sémantique des règles est définie en logique de premier ordre par rapport à une trace d'exécution, pour rester indépendante des langages de modélisation de procédé et de leurs sémantiques d'exécution respectives.

Liu *et al.* [162] proposent un langage graphique de spécification de propriétés métiers appellé Business Property Specification Language (BPSL). BPSL est basé sur LTL et fournit des notations visuelles pour ses opérateurs logiques. BPSL peut être vu comme un DSL graphique pour représenter une expression LTL. De plus, il définit des opérateurs pour les modèles logiques récurrents trouvés dans les procédés métiers.

Awad *et al.* [13] proposent BPMN-Q, un langage visuel basé sur BPMN pour spécifier des règles de conformités. Ces règles sont transformées en logique temporelle (LTL) afin de pouvoir être vérifiées par la suite par un *model-checker*. BPMN-Q est défini pour exprimer des règles sur le flux de contrôle du procédé, mais aussi sur les données comme spécifiées dans [14].

Les travaux de Forster *et al.* dans [85, 86, 87] définissent un ensemble de patrons visuel en utilisant ce qui est appelé Process Pattern Specification Language (PPSL). Ces patrons sont utilisés pour exprimer des contraintes poue un diagramme d'activités. Les patrons PPSL sont ensuite traduits en formules PLTL (Past LTL). Les auteurs utilisent ensuite le *model-checker* NuSMV pour vérifier la propriété sur le système de transition représenté par le diagramme d'activités.

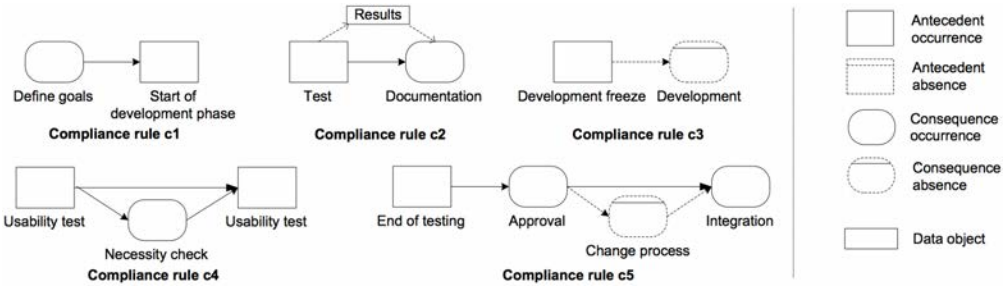


FIGURE 2.15 – Quelques exemples de règles de conformité [166]

Ces approches [166, 162, 13, 85] sont basées sur la définition d'un *graphe visuel* pour représenter la propriété. Quelques exemples tirés des travaux de Ly *et al.* [166] sont visibles sur la figure 2.15. Par exemple, la règle `c1` définit que l'exécution de l'activité "`start of development process`" doit être directement ou indirectement précédée par l'exécution de l'activité "`define goals`". Ainsi, ces approches définissent toutes différents éléments graphiques pour représenter un concept de vérification.

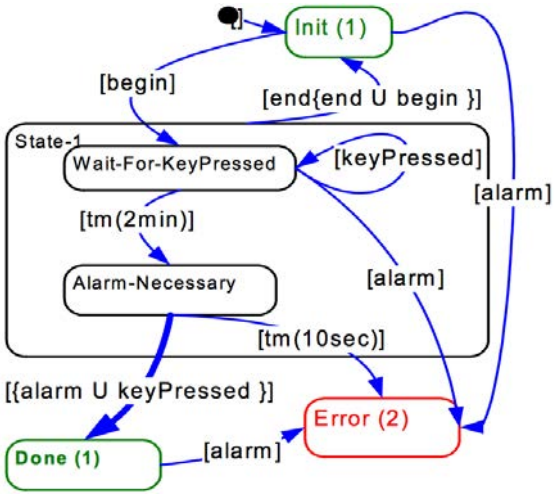


FIGURE 2.16 – Exemple de specification TLChart [63]

Drusinsky [63] proposent TLCharts, un langage de spécification visuelle basé sur une sorte de diagramme d'états afin d'exprimer des propriétés LTL et MTL. La figure 2.16 montre un exemple de spécification TLChart. Les auteurs revendiquent une notation visuelle intuitive pour représenter une propriété temporelle. Cependant, aucune évaluation n'est donnée montrant une simplification pour la spécification de ces propriétés.

Smith *et al.* proposent PROPEL [235], une approche permettant de spécifier des propriétés logiques temporelles à travers l'utilisation des principaux templates décrits dans [66]. Ces templates couvrent les principaux cas d'utilisation des logiques temporelles et sont exprimés à la fois en Finite-State Automaton (FSA) et en phrase du langage naturel.

Ces deux approches [63, 235] ne sont pas destinées à l'expression de propriétés pour le domaine des procédés métiers, mais plus généralement pour tout système dynamique.

2.4.2 Synthèse des approches d'expression de propriétés

Le tableau 2.3 résume l'ensemble de ces approches en comparant le formalisme sous-jacent pour exprimer la propriété, le format proposé, les perspectives du procédé supportées, ainsi que leurs complexités d'utilisation et d'expressivité. Ce dernier critère est classé dans la même colonne ; plus un langage est expressif, plus sa complexité d'utilisation est grande. De toute évidence, ce critère peut être quelque peu subjectif. Nous expliquons les choix de notre notation concernant ce critère dans la suite.

Bien que toutes ces approches pour spécifier des propriétés sur un modèle de procédé fournissent un moyen d'exprimer des propriétés temporelles de manière simplifiée, seule les perspectives du flux de contrôle et de données ont étés prises en compte par ces auteurs. Aucune de ces approches ne permet, par exemple, d'exprimer une propriété permettant de vérifier que la `ResourceA` ou l'`ActivitéA` est disponible/utilisée après x unité de temps.

L'approche proposée dans cette thèse est basée sur le fait d'annoter le modèle (cf. Chapitre 4). Le grand avantage de cette approche provient de sa simplicité pour l'expression de la propriété : en annotant directement les éléments du modèle, au lieu de définir séparément une représentation visuelle de la propriété comme cela est proposé par les approches présentées précédemment. En revanche, leurs pouvoirs d'expressivité sont supérieurs à notre librairie d'annotations, étant donné qu'elles reposent (pour la plupart) sur l'instanciation d'un graphe représentant la propriété.

Le nombre d'éléments n'est donc pas fixé comme sur une annotation. Cependant, ce pouvoir d'expressivité entraîne aussi un accroissement de la complexité d'utilisation. Certaines de ces approches ressemblent fortement à une version visuelle de la logique LTL, qui a été montrée comme compliquée, pour un non-expert, d'exprimer certains comportements subtils attendus par le système [235].

Notre approche proposée à base d'annotations du modèle est très proche de ce que PROPEL [235] propose. La différence majeure est que PROPEL aide à exprimer des propriétés logiques temporelles pour la vérification d'un système *en général*, quand nos annotations sont orientées vérification de procédés et permettent d'exprimer facilement des propriétés sur toutes les dimensions de celui-ci.

Approche	**Nom**	**Formalisme**	**Format**	**Adapté pour procédé ?**	**Perspectives**				**Complexité d'utilisation & Expressivité**
					Controle	Donnée	Ressources	Temps	
Ly *et al.* [166]	-	FOL	Graph based	Yes	✓	✓	-	-	••
Awad *et al.* [13, 14]	BPMN-Q	Past LTL	Graph based	Yes	✓	✓	-	-	••
Liu *et al.* [162]	BPSL	LTL	Graph based	Yes	✓	✓	-	-	•••
Forster *et al.* [85, 86, 87]	PPSL	Past LTL	Graph based	Yes	✓	-	-	-	•••
Drusinsky [63]	TLCharts	LTL, MLT	Statechart	No	n/a	n/a	n/a	n/a	•••••
Smith *et al.* [235]	Propel	FSA	FSA, Textual	No	n/a	n/a	n/a	n/a	••••
Approche de cette thèse									
Laurent *et al.*	-	LTL	Annotations	Yes	✓	✓	✓	✓	•

TABLE 2.3 – Comparatif des approches d'expressions de propriétés

2.5 Conclusion et synthèse générale

Comme visible et résumé sur le tableau 2.2, l'état de l'art souffre grandement du manque de perspectives supportées par leurs formalisations (cf. page 5, problèmes (a1) et (a2)). La majorité des approches se confinent à être utilisées directement dans le *model-checker* et ne proposent pas d'implémentation pour automatiser le processus de vérification ((b2)), ni pour exprimer des propriétés *métiers* sur le procédé ((b1)). De plus, le tableau 2.3 montre qu'il n'existe pas de façon naturelle pour exprimer des propriétés sur les perspectives de *ressources* et de *temps* ((b1)). La plupart des approchent d'expression de propriétés métiers ont un niveau de complexité d'utilisation élevé, proche de ce que serait une version graphique de la logique LTL.

Les dernières lignes des tableaux 2.2 et 2.3 résument les caractéristiques de l'approche proposée dans cette thèse par rapport aux autres approches de la littérature. Pour répondre aux différentes problématiques, nous présentons dans le chapitre 3 une formalisation de UML AD supportant toutes les perspectives des procédés, ainsi que le support de l'expression des propriétés présentées dans la section 2.2. Le chapitre 4 présente l'implémentation de cette formalisation en utilisant le langage Alloy, puis un plug-in intégré à eclipse, construit au-dessus de l'implémentation Alloy, afin de vérifier de manière facile et automatique un procédé. De plus, en formalisant les propriétés présentées dans le tableau 2.1, nous proposons une librairie d'annotations permettant d'exprimer des contraintes métiers (ou propriétés) sur l'ensemble des perspectives du procédé. Ces annotations ont été intégrées dans le plug-in eclipse. Ainsi, en annotant le modèle, il est possible de vérifier de manière automatique les propriétés métiers représentées par ces annotations.

Chapitre 3

Formalisation de UML AD avec la sémantique fUML pour la vérification de procédés

Ce chapitre présente notre contribution pour la formalisation des diagrammes d'activités UML [190] associés à la sémantique fUML [189]. Il est divisé en trois parties. La section 3.1 présente le standard fUML. La section 3.2 présente notre formalisation en définissant la syntaxe du langage, puis sa sémantique. La syntaxe représente la grammaire du langage, c'est-à-dire les constructions correctes du langage. La sémantique traite la signification de ces constructions, c'est-à-dire le comportement d'une instance du langage. La section 3.3 présente la formalisation des propriétés intéressantes à vérifier sur un procédé en utilisant la PLTL [288]. Finalement, la section 3.4 conclut ce chapitre.

3.1 Foundational UML (fUML)

Dans cette section, nous commençons par présenter les motivations derrière la spécification fUML. Ensuite, nous décrivons sa syntaxe ainsi que sa sémantique. Nous présentons la façon dont est défini le moteur d'exécution ainsi que son environnement d'exécution. Enfin, nous donnons un exemple détaillé d'exécution.

3.1.1 Pourquoi fUML ?

La version actuelle de la spécification UML est complexe et utilise une combinaison de diagrammes semi-formelle, des contraintes et un langage naturel afin de définir sa sémantique opérationnelle. L'imprécision et les ambiguïtés du langage naturel rendent difficile la détection et la correction des incohérences de la spécification. De nombreuses études soulignent ces ambiguïtés [83, 207, 233]. Les outils de modélisation implémentant UML sont obligés de faire certaines interprétations basées sur ce langage naturel. De ce fait, ces interprétations compromettent l'interopérabilité des outils basés sur UML. En effet, certains outils peuvent exécuter le même modèle de manière différente, et ne rendent pas, en conséquence, les mêmes résultats pour le même modèle.

Afin de pallier ces problèmes, l'OMG a demandé, en 2005, la proposition d'une définition formelle de la sémantique d'un sous-ensemble d'UML [215]. Cette demande stipule que, pour chacun des éléments du sous-ensemble sélectionné, on doit avoir une sémantique précise et non

ambiguë, de façon à ce que les outils conformes à ce standard puissent complètement interpréter un modèle appartenant à ce sous-ensemble.

Ainsi, en février 2011, l'OMG a publié un nouveau standard appellé "*Semantics of a Foundational Subset for Executable UML Models*" (version 1.0), qui définit une sémantique d'exécution précise pour un sous-ensemble de UML 2.3, souvent appelé "*foundational UML*" (fUML). Cette thèse est basée sur la version 1.1 du standard sortie en août 2013 [189].

Il est important de noter que l'OMG a aussi sorti en 2013 le standard Alf [191], fournissant une syntaxe concrète et textuelle pour décrire un modèle fUML. La sémantique de Alf est définie par un mapping de la syntaxe de Alf vers la syntaxe abstraite de fUML. En essence, Alf est une version textuelle de la représentation des diagrammes d'activités et de classes, permettant ainsi de décrire ces diagrammes sous forme textuelle. La syntaxe de Alf est proche de celle du langage C et de celle de Java [191].

Dans la suite, nous détaillons la syntaxe de fUML ainsi que sa sémantique d'exécution.

3.1.2 Syntaxe de fUML : les éléments de modélisation

Le sous-ensemble sélectionné du méta modèle UML 2, choisi pour être les fondations d'UML, constitue la base pour, éventuellement, définir la sémantique d'exécution d'éléments et de concepts UML de plus haut niveau. Ceci est visible sur la figure 3.1. La surface du sous-ensemble UML (`surface UML subset`) est le sous-ensemble utilisé pour modéliser un système. Cette surface contient généralement plus de concepts que le sous-ensemble de fUML (`Foundational UML subset`). Ceci implique qu'une traduction à partir de la surface du sous-ensemble d'UML vers le sous-ensemble de fUML doit être effectuée. De cette manière, fUML peut être vu comme un intermédiaire entre la surface d'UML pour modéliser le système et le langage de la plateforme cible (`Platform language`) pour exécuter le modèle du système. Ainsi, le modèle est défini en utilisant les concepts de modélisation de UML, et peut être traduit dans le langage de la plateforme cible (ex. Java) pour être exécuté.

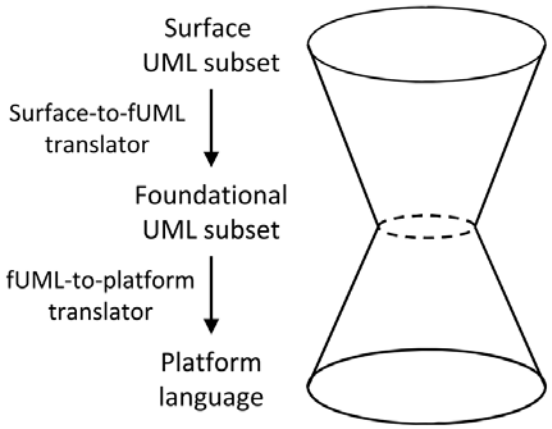


FIGURE 3.1 – fUML comme un intermédiaire entre la surface du sous-ensemble UML et le langage de la plateforme [189]

L'ensemble de fUML a été choisi en considérant trois critères : *compacité*, *facilité de traduction* et *simplicité*. La *compacité* signifie que l'ensemble de fUML doit être petit, mais suffisant pour définir précisément sa sémantique. La *facilité de traduction* signifie que les traductions `surface-to-fUML` et `fUML-to-platform` doivent être faciles à réaliser. La *simplicité* signifie que

la sémantique définie par fUML détermine seulement comment les actions UML sont exécutées en utilisant les fonctions primitives.

Bien sûr, ces critères interagissent entre eux et, en spécifiant l'ensemble de fUML, des compromis doivent être pris en considération. Par exemple, il y a un conflit entre la *compacité* et la *facilité de traduction* : si la traduction peut être simplifiée en ajoutant une traduction un-à-un d'un élément de la surface UML vers le langage de la plateforme, cela cause une détérioration de la *compacité*. En effet, une accumulation de fonctionnalités peut éventuellement détruire le but d'avoir un ensemble compact.

Du fait que fUML est simplement un sous-ensemble de UML, le méta modèle de fUML est un sous-ensemble du méta-modèle UML et est structuré de la même manière. Cela signifie que la structure des packages du méta modèle fUML est la même que la structure des packages du méta modèle UML. Les packages qui ne sont pas inclus dans fUML sont complètement exclus. Les packages inclus peuvent être restreints, comparés au package correspondant dans UML, c'est-à-dire que certains éléments du package peuvent être exclus et certaines contraintes additionnelles peuvent être définies.

Le tableau 3.1 décrit les packages de UML inclus dans fUML. Nous distinguons les packages pour la modélisation structurelle (ex. diagrammes de classes) de ceux pour modéliser le comportement (ex. diagrammes d'activités). Comme en témoigne ce tableau, fUML supporte principalement les diagrammes de classes et d'activités. Le package `Classes` est inclus pour traiter la modélisation des concepts basiques et pour décrire les classes d'un système, ses attributs, ses opérations et ses relations entre elles. Pour modéliser le comportement, `Actions`, `Activities` et `CommonBehaviors` sont partiellement inclus. En effet, certains éléments avec un niveau d'abstraction trop élevé sont retirés de la spécification, ou seront tout simplement inclus dans les futures versions de fUML. Sans lister le détail de chacun des éléments inclus ou non, il est important de noter que :

- Les "central buffer nodes" ou "data store nodes" sont exclus de fUML du fait qu'ils ont été jugés non nécessaires pour la complétude de fUML. Ceci est dû au critère de compacité.
- Les variables sont exclues car le passage de données entre les actions peut être effectué en utilisant du flux de données (input et ouput pin sur les actions).
- Aucun mécanisme de gestion d'exception n'est géré.
- Les actions opaques ne sont pas incluses car elles ne peuvent pas être directement exécutées. En effet, ces actions contiennent généralement du code dans un langage spécifique (ex. C# ou Java).
- fUML est agnostique à propos du temps. Le package `SimpleTime` (nécessaire pour représenter le temps et durée, ainsi que les actions pour observer leurs écoulements) est entièrement exclu de fUML du fait que les évènements de temps et leurs contraintes ne font pas partie du sous-ensemble retenu par fUML. En conséquence, il n'est pas possible, par exemple, d'utiliser des éléments pour *réagir* en fonction du temps.

Ainsi, le package `Activities` permet de décrire comment les actions dans le système seront exécutées en utilisant du flux de contrôle et de données entre les actions. Il est important de noter que toutes les constructions présentées dans la description des diagrammes d'activités de la section 2.1.4 de l'état de l'art sont *incluses* dans fUML.

3.1.3 Sémantique de fUML : un modèle d'exécution

Le modèle d'exécution de fUML est lui-même un modèle, écrit en fUML, qui spécifie comment les modèles fUML sont exécutés. Du fait que cette forme de spécification conduit à de gros diagrammes difficiles a lire, du pseudo-code Java est utilisé pour définir le modèle d'exécution (au lieu de diagramme d'activités fUML). Le modèle d'exécution définit : la sémantique d'exécution de tous les éléments de modélisation de fUML, un moteur d'exécution, et son environnement. La structure des packages pour définir la sémantique des éléments de modélisation est la même

Package UML	Inclus dans fUML ?
Modélisation de la structure	
Classes	✓
Components	
Composite Structures	
Deployments	
Modélisation du comportement	
Actions	✓
Activities	✓
Common Behaviors	✓
Interactions	
State Machines	
Use Cases	

TABLE 3.1 – Paquage UML inclus dans le sous-ensemble fUML

que la structure de la syntaxe abstraite (elle inclut les mêmes packages et sous-packages). Additionnellement, le modèle d'exécution inclut un package appellé `Loci`, afin de représenter le moteur d'exécution ainsi que son environnement. Ainsi, le modèle d'exécution incorpore les packages suivants :

Loci. Ce package spécifie le moteur d'exécution et son environnement.

Classes. Ce package définit la sémantique structurelle de fUML.

CommonBehaviors, Activities, Actions. Ces packages définissent la sémantique comportementale de fUML.

Ces définitions permettent donc de définir une machine virtuelle basique pour l'exécution de modèles fUML. Par modèle fUML, nous entendons un modèle UML comprenant seulement des éléments de modélisation inclus dans le sous-ensemble sélectionné par fUML.

Moteur d'exécution et environnement d'exécution

Comme le montre la figure 3.2, le package `Loci` contient les classes `Locus`, `Executor`, et `ExecutionFactory` représentant un moteur d'exécution pour fUML. L'`Executor` fournit l'abstraction de base pour exécuter un modèle fUML. Il fournit l'interface basique pour évaluer des valeurs et exécuter des comportements (ex. une activité) de manière synchrone ou asynchrone. Chacune des exécutions prend place dans un `Locus` spécifique. Un `Locus` peut être vu comme une abstraction d'un ordinateur physique ou virtuelle, capable d'exécuter des modèles fUML. Chacun des objets et liens créés pendant ou avant l'exécution lui sont associés.

Le modèle d'exécution est basé sur le *visitor pattern* [96] pour instancier la syntaxe. En utilisant ce patron, chacune des méta classes abstraites a un visiteur correspondant dans le modèle d'exécution (nommé `*Execution` et `*Activation`, ex. `CallBehaviorAction` $\mapsto$ `CallBehaviorActionActivation`). Toutes les classes visiteurs dans le modèle d'exécution sont descendantes, directement ou indirectement de la classe racine `SemanticVisitor`. L'`Executor` utilise ces visiteurs afin d'instancier l'équivalent comportemental de l'élément. Afin d'instancier l'élément du visiteur, l'`Executor` utilise une instance de la classe `ExecutionFactory` fournie à l'exécution par le `Locus`. L'`ExecutionFactory` maintient l'ensemble des comportements primitifs qui peuvent être appelés. Dans fUML, ces comportements primitifs sont définis comme des instances de `OpaqueBehavior`. Pour chacune des instances d'`OpaqueBehavior` représentant un

comportement primitif, l'`ExecutionFactory` possède une instance `OpaqueBehaviorExecution` équivalente. L'`ExecutionFactory` contient aussi les instances des `SemanticStrategy` afin de gérer les points de variations de sémantique de UML (ex. ordonnancement des évènements LIFO ou FIFO, etc).

Additionnellement, le modèle d'exécution contient un package appellé `Library` qui contient la Foundational Model Library, une libraire d'éléments définie par l'utilisateur qui peut être référencée dans un modèle fUML. Elle définit les types primitifs tels que *booleen*, *integer*, *string*, *unlimited natural* et les comportements primitifs sur ceux-ci tels que *OR*, *XOR*, *AND*, *NOT* sur le type booleen.

Afin de configurer l'environnement d'exécution, il est nécessaire d'instancier cet ensemble d'objets à l'intérieur du modèle d'exécution pour fournir l'environnement initial d'exécution.

Sémantique opérationnelle de fUML

La sémantique adoptée par le modèle d'exécution de fUML est basée sur le principe d'offre et de consommation de jetons de type contrôles ou objets entre les différents constituants de l'activité (les noeuds et les arcs tels que présentés dans la section 2.1.4), similairement aux réseaux de Petri colorés [131].

Afin d'exécuter un modèle fUML, le moteur d'exécution suit la procédure suivante :

1. **Provision des entrées de l'activité (`ActivityParameterNode`).** Avant que l'exécution de l'activité démarre, les valeurs des paramètres d'entrées de l'activité sont fournies.
2. **Identification des noeuds activables.** Dans un second temps, les noeuds activables sont identifiés par le moteur d'exécution : les noeuds initiaux (`InitialNode`), les paramètres d'entrées de l'activité (`ActivityParameterNode`), et tous les noeuds (`ActivityNode`) qui n'ont aucun arc entrant.
3. **Envoi d'un jeton de contrôle aux noeuds activables.** Une fois les noeuds activables identifiés, des jetons de contrôle leur sont envoyés.
4. **Exécution des noeuds de l'activité.**
   (a) **Vérification si le noeud est prêt à être exécuté.** Quand un noeud de l'activité reçoit un jeton, le moteur d'exécution détermine si tous les prérequis sont remplis afin de démarrer son exécution. Plus précisément, le moteur vérifie, par exemple, si un jeton de contrôle est disponible sur tous ses arcs entrants, et dans le cas d'une action, si des jetons de données sont disponibles sur ses pins d'entrées.
   (b) **Consommation des jetons.** Si un noeud est prêt à être exécuté, c'est-à-dire, si tous les jetons de contrôles et de données sont disponibles, le noeud consomme ces jetons qui lui sont offerts. Ainsi, les jetons sont retirés des arcs entrants et un jeton est ajouté dans le noeud.
   (c) **Execution du comportement du noeud.** Après que les jetons soient consommés par le noeud, le comportement du noeud est exécuté. Le comportement est différent selon le type de noeud qui s'exécute. Par exemple, un noeud de décision (`DecisionNode`) choisira un chemin parmi ses arcs sortants, alors qu'une action pourrait produire des jetons de données et les placer sur ses pins de sorties.
   (d) **Envoi des jetons aux noeuds suivants.** Quand le comportement du noeud a terminé de s'exécuter, un jeton de contrôle est inséré sur chacun des arcs sortants, et, dans le cas d'une action avec des pins, des jetons de données sont produits sur les arcs de sortie de ses pins.
   (e) **Exécution des autres noeuds de l'activité.** Du fait qu'après l'exécution d'un noeud, des jetons ont été propagés sur ses arcs sortants, de nouveaux noeuds pourraient être

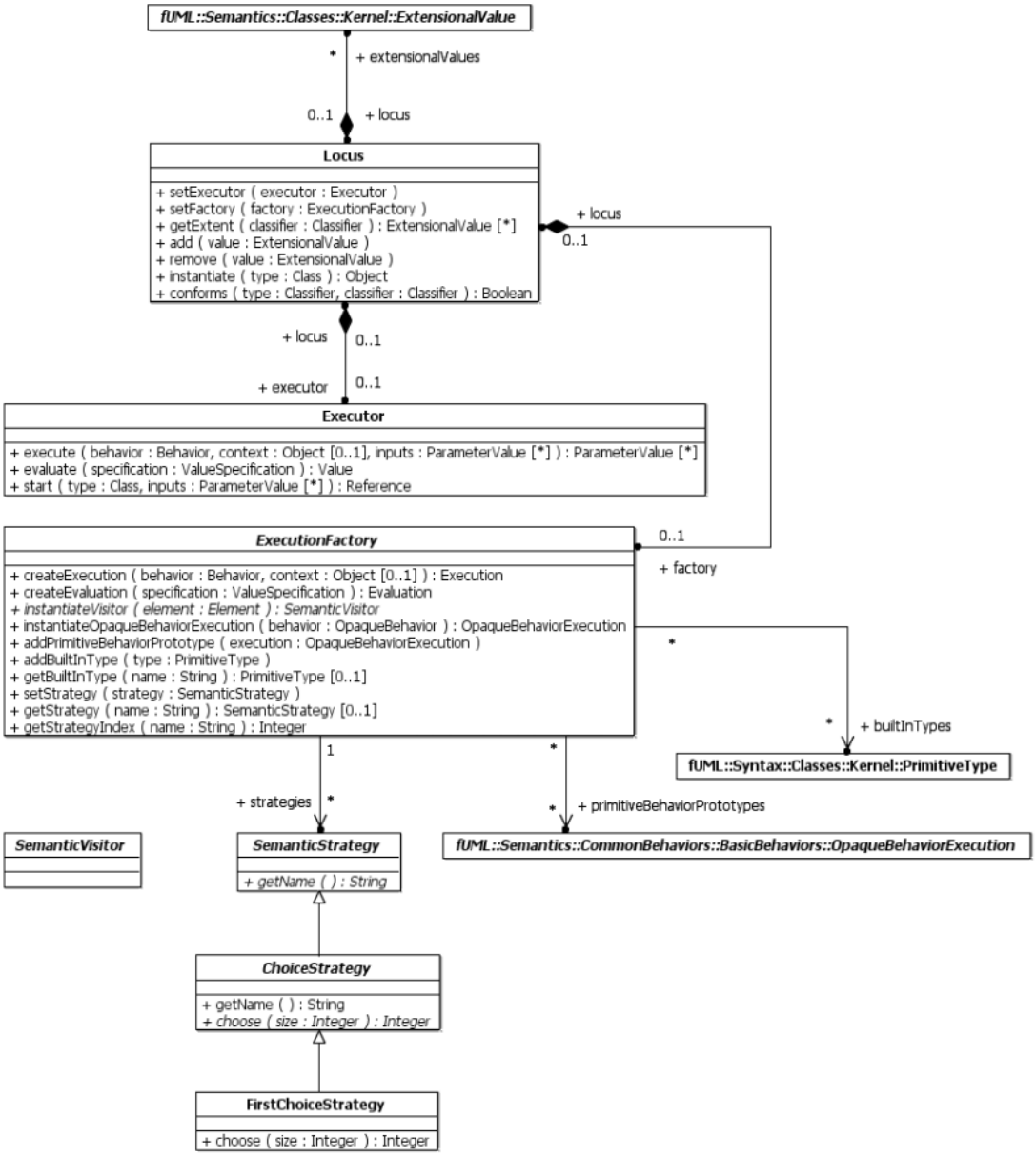


FIGURE 3.2 – `Loci` package du modèle d'exécution de fUML [189]

prêt à s'exécuter. Ainsi, les étapes "a" à "d" sont effectuées pour les noeuds ayant reçu des jetons.

5. **Provision des sorties de l'activité.** Quand aucun autre noeud ne peut être exécuté, ou qu'un noeud `ActivityFinalNode` a été exécuté, l'exécution de l'activité est terminée et les valeurs présentes sur les noeuds de sorties de l'activité sont renvoyées.

Un des points importants de la sémantique d'exécution de fUML et qu'elle possède un concept implicite d'exécution concurrente de threads. Par défaut, les noeuds de l'activité sont ou peuvent être exécutés de façon concurrente. Ainsi, les jetons peuvent être produits et consommés de façon parallèle à travers les arcs d'entrées et de sorties des noeuds. Cependant, la spécification fUML

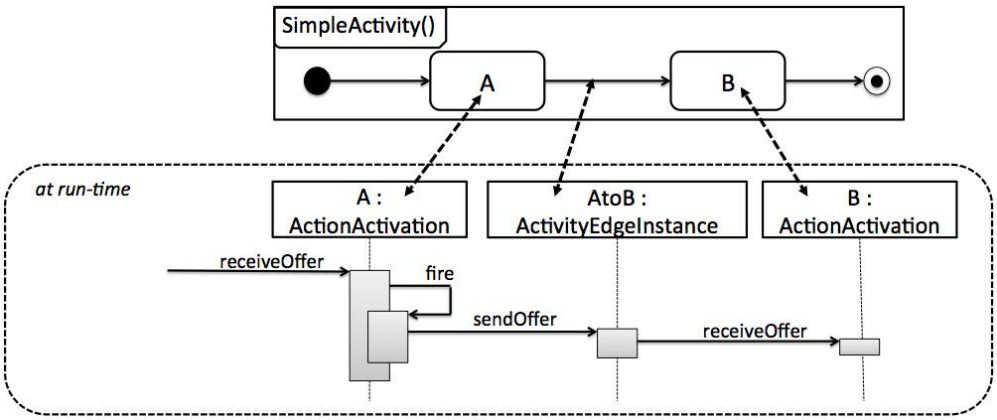


FIGURE 3.3 – Interactions entre les éléments du diagramme d'activité

ne contraint pas à ce que les exécutions soit réellement parallèles, de manière physique. En effet, elle ne spécifie pas précisément comment implémenter le parallélisme. N'importe quelle exécution séquentiellement ordonnée ou partiellement parallèle est valide, à partir du moment que les contraintes du modèle fUML exécuté, en terme de création, terminaison et synchronisation soient respectées. L'implémentation de référence de fUML développé par la "ModelDriven Community" [242] utilise seulement un seul thread pour exécuter les modèles. En essence, cette implémentation ne peut pas exécuter deux noeuds réellement en même temps, mais produit toujours des traces d'exécution légale envers la spécification et le modèle exécuté.

3.1.4 Exemple d'exécution d'une activité

Afin de détailler comment le modèle d'exécution se comporte pour exécuter une activité, la figure 3.3 présente un diagramme d'activité simpliste illustré par un diagramme de séquences UML. Ce diagramme se concentre sur les interactions faites entre les différents éléments du diagramme d'activité, et ne montre pas tous les appels de méthodes effectués pour représenter l'exécution complète. `ActionActivation` et `ActivityEdgeInstance` sont les instanciations de la syntaxe abstraite (réalisée grâce au design pattern visitor). Quand les préconditions de l'action `A` sont remplies, cette action reçoit l'offre (*receiveOffer*), puis commence à exécuter son propre comportement (*fire*). Une fois cette exécution terminée, l'action `A` offre un jeton sur son arc de contrôle sortant `AtoB` (*sendOffer*). Ainsi, l'arc `AtoB` propose à l'action `B` de recevoir l'offre (*receiveOffer*). En essence, l'exécution correspond à une chaîne étendue de *sendOffer*-*receiveOffer*-*fire*-*sendOffer* entre les différents éléments composant le diagramme.

La figure 3.4 présente un diagramme d'activité ainsi qu'une exécution potentielle. Ce diagramme comprend les noeuds les plus basiques pour définir le flux de contrôle : des `Action` (`A, B, B1, B2, C, D`), un noeud `InitialNode`, un noeud `ForkNode`, un noeud `DecisionNode`, un noeud `MergeNode`, un noeud `JoinNode`, et un noeud `ActivityFinalNode`. Les étapes pour exécuter cette activité sont les suivantes :

1. Le modèle d'exécution insère un jeton dans les noeuds activables. Sur cette étape, seul le noeud `InitialNode` est activable.
2. A la fin de l'exécution du noeud `InitialNode`, le jeton est envoyé sur son arc sortant.
3. Les préconditions de l'action `A` sont remplies, il peut commencer à s'exécuter.
4. A la fin de l'exécution de l'action `A`, le jeton est envoyé sur son arc sortant.

5. Le noeud `ForkNode` peut donc commencer à s'exécuter du fait que ses préconditions sont remplies.
6. Le but d'un noeud `ForkNode` étant de réaliser un branchement parallèle, à la fin de son exécution, des jetons sont fournis sur chacun de ses arcs sortants. Il est important de noter que ce comportement est celui par défaut de tous les noeuds des diagrammes d'activités (c'est-à-dire Fork et Join implicites sur les noeuds).
7. Les préconditions pour exécuter `B` et `C` sont remplies. Sur cette exécution, c'est l'action `B` qui a commencé à s'exécuter.
8. Du fait que tout se passe en parallèle, `B` peut terminer de s'exécuter, ou `C` peut commencer à s'exécuter. Sur cette étape, `B` se termine.
9. Les préconditions des noeuds `DecisionNode` et `C` sont remplies. Le noeud `DecisionNode` commence à s'exécuter.
10. Le noeud `DecisionNode` offre un jeton sur seulement un de ses arcs de sortie. Ce comportement est propre à ce type de noeud et est déterminé par rapport à l'évaluation d'un comportement spécifié sur le noeud (l'attribut "*decisionInput*", non visible sur la figure). Sur cette étape, l'évaluation a choisi l'arc allant vers l'action `B2`.
11. Les noeuds `B2` et `C` sont activables. Le noeud `B2` commence à s'exécuter.
12. Le noeud `C` commence à s'exécuter.
13. Le noeud `C` termine son exécution et insère un jeton sur son arc sortant.
14. Les noeuds `B2` et `C` peuvent se terminer. En revanche, le noeud `JoinNode` n'est pas encore activable, seul un de ses arcs entrants possède un jeton. L'action `B2` se termine.
15. Contrairement aux autres types de noeud, afin que les préconditions du noeud `MergeNode` soient remplies, seul un de ses arcs entrants doit avoir une offre. Le noeud `MergeNode` est donc activable et commence à s'exécuter.
16. Le noeud `MergeNode` offre un jeton sur son arc sortant.
17. Les préconditions du noeud `JoinNode` sont remplies. Sur cette étape c'est le seul noeud activable. Le noeud `JoindeNode` commence donc à s'exécuter.
18. A la fin de son exécution, le noeud `JoinNode` offre un jeton sur son arc sortant.
19. Le noeud `D` peut commencer à s'exécuter.
20. A la fin de l'exécution de `D`, le noeud offre un jeton sur son arc sortant.
21. Les préconditions pour exécuter le noeud `ActivityFinalNode` sont remplies, le noeud s'exécute. Du fait que plus aucun noeud n'est activable et qu'un noeud `ActivityFinalNode` a été exécuté, l'exécution de l'activité est terminée.

Il est important de noter que l'exemple décrit ici est seulement *une* des exécutions possibles de ce diagramme d'activité. Dès lors que des noeuds de branchement parallèle sont introduits, l'indéterminisme du parallélisme implique la possibilité de nombreux chemins d'exécutions différents.

L'exemple d'exécution ainsi que les informations sur l'environnement d'exécution présenté dans cette section sont suffisants pour comprendre la formalisation introduite dans la prochaine section. En revanche, ils ne couvrent pas entièrement tous les aspects pertinents de la spécification fUML. Les lecteurs intéressés par une introduction plus poussée peuvent se référer directement à la spécification [189].

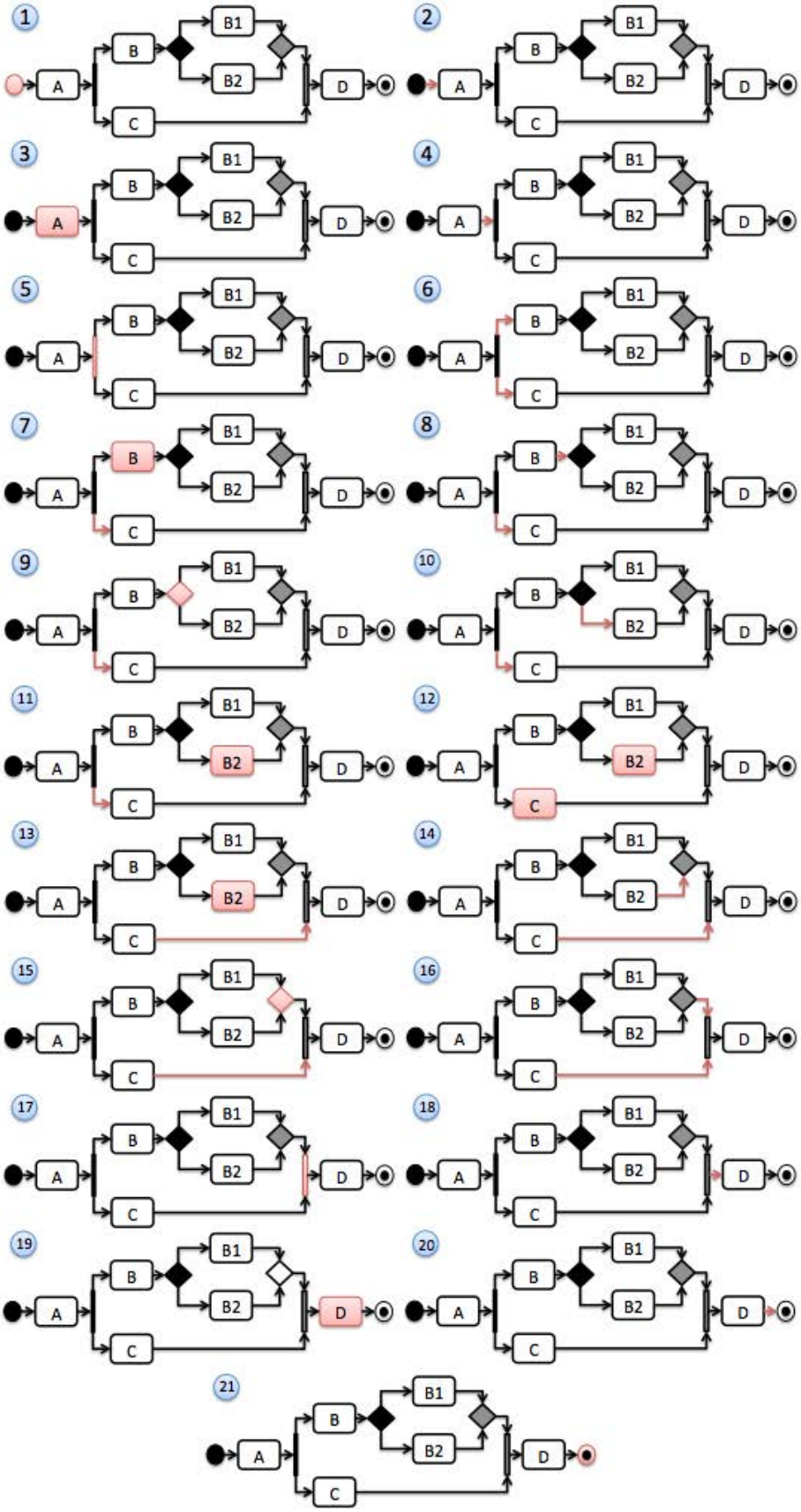


FIGURE 3.4 – Exemple d'exécution d'un modèle fUML

3.2 Formalisation de UML AD et fUML

Comme expliqué dans la section précédente, la sémantique de fUML est donnée en pseudo-code Java. Les auteurs ont donc déjà pris des décisions liées au langage Java afin de lui donner une sémantique opérationnelle. La sémantique donnée est donc restreinte par rapport aux constructions du langage que celui-ci propose (*language dependent*), ce qui n'est pas toujours adapté pour la généralité de la sémantique que l'on souhaite exprimer. De plus, cette forme de spécification basée sur le langage Java augmente la complexité. En effet, il serait en théorie "possible" d'extraire les différents états et fonctions de franchissement de la sémantique opérationnel afin de construire un *model-checker* autour du moteur d'exécution. Cependant, la vérification de ce système de transition extrait aurait une trop grande complexité due à un manque d'*abstraction*.

Ainsi, formaliser fUML dans un langage mathématique de plus haut niveau apporterait de nombreux avantages : (1) se détacher du langage cible choisie par les auteurs de fUML, (2) d'être capable de l'implémenter facilement dans le langage d'un *model-checker* donnée, (3) pouvoir effectuer une vérification plus efficace du système, et (4) pouvoir l'étendre pour prendre en compte les informations organisationnelles associées aux modèles de procédés telles que la perspective des ressources et les contraintes temporelles (cf. section 2.2.2).

Nous avons décidé d'utiliser la logique de premier-ordre (FOL, pour first-order logic) pour effectuer cette formalisation. La FOL est suffisament expressive pour permettre de donner une sémantique mathématique *claire* et *non ambigüe* à fUML, en se basant directement sur sa sémantique opérationnelle, plutôt que de de reposer sur un formalisme tel que les réseaux de Petri pour lesquels les choix sémantiques du langage ne seraient pas forcément adaptés à fUML. Par exemple, il a été montré que les réseaux de Petri classiques ne sont pas adaptés pour modéliser un système tel qu'un procédé contenant des données, ressources et temps [268]. En effet, bien que possible en utilisant différentes extensions des réseaux de Petri, leur expression n'est pas naturelle et engendre des réseaux complexes avec un effort de modélisation important [263].

L'implémentation de cette formalisation permettrait donc d'appliquer des techniques de *model-checking* (non borné [42] ou borné [25]) du fait qu'elle soit basée sur une fondation mathématique, comme tous les langages pris en entrées par les *model-checkers*.

Dans la suite, nous réduisons formellement la représentation d'un procédé en un graphe aux sommets labelisés (vertex-labeled graph) [29]. Chacun des noeuds du graphe correspond à un noeud d'activité UML selon son type (c'est-à-dire `ControlNode`, `ExecutableNode` ou `ObjectNode`). La sémantique d'exécution de ce formalisme est basée sur les notions d'*états*, *activation*, et *tir* de transitions, similairement à celles utilisées dans les réseaux de Petri colorés [131]. Afin de pouvoir exprimer une grande gamme de propriétés sur toutes les perspectives du procédé de façon unifiée (cf. section 2.2), la formalisation couvre à la fois la perspective de *flux de contrôle* et *des données* à travers l'utilisation de la notation des diagrammes d'activités, mais prend aussi en compte les informations organisationnelles associées telles que la perspective des ressources et les contraintes temporelles.

Dans la prochaine section, nous présentons notre modèle formel en commençant par définir sa syntaxe, puis sa sémantique.

3.2.1 Syntaxe

La figure 3.5 montre un extrait du méta modèle des diagrammes d'activités fUML géré par notre formalisation. Cette formalisation adresse seulement un sous-ensemble compris dans le standard fUML et utile pour la modélisation de procédés comme identifié dans [20] et introduit dans la section 2.1.4 page 16 lors de la présentation des diagrammes d'activités UML.

Formellement, nous considérons trois éléments basiques : *Control*, *Executable*, et *Object*.

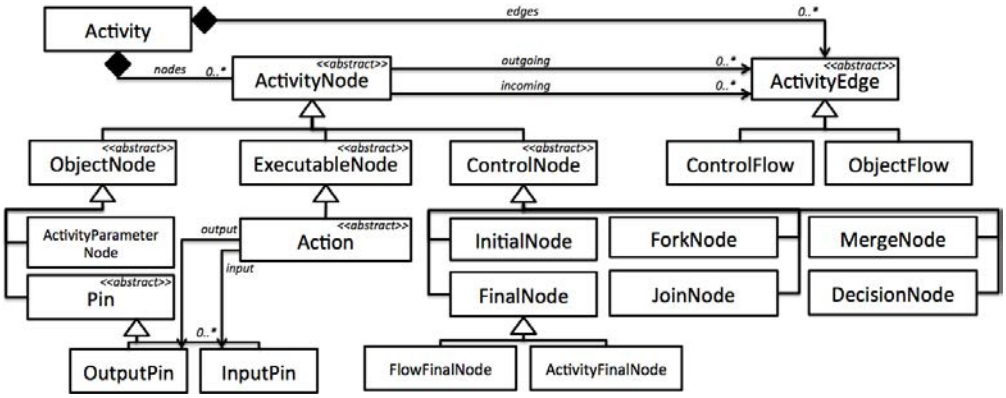


FIGURE 3.5 – Extrait du meta-modèle des diagrammes d'activités fUML géré par notre formalisation

- $Control = \{fork, join, decision, merge, initial, activityFinal, flowFinal\}$ représentent les noeuds de type `ControlNode`,
- $Executable = \{action\}$ représente les noeuds de type `ExecutableNode`,
- $Object = \{activityParameter, outPin, inPin\}$ représentent les noeuds de type `ObjectNode`,
- $Types = Control \cup Executable \cup Object$ représente l'ensemble de tous les types de noeuds.

Ainsi, nous introduisons la notion d'un diagramme comme un *vertex-labeled* graphe :

Definition 5. *Un* ***Diagram*** *est un tuple* $D = (V, E, lab, lower, upper)$ *tel que :*
- V *est l'ensemble des sommets.*
- $E \subseteq V \times V$ *est l'ensemble des arcs.*
- $lab : V \mapsto Types$ *est la fonction d'étiquetage qui associe à chacun des sommets* $v \in V$ *un type.*
- $lower/upper : V \mapsto \mathbb{N} \cup \{\epsilon\}$ *sont les fonctions qui retournent, respectivement, la multiplicité inférieure et supérieure d'un noeud de type objet.*

$$lower(v) \stackrel{def}{=} \begin{cases} n \in \mathbb{N} & \text{if } lab(v) \in Object \\ \epsilon & \text{otherwise} \end{cases}$$

La fonction upper *a la même définition.*

Pour un diagramme $D = (V, E, lab, lower, upper)$, nous introduisons des fonctions auxiliaires afin de nous aider à formaliser un diagramme d'activité :
- $Vlab : Types \mapsto 2^V$ est la fonction qui retourne tous les sommets pour un type donné :

$$Vlab(t) \stackrel{def}{=} \{v \in V \mid lab(v) = t\}$$

- $incoming/outgoing : V \mapsto 2^E$ sont les fonctions qui retournent, respectivement, les arcs entrants et sortants d'un noeud :

$$incoming(v) \stackrel{def}{=} \{(x, y) \in E \mid y = v\}$$
$$outgoing(v) \stackrel{def}{=} \{(x, y) \in E \mid x = v\}$$

– $source/target : E \mapsto V$ sont les fonctions qui retournent, respectivement, la source et la destination d'un arc :

$$source(e) \stackrel{def}{=} \{s \in V \mid e = (s,t)\}$$
$$target(e) \stackrel{def}{=} \{t \in V \mid e = (s,t)\}$$

– $input/output : V \mapsto 2^V$ sont les fonctions qui retournent, respectivement, les entrées et sorties d'un noeud de type action :

$$input(v) \stackrel{def}{=} \begin{cases} in \subseteq V & \text{if } lab(v) = action \wedge \forall v' \in in, lab(v') = inPin \wedge ((v,v') \in E \vee (v',v) \in E) \\ \emptyset & \text{otherwise} \end{cases}$$

$$output(v) \stackrel{def}{=} \begin{cases} out \subseteq V & \text{if } lab(v) = action \wedge \forall v' \in out, lab(v') = outPin \wedge ((v,v') \in E \vee (v',v) \in E) \\ \emptyset & \text{otherwise} \end{cases}$$

Maintenant, nous pouvons définir la notion d'**Activity Diagram** comme un **Diagram** avec certaines contraintes structurelles additionnelles.

Definition 6. *Un **Activity Diagram** est un **Diagram**, $AD = (V, E, lab, lower, upper)$, avec les contraintes additionnelles suivantes :*

– *Aucun noeud n'est déconnecté :*

$$\forall v \in V, incoming(v) \neq \emptyset \vee outgoing(v) \neq \emptyset$$

– *La source et destination d'un arc sont différentes :*

$$\forall e \in E, source(e) \neq target(e)$$

– *Les noeuds initiaux n'ont pas d'arcs entrants :*

$$\forall v \in Vlab(initial) : incoming(v) = \emptyset$$

– *Tous les noeuds de fin d'activité ou de fin de flux n'ont pas d'arcs sortants :*

$$\forall v \in (Vlab(flowFinal) \cup Vlab(activityFinal)) : outgoing(v) = \emptyset$$

– *Les pins sont connectés à un seul autre noeud pin :*

$$\forall v \in Vlab(inPin) : |incoming(v)| = 1 \wedge$$

$$\forall v \in Vlab(inPin), \forall e \in incoming(v), \forall a \in source(e) : lab(a) = outPin \wedge$$

$$\forall v \in Vlab(outPin) : |outgoing(v)| = 1 \wedge$$

$$\forall v \in Vlab(outPin), \forall e \in outgoing(v), \forall a \in target(e) : lab(a) = inPin$$

– *La borne inférieure d'un noeud objet est inférieure ou égale à sa borne supérieure :*

$$\forall v \in Vlab(Object) : lower(v) \leq upper(v)$$

– *La borne supérieure d'un noeud objet est au minimum égale à un :*

$$\forall v \in Vlab(Object) : upper(v) \geq 1$$

Généralement, un procédé est caractérisé par deux parties principales : son *workflow* et les informations organisationnelles associés. Ici, le workflow est représenté en utilisant UML AD. Les informations organisationnelles sont directement attachées aux actions afin de donner des informations à propos de l'exécution. Ces informations sont dépendantes du domaine du procédé. Par exemple, les modèles de procédés *logiciels* peuvent comporter des informations sur le nombre d'agents et leurs compétences requises, tandis que des modèles de procédés *médicaux* comportent les équipements et les médicaments nécessaires pour effectuer la tâche. Ainsi, nous définissons un procédé comme un AD étendu avec les informations organisationnelles les plus communes : les ressources et le temps. Il est important de noter que la définition peut être facilement étendue pour prendre en acompte d'autres informations dépendantes du domaine.

Definition 7. *Un* ***Process*** *est un tuple* $P = (V, E, lower, upper, lab, Resource, Use, Capacity, Timing)$ *où :*

– $(V, E, lower, upper, lab)$ *forme un* ***Activity Diagram*** *tel que :*
  – V *contient au moins un noeud initial :* $\exists v \in Vlab(initial)$.
  – V *contient au moins un noeud final :* $\exists v \in Vlab(activityfinal)$.

– $Resource$ *est un ensemble fini de ressource,*

– $Use : V \mapsto 2^{Resource} \cup \{\epsilon\}$ *est la fonction qui associe à chaque action un ensemble de ressource :*

$$Use(v) \stackrel{def}{=} \begin{cases} r \subseteq Resource & \text{if } lab(v) = action \\ \epsilon & \text{otherwise} \end{cases}$$

– $Capacity : Resource \mapsto \mathbb{N} \cup \{\epsilon\}$ *est la fonction qui associe à chaque ressource une capacité maximale d'utilisation :*

$$Capacity(r) \stackrel{def}{=} \begin{cases} r \in \mathbb{N} & \text{if } lab(v) = action \\ \epsilon & \text{otherwise} \end{cases}$$

– $Timing : V \mapsto \mathbb{N} \cup \{\epsilon\}$ *est la fonction qui associe à chaque action un temps pour l'effectuer :*

$$Timing(v) \stackrel{def}{=} \begin{cases} v \in \mathbb{N} & \text{if } lab(v) = action \\ \epsilon & \text{otherwise} \end{cases}$$

3.2.2 Sémantique

La sémantique de notre modèle suit le standard fUML [189]. Nous formalisons la façon dont les jetons transitent entre les arcs et les noeuds qui composent un AD.

Généralement, pour appliquer des techniques de *model-checking*, le système est représenté par un *système de transitions* [42]. Un *système de transitions* est un modèle qui définit les états et actions qui provoquent les transitions entre ces états. Principalement, il y a deux façons de modéliser un *système de transitions*. Dans l'idiome *opérationnel*, les transitions sont exprimées en utilisant des instructions d'affectations, soit avec du flux de contrôle de langage fonctionnel impératif (comme Promela, le langage du *model-checker* Spin [115]), soit en utilisant des variantes des commandes de gardes de Diskstra (tel que Murphi [59] ou SMV [38]). Dans l'idiome *déclaratif*, les transitions sont exprimées avec des contraintes, soit sur l'exécution complète, soit, plus souvent,

sur des étapes individuelles. L'idée de cet idiome prend racine dans les premiers travaux sur la vérification de programmes : la spécification d'opération des langages déclaratifs VDM [132], Larch [106] ou Z [236], qui sont essentiellement les *pre* et *post* conditions des triplets de Hoare [112].

Dans cette section, nous exprimons la sémantique de notre formalisme en utilisant la FOL et l'idiome *déclaratif*. Nous définissons les notions (1) d'*état*, (2) d'*activation* et (3) de *tir* de transitions. Comme identifié dans le chapitre 2.2, il est important d'avoir des informations concernant l'écoulement du temps dans le système pour vérifier certaines propriétés temporelles. Afin de pouvoir vérifier ce type de propriétés, nous adoptons l'approche des Clocked Transition System (CTS) [167, 138], une extension des systèmes de transitions avec des variables supplémentaires, appellée horloges, pour mesurer l'écoulement du temps dans le système. Les horloges actives s'incrémentent uniformément quand le temps progresse. Certaines horloges "locales" peuvent être remises à zéro dans certaines transitions. Il y a toujours une horloge "maître", représentant le temps global, qui, elle, ne peut jamais être remise à zéro.

(1) État. Un état formalise la configuration dans laquelle le procédé se trouve à n'importe quel moment de son exécution.

Definition 8 (State)**.** *Un état d'un procédé* $P = (V, E, lower, upper, lab, Resource, Use, Capacity, Timing)$ *est un tuple* $s = (m, gc, lc)$ *tel que :*

- $m : V \cup E \mapsto \mathbb{N}$ *est la fonction, appelée "marquage", qui associe à chacun des noeuds et arcs un entier :*
  - *pour* $v \in V$, $m(v)$ *est le nombre de jetons,*
  - *pour* $e \in E$, $m(e)$ *est le nombre d'offres.*
- $gc \in \mathbb{N}$ *est l'horloge discrète globale représentant le temps courant passé sur le procédé,*
- $lc : V \mapsto \mathbb{N} \cup \{\epsilon\}$ *est l'horloge discrète locale représentant le temps courant passé sur l'action donnée :*

$$lc(v) \stackrel{def}{=} \begin{cases} n \in \mathbb{N} & \text{if } lab(v) = action \\ \epsilon & \text{otherwise} \end{cases}$$

L'ensemble de tous les états d'un procédé P est noté ***States****.*

Definition 9 (Initial State)**.** *Un état initial* $s_0 = (m_0, gc_0, lc_0)$ *du système est toujours défini comme suit :*

- *Tous les noeuds ont 0 jeton, sauf (i) les noeuds initiaux qui commencent avec 1 jeton et (ii) les paramètres d'entrées de l'activité démarrent avec un nombre de jetons qui varie entre leurs bornes inférieures et supérieures :*

$$m_0(v) = \begin{cases} 1 & \text{if } lab(v) = initial \\ n \in \{lower(v), ..., upper(v)\} & \text{if } lab(v) = activityParameter \wedge incoming(v) = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

- *L'horloge globale est initialisée à 0 :* $gc_0 = 0$.
- *Les horloges locales sont initialisées à 0 :*

$$lc_0(v) = \begin{cases} 0 & \text{if } lab(v) = action \\ \epsilon & \text{otherwise} \end{cases}$$

La dynamique d'un procédé, c'est-à-dire son exécution, est définie à travers la notion de *transition*. Pour passer d'un état à un autre, une transition est d'abord *activée* avant d'être *tirée*. Ainsi, la notion d'*activation* correspond à une précondition, tandis que la notion de *tir* correspond à une post-condition. Nous définissons en premier la notion d'*activation*, puis nous formalisons le concept de *tir*.

(2) Activation de transition. Une transition est dite *active* quand certaines préconditions sont remplies (pour autoriser le tir de la transition). En faisant une abstraction de la façon dont le modèle d'exécution de fUML exécute un AD, deux cas peuvent être distingués : (i) un noeud est prêt à être exécuté ; (ii) un noeud est prêt à se terminer. Dans notre formalisation, ces préconditions sont représentées respectivement par les prédicats **enabledStart** et **enabledFinish**. De plus, il est important de noter que pour gérer l'écoulement du temps, le système peut aussi progresser en utilisant le prédicat **enabledTime**.
Considérons un procédé $P = (V, E, lower, upper, lab, Resource, Use, Capacity, Timing)$ et un état $s = (m, gc, lc)$. Pour simplifier notre notation, nous assumons que s est implicitement disponible dans les prédicats suivants.

1. **enabledStart** est le prédicat qui détermine si un noeud v est prêt pour être exécuté. Sa définition formelle repose sur les prédicats auxiliaires suivants :
   - La première condition correspond à vérifier si le noeud n'est pas déjà en train de s'exécuter (ne possède pas de jeton) et possède des arcs entrants :

$$pAll(v) \stackrel{def}{=} (m(v) = 0) \wedge incoming(v) \neq \emptyset$$

   - Un noeud *activity* a besoin d'une offre sur tous ses arcs entrants :

$$pNode(v) \stackrel{def}{=} pAll(v) \wedge \bigwedge_{e \in incoming(v)} (m(e) > 0)$$

   - Un noeud *action* étend ce comportement avec des pins entrants, sortants et une liste de ressources utilisées, ainsi que le nombre d'offres sur les noeuds entrants et la disponibilité des ressources sont aussi vérifiés. Nous introduisons d'abord $actionResource : Resource \mapsto 2^V$, la fonction qui renvoie toutes les actions utilisant une ressource r :

$$\begin{aligned} actionResource(r) &\stackrel{def}{=} \{v \in Vlab(action) \mid r \in Use(v)\} \\ pAction(v) &\stackrel{def}{=} pNode(v) \wedge \bigwedge_{e \in incoming(input(v))} (m(e) \geq lower(v)) \\ &\wedge \bigwedge_{r \in Use(v)} (|\{o \in actionResource(r) \mid m(o) > 0 \\ &\qquad \wedge o \neq v\}| < Capacity(r)) \end{aligned}$$

   - Contrairement aux autres noeuds, un noeud *merge* a besoin qu'au moins un de ses arcs entrants possède une offre :

$$pMerge(v) \stackrel{def}{=} pAll(v) \wedge \bigvee_{e \in incoming(v)} (m(e) > 0)$$

   Ainsi, le test d'activation pour déterminer si un noeud peut s'exécuter correspond à :

$$enabledStart(v) \stackrel{def}{=} \begin{cases} pAction(v) & \text{if } lab(v) = action \\ pMerge(v) & \text{if } lab(v) = merge \\ pNode(v) & \text{otherwise} \end{cases}$$

2. **enabledFinish** est le prédicat qui détermine si un noeud est prêt à se terminer et repose sur les prédicats auxiliaires suivants :
   – Le noeud doit posséder des jetons :

$$haveTokens(v) \stackrel{def}{=} (m(v) > 0)$$

   – Un noeud *action* doit avoir son horloge locale au moins égale à son timing défini dans le procédé :

$$pTiming(v) \stackrel{def}{=} (lc(v) \geq Timing(v))$$

   Ainsi, le test d'activation pour déterminer si un noeud peut se terminer correspond à :

$$enabledFinish(v) \stackrel{def}{=} \begin{cases} haveTokens(v) \wedge pTiming(v) & \text{if } lab(v) = action \\ haveTokens(v) & \text{otherwise} \end{cases}$$

3. **enabledTime** détermine si les horloges locales peuvent être augmentées. Les horloges peuvent être augmentées seulement quand il y a au moins une action qui s'exécute :

$$enabledTime() \stackrel{def}{=} \bigvee_{v \in V, lab(v)=action} ((m(v) > 0) \wedge (lc(v) < Timing(v)))$$

(3) tir de transition. Le tir d'une transition et l'effet que cela a sur un état peut être définis comme suit. Trois cas sont à distingués : (i) le tir d'une transition sur un noeud qui peut démarrer ; (ii) le tir d'une transition sur un noeud qui peut terminer ; (iii) le tir d'une transition pour représenter le temps qui s'écoule.
Considérons un second état $s' = (m', gc', lc')$. **fireStart**, **fireFinish** et **fireTime** expriment les contraintes qui doivent être satisfaites pour s'assurer que s' est un successeur de s. **fireStart** est un prédicat lié au démarrage d'un noeud (un noeud satisfaisant le prédicat d'activation **enabledStart**), **fireFinish** est un prédicat lié à la terminaison d'un noeud (un noeud qui satisfait le prédicat d'activation **enabledFinish**), et **fireTime** est un prédicat lié à l'incrémentation des horloges (si l'état courant satisfait le prédicat d'activation **enabledTime**). À des fins de clarté, nous assumons que s et s' sont implicitement disponibles dans les prédicats d'activation ci-dessous.
Nous introduisons d'abord le prédicat fz qui contraint à l'égalité le marquage de tous les noeuds et arcs de s et s', à l'exception de ceux donnés comme paramètre p :

$$fz(p \subseteq V \cup E) \stackrel{def}{=} \bigvee_{v \in V \cup E \setminus p} (m'(v) = m(v))$$

De la même manière, le prédicat $fzTime$ contraint à l'égalité l'horloge globale ainsi que les horloges locales de s et s', à l'exception de celles données comme paramètre p :

$$fzTime(p \subseteq Vlab(action)) \stackrel{def}{=} (gc' = gc) \wedge \bigvee_{v \in V \setminus p} (lc'(v) = lc(v))$$

1. **fireStart** est basé sur les prédicats auxiliaires suivants.
   – Un noeud *activity* est exécuté en ajoutant un jeton et en retirant les offres de ses arcs entrants :

$$sNode(v) \stackrel{def}{=} (m'(v) = m(v) + 1) \wedge \bigwedge_{e \in incoming(v)} (m'(e) = m(e) - m'(v)) \wedge fz(\{v\} \cup incoming(v))$$

– Un noeud *action* demande des conditions additionnelles dues à la présence des pins d'entrées et de sorties. Les offres des arcs entrants de ses pins d'entrées sont consommées jusqu'à la borne maximale autorisée par leurs multiplicités. Puis, des jetons sont produits sur les pins de sorties entre leur borne minimale et maximale :

$$sActionIPin(v) \stackrel{def}{=} \bigwedge_{\substack{i \in input(v),\\ inc \in incoming(i)}} ((\neg(upper(i) \geq m(inc)) \wedge m'(i) = m(inc)) \vee ((upper(i) \geq m(inc)) \wedge m'(i) = upper(i)))$$

$$sActionOPin(v) \stackrel{def}{=} \bigwedge_{o \in output(v)} (m'(o) \geq lower(o) \wedge m'(o) < upper(o))$$

$$sActionEdge(v) \stackrel{def}{=} \bigwedge_{\substack{e \in incoming(v)\\ \cup incoming(input(v))}} (m'(e) = m(e) - m'(v))$$

$$\begin{aligned} sAction(v) \stackrel{def}{=} & (m'(v) = m(v) + 1) \wedge sActionIPin(v) \\ & \wedge sActionOpin(v) \wedge sActionEdge(v) \\ & \wedge fz(\{v\} \cup input(v) \cup output(v) \\ & \quad \cup incoming(v) \cup incoming(input(v)) \end{aligned}$$

– Contrairement aux autres noeuds, un noeud *merge* est exécuté en retirant l'offre de seulement un seul de ses arcs entrants :

$$sMerge(v) \stackrel{def}{=} (m'(v) = m(v) + 1) \wedge (\bigvee_{\substack{e \in incoming(v),\\ m(e) > 0}} (m'(e) = m(e) - m'(v)) \wedge fz(\{e\} \cup \{v\}))$$

Ainsi, la transition pour démarrer l'exécution d'un noeud est caractérisée par :

$$fireStart(v) \stackrel{def}{=} fzTime(\emptyset) \wedge \begin{cases} sAction(v) & \text{if } lab(v) = action \\ sMerge(v) & \text{if } lab(v) = merge \\ sNode(v) & \text{otherwise} \end{cases}$$

2. **fireFinish** est basé sur les prédicats auxiliaires suivants.
   – Un noeud *activity* retire le jeton qu'il possède et l'offre sur tous ses arcs sortants :

$$fNode(v) \stackrel{def}{=} (m'(v) = m(v) - 1) \wedge \bigwedge_{e \in outgoing(v)} (m'(e) = m(e) + m(v)) \wedge fz(v \cup outgoing(v))$$

   – Un noeud *flowFinal* retire le jeton qu'il possède, mais ne l'offre pas :

$$fFlowFinal(v) \stackrel{def}{=} (m'(v) = m(v) - 1) \wedge fz(\{v\})$$

   – Un noeud *decision* retire le jeton qu'il possède et l'offre seulement sur un seul de ses arcs sortants :

$$fDecision(v) \stackrel{def}{=} (m'(v) = m(v) - 1) \wedge \bigvee_{e \in outgoing(v)} (m'(e) = m(e) + m(v) \wedge fz(\{v\} \cup \{e\}))$$

– Un noeud *action* demande en plus des conditions d'un noeud *activity*, de remettre à zéro à la fois les jetons sur ses pins de sorties et d'entrées, et de les offrir sur les arcs sortant de ses pins de sorties. De plus, son horloge locale est réinitialisée à zéro :

$$fActionPin(v) \stackrel{def}{=} \bigwedge_{p \in (output(v) \cup input(v))} (m'(p) = 0)$$

$$fActionEdge(v) \stackrel{def}{=} \bigwedge_{\substack{e \in (outgoing(v) \\ \cup outgoing(output(v)))}} (m'(e) = m(e) + m'(v))$$

$$\begin{aligned} fAction(v) \stackrel{def}{=} & (m'(v) = m(v) - 1) \wedge fActionPin(v) \wedge fActionEdge(v) \\ & \wedge (lc'(v) = 0) \wedge fz(\{v\} \cup input(v) \cup output(v) \\ & \cup outgoing(v) \cup outgoing(output(v)) \end{aligned}$$

Ainsi, la transition pour terminer un noeud est définie par :

$$fireFinish(v) \stackrel{def}{=} \begin{cases} fFlowFinal(v) \ \wedge fzTime(\emptyset) & \text{if } lab(v) = flowFinal \\ fDecision(v) \ \wedge fzTime(\emptyset) & \text{if } lab(v) = decision \\ fAction(v) \ \wedge fzTime(\{v\}) & \text{if } lab(v) = action \\ fNode(v) \ \wedge fzTime(\emptyset) & \text{otherwise} \end{cases}$$

3. **fireTime** incrémente en plus de l'horloge globale du système, les horloges locales de chacune des actions en cours d'exécution :

$$fireTime() \stackrel{def}{=} (gc' = gc + 1) \wedge \bigwedge_{\substack{v \in Vlab(action) \\ \wedge (m(v) > 0)}} (lc' = lc + 1) \wedge fz(\emptyset)$$

Il est maintenant possible de définir la relation de transition complète, c'est-à-dire la condition sous laquelle un état est le successeur d'un autre état. Globalement, quand un noeud *activityFinal* est exécuté, ou quand il n'y a aucun noeud qui peut soit commencer, soit terminer, l'exécution est terminée.

Definition 10 (Successor Relation)**.** *Soit* $P = (V, E, lower, upper, lab, Resource, Use, Capacity, Timing)$ *un procédé. Soit* $s = (m, gc, lc)$ *et* $s' = (m', gc', lc')$ *deux états de* P*.* s' *est le successeur de* s*, si et seulement si le prédicat* $transition(s, s')$ *est vrai :*

$$\begin{aligned} stepTokens() \stackrel{def}{=} & \bigvee_{v \in V \setminus Vlab(outPin) \cup Vlab(inPin)} (enabledStart(v) \wedge fireStart(v) \\ & \vee enabledFinish(v) \wedge fireFinish(v)) \end{aligned}$$

$$stepTime() \stackrel{def}{=} (enabledTime() \wedge fireTime())$$

$$\begin{aligned} transition(s, s') \stackrel{def}{=} & \bigwedge_{v \in Vlab(activityFinal)} (m(v) = 0) \\ & \wedge (stepTime() \vee stepTokens()) \end{aligned}$$

Ainsi, pour représenter une exécution de procédé, nous définissons la notion de trace :

Definition 11 (Trace)**.** *Soit* $P = (V, E, lower, upper, lab, Resource, Use, Capacity, Timing)$ *un procédé. Une* **Trace** *est une séquence d'état ordonné notée* $\sigma = \langle s_0, s_1, ..., s_n \rangle \in States^*$ *t.q. :* $\forall i \in \mathbb{N}, transition(\sigma[i], \sigma[i+1])$ *est vrai, avec* $\sigma[i] = s_i$ *et* s_0 *est l'état initial. L'ensemble de toutes les traces est noté* **Traces***(P).*

3.2.3 Exemple d'exécution

Prenons en exemple l'exécution du procédé de la figure 3.4. Nous allons décrire les fonctions du tuple de façon enumérative : le premier élément du couple est un élément du domaine de la fonction et le deuxieme élément est le résultat de la fonction sur l'élément. Soit le procédé $P = (V, E, lower, upper, lab, Resource, Use, Capacity, Timing)$ tel que :

$$
\begin{aligned}
V =& \{Initial, A, Fork, B, Decision, B1, B2, Merge, C, Join, D, Final\},\\
E =& \{(Initial, A), (A, Fork), (Fork, B), (Fork, C), (B, Decision),\\
& (Decision, B1), (Decision, B2, (B1, Merge), (B2, Merge),\\
& (Merge, Join), (C, Join), (Join, D), (D, Final))\}\\
lower =& \{(Initial, \epsilon), \ldots, (Final, \epsilon)\}\\
upper =& \{(Initial, \epsilon), \ldots, (Final, \epsilon)\}\\
lab =& \{(Initial, initial), (A, action), (Fork, fork), (B, action),\\
& (Decision, decision), (B1, action), (B2, action), (Merge, merge),\\
& (C, action), (Join, join), (D, action), (Final, activityFinal)\}\\
Resource =& \emptyset\\
Use =& \{(Initial, \emptyset), \ldots, (Final, \emptyset)\}\\
Capacity =& \emptyset\\
Timing =& \{(Initial, \epsilon), (A, 1), (Fork, \epsilon), (B, 1),\\
& (Decision, \epsilon), (B1, 1), (B2, 1), (Merge, \epsilon),\\
& (C, 1), (Join, join), (D, 1), (Final, activityFinal)\}
\end{aligned}
$$

Dans la suite, nous détaillons la trace de d'états nécessaire pour représenter l'exécution de la figure 3.4 selon le modèle formel défini précédemment. Par souci de clarté, seul le marquage des noeuds et arcs possédant au moins 1 jeton est noté. De la même façon, seules les horloges locales des actions dont leurs valeurs sont différentes de 0 sont notées. Entre chacun des états, nous notons les prédicats d'activation évaluée à *vrai*, et soulignons celui choisi. Ce choix est fait normalement de façon non déterministe dans le modèle présenté. Cependant, l'exemple d'exécution donné ci-dessous suit l'exécution de la figure 3.4.

Certains états supplémentaires sont nécessaires pour représenter l'écoulement du temps. De ce fait, bien que sur la figure 3.4 l'exécution se déroule en 21 étapes, notre modèle formel nécessite plus d'états pour prendre en compte le temps. Ainsi, pour faire coïncider les numéros d'état de la figure 3.4 avec ceux ci-dessous, nous nommons les états incrémentant le temps avec la notation $State_{Xbis}$. En revanche, nous avons délibérément associé au procédé P un besoin d'une seule unité de temps pour effectuer chacune des actions du procédé, afin de ne pas avoir une trace trop longue.

$$
\begin{aligned}
State_1 =& \{\{(Initial, 1), \ldots\},\\
& \{0\}\\
& \{\ldots\}\}\\
& \qquad \underline{enabledFinish(Initial)}\\
State_2 =& \{\{((Initial, A), 1), \ldots\},\\
& \{0\}
\end{aligned}
$$

$\{\ldots\}\}$

$\underline{enabledStart(A)}$

$State_3 = \{\{(A,1),\ldots\},$
$\{0\}$
$\{\ldots\}\}$

$\underline{enabledTime}$

$State_{3bis} = \{\{(A,1),\ldots\},$
$\{1\}$
$\{(A,1),\ldots\}\}$

$\underline{enabledFinish(A)}$

$State_4 = \{\{((A,Fork)),1),\ldots\},$
$\{1\}$
$\{\ldots\}\}$

$\underline{enabledStart(Fork)}$

$State_5 = \{\{(Fork,1),\ldots\},$
$\{1\}$
$\{\ldots\}\}$

$\underline{enabledFinish(Fork)}$

$State_6 = \{\{((Fork,B),1),((Fork,C),1),\ldots\},$
$\{1\}$
$\{\ldots\}\}$

$\underline{enabledStart(B)}, enabledStart(C)$

$State_7 = \{\{(B,1),((Fork,C),1),\ldots\},$
$\{1\}$
$\{\ldots\}\}$

$\underline{enabledTime}, enabledStart(C)$

$State_{7bis} = \{\{(B,1),((Fork,C),1),\ldots\},$
$\{2\}$
$\{(B,1),\ldots\}\}$

$\underline{enabledFinish(B)}, enabledStart(C)$

$State_8 = \{\{((B,Merge),1),((Fork,C),1),\ldots\},$
$\{2\}$
$\{\ldots\}\}$

$\underline{enabledStart(Decision)}, enabledStart(C)$

$State_9 = \{\{(Decision,1),((Fork,C),1),\ldots\},$
$\{2\}$
$\{\ldots\}\}$

$\underline{enabledFinish(Decision)}, enabledStart(C)$

$State_{10} = \{\{((Decision,B2),1),((Fork,C),1),\ldots\},$

$$
\begin{array}{rl}
 & \{2\} \\
 & \{\ldots\}\} \\
 & \underline{enabledStart(B2)}, enabledStart(C) \\
State_{11} = & \{\{(B2,1),((Fork,C),1),\ldots\}, \\
 & \{2\} \\
 & \{\ldots\}\} \\
 & \underline{enabledTime}, enabledStart(C) \\
State_{11bis} = & \{\{(B2,1),((Fork,C),1),\ldots\}, \\
 & \{3\} \\
 & \{(B2,1),\ldots\}\} \\
 & enabledFinish(B2), \underline{enabledStart(C)}
\end{array}
$$

$$
\begin{array}{rl}
State_{12} = & \{\{(B2,1),(C,1),\ldots\}, \\
 & \{3\} \\
 & \{(B2,1),\ldots\}\} \\
 & enabledFinish(B2), \underline{enabledTime} \\
State_{12bis} = & \{\{(B2,1),(C,1),\ldots\}, \\
 & \{4\} \\
 & \{(B2,2),(C,1),\ldots\}\} \\
 & enabledFinish(B2), \underline{enabledFinish(C)} \\
State_{13} = & \{\{(B2,1),((C,Join),1),\ldots\}, \\
 & \{4\} \\
 & \{(B2,2),\ldots\}\} \\
 & \underline{enabledFinish(B2)} \\
State_{14} = & \{\{((B2,Merge),1),((C,Join),1),\ldots\}, \\
 & \{4\} \\
 & \{\ldots\}\} \\
 & \underline{enabledStart(Merge)} \\
State_{15} = & \{\{(Merge,1),((C,Join),1),\ldots\}, \\
 & \{4\} \\
 & \{\ldots\}\} \\
 & \underline{enabledFinish(Merge)} \\
State_{16} = & \{\{((Merge,Join),1),((C,Join),1),\ldots\}, \\
 & \{4\} \\
 & \{\ldots\}\} \\
 & \underline{enabledStart(Join)} \\
State_{17} = & \{\{(Join,1),\ldots\}, \\
 & \{4\} \\
 & \{\ldots\}\}
\end{array}
$$

$$
\begin{aligned}
& & & \underline{enabledFinish(Join)} \\
State_{18} =& \{\{((Join, D), 1), \ldots\}, \\
& \{4\} \\
& \{\ldots\}\} \\
& & & \underline{enabledStart(D)} \\
State_{19} =& \{\{(D, 1), \ldots\}, \\
& \{4\} \\
& \{\ldots\}\} \\
& & & \underline{enabledTime} \\
State_{19bis} =& \{\{(D, 1), \ldots\}, \\
& \{5\} \\
& \{(D, 1), \ldots\}\} \\
& & & \underline{enabledFinish(D)} \\
State_{20} =& \{\{((D, Final), 1), \ldots\}, \\
& \{5\} \\
& \{\ldots\}\} \\
& & & \underline{enabledStart(Final)} \\
State_{21} =& \{\{(Final, 1), \ldots\}, \\
& \{5\} \\
& \{\ldots\}\}
\end{aligned}
$$

La trace d'exécution $\mathcal{T}$ correspond donc à :

$$\mathcal{T} = \langle State_1, State_2, \ldots, State_{21} \rangle$$

La prochaine section présente la formalisation des propriétés intéressantes à vérifier sur un procédé.

3.3 Formalisation des propriétés pour la vérification d'un procédé

Afin d'étudier les propriétés du procédé modélisé en utilisant notre formalisme, nous avons besoin d'une logique formelle [76]. Beaucoup de logiques existent et peuvent exprimer différents types de propriétés, telles que CTL [107] ou LTL [204]. En raison de sa sémantique de temps linéaire, qui est plus appropriée dans le contexte de la vérification de procédé métiers, LTL a été clairement préférée par rapport à la sémantique de branchement du temps de CTL [162]. Dans notre cas, toutes nos propriétés peuvent être exprimées en utilisant LTL. En effet, LTL est généralement utilisée pour la spécification et la vérification de propriétés formelles par rapport à une trace d'exécution.

Les spécifications (formules) LTL sont construites à partir de propositions atomiques, de connecteurs booléens ($\wedge, \neg$) et d'opérateurs temporels (X "suivant", U "jusqu'à").

Definition 12 (Syntaxe de LTL)**.** *Soit AP un ensemble de propositions atomiques, alors les formules de LTL sont définies inductivement par les règles suivantes :*

– *chaque proposition atomique $p \in AP$ est une formule,*

– *si f et g sont des formules alors $f \wedge g$, $\neg f$, Xf et fUg en sont aussi.*

Les abréviations habituellement utilisées sont :

(ou)	$f \vee g$	$\Leftrightarrow$	$\neg(\neg f \wedge \neg g)$
(vrai)	True	$\Leftrightarrow$	$f \vee \neg f$
(faux)	False	$\Leftrightarrow$	$\neg$ True
(finalement)	$\mathbf{F}g$	$\Leftrightarrow$	True Ug
(toujours)	$\mathbf{G}g$	$\Leftrightarrow$	$\neg\mathbf{F}\neg g$

Example 1. *La propriété de sûreté "durant toute l'exécution, x est supérieur à 0" se traduit en logique LTL par* $\mathsf{G}\ (x > 0)$

Il existe deux sémantiques pour la LTL : (1) la sémantique classique LTL est définie sur les executions infinies [204] du système considéré, et (2) la sémantique LTL borné où la formule est anaysée seulement sur une profondeur borné k [25].

Dans notre cas, il est plus naturel d'utiliser la deuxieme sémantique car nos exécutions (trace def. 11) sont finies. [1] Cependant, l'utilisation de la première sémantique ne pose pas de problème majeur car il suffit d'étendre les traces finies en trace infinies en bégayant sur le dernier état de la trace.

Pour la généralité du propos, nous considérons que nos traces sont infinies et nous donnons la sémantique LTL classique dans la définition suivante. [2] Nous étendons donc l'ensemble $Traces(P)$ de la definition 11 en $Traces(P)^\omega$

Definition 13 (Sémantique de LTL)**.** *Soit P un procédé, s un état de P et $\sigma = \langle s_0, s_1, \ldots\rangle$ une trace de P, alors pour une proposition atomique p et deux formules LTL quelconques f et g nous avons :*

- $\sigma \models p \quad \Leftrightarrow \quad p \in \Pi(\sigma_{in}(0))$,
- $\sigma \models f \wedge g \quad \Leftrightarrow \quad (\sigma \models f) \wedge (\sigma \models g)$,
- $\sigma \models \neg f \quad \Leftrightarrow \quad \neg(\sigma \models f)$,
- $\sigma \models Xf \quad \Leftrightarrow \quad \sigma^1 \models f$,
- $\sigma \models fUg \quad \Leftrightarrow \quad \exists i$ *tel que* $(\forall j < i, \sigma^j \models f) \wedge (\sigma^i \models g)$.

Nous définissons alors la satisfaction de f par P comme suit :

- $P \models f \quad \Leftrightarrow \quad \forall \sigma \in Traces(P)^\omega,\ \sigma \models f$

La PLTL (Past LTL) [288] étend la LTL avec des opérateurs homologues permettant des déclarations par rapport aux états passés. Elle introduit les opérateurs temporels supplémentaires suivants :

– Y ou X^{-1} : *precedent state (hier)* : $\mathsf{Y}\ p$ signifie que p est vrai dans l'état précédent le long de l'exécution.
– P ou F^{-1} : *previously (avant)* : $\mathsf{P}\ p$ signifie que p est vrai avant dans un état de l'exécution.
– H ou G^{-1} : *historically (toujours avant)* : $\mathsf{H}\ p$ signifie que p est vrai dans tous les précédents états de l'exécution.
– S ou U^{-1} : *since* (depuis) : $p\ \mathsf{S}\ q$ signifie que p est toujours vrai jusqu'à un état précédent où q est vrai.

La PLTL est particulièrement utile pour exprimer des spécifications en langages naturels qui utilisent souvent des références vers des évènements qui se sont produit dans le passé. Gabbay *et al.* [94, 93] ont prouvé que n'importe quelle propriété temporelle exprimée en utilisant la PLTL peut être traduite en propriété équivalente LTL (quand évaluée au début du chemin). En d'autres

1. Par essence, les procédés sont des procédures ayant un début et forcément une fin, et donc amenés à se terminer.
2. Pour le cas borné, nous renvoyons le lecteur intéressé vers [25].

termes, LTL avec les opérateurs du passé n'est pas plus expressif que la LTL classique. L'argument principal pour l'utilisation de la PLTL est motivé par la pratique : elle permet de spécifier les propriétés de façon plus succincte, directe et naturelle que la LTL classique.

Example 2. *La propriété "toute subvention (grant) est précédée par une demande (request)" se traduit en logique LTL classique par :*

$$\neg((\neg request)\ \mathsf{U}\ (grant \wedge \neg request))$$

et peut-être aussi exprimée de manière plus directe en utilisant les opérateurs de la PLTL telle que :

$$\mathsf{G}\ (grant \implies \mathsf{F}^{-1}\ (request))$$

Ainsi, nous utilisons la (P)LTL afin d'exprimer nos propriétés dans la suite. Dans notre formalisme, les propositions atomiques sont soit statiques (liées à la structure du procédé) soit dynamiques lié à un état.

Definition 14 (Propositions atomiques statiques)**.** *Une proposition atomique statique peut avoir la forme suivante : $lower(v)$ op n, $upper(v)$ op n, $r \in Use(v)$, $Capacity(v)$ op n, ou $Timing(v)$ op n, avec $v \in V$, $r \in Ressource$, op $\in \{=, \neq, <, \leq, >, \geq\}$ et $n \in \mathbb{N}$.*

Definition 15 (Propositions atomiques dynamiques)**.** *Une proposition atomique dynamique peut avoir la forme suivante : $m(v)$ op n ou gc op n avec $v \in V \cup E$, op $\in \{=, \neq, <, \leq, >, \geq\}$ et $n \in \mathbb{N}$.*

Avant de passer à la formalisation des propriétés, nous introduisons d'abord quelques prédicats et ensembles pour simplifier leurs expressions. Encore une fois, nous assumons que s est implicitement disponible :

- $token : V \cup E \mapsto bool$ est le prédicat qui détermine si un noeud ou un arc v possède un jeton :

$$token(v) \stackrel{def}{=} (m(v) > 0)$$

- $start : V \mapsto bool$ est le prédicat qui determine si un noeud v commence à s'executer :

$$start(v) \stackrel{def}{=} (\neg token(v)) \wedge \mathsf{X}\ (token(v))$$

- $end : V \mapsto bool$ est le prédicat qui determine si un noeud v termine son execution :

$$end(v) \stackrel{def}{=} (\neg token(v)) \wedge \mathsf{X}^{-1}\ (token(v))$$

- $data : Vlab(outPin) \mapsto bool$ est le prédicat qui détermine si une donnée v est présente :

$$data(v) \stackrel{def}{=} (token(v)) \vee \bigwedge_{o \in outgoing(v)} (token(o))$$

- $parallel \subseteq 2^V$ est l'ensemble qui regroupe toutes les actions en cours d'execution :

$$parallel \stackrel{def}{=} \{v \in Vlab(action) \mid token(v)\}$$

3.3.1 Propriétés de flux de contrôle

(1.1) Possibilité de complétion. Cette propriété est exprimée en vérifiant que, finalement, au moins un noeud de type *activityFinal* sera exécuté :

$$\mathsf{F}\,(\bigvee_{v \in Vlab(activityFinal)} (token(v)))$$

(1.2) Complétion propre. Cette propriété est exprimée en vérifiant que, globalement, dès qu'un noeud de type *activityFinal* est exécuté, cela implique qu'il n'y a pas d'autre noeud de type *action* en cours d'exécution.

$$\mathsf{G}\,(\bigvee_{v \in Vlab(activityFinal)} (token(v)) \implies \bigwedge_{v' \in Vlab(action)} (\neg token(v')))$$

(1.3) Transitions mortes. Vérifier que le procédé ne contient pas d'activité morte correspond à une propriété d'*accessibilité*. En effet, il est nécessaire de répondre à la question : "existe-t-il une exécution telle que `actionA` est exécutée". Pour répondre à cette question, la propriété est exprimée de manière à vérifier que `actionA` n'est *jamais* exécutée. Ainsi, dans ce cas-là, un contre-exemple prouve la possibilité d'exécuter cette action, et montre que la transition n'est pas morte :

$$\forall v \in Vlab(action), \mathsf{G}\,(\neg token(v))$$

3.3.2 Propriétés de flux de données

(2.1) Données manquantes. Cette propriété peut être vérifiée en s'assurant que quand un noeud a des offres sur ses flux de contrôle entrants, il aura finalement des offres sur ses pins d'entrées :

$$\mathsf{G}\,(\bigwedge_{v \in Vlab(action)} (pNode(v) \implies \mathsf{F}\,(pAction(v)))$$

(2.1) Données redondantes. Cette propriété peut être vérifiée en s'assurant que quand le procédé se termine, toutes les offres sur le flux de données doivent avoir été consommées :

$$\mathsf{G}\,(\bigvee_{v \in Vlab(activityFinal)} (token(v)) \implies \bigwedge_{e \in E, lab(target(e)) \in Object} (\neg token(e)))$$

3.3.3 Propriétés de Ressources

(3.1) Ressource manquante. Cette propriété peut être vérifiée en s'assurant que quand une action est prête pour être exécutée, ses ressources sont disponibles (ou avec une capacité suffisante) :

$$\mathsf{G}\,(\bigwedge_{\substack{v \in Vlab(action),\\ r \in Resource \wedge \\ r \in Use(v)}} (enabledStart(v) \implies (\,|\{o \in parallel \mid o \neq v \wedge r \in Use(o)\}| < Capacity(r))))$$

(3.1) Utilisation inefficace des ressources.

– Cette propriété peut être vérifiée en s'assurant que chaque ressource affectée au procédé est toujours utilisée au moins une fois :

$$\forall r \in Resource, \mathsf{F}\ (\bigvee_{v \in VLab(action), r \in Use(v)} (token(v)))$$

– Une version plus précise de cette propriété peut être définie en s'assurant que la capacité maximale d'utilisation de la ressource finit toujours par être atteinte :

$$\forall r \in Resource, \mathsf{F}\ (|\{v \in \{parallel\} \mid r \in Use(v)\}| = Capacity(r))$$

3.3.4 Propriétés de temps

(4.1) Durée du procédé. Soit $t \in \mathbb{N}$ la valeur représentant le nombre maximal d'unités de temps pour effectuer le procédé. Pour vérifier que le procédé termine toujours avant t correspond à :

$$\mathsf{F}\ (\bigvee_{v \in Vlab(activityFinal)} (token(v)) \wedge (gc < t))$$

(4.2) Contrainte de début/fin d'activité. Soit $t \in \mathbb{N}$ la valeur représentant un nombre maximal d'unités de temps.

– Pour vérifier que l'action "a" *commence* toujours *avant* t unité de temps correspond à :

$$\mathsf{G}\ (start(a) \implies (gc < t)))$$

– Pour vérifier que l'action "a" *commence* toujours *après* t unité de temps correspond à :

$$\mathsf{G}\ (start(a) \implies (gc > t)))$$

– Pour vérifier que l'action "a" *termine* toujours *avant* t unité de temps correspond à :

$$\mathsf{G}\ (end(a) \implies (gc < t)))$$

– Pour vérifier que l'action "a" *termine* toujours *après* t unité de temps correspond à :

$$\mathsf{G}\ (end(a) \implies (gc > t)))$$

(4.3) Contrainte relative de début/fin d'activité. Pour définir ces propriétés, nous utilisons le domaine infini des entiers, qui en font des formules (P)LTL non standard. Le problème provient d'une des limitations de la (P)LTL : il n'est pas possible de comparer les propositions atomiques de plusieurs états à la fois. Ainsi, la comparaison de deux états donnés (ex. comparer l'horloge globale de deux états différents) est impossible. Une solution possible pour pallier cette limitation consiste à augmenter la définition de l'état (`State`) avec l'information nécessaire pour comparer les états. Par exemple, stocker dans chaque état un ensemble contenant chacun des noeuds déjà exécutés ainsi que l'horloge globale à ce moment-là. Par souci de lisibilité, nous allons garder l'écriture avec les données infinies. Soit $t \in \mathbb{N}$ la valeur représentant un nombre maximal d'unité de temps.

– Pour vérifier que l'action "a" *commence* toujours *avant* t unité de temps après que l'action "b" se soit exécutée correspond à :

$$\forall i \in \mathbb{N}, \mathsf{G}\ (end(b) \wedge (i = gc) \implies \mathsf{F}\ (start(a) \wedge (gc < i + t)))$$

– Pour vérifier que l'action "a" *commence* toujours *après* t unité de temps après que l'action "b" se soit exécutée correspond à :

$$\forall i \in \mathbb{N}, \mathsf{G}\ (end(b) \wedge (i = gc) \implies \mathsf{F}\ (start(a) \wedge (gc > i + t)))$$

– Pour vérifier que l'action "a" *termine* toujours *avant* t unité de temps après que l'action "b" se soit exécutée correspond à :

$$\forall i \in \mathbb{N}, \mathsf{G}\ (start(b) \wedge (i = gc) \implies \mathsf{F}\ (end(a) \wedge (gc < i + t)))$$

– Pour vérifier que l'action "a" *termine* toujours *après* t unité de temps après que l'action "b" se soit exécutée correspond à :

$$\forall i \in \mathbb{N}, \mathsf{G}\ (start(b) \wedge (i = gc) \implies \mathsf{F}\ (end(a) \wedge (gc > i + t)))$$

3.3.5 Propriétés métiers

Cette section présente la formalisation de l'ensemble des propriétés métiers du tableau 2.1 de la section 2.2.2 par rapport au modèle formel présenté précédemment.

Existence Permet d'exprimer des cardinalités concernant le nombre d'exécutions des noeuds :

– $existence(a)$ est la propriété qui spécifie que le noeud $a \in V$ est exécuté au moins 1 fois :

$$\mathsf{F}\ (token(a))$$

– $existence\_2(a)$ est la propriété qui spécifie que le noeud $a \in V$ est exécuté au moins 2 fois :

$$\mathsf{F}\ (token(a) \wedge \mathsf{X}\ (existence(a)))$$

– $existence\_3(a)$ est la propriété qui spécifie que le noeud $a \in V$ est exécuté au moins 3 fois :

$$\mathsf{F}\ (token(a) \wedge \mathsf{X}\ (existence\_2(a)))$$

– $existence\_N(a)$ est la propriété qui spécifie que le noeud $a \in V$ est exécuté au moins N fois :

$$\mathsf{F}\ (token(a) \wedge \mathsf{X}\ (existence\_N\text{-}1(a)))$$

– $abscence(a)$ est la propriété qui spécifie que le noeud $a \in V$ est exécuté 0 fois :

$$\mathsf{G}\ (\neg token(a))$$

– $abscence\_2(a)$ est la propriété qui spécifie que le noeud $a \in V$ est exécuté au plus 1 fois :

$$\neg existence\_2(a)$$

– $abscence\_3(a)$ est la propriété qui spécifie que le noeud $a \in V$ est exécuté au plus 2 fois :

$$\neg existence\_3(a)$$

– $abscence\_N(a)$ est la propriété qui spécifie que le noeud $a \in V$ est exécuté au plus N fois :

$$\neg existence\_N{+}1(a)$$

– $range\_N\_M(a)$ est la propriété qui spécifie que le noeud $a \in V$ est exécuté entre N et M fois :

$$existence\_N(a) \wedge absence\_M(a)$$

Relation Permet d'exprimer des contraintes sur l'ordre d'exécution des noeuds :

– $relation\_before(a, b)$ est la propriété qui spécifie que $a \in V$ est toujours exécuté avant $b \in V$:

$$\mathsf{G}\ (token(b) \implies \mathsf{F}^{-1}\ (token(a)))$$

– $relation\_after(a, b)$ est la propriété qui spécifie que $a \in V$ est toujours exécuté après $b \in V$:

$$\mathsf{G}\ (token(b) \implies \mathsf{F}\ (token(a)))$$

– $relation\_par(a, b)$ est la propriété qui spécifie que $a \in V$ est toujours exécuté en même temps que $b \in V$:

$$\mathsf{G}\ (token(a) \implies \mathsf{F}\ (token(a) \wedge token(b)))$$

– $relation\_excl(a, b)$ est la propriété qui spécifie que $a \in V$ est toujours exécuté en exclusion de $b \in V$:

$$\mathsf{G}\ (token(a) \implies \neg token(b))$$

– $relation\_between(a, b, c)$ est la propriété qui spécifie que $a \in V$ est toujours exécuté entre $b \in V$ et $c \in V$:

$$\mathsf{G}\ (token(a) \implies \mathsf{F}^{-1}\ (tokens(b)) \wedge \mathsf{F}\ (token(c)))$$

ExistenceData Permet d'exprimer des contraintes sur la présence ou non de données :

– $existence\_data(d)$ est la propriété qui spécifie que la donnée $d \in Vlab(outPin)$ est disponible à la fin de l'exécution :

$$\mathsf{F}\ (data(d))$$

RelationData Permet d'exprimer des contraintes sur l'ordre de création des données :

– $relation\_data\_before(da, db)$ est la propriété qui spécifie que la donnée $da \in Vlab(outPin)$ est disponible avant la donnée $db \in Vlab(outPin)$:

$$\mathsf{G}\ ((data(db)) \implies \mathsf{F}^{-1}\ ((data(da))))$$

– $relation\_data\_after(da, db)$ est la propriété qui spécifie que la donnée $da \in Vlab(outPin)$ est disponible après la donnée $db \in Vlab(outPin)$:

$$\mathsf{G}\ ((data(db)) \implies \mathsf{F}\ ((data(da))))$$

– $relation\_data\_par(da, db)$ est la propriété qui spécifie que la donnée $da \in Vlab(outPin)$ est disponible en même temps que la donnée $db \in Vlab(outPin)$:

$$\mathsf{G}\ ((data(da)) \implies \mathsf{F}\ ((data(da)) \wedge (data(db))))$$

– $relation\_data\_excl(da, db)$ est la propriété qui spécifie que la donnée $da \in Vlab(outPin)$ est disponible en exclusion de la donnée $db \in Vlab(outPin)$:

$$\mathsf{G}\ ((data(da)) \implies \neg(data(db)))$$

RelationDataActivity Permet d'exprimer des contraintes sur l'ordre de création des données par rapport à des actions :

– $relation\_data\_act\_before(d, a)$ est la propriété qui spécifie que la donnée $d \in Vlab(outPin)$ est disponible avant l'exécution de l'action $a \in Vlab(action)$:

$$\mathsf{G}\ (token(a) \implies \mathsf{F}^{-1}\ (data(d)))$$

– $relation\_data\_act\_after(d, a)$ est la propriété qui spécifie que la donnée $d \in Vlab(outPin)$ est disponible après l'exécution de l'action $a \in Vlab(action)$:

$$\mathsf{G}\ (token(a) \implies \mathsf{F}\ (data(d)))$$

– $relation\_data\_act\_during(d, a)$ est la propriété qui spécifie que $d \in Vlab(outPin)$ est disponible pendant l'exécution de l'action $a \in Vlab(action)$:

$$\mathsf{G}\ (token(a) \implies \mathsf{F}\ (token(a) \wedge data(d)))$$

– $relation\_data\_act\_excl(d, a)$ est la propriété qui spécifie que $d \in Vlab(outPin)$ est disponible en exclusion de l'exécution de l'action $a \in Vlab(action)$:

$$\mathsf{G}\ (data(d) \implies \neg token(a))$$

– $relation\_data\_act\_between(d, a, b)$ est la propriété qui spécifie que $d \in Vlab(outPin)$ est disponible entre l'exécution de l'action $a \in Vlab(action)$ et $b \in Vlab(action)$:

$$\mathsf{G}\ (data(d) \implies \mathsf{F}^{-1}\ (token(a)) \wedge \mathsf{F}\ (token(b)))$$

ExistenceTimeActivity Permet d'exprimer des contraintes de temps réel par rapport à l'exécution des noeuds :

– $time\_activity\_before(a, t)$ est la propriété qui spécifie que $a \in Vlab(action)$ est exécuté avant $t \in \mathbb{N}$ unité de temps :

$$\mathsf{G}\ (token(a) \implies gc < t)$$

– $time\_activity\_after(a, t)$ est la propriété qui spécifie que $a \in Vlab(action)$ est executé après $t \in \mathbb{N}$ unité de temps :

$$\mathsf{G}\ (token(a) \implies gc > t)$$

– $time\_activity\_between(a, t1, t2)$ est la propriété qui spécifie que l'action $a \in Vlab(action)$ doit être executé entre $t1 \in \mathbb{N}$ et $t2 \in \mathbb{N}$ unité de temps :

$$\mathsf{G}\ (token(a) \implies gc > t1 \wedge gc < t2)$$

ExistenceTimeData Permet d'exprimer des contraintes de temps réel par rapport à la disponibilité des données :

- $time\_data\_before(d, t)$ est la propriété qui spécifie que la donnée $d \in Vlab(outPin)$ doit être disponible avant $t \in \mathbb{N}$ unité de temps :

$$\mathsf{G}\ (data(d) \implies gc < t)$$

- $time\_data\_after(d, t)$ est la propriété qui spécifie que la donnée $d \in Vlab(outPin)$ doit être disponible après $t \in \mathbb{N}$ unité de temps :

$$\mathsf{G}\ (data(d) \implies gc > t)$$

- $time\_data\_between(d, t1, t2)$ est la propriété qui spécifie que la donnée $d \in Vlab(outPin)$ doit être disponible entre $t1 \in \mathbb{N}$ et $t2 \in \mathbb{N}$ unité de temps :

$$\mathsf{G}\ (data(a) \implies gc > t1 \wedge gc < t2)$$

ExistenceTimeResource Permet d'exprimer des contraintes de temps réel par rapport à l'utilisation des ressources :

- $time\_resource\_before(r, t)$ est la propriété qui spécifie que la ressource $r \in Resource$ doit être utilisée avant $t \in \mathbb{N}$ unité de temps :

$$\mathsf{G}\ (\bigwedge_{a \in Vlab(action)} (token(a) \wedge r \in Use(a) \implies gc < t))$$

- $time\_resource\_after(r, t)$ est la propriété qui spécifie que la ressource $r \in Resource$ doit être utilisé après $t \in \mathbb{N}$ unité de temps :

$$\mathsf{G}\ (\bigwedge_{a \in Vlab(action)} (token(a) \wedge r \in Use(a) \implies gc > t))$$

- $time\_resource\_between(r, t1, t2)$ est la propriété qui spécifie que la resource $r \in Resource$ doit être utilisé entre $t1 \in \mathbb{N}$ et $t2 \in \mathbb{N}$ unité de temps :

$$\mathsf{G}\ (\bigwedge_{a \in Vlab(action)} (token(a) \wedge r \in Use(a) \implies gc > t1 \wedge gc < t2))$$

Comme expliqué dans la section 2.2.2 page 26, il est possible de définir des variations de cet ensemble de propriétés pour vérifier différents cas plus spécifiques. Par exemple, vérifier que l'activité `A` est toujours exécutée directement après l'activité `B` (sans possibilité d'exécuter d'autre activité entre ces deux exécutions) n'est pour le moment pas exprimable à partir de l'ensemble défini ci-dessus. Seule la contrainte spécifiant que `A` finira forcément par être exécuté dans le futur après `B` ($relation\_after(B, A)$) l'est. Par souci de simplicité d'utilisation, nous nous restreignons à ces propriétés basiques et essentielles, plûtot que de définir toutes les variations possibles. Proposer des centaines de propriétés métiers différents dont chacune d'entre elles est seulement une légère variation de la propriété précédente détruirait notre but concernant la simplicité d'utilisation de notre libraire.

Ainsi, nous définissons nos propriétés métiers par :

1. Un ensemble de propriétés basiques exprimées en PLTL, tel que la librairie proposée précédemment, et
2. des *connecteurs logiques* tels que $\vee, \wedge, \neg, \implies$.

Ainsi, la combinaison de ces propriétés basiques avec les opérateurs de logiques classiques permet d'exprimer un grand éventail de propriétés.

Example 3. *La propriété "l'activité* `A` *doit être exécutée avant* `B` et `C`*" peut être exprimée telle que : "*$relation\_before(A, B) \wedge relation\_before(A, C)$*".*

Il est important de noter que l'utilisation directe de la libraire (c'est-à-dire sans composer les propriétés "basiques" avec des connecteurs de la logique classique) permet déjà de couvrir la grande majorité des cas nécessaires.

3.4 Conclusion

Ce chapitre propose une formalisation en utilisant la FOL de la nouvelle spécification fUML, dans le contexte de la vérification de procédé basé sur UML AD. La formalisation est capable de gérer le flux de contrôle, de données, les ressources, et les contraintes temporelles du procédé de manière unifiée. Un grand nombre de propriétés sur toutes les différentes perspectives du procédé ont été formalisées en utilisant la PLTL. Ainsi, il est possible de vérifier des propriétés impliquant toutes les dimensions du procédé. L'ensemble de ces propriétés permet de garantir la validité d'un procédé basé sur UML AD.

Une fois la formalisation établie et l'ensemble des propriétés défini, la prochaine étape consiste à choisir un langage d'implémentation. Alloy [177] a été choisi dans ce but. Le prochain chapitre présente ALLOY4PV, notre implémentation de la formalisation présentée dans ce chapitre.

Chapitre 4

Alloy4PV : Un framework basé sur Alloy pour la vérification de procédés métiers

Ce chapitre présente Alloy4PV, un framework basé sur la logique Alloy et implémentant la formalisation du chapitre 3, permettant de vérifier un large éventail de propriétés métiers sur un modèle de procédé. La section 4.1 présente le langage Alloy ainsi que son outil associé pour analyser et vérifier une spécification Alloy. La section 4.2 motive le choix d'Alloy comme langage d'implémentation, puis présente les différents modules Alloy afin d'implémenter la formalisation. La section 4.3 présente l'outil graphique, intégré à eclipse, basé sur l'API d'Alloy, afin d'automatiser tout le processus de vérification.

4.1 Alloy : un langage et un outil pour les modèles relationnels

Alloy [126], souvent appelé *"lightweight formal methods"* [125] par son auteur, provient du fait qu'il apporte le bénéfice des approches traditionnelles des méthodes formelles à un coût plus léger, sans avoir besoin d'un grand investissement initial. Il permet de modéliser le système de façon incrémentale en fonction de la perception de son utilisateur, en explorant les problèmes potentiels le plus tôt possible de façon automatique. Cette section présente non seulement le langage Alloy et son outil associé, mais aussi un cas d'étude simple de modélisation statique et dynamique de modèle Alloy afin d'illustrer son utilisation sur un cas concret.

4.1.1 Le langage Alloy

Alloy est un langage de modélisation déclaratif développé par le MIT basé sur la logique de premier ordre (FOL) associé aux opérateurs du calcul relationnel. Le langage Alloy a été grandement inspiré par la notation Z [236]. De la même façon, il décrit toutes les structures (dans l'espace et le temps) avec un minimum de notions mathématiques. Cependant, Alloy est encore plus simple que Z. Alloy est très fortement influencé par la modélisation orientée-objet. De la même façon, il permet de facilement classifier les objets, et associer des propriétés aux objets selon leur classification. Alloy supporte les "expressions de navigation", qui sont dominantes dans la modélisation objet, en utilisant une syntaxe particulièrement simple et uniforme. La logique a été maintenue exempte de toute notion qui l'aurait rattachée à un langage de programmation ou modèle d'exécution particulier.

Un modèle Alloy consiste en un nombre de déclarations de *signature* (`sig`), *relations*, *faits* (`fact`) et *prédicats* (`pred`). Chacune des signatures indique un ensemble d'*atomes*, représentant les entités basiques d'Alloy. Un atome est une entité primitive ayant trois propriétés : (1) indivisible, elle ne peut être fracturée en plusieurs sous partie ; (2) immutable, ses propriétés ne changent pas au cours du temps ; (3) non interprétée, il n'a pas de propriété innée (ex. comme les chiffres). Chacune des *relations* est déclarée sous une signature, afin de représenter les liens entre les signatures. Alloy introduit les *faits* comme des déclarations définissant des contraintes sur les éléments du modèle. Les *prédicats* peuvent être vus comme des contraintes paramétrables, et être utilisés à partir d'autres prédicats et faits.

Une des caractéristiques importantes du langage Alloy est qu'il gère les scalaires et les ensembles comme des relations, et ce afin de généraliser la notion de *jointure relationnelle*. Par exemple, une relation entre deux atomes `A1` et `A2` est représentée par la paire `{(A1,A2)}`. Un ensemble comme `{A1,A2}` est représenté comme un ensemble de relations unaires tel que `{(A1),(A2)}`. Finalement, un scalaire est représenté comme une relation unaire à un seul élément (un singleton). Par exemple, le scalaire `A1` sera représenté dans Alloy par `{(A1)}`. Traité à la fois les scalaires et les ensembles comme des relations est une propriété intéressante d'Alloy, qui a été introduite pour la première fois dans les travaux de Tarski [241]. En conséquence, le même opérateur de jointure (".") peut être appliqué sur les scalaires, sur les ensembles et sur les relations. Ainsi, changer la "multiplicité" d'une relation dans sa déclaration n'implique pas de devoir changer la contrainte dans laquelle elle apparaît (et ce, peu importe si l'opérateur de jointure fait correspondre un élément vers un scalaire ou vers un ensemble). Se passer de distinction entre les ensembles et les scalaires permet aussi de rendre les contraintes plus uniformes et faciles à écrire, tout en éliminant le problème d'application partielle de fonction, permettant d'éliminer le besoin de valeurs spéciales "non définies". Cependant, présenter tous les détails du langage Alloy dépasse la portée de ce travail. Jackson, l'auteur d'Alloy, a minutieusement expliqué tous ses principes et concepts dans son remarquable ouvrage [126].

4.1.2 Alloy Analyzer

L'Alloy Analyzer (figure 4.1) est un outil complètement automatique qui trouve des *instances* de spécifications Alloy (ou modèle Alloy), c'est-à-dire qu'il est capable d'assigner des valeurs aux ensembles et relations de la spécification de manière à ce que, pour chaque affectation, toutes les formules de la spécification soient valides. Cet outil est capable d'effectuer deux types d'analyse. Il permet de produire des instances du modèle (spécification Alloy) satisfaisant une condition, ou encore de vérifier que certaines propriétés sont satisfaites par le modèle. L'Alloy Analyzer est donc capable d'effectuer 2 types d'analyses : (1) de la *simulation* en utilisant le mot clé "`run`" et (2) de la *vérification* en utilisant le mot clé "`check`".

L'Alloy Analyzer fonctionne en transformant automatiquement une spécification Alloy en un problème de satisfiabilité booléenne (SAT) en utilisant le *model-finder* Kodkod [245]. Ce problème est ensuite résolu par un SAT solveur tel que SAT4J [158], MiniSat [70] ou Berkmin [101], et le résultat de l'analyse est affiché à l'utilisateur. La satisfiabilité booléenne est le problème de trouver une affectation à une formule booléenne de manière à ce que toute la formule soit évaluée à vrai. Aussi facile que cela puisse paraître, il fait partie des problèmes complexes et toujours en suspend dans l'informatique pour répondre à ce problème de manière *efficace*. Il existe une grande et florissante communauté autour de la construction d'algorithme pour la résolution SAT afin de résoudre des formules toujours plus complexes. Chaque année, une compétition [1] est tenue pendant laquelle les dernières améliorations concernant la résolution SAT sont mises en compétitions entre elles sur des problèmes communs. Grâce à un grand nombre de compétiteurs,

1. *"The SAT Competition"*, http ://www.satcompetition.org/

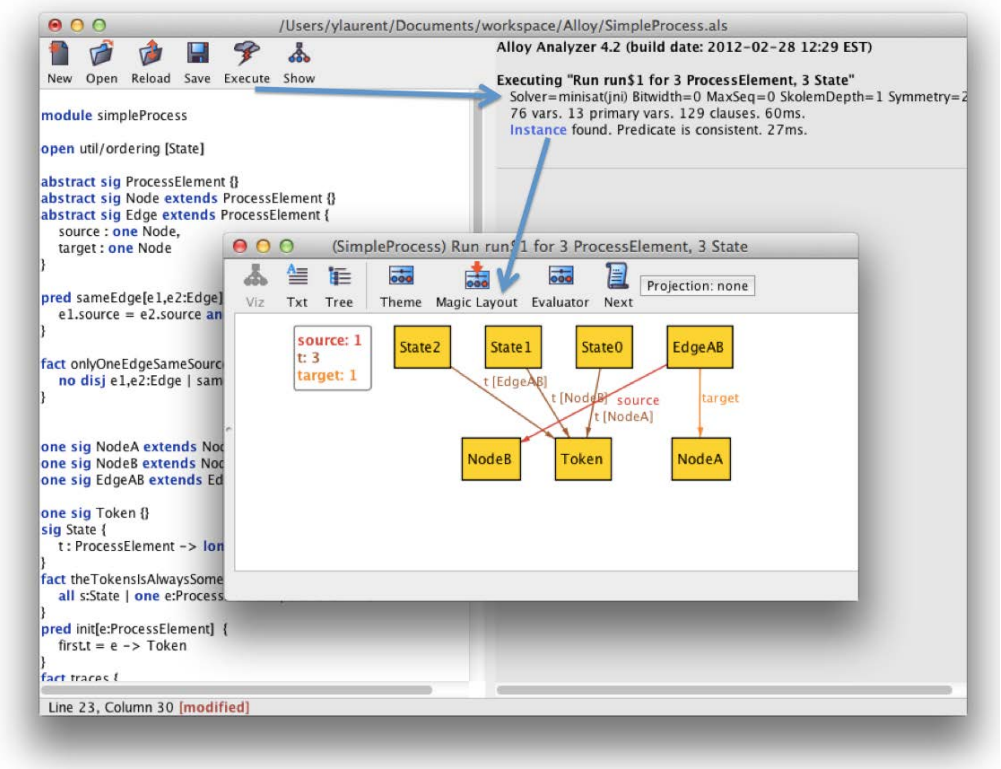


FIGURE 4.1 – Screenshot de l'Alloy Analyzer

un format standard d'entrée ainsi qu'une facilité pour évaluer la performance des SAT solveur, de nombreuses améliorations ont été introduites au cours de ces dix dernières années. De nos jours, les SAT solveurs ont étés appliqués sur une grande variété de problèmes qui étaient inconcevables quelques années auparavant [206].

Afin de gérer le problème d'explosion d'état [250], l'utilisateur spécifie une *borne* sur les éléments du modèle (les signatures) afin de limiter le domaine. Une borne est un entier positif limitant le nombre d'instances de chacun des éléments d'une instance du système analysé. Cette borne est nécessaire du fait que *"la logique de premier ordre n'est pas décidable"* [124]. Il est impossible de créer un outil automatique permettant de dire, avec fiabilité, si une affirmation est valide, et ce quelles que soient les affectations possibles. Ainsi, dans le cas d'une vérification (`check`), si aucun contre-exemple n'est trouvé, l'affirmation *pourrait* toujours être invalide pour une borne supérieure. De la même manière, dans le cas d'une simulation (`run`), si aucune instance n'est trouvée, la condition pourrait être valide pour une borne supérieure.

Une autre fonction importante de l'Alloy Analyzer est le débogage de modèle sur-contraint (aussi connu sous le nom de UnSAT, pour *"unsatisfiable core"*) [230]. Plus particulièrement, si un modèle est trop contraint de façon à ce que l'analyseur ne puisse trouver une instance satisfaisant le modèle, l'outil est capable d'indiquer les déclarations conflictuelles.

Finalement, l'Alloy Analyzer peut énumérer les instances ou contre-exemples de façon à produire un résultat différent à chaque fois.

Il est important de noter que les auteurs d'Alloy fournissent l'API Java sur laquelle l'Alloy Analyzer a été construit. En conséquence, il est très facile d'intégrer la puissance d'analyse d'Alloy dans les outils de l'IDM.

```
1  module simpleProcess
2
3  abstract sig ProcessElement {}
4  abstract sig Node extends ProcessElement {}
5  abstract sig Edge extends ProcessElement {
6    source : one Node,
7    target : one Node
8  }
9  pred sameEdge[e1,e2:Edge] {
10   e1·source = e2·source and e1·target = e2·target
11 }
12 fact onlyOneEdgeSameSourceTarget {
13   no disj e1,e2:Edge | sameEdge[e1,e2]
14 }
15
16 one sig NodeA extends Node {}
17 one sig NodeB extends Node {}
18 one sig EdgeAB extends Edge {}
19
20 run {} for 3 ProcessElement
21 check { all e:Edge | not (e·source = e·target) } for 3 ProcessElement
```

Listing 4.1 – Modèle Alloy représentant des noeuds et arcs

4.1.3 Exemple statique et dynamique de modèles Alloy

Alloy permet de modéliser des systèmes à la fois statiques et dynamiques. Un modèle statique décrit des états, et non un comportement. Leurs propriétés peuvent être vues comme des invariants ; par exemple, vérifier qu'une liste est ordonnée. Un modèle dynamique décrit les *transitions* entre les états. Les propriétés sont des *opérations* ; par exemple, *comment* fonctionne un algorithme de tri.

Dans cette partie, nous présentons un cas d'utilisation de Alloy basé sur un modèle simple comprenant des noeuds et des arcs. Nous commençons par modéliser le système statiquement, puis nous lui rajouterons une dynamique basée sur un jeton transitant entre les noeuds et les arcs. Le but est de comprendre les principaux concepts d'Alloy sur un exemple simpliste, pour pouvoir appréhender son utilisation lors de la définition de ALLOY4PV (dans la section suivante).

Modélisation statique

Le listing 4.1 montre un modèle Alloy représentant des noeuds et des arcs. La ligne 1 déclare le nom du module, c'est l'équivalent d'un package en Java ; leur but est de regrouper les signatures et prédicats ensemble afin de faciliter la modularité. La ligne 3 déclare une signature abstraite appellée `ProcessElement`. Une signature abstraite est une signature sans aucune instance directe. Les lignes 4 et 5 déclarent des signatures en tant que sous-signatures (ou sous-ensembles) de `ProcessElement`, dues à la présence de la déclaration "`extends ProcessElement`". Cependant, la signature `Edge` a deux relations déclarées, `source` et `target` représentant chacune un seul `Node`. Le mot clé "`one`" implique la multiplicité de la relation, ici seulement un noeud ([1,1]). Il existe d'autres mots clé pour d'autres multiplicités, tels que "`lone`" ([0,1]), "`some`" ([1,*]) ou "`set`" ([0,*]). La Ligne 9 déclare un prédicat `sameEdge` prenant en paramètre deux `Edge`s. Ce prédicat spécifie si les deux arcs sont identiques, c'est-à-dire si leurs `source` et `target` sont identiques. La ligne 12 déclare un fait (`fact`) contraignant le système ; plus précisément aucun ("`no`") `Edge e1`

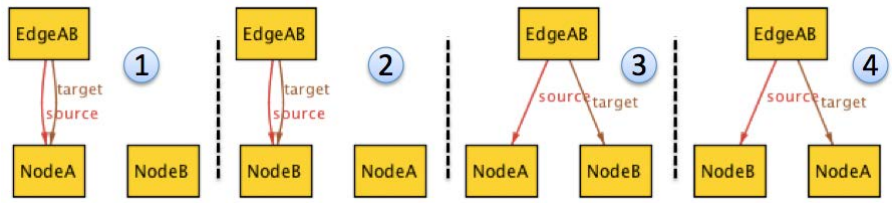


FIGURE 4.2 – Résultat de la commande "`run  for 3 ProcessElement`" sur le modèle Alloy de la figure 4.1

```
1 [···]
2
3 some sig NodeI extends Node {}
4 some sig EdgeI extends Edge {}
5
6 run { all e:Edge | not (e·source = e·target) } for 10 ProcessElement
```

Listing 4.2 – Modification du modèle Alloy de la figure 4.1

et `e2` disjoint (`disj`, c'est-à-dire `e1` $\neq$ `e2`) tel que le prédicat `sameEdge` est évalué à vrai. Ainsi, le système n'accepte aucun `Edge` identique. Les lignes 16 à 18 déclarent des signatures qui seront représentées comme des singletons due à la présence du littéral "`one`". Ainsi, un seul `NodeA` et `NodeB`, sous-ensemble de `Node`, est autorisé. De la même façon, un seul `EdgeAB` sous-ensemble de `Edge` est autorisé.

La ligne 20 définit une simulation ("`run`") vide, produisant une instance aléatoire du modèle satisfaisant les faits. L'analyseur essaie de trouver des affectations de variables pour chacune des relations de façon à ce que les contraintes soient vraies. La partie "`for 3 ProcessElement`" spécifie la borne pour les éléments du système. Plus particulièrement, l'Alloy Analyzer essaie de produire une instance du modèle en utilisant seulement trois atomes pour la signature `ProcessElement`. Dans cet exemple, nous avons déjà déclaré trois atomes, c'est-à-dire `NodeA`, `NodeB` et `Edge`, ce qui limite grandement le résultat de recherche.

La figure 4.2 montre les résultats de cette simulation pour ce problème. Seules 4 solutions existent. Les instances (1) et (2) définissent les relations "source" et "target" sur la même signature, `NodeA` ou `NodeB`. Les instances (3) et (4) définissent "source" soit sur `NodeA` ou `NodeB`, et inversement pour la relation "target".

Il est aussi possible de faire de la vérification. La ligne 21 définit une vérification ("`check`") pour s'assurer que, quel que soit l'`Edge`, "`source`" et "`target`" sont différents. Sur cet exemple, cette vérification trouve les 2 contre-exemples (1) et (2) de la figure 4.2.

Dans ces exemples, la borne des signatures est gardée faible à des fins de compréhension. Cependant, en modifiant les lignes 16 à 18 par les lignes 3 et 4 du listing 4.2 et en utilisant la simulation de la ligne 6, les modèles trouvés par l'Alloy Analyzer peuvent devenir plus complexes. En effet, les lignes 3 et 4 définissent deux signatures `NodeI` et `EdgeI` de type respectivement `Node` et `Edge` avec une multiplicité "`some`" ([0,*]). Cette fois-ci, la borne maximale des `ProcessElement` est de 10 éléments, avec comme contrainte à satisfaire que tous les `Edge`s doivent avoir une "`source`" et "`target`" différentes (ligne 6). Quelques exemples de résultat sont visibles sur la figure 4.3. Dans ce cas-ci, il n'est pas possible de lister exhaustivement dans ce document les solutions possibles.

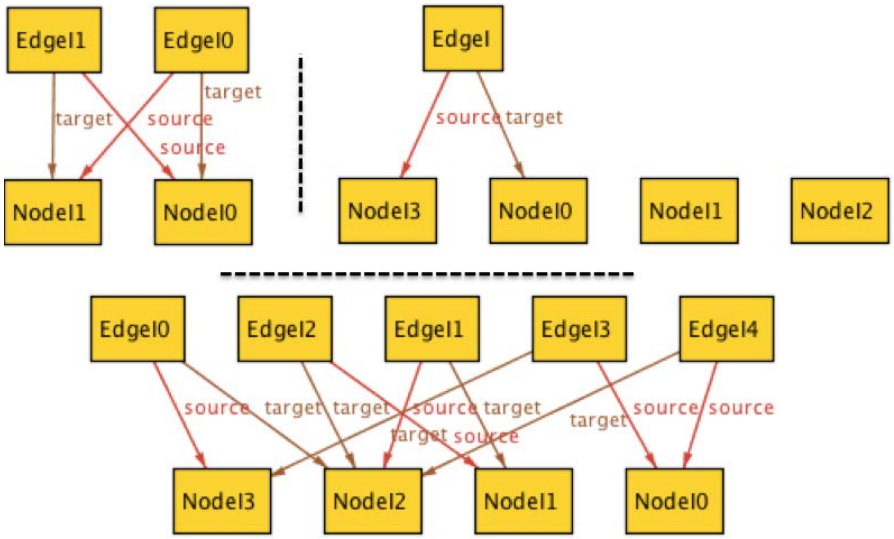


FIGURE 4.3 – Quelques exemples de simulation aléatoire du modèle Alloy de la figure 4.2

Modélisation dynamique : traces pattern

Alloy ne fournit pas de support direct pour spécifier des systèmes dynamiques dont l'état évolue au cours du temps ; il n'y a pas de notion de temps ou d'état mutable. Cependant, plusieurs idiomes et extensions ont été proposés pour aborder ce problème [99, 90]. Le "*traces pattern*" est l'un des plus populaires pour modéliser et raisonner par rapport à des opérations et des traces d'exécution [126]. Ce patron permet de modéliser une séquence d'exécution d'une machine abstraite. L'idée est de modéliser *explicitement* les états du système et de créer un *ordre total* et *linéaire* entre eux. Chaque paire d'états consécutifs est liée de façon à ce qu'ils satisfassent une opération donnée. Une opération est généralement modélisée comme un prédicat spécifiant la relation entre l'état avant et après la transition. Deux variantes de ce patron existent, connues respectivement comme "*global state*" et "*local state*" [126]. Dans le premier, tous les champs mutables sont définis dans une signature globale. Cela permet de grouper artificiellement tous les champs mutables ensemble. Dans le second, la signature de l'état est ajoutée localement à chacun des champs mutables. Cela permet d'être plus modulaire parce que les champs sont déclarés dans la signature à laquelle ils appartiennent naturellement. Dans ce document, nous utilisons seulement la variante "*global state*" pour modéliser nos systèmes dynamiques. Ce choix facilite la compréhension en faisant une séparation complète de la partie statique et dynamique du modèle Alloy.

Alloy propose un module (l'équivalent d'une bibliothèque dans un langage de programmation traditionnelle) permettant d'établir un ordre total et linéaire au-dessus d'une signature donnée. La figure 4.4 présente ce module. Ce module est générique, il permet d'imposer un ordre total sur n'importe quel type de signature (signature `S` sur cet exemple). Ce module propose plusieurs prédicats par rapport à l'ensemble des signatures ordonnées, c'est-à-dire par rapport à la *trace*. Par exemple, `first` et `last` retournent le premier et le dernier élément de la trace. À partir d'un élément de la trace, il est possible de savoir le prochain (`next`) ou le précédent (`prev`) élément. De la même manière, `nexts` et `prevs` retournent l'ensemble des éléments précédents et suivants d'un élément de la trace. Comme cela est visible sur la figure 4.4, d'autres fonctions et prédicats existent pour comparer les positions des éléments de la trace.

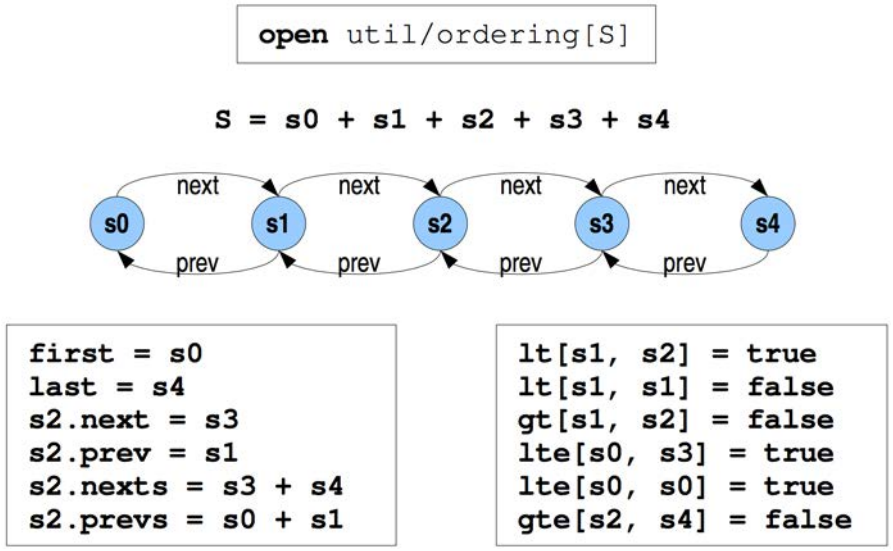


FIGURE 4.4 – Module d'ordonnancement de Alloy [178]

```
1  open util/ordering [State]
2
3  [···]
4
5  sig State {
6    t : ProcessElement →lone Token
7  }
8  one sig Token {}
9
10 fact traces {
11   init [NodeA]
12   all s: State - last | let s' = s·next | {
13     transition[s,s']
14   }
15 }
16 pred init[e:ProcessElement] { first·t = e →Token }
17 pred transition[s,s':State] { s·t ≠ s'·t }
18
19 fact theTokensIsAlwaysSomewhere {
20   all s:State | one e:ProcessElement | s·t·Token = e
21 }
22
23 run {} for 3 ProcessElement, 3 State
24 check { some s:State | s·t·Token = NodeB } for 3 ProcessElement, 3 State
```

Listing 4.3 – Excerpt of `Semantic.als`

Dans la suite, nous rajoutons de la "dynamique" aux modèles Alloy de la figure 4.1. L'idée est d'ajouter un jeton qui transite aléatoirement entre les noeuds et les arcs (`ProcessElement`). La seule contrainte importante est qu'un jeton ne peut pas rester deux fois de suite sur le même élément.

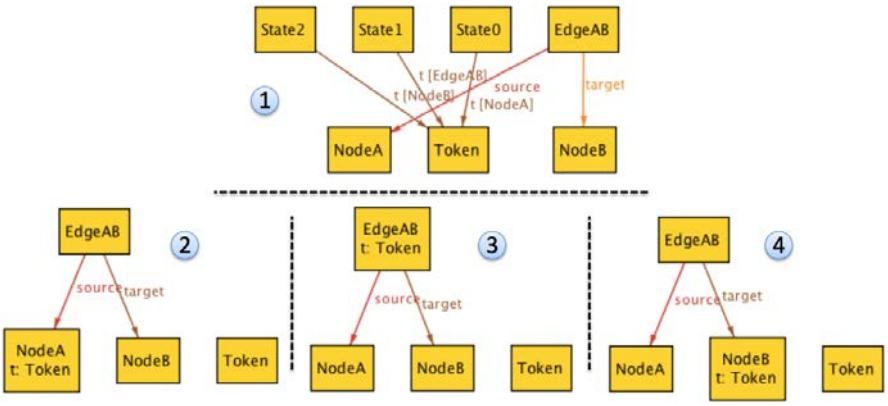


FIGURE 4.5 – Simulation du modèle Alloy figure 4.3

Le code Alloy de la figure 4.3 donne la solution au problème précédent en utilisant le *traces pattern*. Ligne 1, le module `ordering` d'Alloy est utilisé sur la signature `State`. Les lignes 5 à 7 définissent la signature `State` représentant *un* état de la trace. Sur cet exemple, l'état est défini par la relation "`t`" associant pour chaque `ProcessElement`, zéro ou un `Token` (`lone`). Ligne 8, le jeton est modélisé en déclarant une simple signature de type singleton nommé `Token`. Ligne 10, le fait `traces` contraint la trace de `State`s. Le prédicat `init` initialise l'état initial du système en imposant que le premier état (`first`) soit initialisé de façon à ce que son paramètre "`e`" détienne le jeton. Ici, c'est le noeud `NodeA` qui est initialisé avec le jeton. Les lignes 12 à 14 contraignent un à un chacun des états du système, quand "`s`" représente l'état courant et "`s'`" l'état suivant. Dans cet exemple, toutes ces paires d'états successeurs doivent satisfaire le prédicat "`transition`". Ce prédicat, défini dans la ligne 17, impose simplement que la relation "`t`" de l'état courant soit différente de celle de l'état suivant. Finalement, un fait est rajouté sur la ligne 19 afin d'imposer que sur tous les états, seulement *un* `ProcessElement` possède le jeton. Ligne 23, la simulation est la même que précédemment, sauf qu'une borne de 3 `State`s est utilisée. Ainsi, le système peut être analysé sur une longueur de trois transitions. La figure 4.5 montre le résultat de la simulation. L'étiquette (1) montre un modèle aléatoire trouvé par l'Alloy Analyzer. Les étiquettes (2), (3) et (4) présentent le même résultat quand l'affichage est projeté sur respectivement l'état `State0`, `State1` et `State2`. Sur cette simulation, le jeton est au début sur `NodeA`, puis transite vers `EdgeAB`, pour terminer sur `NodeB`. Bien sûr, beaucoup d'autres simulations sont possibles. Cet exemple est une simple simulation satisfaisant tous les faits et contraintes du modèle Alloy.

Ligne 24, la vérification permet de s'assurer que, quelle que soit l'exécution, il y aura des `States` tel que `NodeB` aura le jeton. Cette vérification retourne un contre-exemple. En effet, il existe beaucoup d'exécutions du modèle, de sorte que cette affirmation soit fausse. Un contre-exemple correspondrait à une trace où le jeton transite parmi ces éléments : {`NodeA`, `EdgeAB`, `NodeA`}. Ce type de propriété exprimé par rapport à une trace d'exécution peut être assimilé à une logique temporelle. De cette manière, Alloy est capable d'exprimer non seulement des propriétés LTL [47], mais aussi CTL avec des contraintes d'équité [249]

Cet exemple, bien que simpliste, montre un sous-ensemble des possibilités du langage Alloy et de son outil afin de faire de la simulation et de la vérification de modèles. La prochaine section présente notre framework ALLOY4PV. Ce dernier implémente la formalisation du chapitre précédent.

4.2 Alloy4PV : Un framework pour la vérification de procédés

Cette section commence par présenter les raisons qui nous ont poussés à choisir Alloy comme langage formel, puis présente les différents modules Alloy nécessaires pour représenter et vérifier un modèle de procédé avec l'Alloy Analyzer.

4.2.1 Pourquoi Alloy ?

Ci-dessous, nous mettons en avant les points qui ont motivé notre choix pour Alloy :

- Il supporte une large variété de propriétés telles que les invariants, affirmations définies par l'utilisateur, LTL [47] et formules CTL [249] avec des contraintes d'équité.
- L'expressivité du langage est suffisante pour représenter un modèle UML associé à des contraintes OCL de manière presque directe, comme présenté par l'outil UML2Alloy [9]. Ce critère est particulièrement important pour la communauté de l'ingénierie dirigée par les modèles (MDD pour Model-Driven Development).
- La logique d'Alloy est très générique et n'oblige pas l'utilisateur à un style particulier de spécification pour modéliser et vérifier des systèmes réactifs.
- Il permet de spécifier le SAT solveur à utiliser pour résoudre le problème, permettant de mettre à profit les avancées toujours plus importantes en temps de vérification pour la résolution SAT.
- Il possède la capacité de produire l'*"unsatisfiable core"* (UnSAT) lorsqu'il échoue à trouver une instance pour un problème donné (c'est-à-dire l'ensemble des clauses pour lesquelles aucune instance satisfaisante n'existe). Ainsi, lorsque l'évaluation d'une propriété échoue, il est possible de donner des explications sur la *raison* de cet échec à partir des clauses compromettantes.
- Il possède un outil graphique ainsi qu'une API Java pour intégrer de façon transparente le processus d'analyse et de vérification dans un BPMS.
- Il jouit d'une grande et active communauté avec un développement continu de l'équipe du MIT.
- Etant basée sur la logique de premier ordre, l'implémentation de la formalisation telle que décrite dans le chapitre 3 peut être effectuée de manière (presque) directe.
- Beaucoup d'extensions sont proposées par la communauté, telles que : (1) DynAlloy [91], une version dynamique d'Alloy proposant des constructions opérationnelles pour modéliser l'exécution d'opérations et raisonner par rapport à des traces d'exécutions. (2) Moolloy [127], une extension implémentant un algorithme d'amélioration guidée (GIA, pour Guided Improvement Algorithm) pour résoudre des problèmes d'optimisation multi objectifs. (3) Kelloy [248], un outil pour vérifier des spécifications Alloy sur des domaines infinis en les traduisant dans le langage d'entrée du *theorem proving* KeY [6].
  Ces outils permettront, dans le futur, la possibilité d'améliorer le framework ALLOY4PV de façon presque "gratuite". Par exemple, Moolloy permettrait de résoudre des propriétés telles que "quelle est l'exécution la *plus rapide* pour exécuter l'activité `A`", quand Alloy est simplement capable de répondre à la question de satisfiabilité telle que "quel chemin exécute l'activité `A`", sans notion d'optimisation de la solution trouvée.

Ainsi, afin d'implémenter la formalisation présentée dans le chapitre 3, nous utilisons le langage Alloy. La popularité croissante d'Alloy en tant que méthode formelle pour la communauté MDD provient de sa notation orientée-objet basée sur une logique simple de relations, en incorporant les notions d'abstraction et d'héritage. Cette notation partage beaucoup de ressemblance avec UML enrichi de contraintes OCL, tout en proposant un outil puissant d'analyse automatique basé sur un SAT-solver, mettant à disposition la puissance de la vérification formelle aux ingénieurs logiciels.

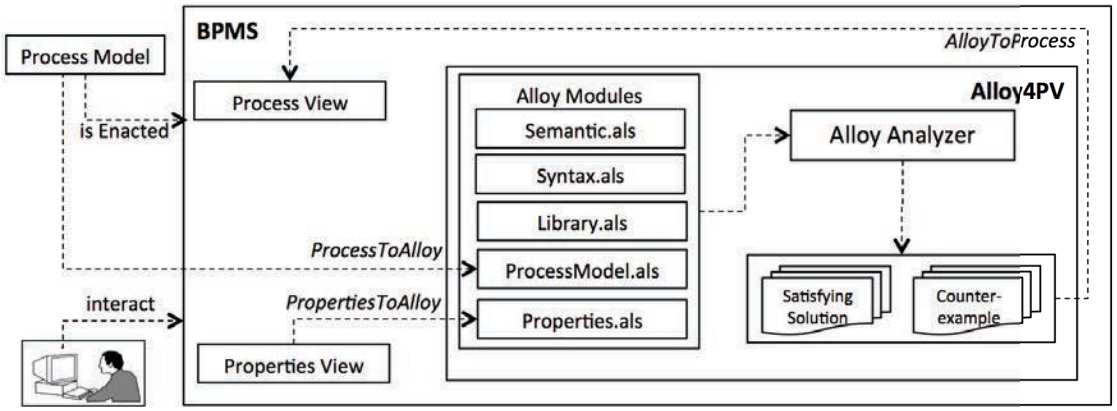


FIGURE 4.6 – Aperçus au niveau du framework ALLOY4PV

Dans la prochaine section, nous présentons la vision haut niveau de notre framework ALLOY4PV.

4.2.2 Aperçu haut niveau de Alloy4PV

ALLOY4PV (**A**lloy for **P**rocess **V**erification) est le nom donné à notre framework permettant la vérification de procédés métiers. Ce framework est basé sur notre formalisation de fUML et est implémenté en utilisant différents modules Alloy. La figure 4.6 montre un aperçu du processus pour effectuer la vérification avec ALLOY4PV dans un BPMS. Le BPMS prend en entrée un modèle de procédé (`Process Model`) dans le format UML AD. La vue des propriétés (`Properties View`) permet aux modélisateurs de sélectionner et d'exprimer des propriétés à travers une interface graphique. La vue du procédé (`Process View`) affiche les résultats de la vérification.

ALLOY4PV est composé de cinq modules : `Semantics.als`, `Syntax.als`, `Library.als`, `ProcessModels.als` et `Properties.als`. Dans la suite, nous détaillons le contenu de ces modèles statiques et dynamiques (contextuellement générés) requis par l'Alloy Analyzer afin de vérifier un modèle de procédé. Le but est de donner un aperçu global de la façon dont ALLOY4PV est implémenté en utilisant le langage Alloy, plutôt que de donner une définition exhaustive de ceux-ci. De plus, trois routines sont nécessaires :

ProcessToAlloy pour représenter le modèle de procédé dans le langage Alloy.

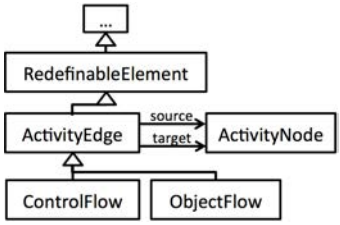
PropertiesToAlloy pour représenter les propiétés à analyser dans le langage Alloy.

AlloyToProcess pour analyser les résultats retournés par l'Alloy Analyzer et les afficher à l'utilisateur.

Ainsi, nous détaillons les différents modules Alloy et routines pour pouvoir vérifier de manière automatique un procédé. Un listing complet de ces modules est disponible dans l'annexe A.

4.2.3 Modules statiques

Syntax.als Ce module représente la syntaxe du PML. Il contient les signatures et relations Alloy pour représenter l'équivalent des classes et attributs du méta modèle, en l'occurrence, le méta modèle de UML AD. Le listing de la figure 4.7 montre un exemple se concentrant sur les `ActivityEdge`s du méta modèle. Grâce à la notation orientée-objet de Alloy, il est naturel et direct de représenter un méta modèle. Une méta classe correspond à une signature, et les attributs des méta classes à des relations. De toute évidence, une signature existe pour représenter chacun des éléments d'un AD.




```
1  abstract sig FumlObject extends Object
2  abstract sig Element extends FumlObject {···}
3  abstract sig NamedElement extends Element {···}
4  abstract sig RedefinableElement extends NamedElement {···}

5  abstract sig ActivityEdge extends RedefinableElement {
6    source : ActivityNode,
7    target : ActivityNode
8  }
9  abstract sig ControlFlow extends ActivityEdge {}
10 abstract sig ObjectFlow extends ActivityEdge {}
```

FIGURE 4.7 – Apperçus de la partie pour représenter les `ActivityEdge` du module `Syntax.als`

```
1  abstract sig Resource extends Object {}
2
3  abstract sig Info {
4     use_i : ExecutableNode →set Resource,
5     timing_i : ExecutableNode →Int,
6     capacity_i : Resource →Int
7  }
8  fun Use    : ExecutableNode →set Resource { Info·use_i }
9  fun Timing    : ExecutableNode →Int    { Info·timing_i }
10 fun Capacity  : Resource →Int     { Info·timing_i }
```

Listing 4.4 – Partie organisationnelle d'un procédé

Le listing 4.4 montre comment sont représentées les informations organisationnelles liées au procédé. La signature abstraite `Resource` permet de définir les ressources. La signature abstraite `Info` comprend 3 relations `use_i, timing_i, capacity_i` mappant respectivement des noeuds exécutables vers un ensemble de ressources (pour représenter les ressources utilisées par l'action), des noeuds exécutables vers un entier (pour représenter le nombre d'unités de temps nécessaire pour exécuter le noeud) et une ressource vers un entier (pour exprimer la capacité d'utilisation de la ressource). En plus de ces 3 relations, des fonctions (`Use, Timing, Capacity`) sont définies pour accéder à toutes les informations organisationnelles définies sur le modèle. Définir ces informations correspond donc à instancier la signature `Info`.

Ainsi, le module `Syntax.als` permet de représenter toutes les informations syntaxiques du modèle défini dans la section 3.2 de la formalisation. Plus précisément, ce module contient les signatures nécessaires pour représenter une instance d'un tuple `Process`.

Semantic.als Ce module correspond à la partie comportementale du PML. Il représente les notions d'*états*, *activation*, et *tir* de transition définies dans la formalisation. Comme expliqué dans la section 4.1.3, Alloy ne fournit pas de façon prédéfinie pour modéliser un comportement dynamique, du fait que les instances peuvent être seulement peuplées d'atomes immutables. Il est nécessaire d'utiliser l'idiome *"traces pattern"* [126] en introduisant une signature pour représenter l'état global du système, et de modéliser les opérations comme des prédicats définissant les relations entre les états avant et après. Nous utilisons la variante *"global state"* de ce patron.

Le listing 4.5 présente la signature `State` (ligne 6) pour représenter *un* état du système. Ligne 1, le module `util/ordering` est utilisé sur la signature `State` pour créer un ordre total et linéaire entre les différents états. La ligne 3 définit la signature `Status`, utilisée comme relation dans la

```
1 open util/ordering[State]
2
3 abstract sig Status {}
4 one sig Running, Finished extends Status {}
5
6 sig State {
7   v : ActivityNode →one Int, // tokens
8   e : ActivityEdge →one Int, // offers
9
10  lc : ExecutableNode →one Int, // local clock
11  gc : Int, // global clock
12
13  running : Status
14 }
```

Listing 4.5 – Représentation d'un `State`

signature `State` (ligne 13). La ligne 4 définit deux singletons de `Status` : `Running` et `Finished`. Quand chacune des relations (`v, e, lc, gc`) de `State` correspond aux informations représentant un état tel que défini dans la formalisation (cf. definition `State` page 62), la relation `running` est nécessaire seulement au niveau de l'implémentation. Cette relation permet de définir si la relation de transition a pu se dérouler complètement. Lorsqu'il y a toujours des prédicats d'activation évalués à vrai, `running` est affecté à `Running` ; quand aucun prédicat d'activation n'est évalué à vrai, `running` est affecté à `Finished`.

Comme montré sur le listing 4.6, afin de simplifier la notation des futurs prédicats, il est possible de définir différentes fonctions et prédicats. Par exemple, le prédicat `hasTokens` détermine si un noeud donné possède au moins un jeton. Il est important de noter que nous utilisons ici la *"receiver syntax"*. `State.hasTokens[node:ActivityNode]` est l'équivalent de `hasTokens[this:State,node:ActivityNode]`. Cette notation permet d'avoir une syntaxe *"orientée-objet"* et plus naturelle lors de la définition du système.

Le listing 4.7 présente le fait `traces` pour contraindre un à un tous les états (`State`) du système. "`s`" correspond à l'état courant, et "`s'`" à son état successeur (`s.next`). Ainsi, chacun des `State`s ayant sa relation `running` à `Running` est contraint par le prédicat `transition`. Ce prédicat correspond au même prédicat que celui défini respectivement dans la formalisation et est visible sur les lignes 16 à 22. De la même façon, il appelle les prédicats `stepTime` (ligne 25) et `stepTokens` (ligne 28) afin de faire évoluer les états du système. Dans le cas où la relation `running` est à faux, le prédicat `endLoop` (lignes 11 à 15) bégaye le dernier état en forçant l'égalité entre `s` et `s'`. En effet, lorsque la borne pour la signature des `State`s est supérieur, il arrive un moment où la relation de transition ne peut plus être déroulée étant donné qu'aucun prédicat d'activation ne peut être évalué à vrai. Dans ce cas-ci, comme préconisé par Jackson [126], la meilleure solution consiste à simplement bégayer le dernier état, pour *"remplir"* les `State`s supplémentaire.

Dans la suite, nous ne détaillons pas tous les prédicats d'activation et de tir, mais nous présentons seulement les prédicats `enabledFinish` et `fireFinish`. Comme visible sur le listing 4.8, les prédicats `enabledFinish` et `fireFinish` suivent exactement les définitions de la formalisation.

Finalement, le listing 4.9 présente le prédicat `init`, prenant en paramètre `n`, `e` et `global` correspondant respectivement à la relation entre les `ActivityNode`s et leurs nombres de jetons, les `ActivityEdge`s et leurs nombres d'offres, et l'horloge globale du système. Ce prédicat initialise le *premier* état du système. En effet, la relation `first` définie dans le module `util/ordering` renvoie le premier état. Ainsi, tous les noeuds et arcs sont initialisés à 0 à l'exception de ceux donnés en argument dans `n` et `e` (`++` correspond à l'opérateur de recouvrement de relation, *relational*

```
// getters
pred State·hasOffers[edge:ActivityEdge]
  { this·e[edge] > 0 }
fun State·getTokens[node:ActivityNode] : Int
  { this·v[node] }
pred State·hasTokens[node:ActivityNode]
  { this·v[node] > 0 }
fun State·getOffers [edge:ActivityEdge] : Int
  { this·e[edge] }
fun State·getLocalClock[node:ExecutableNode] : Int
  { this·lc[node] }

// setters
pred State·setTokens[node:ActivityNode, i : Int]
  { this·v[node] = i}
pred State·setOffers [edge:ActivityEdge, i : Int]
  { this·e[edge] = i}
pred State·setLocalClock[node:ExecutableNode, i : Int]
  { this·lc[node] = i }

// block Edge and Node
pred freezeAll[s,s':State]
  { freezeAllNode[s,s',ActivityNode] and freezeAllEdge[s,s',ActivityEdge]}
pred freezeAllEdge[s,s':State,edge: set ActivityEdge]
  { all e : edge | freezeEdge[s,s',e] }
pred freezeAllNode[s,s':State,node: set ActivityNode]
  { all n : node | freezeNode[s,s',n] }
pred freezeEdge[s,s':State, edge:ActivityEdge]
  {s·getOffers[edge] = s'·getOffers[edge] }
pred freezeNode[s,s':State, node:ActivityNode]
  { s·getTokens[node] = s'·getTokens[node] }
pred freezeTime[s,s':State]
  { s'·lc = s·lc and s'·gc = s·gc }
[···]

pred State·start[n:ExecutableNode]
  { not this·hasTokens[n] and this·next·hasTokens[n] }
pred State·end[n:ExecutableNode]
  { not this·hasTokens[n] and this·prev·hasTokens[n] }
```

Listing 4.6 – Quelques fonctions et prédicats pour simplifier l'écriture

```
fact traces {
  all s: State - last | let s' = s.next | {
    s.running = Running implies {
      transition[s,s]
    } else {
      // stuttering the last state
      endLoop[s,s']
    }
  }
}
pred endLoop[s,s':State] {
  freezeAll[s,s']
  freezeTime[s,s']
  s'.running = Finished
}
pred transition[s,s':State] {
  some f:ActivityFinalNode | s.hasTokens[f] implies {
    endLoop[s,s']
  } else {
    s'.running = Running
    (stepTime[s,s'] or stepTokens[s,s'])
  }

}
pred stepTime[s,s':State] {
  s.enabledTime and fireTime[s,s']
}
pred stepTokens[s,s':State] {
  one n:(ActivityNode-Pin) | {
    (s.enabledStart[n] and fireStart[s,s',n])
    or
    (s.enabledFinish[n] and fireFinish[s,s',n])
  }
}
```

Listing 4.7 – *"traces pattern"* pour dérouler la relation de transition

```
pred State·enabledFinish[node:ActivityNode] {
  this·hasTokens[node] // the node is already executing
  some node·outgoing // the node owns outgoing edge
  node in Action implies { // enough time spent on this action?
    this·getLocalClock[node]·gte[Timing[node]]
  }
}
pred fireFinish[s , s' : State, node:ActivityNode] {
  freezeTime[s,s',node] // freeze all the clock except "node"
  node in Action implies {
    s'·setTokens[node, 0]
    // add offers to both outgoing edge and outgoing edge of output
    // the number of added offers corresponds to the number of tokens on the node
    all edge : (node·outgoing+ node·output·outgoing) | {
      s'·setOffers[edge, s·getOffers[edge]·add[s·getTokens[edge·source]]]
    }
    // reset input and output pin
    all pin : (node·output+ node·input) | {
      s'·setTokens[pin, 0]
    }
    freezeAllEdge[s,s',(ActivityEdge-(node·outgoing+ node·output·outgoing))]
    freezeAllNode[s,s',(ActivityNode-(node+ node·input+ node·output))]
  }
  else node in DecisionNode implies {
    s'·setTokens[node, 0]
    // offer tokens on only one outgoing edge
    one edge : node·outgoing | {
      s'·setOffers[edge, s·getOffers[edge]·add[s·getTokens[edge·source]]]
      freezeAllEdge[s,s',(ActivityEdge-edge)]
    }
    freezeAllNode[s,s',(ActivityNode-node)]
  }
  else node in ActivityNode implies {
    s'·setTokens[node, 0]
    all edge : node·outgoing | {
      s'·setOffers[edge, s·getOffers[edge]·add[s·getTokens[edge·source]]]
    }
    freezeAllEdge[s,s',(ActivityEdge-node·outgoing)]
    freezeAllNode[s,s',(ActivityNode-node)]
  }
}
```

Listing 4.8 – Prédicats d'activation et de tir pour terminer l'exécution d'un noeud

override operator). En outre, toutes les horloges locales sont initialisées à 0 et l'horloge globale est initialisée à `global`.

Library.als contient l'ensemble des propriétés définies dans la section 3.3 de la formalisation, représenté dans le langage Alloy. Alloy ne propose pas de représentation directe de la (P)LTL. Cependant, la représentation en FOL des formules (P)LTL peut être effectuée de manière (presque) directe en suivant la traduction présentée dans [25], pour effectuer du *model-checking* borné de formule LTL. Dans [47], Cunha *et al.* expliquent en détail la traduction dans le langage Alloy

```
pred init[ nodes:ActivityNode→Int, edges:ActivityEdge→Int, global : Int] {
  first·v = (ActivityNode →0) +/+ nodes
  first·e = (ActivityEdge →0) +/+ edges

  first·gc = global
  first·lc = (ExecutableNode →0)

  first·running = Running
}
```

Listing 4.9 – Predicat d'initialisation du système

des formules LTL en se basant sur les travaux de [25]. Dans cette section, nous ne redonnons pas l'ensemble des propriétés de la section 3.3 exprimées avec Alloy, mais donnons quelques exemples. En effet, redonner l'ensemble des propriétés exprimées en FOL (Alloy) serait redondant par rapport à la formalisation des propriétés en PLTL présentée dans la section 3.3.

Il est important de comprendre que l'expression des propriétés repose grandement sur le module `util/ordering` d'Alloy (tel que visible sur la figure 4.4) pour "accéder" aux différents états de la trace d'exécution. Les fonctions les plus couramment utilisées correspondent à :

- `first` pour retourner le premier état de la trace,
- `last` pour retourner le dernier état de la trace,
- `s.next` pour retourner l'état suivant de s,
- `s.prev` pour retourner l'état précédent de s,
- `s.nexts` pour retourner l'ensemble des états successeurs de s, et
- `s.prevs` pour retourner l'ensemble des états prédécesseurs de s.

(1.1) Possibilité de complétion.

$$\mathsf{F}\,(\bigvee_{v \in Vlab(activityFinal)} (token(v)))$$

Cette propriété peut être traduite dans le langage Alloy tel que :

```
pred Completion {
  some s:State, f:ActivityFinalNode | s·hasTokens[f]
}
```

spécifiant qu'il existe au moins un état (`some s:State`) et au moins un noeud final (`some f:ActivityFinalNode`) tel que l'état `s` possède un jeton sur le noeud `f`.

(2.1) Données redondantes.

$$\mathsf{G}\,(\bigvee_{v \in Vlab(activityFinal)} (token(v)) \implies \bigwedge_{e \in E, lab(target(e)) \in Object} (\neg token(e)))$$

Cette propriété peut être traduite dans le langage Alloy comme suit :

```
pred RedondantData {
  all s:State | some f:ActivityFinalNode | s·hasTokens[f] implies {
    all o:ObjectFlow | not s·hasOffers[o]
  }
}
```

spécifiant que pour tous les états (`all s:State`) dont au moins un noeud final (`some f:ActivityFinalNode`) possède un jeton (`s.hasTokens[f]`), les arcs de flux de données (`all o:ObjectFlow`) n'ont pas d'offres (`not s.hasOffers[o]`).

(3.1) Ressource manquante.

$$\mathsf{G}\,\Big(\bigwedge_{\substack{v \in Vlab(action),\\ r \in Resource \wedge\\ r \in Use(v)}} (enabledStart(v) \implies (\,|\{o \in parallel \mid o \neq v \wedge r \in Use(o)\}| < Capacity(r))))$$

Cette propriété peut être traduite dans le langage Alloy tel que :

```
pred LackOfResource {
  all s:State, en:ExecutableNode, r:Resource | s.enabledStart[en] and (r in Use[en])
      implies {
    #{ o : ExecutableNode | o != en and r in Use[o] and s.hasTokens[o] } < Capacity[r
        ]
  }
}
```

spécifiant que pour tous les états (`all s:State`) et pour toutes les ressources (`r:Resource`) sur lesquels un noeud exécutable (`en:ExecutableNode`) est prêt à être exécuté (`s.enabledStart[en]`) utilisant une ressource donnée (`r in Use[en]`), le nombre (`#{`*expr*`}` représente la cardinalité de l'ensemble *expr*) d'autres noeuds (`o:ExecutableNode | o != en`) actuellement en cours d'exécution (`s.hasTokens[o]`) et utilisant la même ressource (`r in Use[o]`) est inférieur à la capacité totale d'utilisation de la ressource (`< Capacity[r]`).

(4.3) Contrainte relative de début/fin d'activité.

$$\forall i \in \mathbb{N}, \mathsf{G}\,(end(b) \wedge (i = gc) \implies \mathsf{F}\,(start(a) \wedge (gc < i + t)))$$

Nous avons choisi spécifiquement de présenter cette propriété étant donné qu'il est normalement nécessaire de comparer les propositions atomiques de deux états entre eux (ce qui n'est pas possible en LTL classique). Grâce au *"traces pattern"*, tous les états du système sont modélisés explicitement. Ainsi, il est possible de s'adresser directement à n'importe quel `State` de la trace d'exécution pour les comparer entre eux. Cette propriété peut être traduite dans le langage Alloy de la façon suivante :

```
pred TimeRelativeNode[a,b:ExecutableNode, t:Int]{
  all s1:State | s1.start[b] implies {
    some s2 : (s1+ s1.nexts) | s2.start[a] and s2.gc.lt[s1.gc.add[t]]
  }
}
```

spécifiant que pour tous les états (`all s1:State`) tel que le noeud exécutable "`b`" démarre (`s1.start[b]`), il existe des états dans le futur (`some s2 : (s1+s1.nexts)`) tel que "`a`" commence à s'exécuter (`s2.start[a]`) et que l'horloge globale à ce moment-là est inférieure à l'état global plus `t` lorsque `b` avait démarré (`s2.gc.lt[s1.gc.add[t]]`). Le prédicat `lt` correspondant à la comparaison "inférieure à" entre deux entiers. La fonction `add` correspondant à l'addition entre deux entiers. Ces deux prédicats/fonctions ont la syntaxe "receiver" (`Int.lt[Int]` et `Int.add[Int]`) et sont définis dans le module des entiers d'Alloy (`util/integer`).

Propriété métier : relation_between

$$relation\_between(a, b, c) \stackrel{def}{=} \mathsf{G}\ (token(a) \implies \mathsf{F}^{-1}\ (tokens(b)) \wedge \mathsf{F}\ (token(c)))$$

Nous avons choisi cette propriété pour montrer comment sont exprimées les propriétés PLTL utilisant un opérateur du passé. Cette propriété peut être traduite dans le langage Alloy de la façon suivante :

```
1 pred relation_between[a,b,c:ExecutableNode] {
2   all s:State | s.hasTokens[a] implies {
3     (some s1:(s+ s.prevs) | s1.hasTokens[b] )
4     and
5     (some s2:(s+ s.nexts) | s2.hasTokens[c] )
6   }
7 }
```

spécifiant que pour tous les états (`all s:State`) tel que le noeud exécutable "`a`" possède un jeton (`s.hasTokens[a]`), il existe des états dans le passé (`some s1:(s+s.prevs)`) et dans le futur (`some s2:(s+s.nexts)`) tel que les noeuds "`b`" et "`c`" possèdent respectivement un jeton (`s1.hasTokens[b]`/`s2.hasTokens[c]`). Comme visible ici, les opérateurs du passé de la PLTL peuvent être facilement représentés grâce aux fonctions `State.prev` et `State.prevs` du module `ordering` d'Alloy.

4.2.4 Modules dynamiques

Alors que la section précédente présentait les modules statiques, cette section présente les modules dynamiques, c'est-à-dire les modules qui sont générés selon le procédé et les propriétés à analyser.

ProcessModel.als représente l'instance du procédé à analyser. La figure 4.8 montre un simple procédé avec du flux de contrôle et de données, des ressources, et des contraintes de temps. La routine `ProcessToAlloy` permet de généré une spécification Alloy à partir du modèle de procédé, tel que visible sur le listing 4.10. Chacun des éléments du procédé est représenté comme un *singleton* (`one sig`) de son type équivalent, défini dans `Syntax.als`. Certains éléments, tels que les noeuds exécutables et les pins, spécifient leurs relations vers l'élément correspondant (ex. `input = InB2` sur la ligne 17). Les informations organisationnelles telles que les ressources utilisées par les actions (`use_i`), le temps pour l'effectuer (`timing_i`) et la capacité d'utilisation de chaque ressource (`capacity_i`) sont renseignées en instanciant la signature `Info`. Les lignes 28 et 29 définissent quelques `MultiplicityElement` pour spécifier la borne basse et haute des pins. Les multiplicités les plus communes ([0,1], [1,1], [0,*], [1,*]) sont déjà définies dans le module `Syntax.als`. Cependant, comme cela est visible sur ses lignes, il est possible de définir ses propres multiplicités (ex. [1,2]). Il est important de noter que la fonction `max` (ligne 29) représente l'entier maximal exprimable sur le système. Cette fonction est définie dans le module standard `util/integer`. L'analyse des modèles Alloy reposant sur un SAT solveur, une multiplicité "infinie" (ex. `*`) n'est pas directement représentable. Comme visible sur ce listing, la représentation du procédé dans le langage Alloy est naturelle et directe. Chaque élément et concept correspond à une nouvelle instanciation d'une signature définie dans le module `Syntax.als`.

Properties.als contient les commandes pour lancer l'Alloy Analyzer par rapport à un ensemble de propriétés à vérifier. Le listing 4.11 montre le fait (`fact`) appelant le prédicat `init`, permettant d'initialiser le premier état du système. Sur cet exemple, tous les noeuds initiaux (`InitialNode`) sont initialisés avec un jeton, tous les arcs sont initialisés à zéro, et l'horloge globale est initialisée

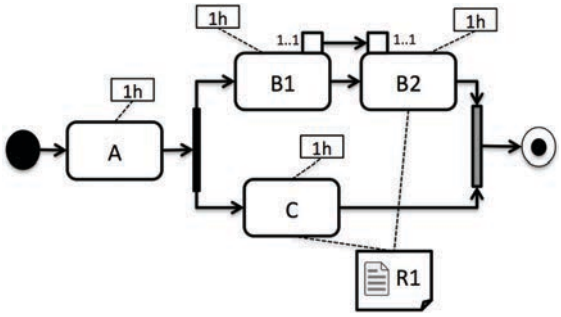


FIGURE 4.8 – Exemple de simple procédé avec du flux de données, des ressources, et des contraintes de temps

```
1  open semantic
2  open syntax
3
4  // Organisational information
5  one sig R1 extends Resource {}{}
6  one sig InfoProcess extends Info {} {
7    use_i    = (B2 →BankConnector) + (C →R1)
8    timing_i  = (A →1) + (B1 →1) + (B2 →1) + (B3 →1)
9    capacity_i = (R1 →1)
10 }
11
12 // UML AD
13 one sig Initial extends InitialNode {} {}
14 one sig A extends OpaqueAction {}{input = none and output = none}
15 one sig Fork extends ForkNode {} {}
16 one sig B1 extends OpaqueAction {}{input = none and output = OutB1}
17 one sig B2 extends OpaqueAction {}{input = InB2 and output = none}
18 one sig C extends OpaqueAction {}{input = none and output = none}
19 one sig Join extends JoinNode {} {}
20 one sig Final extends ActivityFinalNode {} {}
21 one sig OutB1 extends OutputPin {}{multiplicityElement = MultiOneOne}
22 one sig InB2 extends InputPin {}{multiplicityElement = MultiOneOne}
23 one sig f1 extends ObjectFlow {} {source = Initial and target = A}
24 [···]
25 one sig f8 extends ControlFlow {} {source = Join and target = Final}
26
27 // (common MultiplicityElement defined in Syntax.als)
28 one sig MultiOneOne extends MultiplicityElement {} {lower = 1 and upper = 1}
29 one sig MultiZeroStar extends MultiplicityElement {} {lower = 0 and upper = max[]}
```

Listing 4.10 – Représentation du procédé de la figure 4.8 en utilisant Alloy

à zéro. Ce prédicat est particulièrement utile pour pouvoir initialiser le système avec différents états initiaux. Par exemple, si le modèle de procédé est modifié à la volée, où si une propriété temporelle est revérifiée pendant l'exécution, la possibilité d'initialiser l'état initial avec l'état courant d'exécution est primordiale.

```
1 fact initialize {
2   init[
3     InitialNode →1,
4     ActivityEdge →0,
5     0
6   ]
7 }
```

Listing 4.11 – Fait d'initialisation du système

```
1 run {Completion} for 0 but 15 State
2 check {Completion} for 0 but 15 State
3 check {relation_between[B1,A,B2]} for 0 but 15 State
4
5 // same as first line
6 run {Completion} for 0 but 15 State, 10 ActivityNode, 9 ActivityEdge, 1 Resource
7 run {Completion} for 0 but 15 State, 1 ActivityFinalNode, 1 InitialNode, 4
      ExecutableNode, 8 ControlFlow, 1 ObjectFlow, ···, 1 Resource
```

Listing 4.12 – Commandes pour vérifier le système

Le listing 4.12 montre les commandes pour lancer l'analyse des propriétés. Comme expliqué dans la section 4.1, il y a deux façons de lancer une analyse avec l'Alloy Analyzer. La commande `run` (ligne 1) permet de chercher un modèle qui satisfait la formule, alors que la commande `check` (lignes 2 et 3) permet de chercher un contre-exemple par rapport a la formule. Dans la section 2.2.3, nous avons discuté de la différence entre les propriétés *faible* et *forte* : une propriété *faible* correspond à s'assurer qu'au moins *une* exécution satisfait la propriété quand une propriété *forte* s'assure que la propriété est vraie, et ce quelle que soit l'exécution. Alloy permet donc de répondre très facilement à ce besoin en utilisant respectivement la commande `run` et `check` pour les propriétés *faible* et *forte*. La ligne 1 permet de chercher pour une exécution, tel que le procédé termine (un noeud `ActivityFinalNode` possède un jeton) et de répondre à la propriété *faible* `Completion`. La ligne 2 permet de s'assurer que, quelle que soit l'exécution, le procédé termine toujours et répond à la propriété *forte* `Completion`. La ligne 3 permet de s'assurer que l'action `B1` est toujours exécutée entre `A` et `B2`. Sur cette propriété, le prédicat prend en paramètres les actions instanciées dans le module `ProcessModel.als`.

Sur chacune des commandes, il est nécessaire de spécifier une borne sur les signatures pour fermer le domaine. Toutes les bornes sont contraintes par le modèle de procédé pris en entrée afin d'être analysé. Par exemple, si le procédé possède 10 noeuds, la borne de la signature `ActivityNode` sera de 10. De la même façon, si le procédé possède 2 types de ressources différentes, la borne des `Resource` sera de 2. La partie "`for 0`" implique que, par défaut, toutes les signatures ont une borne de 0, sauf (`but`) les signatures précisées après. Comme montré sur le listing (lignes 1, 2 et 3), seule la signature du nombre de `State` est précisée (cf. section suivante). Alloy possède de nombreuses règles pour définir ces bornes de façon implicite lors de la définition des signatures ou explicitement au niveau des commandes [126]. Du fait que le module `ProcessModel.als` possède de nombreuses signatures de type singleton (`one sig`), l'Alloy Analyzer détermine ces bornes implicitement à partir de la spécification soumise, et outrepasse la borne par défaut de 0. Ainsi, les lignes 1, 6 et 7 sont équivalentes sur la manière dont l'Alloy Analyzer sera configuré pour lancer l'analyse.

Ainsi, la génération des propriétés par la routine `PropertiesToAlloy` est très simple à réaliser et correspond seulement à générer les bons prédicats à analyser (avec ses paramètres si requis par la propriété). La seule difficulté consiste à connaître le nombre de `State` sur laquelle analyser le procédé. En effet, l'Alloy Analyzer n'est pas capable de déterminer, de lui-même, le nombre total de `State`s nécessaire pour analyser *"complètement"* le modèle. C'est à l'utilisateur de choisir cette borne. Il y a deux cas à distinguer :

1. La borne des `State`s est supérieure au nombre de `State`s normalement requis pour dérouler complètement la relation de transition.
2. La borne des `State`s n'est pas suffisante pour dérouler complètement la relation de transition.

Nous avons vu que le premier cas est géré en bégayant le dernier état afin de "remplir" la trace. Dans le second cas, nous expliquons dans la prochaine section comment calculer la borne de la signature `State` (c'est-à-dire la longueur de la trace) afin de toujours pouvoir analyser de manière *"complète"* le modèle de procédé.

4.2.5 Longueur de trace

En reprenant le cas d'exécution de la figure 3.4 présenté dans la formalisation, il est montré qu'une borne de 10 `State`s n'est pas suffisante pour pouvoir dérouler complètement la relation de transition. En revanche, une borne de 100 `State`s est de toute évidence trop "grande" et non nécessaire pour pouvoir analyser complètement le procédé.

Du fait que l'Alloy Analyzer repose sur un SAT solveur, il effectue du *model-checking borné* (BMC, pour Bounded Model Checking). Ainsi, il est seulement capable de garantir l'absence de contre-exemple jusqu'à une certaine borne k. Cependant, dans le cas de la vérification de procédés, le système vérifié n'est pas un système dont les executions sont infinies (tel que l'exécution d'un système d'exploitation). Un procédé a un début et une fin, généralement marqués respectivement par un noeud initial et un noeud final (`InitialNode` et `ActivityFinalNode`).

Du fait que nous savons qu'un procédé est amené à se terminer un jour, la relation de transition définie est, en conséquence, elle aussi amenée à se terminer. Tel que défini dans le listing 4.7, le déroulement de la relation de transition est terminé lorsque la relation `running` de l'état courant est affectée à `Finished`. Une fois affecté à `Finished`, l'état est bégayé grâce au prédicat `endLoop`. Ainsi, une façon de vérifier que le système possède toujours un nombre d'états suffisant pour dérouler complètement la relation de transition consiste à vérifier que pour chaque execution, finalement, il existe au moins un état tel que sa relation `running` est affectée à `Finished`. Dans le cas contraire, cela implique qu'il existe au moins une exécution telle que la relation de transition n'a pu se terminer, démontrant un besoin d'analyse sur un plus grand nombre d'états.

Le listing 4.13 présente le code Alloy pour effectuer cette vérification, et, ainsi, calculer la longueur de trace (c'est-à-dire le nombre de `State`) nécessaire pour effectuer une analyse complète du procédé. Ainsi, le prédicat `transitionOver` (ligne 1) spécifie qu'il existe des états (`some s:State`) tels que la relation de transition est déroulée complètement (`s.running = Finished`). Les lignes 6, 7 et 8 indiquent à l'Alloy Analyzer de vérifier que, quelle que soit l'exécution, le prédicat `transitionOver` est toujours vrai. Ainsi, un contre-exemple est disponible seulement quand le déroulement de la relation de transition n'a pu être terminée. Sur ces lignes, à partir de 40 `State`s, aucun contre-exemple n'est trouvé. Il n'est pas possible de trouver une exécution telle qu'en 40 `State`s, la relation ne soit pas terminée.

Le calcul de la longueur de la trace consiste donc à vérifier le prédicat `transitionOver` en incrémentant le nombre de `State`s jusqu'à ce qu'aucun contre-exemple ne soit trouvé, c'est-à-dire en utilisant des techniques d'*incremental-scoping* [26]. Ainsi, bien que nous effectuons avec Alloy du *model-checking borné*, nous analysons le modèle sur une longueur de trace suffisante pour prendre en compte toutes les exécutions possibles.

```
1 pred transitionOver {
2   some s:State | s.running = Finished
3 }
4
5 [...]
6 check {transitionOver} for 0 but 20 State // counter-example
7 check {transitionOver} for 0 but 30 State // counter-example
8 check {transitionOver} for 0 but 40 State // NO counter-example
```

Listing 4.13 – Calcul de la longueur de trace

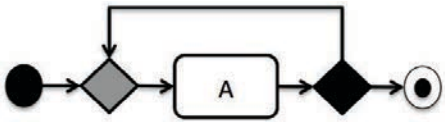


FIGURE 4.9 – Exemple de procédé potentiellement infini

Dans la pratique, le nombre d'analyses du prédicat `transitionOver` n'est jamais très important. En utilisant certaines heuristiques, il est possible de déterminer le nombre d'états approximatifs du modèle. Un diagramme d'activité étant un graphe orienté, il est possible de le transformer en graphe orienté pondéré (weighted graph). La pondération sur les arcs dépend des temps associés aux actions. À partir de ce graphe, un simple calcul du plus long chemin vers un des noeuds finaux permet d'avoir une approximation *minimal* du nombre d'états. À partir de cette approximation, et en associant la technique d'*incremental-scoping* sur le nombre de `State`, il est possible de déterminer la longueur de la trace pour tester toutes les exécutions possibles.

Il est important de noter que notre façon de calculer la longueur de trace est possible seulement car nous vérifions un système fini, c'est-à-dire amené à se terminer.

La prochaine section discute des problèmes liés aux boucles dans les procédés.

4.2.6 Gestion des boucles dans le procédé

La figure 4.9 montre un exemple de procédé dont l'exécution est potentiellement infinie. En effet, le noeud `DecisionNode` peut potentiellement toujours choisir la transition vers le noeud `MergeNode`. Ainsi, en vérifiant, par exemple, la propriété `Completion` sur ce procédé, un contre-exemple est toujours généré montrant que la transition vers le noeud `MergeNode` est toujours choisie par le noeud `DecisionNode`.

Certains *model-checker* permettent de spécifier l'équité (*fairness*) au niveau du système. Une condition d'équité permet de préciser que quelque chose ne se produit pas infiniment souvent. Ainsi, une condition d'équité précisant que la transition vers le noeud `MergeNode` ne peut être choisie indéfiniment résoudrait le problème des boucles dans le procédé, tel qu'utilisé par Eshuis dans [79].

Cependant, Alloy ne propose pas de moyen de spécifier les conditions d'équité. Il est nécessaire de restreindre le système pour ne pas prendre en compte les exécutions de sorte que cette transition est choisie indéfiniment. Une façon de restreindre le système de cette manière est de rajouter un fait (`fact`) s'assurant que seules les exécutions telles que la relation de transition a pu être déroulée complètement (`transitionOver`) seront analysées :

```
1 fact fairness {
2   transitionOver
```

```
3 }
```

Ainsi, le contre-exemple présenté précédemment ne peut plus être produit par l'Alloy Analyzer du fait que, sur ce contre-exemple, la relation de transition n'est pas terminée.

En revanche, le problème de cette solution est qu'elle ne peut pas être utilisée pour calculer la *longueur de la trace* telle que présentée dans la section précédente. En effet, parce qu'un fait (`fact`) implique une restriction directement au niveau du système lui-même, vérifier le prédicat `transitionOver` deviendrait trivialement vrai. Ainsi, lors du calcul de la longueur de trace sur laquelle analyser le procédé, le système doit être restreint de façon à ce que, par exemple, la même transition de sortie de noeud `DecisionNode` ne puisse pas être choisie plus de x fois (x étant fixé par l'outil ou personnalisables selon les besoins de l'utilisateur) :

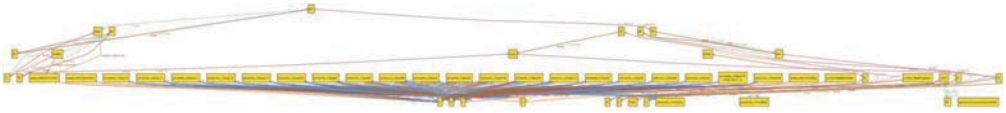
```
1  pred fairness[n:Int] {
2    all decision:DecisionNode | fairDecision[decision,n]
3  }
4  pred fairDecision[node:DecisionNode,n:Int] {
5    all edge:node·outgoing |
6      #{getNumberOfEdgeActivation[edge]} < = n
7  }
8  fun getNumberOfEdgeActivation[edge:ActivityEdge] : State {
9    { s : State | not s·hasOffers[edge] and s·next·hasOffers[edge] }
10 }
11
12 fact fairnessTransitionOver {
13   fairness[2]
14 }
15 fact fairnessTransitionOverUserDefined {
16   fairDecision[DecisionA, 1]
17   fairDecision[DecisionB, 5]
18 }
```

Le prédicat `fairness` impose que tous les noeuds de décision (`all decision:DecisionNode`) ne puissent choisir la même transition indéfiniment (`fairDecision`). En effet, ce prédicat spécifie que le nombre de fois (`getNumberOfEdgeActivation`) que ses arcs de sorties ont un jeton offert soit inférieur à n. Finalement, le fait `fairnessTransitionOver` (lignes 12 à 14) restreint le système de manière à ce que tous les noeuds `DecisionNode` ne puissent choisir le même arc de sortie plus de 2 fois. Il est aussi possible de laisser l'utilisateur définir manuellement, sur chacun des différents noeuds, le nombre maximal d'activations de ses arcs de sorties, tel que visible sur le prédicat `fairnessTransitionOverUserDefined`.

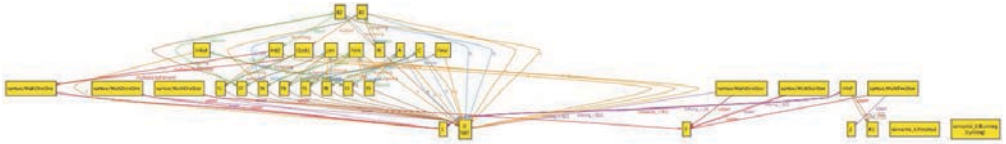
La prochaine section présente la façon d'analyser le résultat renvoyé par l'Alloy Analyzer.

4.2.7 Analyse du résultat renvoyé par l'Alloy Analyzer

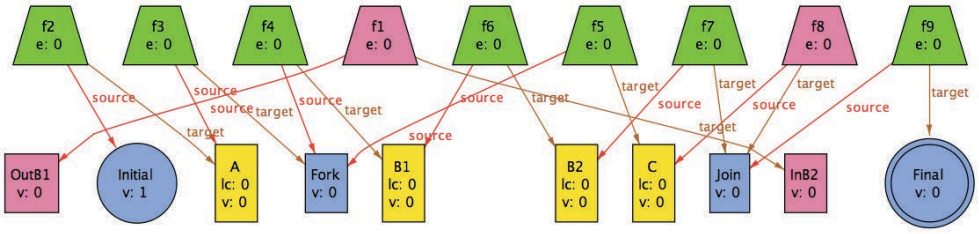
Quand l'ensemble des solutions ou des contre-exemples sont calculés par l'Alloy Analyzer, les résultats sont renvoyés vers la `Process View` en utilisant la routine `AlloyToProcess`. Cette routine analyse le résultat retourné par l'Alloy Analyzer (ex. en extrayant le chemin menant à l'interblocage) et le montre sur la `Process View`. La figure 4.10 montre un modèle trouvé par l'Alloy Analyzer. La sous-figure (a) présente l'ensemble des signatures et des relations trouvées pour satisfaire la propriété *faible* `Completion` (c'est-à-dire un exemple d'exécution tel que le procédé termine) par rapport au procédé de la figure 4.8. Toutes les `States` sont mises "à plat" sur la visualisation. L'Alloy Analyzer permet de faire des projections par-dessus une signature donnée, afin de pouvoir parcourir et visualiser le résultat de manière plus simple. Une projection sur la première `State` est visible sur la sous-figure (b). L'Alloy Analyzer permet aussi



(a) Résultat global



(b) Projection sur la première `State`



(c) Projection sur la première `State` avec personnalisation du thème pour les signatures et relations

FIGURE 4.10 – Résultat retourné par l'Alloy Analyzer sur la propriété *faible* `Completion`, dans son ensemble (a), puis projeté sur le premier état (b) et avec un thème pour une meilleure visualisation (c)

de personnaliser l'affichage des différentes signatures et relations trouvées. La sous-figure (c) montre la même projection que la sous-figure (b) mais avec une personnalisation de l'affichage. Toutes les signatures autres que descendantes de `ActivityNode` ou `ActivityEdge` ont été cachées. Les signatures `ActivityEdge` et `ActivityNode` sont respectivement mises sous forme de trapèzes et rectangles. Les trapèzes de couleur rouge et verte représentent respectivement les `ObjectFlow` et `ControlFlow`. Les rectangles de couleur rouge, jaune et bleu représentent les `ObjectNode`, `ExecutableNode` et `ControlNode`. Une personnalisation supplémentaire a été apportée aux noeuds `InitialNode` et `ActivityFinalNode` en leur donnant une forme de cercle simple et double. De plus, les relations liées aux états ont été affichées directement sur leurs signatures associées, c'est-à-dire que chacune des `ActivityNode` possède un attribut v représentant son nombre de jetons. Ainsi, en parcourant dans l'ordre les `State`s, il est possible de "rejouer" l'exécution directement sous l'Alloy Analyzer.

L'API Alloy permet de récupérer un graphe d'objets représentant le graphe des signatures et relations visibles sur la figure 4.10 (a). Récupérer la solution ou le contre-exemple représentant l'exécution satisfaisant ou invalidant la propriété (`run` ou `check`) correspond donc à parcourir, dans l'ordre, l'ensemble des signatures `State` renvoyé par l'API en regardant ou se situe les jetons et les offres tant que la relation `running` est affectée à `Running`. Une fois ce chemin récupéré, il est possible de l'afficher à l'utilisateur dans la `Process View`, soit sous forme textuelle (ex. $\{Initial, A, B, \ldots, Final\}$) ou mieux encore en l'affichant graphiquement sur le procédé.

L'annexe A présente l'ensemble du code Alloy présenté dans cette section, dont l'ensemble des propriétés présenté dans la formalisation. La prochaine section présente notre outil, implémenté au-dessus de la spécification Alloy afin d'automatiser tout le processus de vérification.

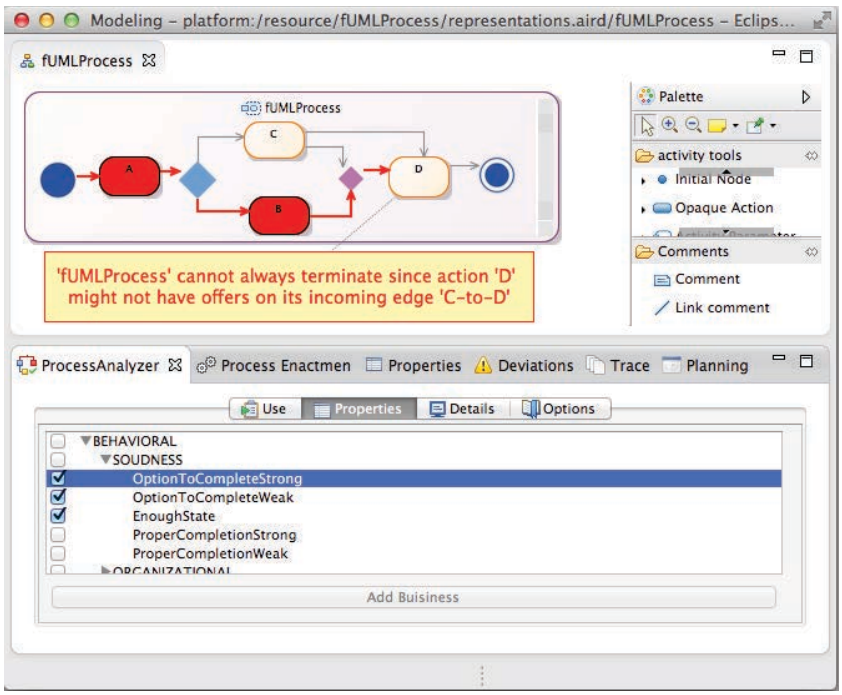


FIGURE 4.11 – Outil graphique associé à ALLOY4PV

4.3 Outil graphique associé à Alloy4PV

Au-dessus des spécifications Alloy de ALLOY4PV, nous avons développé un outil, fourni comme un plug-in Eclipse Modeling Framework (EMF). Il vient avec une librairie de propriétés prédéfinies prêtes à être utilisées directement sur le modèle et permet d'ajouter des propriétés métiers à travers une interface graphique. L'utilisateur a seulement à choisir dans l'interface les propriétés qui l'intéressent, et remplir, si nécessaire, les paramètres (ex. le temps maximal pour terminer le procédé). La figure 4.11 montre une impression d'écran de notre outil pour la modélisation, l'exécution [155] et la vérification [153, 152] de procédé, en mettant en avant la partie visuelle du procédé ainsi que son analyseur. Le prototype repose sur Obeo UML Designer pour modéliser et afficher graphiquement le procédé. Afin de tirer parti des processeurs multicoeurs, il incorpore un pool de threads afin de vérifier directement plusieurs propriétés en parallèle.

Modèle pris en entrée. Afin de définir et modéliser un procédé pris en entrées par notre prototype, nous avons créer un méta modèle basé sur le méta modèle UML 2 afin de prendre en compte certaines informations organisationnelles non représentables directement avec un simple diagramme d'activité. Ce méta modèle est visible sur la figure 4.12. Nous avons volontairement gardé le méta modèle simple en prenant seulement en compte les informations organisationnelles telles que présentées dans le chapitre 3. Cette option a été choisie parce que les autres informations ne sont pas nécessaires pour vérifier le *comportement* du procédé (ex. vérifier qu'une ressource donnée est bien affectée à un type d'action est un problème syntaxique). La méta classe `ProcessActivity` étend `Activity` en ajoutant un attribut *totalTime* représentant le temps total alloué pour effectuer l'activité. La méta classe `Resource` permet de définir les ressources allouées au procédé, tout en prenant en compte leurs capacités d'utilisation (*capacity*). La méta class `ProcessAction` étend `ExecutableNode` en ajoutant un attribut permettant de spécifier le temps pour effectuer

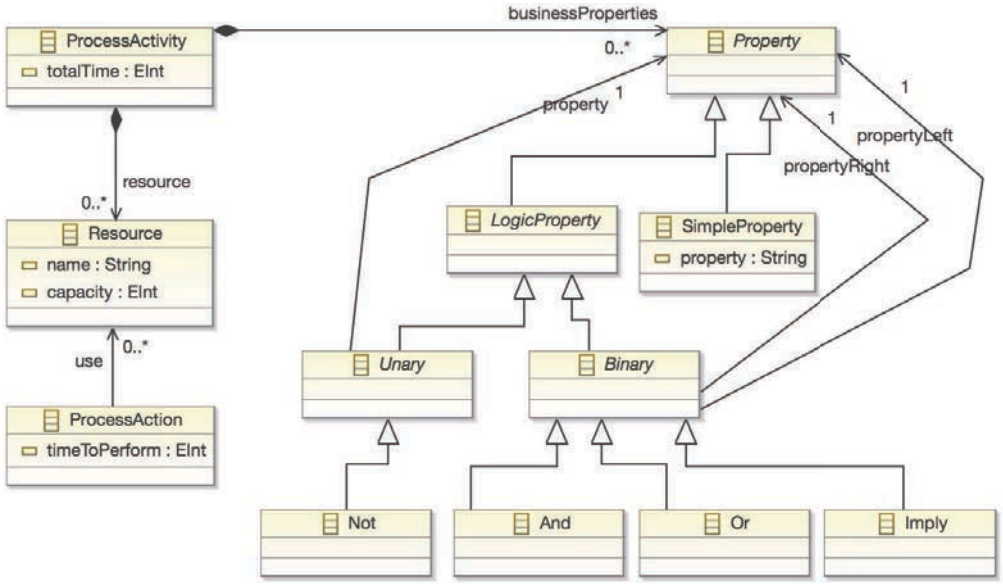


FIGURE 4.12 – Méta modèle étendant le méta modèle de UML AD pour la définition des informations organisationnelles

l'action (*timeToPerform*) tout en associant les `Resources` utilisées par celle-ci (*use*). La méta class `ProcessActivity` contient aussi un ensemble de propriétés métiers (`Property`). Une méta classe `Property` étend `CommentNode` et peut être de deux types différents : `SimpleProperty` ou `LogicProperty`. Une `SimpleProperty` correspond à une propriété basique telle que définie dans la librairie de propriétés métiers présentée dans la section 3.3.5. Son attribut *property* sert à contenir directement la propriété (ex. *"existence(A)"*). La méta classe `LogicProperty` permet de définir des propriétés logiques composées de `SimpleProperty`. Une `LogicProperty` peut être de type `Unary` ou `Binary` selon l'opérateur de la logique représenté :

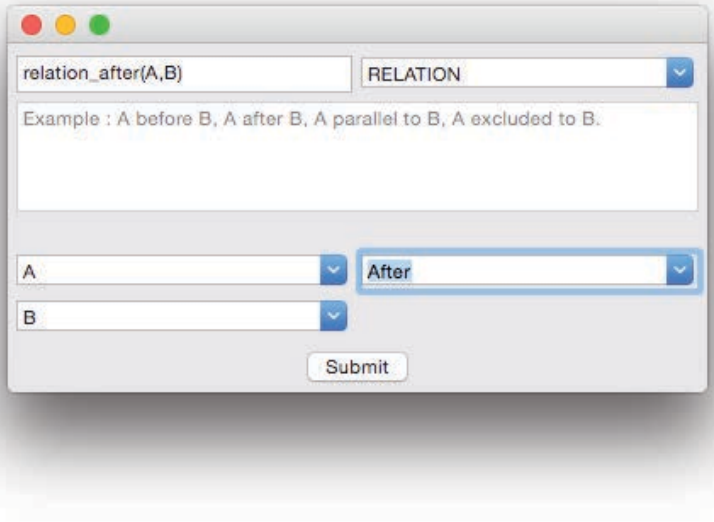
Not permet de spécifier la négation d'une `SimpleProperty`.

And, Or, Imply permettent de spécifier les opérateurs AND, OR, et d'implication de la logique entre deux `SimpleProperty`.

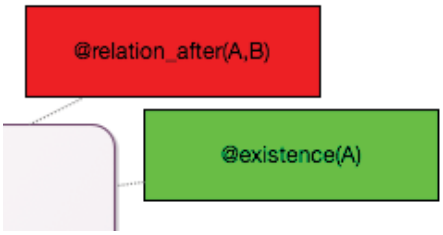
Les méta classes `Property`, `LogicProperty`, `Unary` et `Binary` sont abstraites. Ainsi, `SimpleProperty` représente les propriétés métiers de notre librairie, et les méta classes `Not, And, Or, Imply` permettent de composer des `Property`.

Il est important de noter qu'il aurait été possible de définir une méta classe par propriété métier (ex. `Existence` contenant un attribut de type `ProcessAction`, etc). Cependant, nous avons choisi de garder le méta modèle petit et simple, afin de ne pas avoir à modifier le méta modèle à chaque modification de la librairie de propriétés métiers. La validité de l'attribut *property* de `SimpleProperty` est assurée par l'outil.

Expression de propriétés métiers. Les propriétés métiers peuvent être ajoutées à travers des templates prédéfinis, par exemple en sélectionnant que l'`ActionA` doit toujours être exécutée avant l'`ActionB`. La figure 4.13 (a) montre un exemple d'expression de propriétés métiers à travers l'interface graphique : l'utilisateur sélectionne le type de propriétés (`RELATION` sur cet exemple), puis les différents paramètres nécessaires sont ensuite sélectionnables par l'utilisateur. Dans ce cas-ci, seuls les noeuds `ExecutableNode` sont sélectionnables dans les listes déroulantes. Il est



(a)



(b)

FIGURE 4.13 – Expression de propriétés métiers sur le procédé (a) avec affichage des propriétés sur le procédé sous forme d'annotations (b)

aussi possible de sélectionner le type de propriétés logiques, et de sélectionner si l'on souhaite faire une relation `AND, OR, Not` ou `Imply` entre deux propriétés définies précédemment. Une fois la propriété validée, un élément de type `Property` est ajouté sur le modèle. Comme visible sur la figure 4.13 (b), les propriétés métiers sont visibles sous forme d'annotations, où plus précisément de noeud UML `CommentNode`.

Résultat de vérification. Quand la vérification est effectuée, le chemin menant vers le contre-exemple (s'il y en a un) est mis en évidence en vert pour les propriétés *faible* et en rouge pour les propriétés *forte*. De plus, des noeuds UML `CommentNode`s sont directement insérés dans le modèle afin d'afficher les erreurs qui doivent être corrigées sur le modèle. Finalement, les annotations sont coloriées en vert quand la propriété est valide, et en rouge quand la propriété est invalide.

Il est important de noter que le prototype ne demande *aucun* bagage formel pour être utilisé par le modélisateur. Tout est *automatisé* à travers l'utilisation de l'interface graphique, et ce, afin de faciliter l'adoption de l'outil.

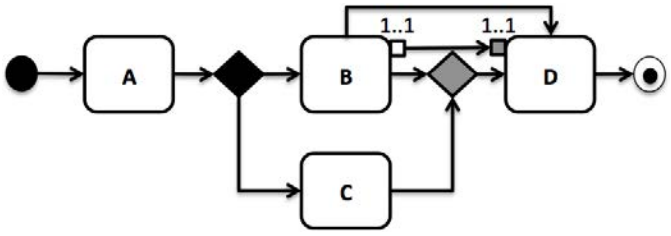


FIGURE 4.14 – Procédé souffrant à la fois d'*impossibilité de complétion* et de *données manquantes*

4.3.1 Ordre de vérification des propriétés

Afin de faire un diagnostic précis des erreurs d'un procédé, il est nécessaire de suivre un ordre pour vérifier les propriétés du procédé. En effet, certaines propriétés non valides entraînent automatiquement l'invalidité d'autres propriétés dépendant des caractéristiques assurées par cette dernière. Par exemple, un problème *syntaxique* tel que l'exemple (f) de la figure 2.4 (page 20) implique directement des problèmes comportementaux tels que des inters blocages et/ou des transitions mortes. Ainsi, il est nécessaire de corriger les erreurs syntaxiques avant la vérification des propriétés comportementales.

Le *flux de contrôle* du procédé définit les chemins d'exécution possibles du procédé. Les données, l'allocation des ressources et la durée des activités ne font que *restreindre* ces chemins d'exécution, c'est-à-dire qu'ils ne peuvent pas ajouter de nouveaux chemins d'exécution qui ont été exclus par la perspective du flux de contrôle [271]. Prenons en exemple le procédé de la figure 4.14. Ce procédé souffre à la fois du problème de la figure 2.6 (impossibilité de complétion, page 22) et de la figure 2.9 (données manquantes, page 23). En analysant ces 2 propriétés, on obtient le même contre-exemple : $\{A, C\}$. Il est, dès lors, plus compliqué de comprendre la *raison* de l'insatisfiabilité des propriétés. Ainsi, de la même manière qu'il est nécessaire de vérifier les propriétés syntaxiques avant les propriétés comportementales, un ordre existe parmi ces dernières. En effet, un problème au niveau du flux de contrôle a pour conséquence d'entraîner automatiquement une insatisfiabilité d'autres propriétés. Par exemple, une propriété de temps sur la *durée du procédé* pourrait être insatisfiable, car si le procédé est sujet aux interblocages, il ne peut tout simplement pas se terminer. En reprenant l'exemple de la figure 4.14, si ce procédé souffrait seulement du problème de *données manquantes* (en retirant le `ControlFlow` de `B` vers `D`), la vérification de la propriété d'*impossibilité de complétion* retournerait quand même un contre-exemple, bien que le problème se situe sur le flux de données. Ainsi, il est non seulement nécessaire d'effectuer l'analyse des propriétés selon un ordre défini, mais aussi nécessaire de faire une abstraction de certaines perspectives lors de l'analyse.

Ainsi, il est nécessaire d'analyser en premier lieu le *flux de contrôle* en isolation, puis les propriétés de *flux de données* combinées avec la perspective de *flux de contrôle*. Quand ces deux analyses sont terminées, il est possible d'analyser les propriétés liées aux *ressources* et au *temps* de manière unifiée avec toutes les perspectives. Prenons en exemple le procédé de la figure 2.11. Ce procédé est annoté avec l'utilisation des ressources ainsi que des contraintes temporelles. "Est-il possible de terminer le procédé en 4h ?". Dans le cas où l'on fait abstraction de l'utilisation des ressources, cette propriété est vraie. Toutes les actions sont exécutées (`A, B, C, D`), mais `C` est exécuté *en même temps* que `B`. Cependant, dans le cas où l'on prend en compte l'utilisation des ressources, il est impossible d'exécuter `B` et `C` en même temps du fait que ces actions utilisent la même ressource. Bien que le flux de contrôle définisse ces actions après un `ForkNode`, les autorisant à s'exécuter en parallèle, la ressource `R1` restreint ce comportement. Ainsi, la vérification des

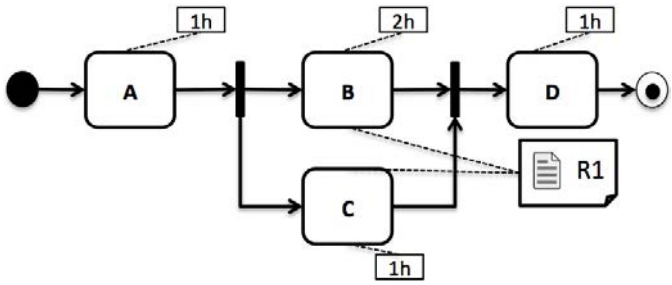


FIGURE 4.15 – Procédé annoté avec l'utilisation des ressources et contraintes temporelles

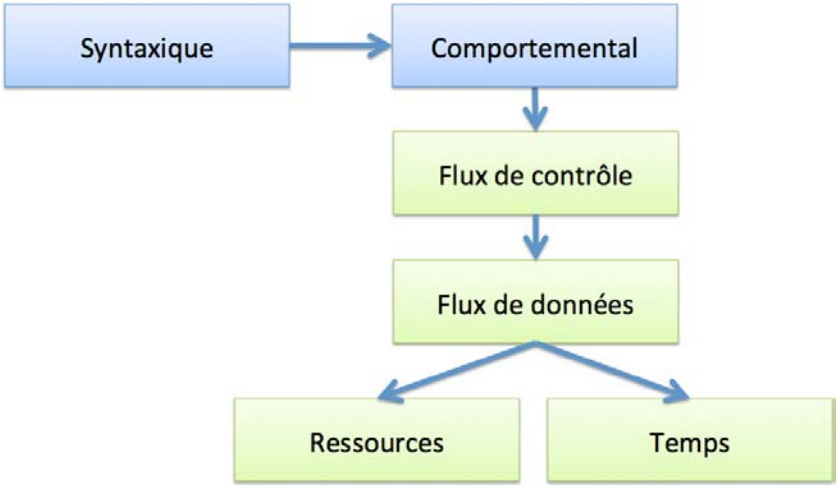


FIGURE 4.16 – Ordre d'analyse des propriétés mis en oeuvre par ALLOY4PV

propriétés liées aux *ressources* et aux *temps* nécessite de prendre en compte toutes les perspectives du procédé de façon combinée. La figure 4.16 résume l'ordre d'analyse des propriétés sur un procédé.

Ainsi, nous avons en fait implémenté 2 versions du module `Semantics.als` :

Flux de contrôle et flux de données. Toutes les conditions dans les différents prédicats (pré conditions et activation) liées aux ressources et aux temps (horloges locales et globales) ont été complètement retirées.

Complet Implémentation complète de la formalisation ne faisant aucune abstraction.

L'intérêt principal de ces deux versions concerne l'efficacité de la vérification. En effet, dû aux nombreux tics de l'horloge globale et des horloges locales, de nombreux états sont introduits pour supporter l'écoulement du temps dans le système. Cependant, pour une grande gamme de propriétés (celle liée au flux de contrôle et de données), le calcul de ces états intermédiaires n'est pas nécessaire.

L'outil associé à ALLOY4PV guide l'utilisateur pour la correction du procédé. En suivant l'ordre établi sur la figure 4.16, l'utilisateur est encouragé à corriger en premier lieu les erreurs trouvées sur la dite perspective avant de passer à la perspective suivante. De cette façon, il est beaucoup plus facile de comprendre et de raisonner sur la raison de cette erreur comportementale qui est présente sur le procédé.

4.4 Conclusion

Ce chapitre présente une implémentation de notre formalisation en utilisant le langage Alloy. Afin d'automatiser tout le processus de vérification, un outil basé sur Eclipse EMF, Obeo Designer et l'API d'Alloy a été développé. L'outil permet de vérifier une grande gamme de propriétés sur le flux de contrôles, de données, des ressources et du temps. Il permet aussi de spécifier et vérifier des propriétés métiers basées sur une librairie paramétrable.

Un des points importants concernant l'utilisation de l'outil est qu'aucun bagage mathématique n'est requis par l'utilisateur pour pouvoir vérifier son procédé. Aucune connaissance du langage Alloy, ni en expression de logique temporelle n'est requise. L'Alloy Analyzer est complètement caché à l'utilisateur. Tout est automatisé à travers l'utilisation de l'interface graphique pour faciliter l'adoption de l'outil.

Bien que l'approche proposée ici soit basée sur du *model-checking* borné, nous prenons en compte les boucles dans le procédé et calculons la longueur de trace nécessaire pour vérifier toutes les exécutions pertinentes du modèle.

De toute évidence, une certaine compréhension de la sémantique de fUML est requise par l'utilisateur pour "comprendre" les erreurs et corriger le modèle. La correction automatique d'erreur comportementale reste un problème ouvert, auquel certains auteurs tentent de répondre [95]. Cependant, sans connaître les intentions réelles du modélisateur, il peut être difficile de proposer une correction adaptée à ses besoins réels.

Le prochain chapitre présente une approche permettant de générer aléatoirement des procédés, afin de pouvoir générer un ensemble de modèles de test pour évaluer Alloy4PV.

Chapitre 5

Generation de procédés basée sur un algorithme génétique multi objectif

La complexité croissante des procédés (ex. métiers, logiciels, médicaux ou militaires), stimule l'adoption de toujours plus de techniques d'exécution, d'analyse et de vérification. Cependant, ces techniques ne peuvent pas être validées précisément du fait qu'il n'est pas possible d'obtenir de nombreux modèles de procédés réalistes afin de les mettres à l'épreuve. Le petit ensemble de modèles de procédés disponible publiquement dans la littérature est généralement insuffisant pour conduire une étude empirique sérieuse et, ainsi, valider complètement les travaux autour de l'analyse et la vérification de procédés.

Dans ce chapitre, nous répondons à ce problème en proposant un générateur aléatoire de procédés basé sur un algorithme génétique multi objectif. L'originalité de notre approche provient du fait que le modèle de procédés est construit au travers d'une séquence d'opérations de haut niveau (appelées *change patterns*) inspirée de la façon dont un modélisateur aurait pu travailler pour modéliser un procédé. De plus, l'approche utilisée est générique et complètement indépendante du langage de modélisation. Un prototype basé sur cet algorithme a été implémenté et montre qu'il est possible de générer rapidement de gros procédés, syntaxiquement corrects, et personnalisés selon le besoin de l'utilisateur.

La section 5.1 présente les motivations pour la création d'une approche permettant de générer des procédés. La section 5.2 présente notre idée pour représenter et générer un procédé. La section 5.3 présente le fonctionnement des algorithmes génétiques et la section 5.4 les utilise pour générer des procédés. La section 5.5 discute de la validité des procédés générés. La section 5.6 présente le prototype que nous avons développé. La section 5.7 présente l'évaluation de ce prototype. La section 5.8 présente l'état de l'art concernant la génération de modèles. Finalement, la section 5.9 conclut ce chapitre en présentant les futures perspectives de ce travail.

5.1 Motivations

Une situation frustrante à laquelle doivent faire face beaucoup de scientifiques est l'incapacité de valider leurs approches par manque de données et de modèles réalistes. Par modèles réalistes, nous entendons de modèles non seulement d'une taille donnée, mais aussi exposant certaines propriétés spécifiques et des constructions complexes, comparables à ce qu'un modélisateur aurait pu produire dans un projet réel. La communauté de modélisation et d'analyse de procédés souffre particulièrement de ce manque. En effet, beaucoup d'approches ont été proposées pour vérifier et analyser des procédés. Cependant, ces approches ne peuvent pas être validées précisément

parce qu'il n'est pas possible d'obtenir de nombreux procédés réalistes afin de les tester. Le petit ensemble d'"échantillons" de modèles publiquement disponibles dans la littérature est généralement insuffisant pour conduire une étude empirique sérieuse. De plus, certains procédés disponibles sont souvent sous forme textuelle ou dans un formalisme inadapté, ce qui entraîne le besoin de convertir le procédé dans le format d'entrée attendu par l'outil d'analyse.

Une des solutions aurait pu être de promouvoir des initiatives afin de mettre en place un dépôt de procédés. Cependant, ce genre d'initiatives rencontre des problèmes de confidentialité. Peu d'organisations acceptent de partager leurs modèles (ex. seuls 150 modèles ont été soumis sur le dépôt Moogle [165]). Additionnellement, aucune garantie n'est disponible sur la qualité et la sûreté de ces modèles, ou, s'ils disposent de propriétés intéressantes, pour tester la validité d'approches de vérification.

Dans la littérature, beaucoup d'approches ont été proposées pour générer des modèles à des fins de tests [177, 180, 74, 202, 31]. Toutes ces approches concernent la génération de modèles ayant des préoccupations structurelles, c'est-à-dire des diagrammes de classes ou autres instances de méta-modèles EMF. Généralement, le but principal de ces approches est de tester le passage à l'échelle de certains outils ou approches. Ces générateurs ont généralement été utilisés pour produire de gros modèles pour tester, par exemple, le passage à l'échelle de la vérification de langage ou la comparaison de modèles. Un des problèmes majeurs concernant ces approches de génération concerne l'impossibilité de définir certains paramètres pendant la génération : (1) des contraintes syntaxiques, (2) des contraintes entre les différents éléments générés, (3) des pondérations sur la façon dont les éléments seront générés, et (4) des objectifs sur le modèle généré. Par exemple, l'application de l'approche proposée par Mougenot *et al.* [180] pourrait potentiellement générer le procédé visible sur la figure 5.1. Cependant, ce procédé, en plus de posséder des erreurs syntaxiques, ne ressemble en rien à un procédé qu'un modélisateur aurait pu avoir modélisé. Le principal problème de toutes ces approches est que, à notre connaissance, aucune d'entre elles n'abordent la génération de modèles *comportementaux*. Un modèle comportemental tel qu'un procédé possède une logique d'execution exprimé en composant le modèle de différent éléments de manière logique pour exprimer une certaine dynamique. Une simple génération d'éléments aléatoire ne garantis pas des modèles réalistes.

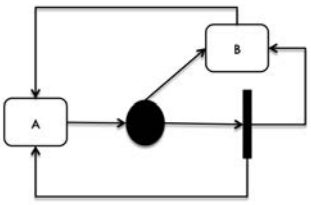


FIGURE 5.1 – Exemple de procédé qui aurait pu être généré par les approches de la littérature

La vérification de procédés est plus préoccupée par la vérification des propriétés comportementales (telles que présentées dans la section 2.2) présentes dans le modèle de procédé, plutôt que simplement leur taille. C'est pourquoi, une approche générant aléatoirement un certain nombre de différents noeuds et arcs n'est pas suffisante. Afin de valider une approche de vérification, il est nécessaire d'avoir des modèles réalistes, comportant certains workflow patterns [259], ayant des constructions complexes telles que les boucles, routage parallèle et conditionnel, synchronisation et fusion de flux alternatifs, combinées d'une façon cohérente, de la façon dont un modélisateur aurait pu les créer.

De plus, il y a généralement différents objectifs derrière chacune des génération de procédés. Par exemple, si le but de la génération est de tester le passage à l'échelle d'approche d'exécution

de procédé, il est important de pouvoir influencer la génération vers un procédé massivement parallèle. Ainsi, nous avons identifié plusieurs *objectifs* de générations :

1. la taille du procédé généré (ex. le procédé doit contenir environ 100 noeuds),
2. le nombre de chacun des éléments du méta-modèle (ex. le procédé doit contenir plus de trois structures de branchement parallèles, et moins de trente activités),
3. des contraintes syntaxiques (ex. les noeuds de branchement conditionnel n'ont pas plus de trois arcs sortants), et
4. la présence (ou non) de *workflow patterns* [259]. Les *workflow patterns* representent un ensemble de patrons récurrents que l'on peux rencontrer quand on définie un modèle de procédé. Ils définissent donc des structures que l'on peux appliquer sur les différentes perspectives d'un procédé, pour répondre a différent problèmes et situations lors de la modélisation.

La section suivante présente notre intuition pour représenter et générer un procédé.

5.2 Idées pour représenter et générer un procédé

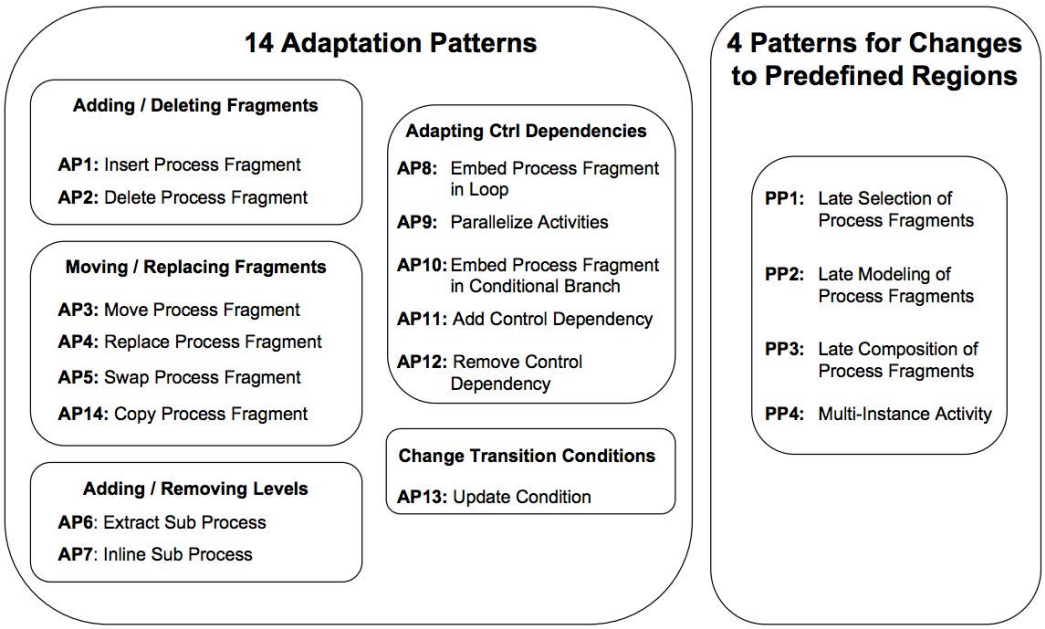


FIGURE 5.2 – Vue d'ensemble des *change patterns* [275]

Dans [275], Weber *et al.* proposent un ensemble d'opérations de haut niveau (*change patterns*) pour adapter un procédé. Cet ensemble a été trouvé empiriquement en utilisant des procédés métiers de différents domaines (santé, automobile, etc). Une vue d'ensemble de ces *change patterns* est visible sur la figure 5.2. Ces *change patterns* permettent donc d'adapter un procédé en ajoutant, supprimant, déplaçant ou modifiant une partie d'un procédé. Dans cette section, nous ne présentons pas l'ensemble de ces *change patterns*, mais donnons simplement un aperçu du change pattern AP1 pour ajouter un fragment de procédé. Comme visible sur la figure 5.3, ce *change pattern* possède trois variantes d'insertion : sequentielle, parrallèle et conditionnelle.

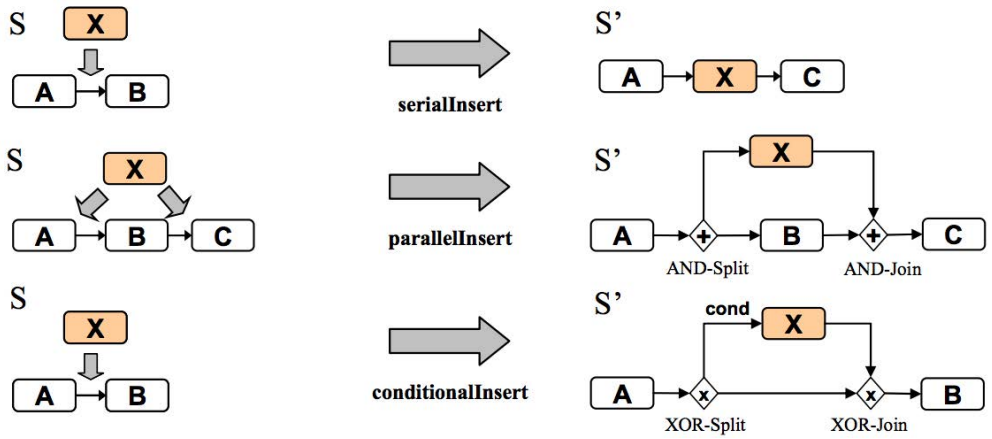


FIGURE 5.3 – "AP1 Insert Process Fragment" change pattern [275]

Le fragment `X` peut être un simple noeud ou un ensemble d'éléments. L'application d'un *change pattern* transforme donc un procédé S en un autre procédé S' contenant le nouveau fragment `X`.

Un procédé peut être représenté comme une séquence d'application de *change patterns*. La figure 5.4 montre la construction d'un procédé en 6 étapes en utilisant seulement les *change patterns* de la figure 5.3. Il est important de noter que les *change patterns* n'ont pas été définis spécifiquement pour les diagrammes d'activité UML, mais sont indépendants du langage de modélisation de procédés. Cependant, générer des procédés en appliquant aléatoirement des *change patterns* ne garantit en rien des procédés réalistes, ou des procédés exposant certains *objectifs* définis (présentés dans la section précédente). Comment atteindre les objectifs fixés par l'utilisateur lors de la génération ? La section suivante présente les algorithmes génétiques, une solution pour générer aléatoirement des procédés atteignant des objectifs désirés.

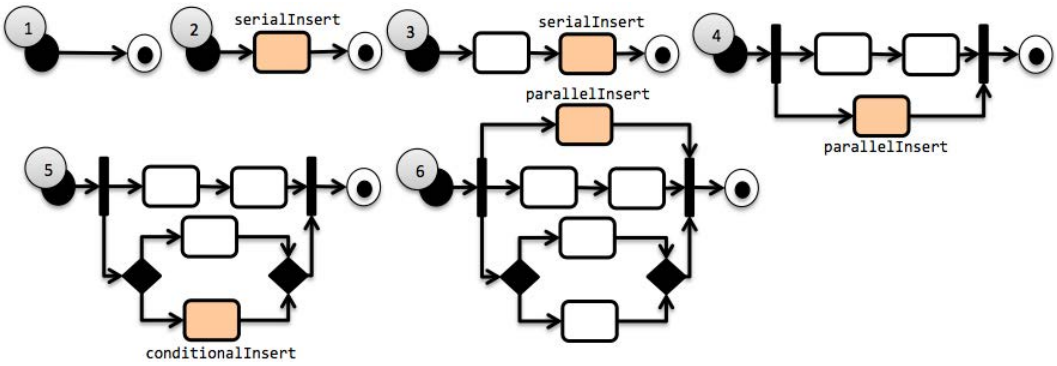


FIGURE 5.4 – Construction d'un procédé en 6 étapes, en utilisant seulement le "Insert Process Fragment" pattern

5.3 Genetic Algorithm

Les algorithmes génétiques (GA, pour Genetic Algorithm) [53] sont des algorithmes de recherche probabilistes qui utilisent le principe de *sélection naturelle* (basé sur la théorie de

l'évolution de Darwin) pour résoudre itérativement un ensemble de solutions (appellé population) vers une solution optimum. Une solution potentielle est appelée un *chromosome*. Un *chromosome* est composé de multiples *gènes*. Un *gène* est une composante distincte d'une solution potentielle.

Durant chaque évolution, le principe de *sélection naturelle* est appliqué pour déterminer quelles solutions survivent, et quelles solutions sont écartées.

Afin d'effectuer le processus de sélection, une *fonction d'évaluation* (*fitness* en anglais) est requise afin d'être capable d'évaluer la qualité d'une solution parmi les autres. La fonction d'évaluation affecte un entier positif, ou *score* (*fitness value* en anglais), reflétant la qualité de la solution donnée (ex. un grand nombre implique une meilleure solution). Ce *score* est ainsi utilisé dans le processus de *sélection naturelle* pour choisir quelle solution potentielle continuera sur la prochaine génération, et lesquelles disparaitront. Cependant, le processus de sélection ne choisit pas forcément les x meilleures solutions. Les solutions sont en réalité choisies statistiquement de manière à ce qu'une solution avec un *score* élevé ait plus de chance d'être choisi, mais rien n'est garanti. Ce comportement est dû au fait qu'une solution peut être temporairement inférieure aux autres, mais pourrait évoluer dans plusieurs générations en une solution encore meilleure que les précédentes "meilleures" solutions. La taille de la population est fixe, ainsi un candidat peut être sélectionné plusieurs fois.

Pour faire évoluer la population en une nouvelle, des *opérations génétiques* sont appliquées sur la population telles que : (1) la *reproduction*, c'est-à-dire faire une copie d'une solution potentielle (2) le *croisement*, c'est-à-dire échanger les gènes entre deux solutions potentielles, simulant l'"accouplement" des deux solutions et (3) la *mutation*, c'est-à-dire l'altération aléatoire des gènes d'une solution potentielle.

L'évolution continue jusqu'à ce qu'une condition de *terminaison* soit atteinte telle qu'une limite de temps ou qu'un *score* désiré soient atteints par un membre de la population.

Ainsi, le plan pour appliquer un algorithme génétique correspond à :

1. **Genèse** : Création d'un ensemble initial de n candidats (population), ou solutions potentielles (fournis ou générés aléatoirement).
2. **Evaluation** : Évaluation de chacun des membres de la population en utilisant une fonction d'évaluation.
3. **Survie du plus apte** : Sélection d'un certain nombre de membres de la population, favorisant ceux ayant un score élevé.
4. **Evolution** : Génération d'une nouvelle population en appliquant des *opérations génétiques*.
5. **Iteration** : Répétition des étapes 2 à 4 jusqu'à ce que la condition de *terminaison* soit atteinte.

5.4 Utilisation d'un algorithme genetique pour la génération de procédés

L'intérêt d'utiliser un GA pour générer des procédés provient de leur habilité à optimiser une population de solutions potentielles vers les objectifs de génération désirés. En donnant à l'algorithme un moyen de faire évoluer un procédé et en donnant une fonction d'évaluation des objectifs, le GA est capable de faire évoluer une population de candidats vers une solution potentielle à travers le principe de sélection naturelle.

La figure 5.5 montre comment le GA est utilisé pour générer un procédé et est expliquée dans la suite en reprenant les étapes décrites précédemment pour appliquer un GA. Un *chromosome* correspond à un procédé quand un élément de celui-ci (c'est-à-dire des noeuds et arcs) représente ses *gènes*.

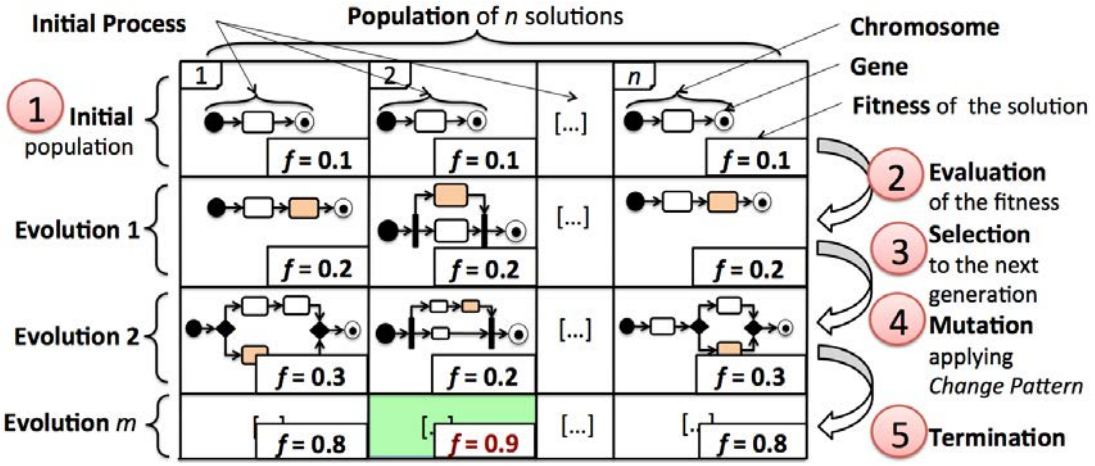


FIGURE 5.5 – Vue globale de l'utilisation d'un GA pour la génération d'un procédé [156]

Genèse. Afin de configurer la population initiale, certaines informations sont requises :

- La taille de la population, qui correspond au nombre maximal de solutions possibles. Une taille importante augmente les chances de converger plus rapidement vers les objectifs, au coût d'un temps de calcul supérieur pour faire évoluer la population.
- Le procédé initial qui évoluera pendant la phase d'évolution. Par défaut, le procédé correspond à un simple procédé avec un noeud initial (ex. `InitialNode`) et un noeud final (ex. `ActivityFinalNode`). Ce choix est naturel du fait que tous les procédés possèdent un début et une fin. Par ailleurs, il est aussi possible d'utiliser un procédé initial défini par l'utilisateur. Dans ce cas, les procédés générés seront une *dérivation* de ce procédé. Ainsi, ce procédé initial est copié dans toute la population initiale.

Evaluation. La *fonction d'évaluation* évalue une solution par rapport aux objectifs définis. En partant du principe que p est une solution potentielle, C_s est la taille désirée du procédé, C_m est la marge acceptée sur la taille C_s, C_e est l'ensemble qui contient le nombre désiré de chacun des éléments du méta-modèle, C_w est l'ensemble qui contient le nombre désiré de *workflow patterns*, C_c est l'ensemble des contraintes syntaxiques, et W_s, W_e, W_w, W_c sont les poids respectifs à chacun de ces objectifs. Soit W la somme de tous les poids. Soit $hold(candidate, objective)$ une fonction qui retourne 1 si l'objectif ($objective$) est atteint par le candidat ($candidate$), et 0 dans le cas contraire. Soit $size(candidate)$ une fonction qui retourne la taille (en terme de nombre de noeuds et arcs) d'un candidat ($candidate$). Soit $margin(x)$ une fonction de seuil utilisé pour connaître les tailles de procédé acceptées selon la marge C_m. La fonction $fitness(candidate)$ retourne un réel entre $[0, 1]$ qui reflète la qualité du candidat (une valeur élevée veut dire que la solution est meilleure) :

$$\mathsf{margin}(x) = \begin{cases} 0 & \text{iff} x \leqslant C_m \\ 1 & \text{iff} x > C_m \end{cases}$$

$$\mathsf{fitness}(p) = \left(\frac{1}{1 + \mathsf{margin}(|\mathsf{size}(p) - C_s|)} \right) * \frac{W_s}{W} + \left(\sum_{w \in C_w} \frac{\mathsf{hold}(p, w)}{\mathsf{card}(C_w)} \right) * \frac{W_w}{W}$$

$$+ \quad \left(\sum_{e \in C_e} \frac{\mathsf{hold}(p,e)}{\mathsf{card}(C_e)} \right) \quad * \quad \frac{W_e}{W} \quad + \quad \left(\sum_{c \in C_c} \frac{\mathsf{hold}(p,c)}{\mathsf{card}(C_c)} \right) \quad * \quad \frac{W_c}{W} \tag{5.1}$$

Soit δ un nombre réel entre $[0, 1]$ représentants le seuil d'acceptation d'un candidat. Soit $fitenough(candidate)$ une fonction qui retourne un booléen déterminant si la solution est considérée suffisamment correcte (c'est-à-dire si les objectifs sont atteints) :

$$\mathsf{fitenough}(p) = 1 - \mathsf{fitness}(p) < \delta \tag{5.2}$$

Il est important de noter qu'une valeur faible affectée à δ implique plus de rigidité afin qu'une solution potentielle p soit jugée suffisamment correcte. Le principal intérêt d'utiliser des poids est que s'il est impossible de remplir tous les objectifs, il est toujours possible de prioriser les objectifs les plus importants en leur associant un poids plus grand. Les *objectifs* peuvent être vus comme des *buts moue* qui doivent être atteint le mieux possible. Plus des objectifs sont ajoutés sur les buts de la génération, plus l'ensemble des solutions potentielles est restreint. Par exemple, il existe moins de solutions possibles correspondant à "générer un procédé de 10 noeuds dont 3 de branchement conditionnel" que simplement "générer un procédé de 10 noeuds". De plus, certain objectifs peuvent être mutuellement exclusifs : "générer un procédé ne contenant aucun noeud de branchement parallèle, mais contenant le workflow pattern *parallel split*". En effet, un procédé contenant le workflow pattern "parallel split" contient forcément un noeud de branchement parallèle. Ainsi dans ce genre de cas, il est impossible de satisfaire tous les objectifs.

Survie du plus apte. La sélection doit favoriser les candidats jugés bons par rapport à ceux jugés plus faibles. Cependant, il n'y a pas de règles générales, aucune stratégie n'est la meilleure pour tous les problèmes. Nous utilisons la technique de sélection proportionnelle aux scores la plus couramment utilisée, appelée "Roulette Wheel Selection" (RWS) [15]. Conceptuellement, chacun des membres de la population se voit attribué une section sur une roulette imaginaire. Une proportion de la roue est attribuée à chacun des candidats en fonction de leurs scores. À l'inverse d'une roue réelle, les sections sont de différentes tailles, proportionnellement à chacune, de manière à ce que les candidats avec un score élevé se voient attribuer une plus grosse portion de la roue. La roue est ainsi tournée, et le candidat associé à la section sélectionnée est choisi. La roue est tournée autant de fois que nécessaire pour sélectionner l'ensemble des parents de la génération suivante. Afin de s'assurer que certains candidats prometteurs ne puissent être perdus lors du passage à la génération suivante, nous utilisons le principe d'*élitisme*. Ce principe implique qu'une portion des meilleurs candidats est directement copiée vers la prochaine génération.

Evolution. Pour chacune des évolutions, certaines *opérations génétiques* sont appliquées sur chacun des candidats de la population. Nous utilisons seulement le principe de *mutation*. Dans notre cas, cette *mutation* correspond à l'application aléatoire d'un *change pattern*. Afin de générer des procédés *réalistes*, il est nécessaire de pouvoir spécifier une probabilité sur la chance d'appliquer un *change pattern* donné sur un candidat. En effet, lorsqu'un modélisateur construit un procédé, il y a plus de chance qu'il effectue un `serialInsert` plutôt qu'un `conditionalInsert`. Ainsi, la fonction d'évolution a besoin de prendre en compte une probabilité définie par l'utilisateur pour chacun des *change patterns*. Il est bien évidemment possible de donner des probabilités par défaut à chacun des *change patterns* en étudiant des procédés de la littérature ou d'un domaine donné. Cependant, il est intéressant de pouvoir modifier ces valeurs. Par exemple, augmenter la probabilité d'application du *change pattern* `parallelInsert` augmentera les chances de générer des procédés massivement parallèles tout en réduisant les chances de générer des procédés plutôt séquentiels. Des probabilités bien affinées sur chacun des *change patterns* permettent de générer des procédés *réalistes* en simulant ce qu'aurait fait réellement un modélisateur. Ainsi, la *mutation*

correspond à appliquer un *change pattern* selon sa probabilité d'être choisie. La population entière évolue en parallèle.

Iteration. La génération termine quand une *condition* est atteinte. Nous proposons plusieurs solutions (non exclusives) pour terminer la génération :

1. Arret quand une solution est trouvée (la fonction $fitenough$ renvoie vraie) ;
2. Arret après x itérations ;
3. Arret après x stagnation du meilleur candidat (aucune amélioration du *score* après plusieurs itérations) ;
4. Arret après x secondes (permets de spécifier un temps d'expiration s'assurant que l'algorithme ne s'exécute pas indéfiniment).

Si plusieurs conditions de terminaison sont spécifiées, l'évolution s'arrête dès que l'une d'entre elles est atteinte. Utiliser simplement la condition n°1 peut impliquer que l'algorithme s'exécute indéfiniment. Par exemple, en cas d'objectifs mutuellement exclusifs, il est possible que le *score* des candidats soit toujours insuffisant pour que la fonction $fitenough$ renvoie vraie. Ainsi, les conditions n°2, 3 et 4 donnent des conditions qui finiront forcément au bout d'un certain temps à être remplies. Quand l'évolution se termine, l'algorithme renvoie le candidat dont le score est le plus élevé. Ainsi, le candidat est la solution, c'est-à-dire le procédé généré.

5.5 Validité des procédés générés

Comme expliqué dans la section 2.2, la validité d'un procédé concerne deux aspects : (1) vérifier si le procédé est bien formé (c'est-à-dire sans les erreurs syntaxiques), et (2) de déterminer en avance, si le procédé exhibe certains comportements désirables (c'est-à-dire sans les erreurs comportementales).

La construction du procédé à travers une séquence de *change pattern* permet d'assurer la sûreté syntaxique. En effet, l'application d'un *change pattern* ne peut entraîner l'ajout d'erreurs syntaxiques [275]. La seule façon de générer des procédés contenant des erreurs syntaxiques est de fournir un procédé en contenant lors de la phase d'initialisation de la population (cf. la phase de *genèse*).

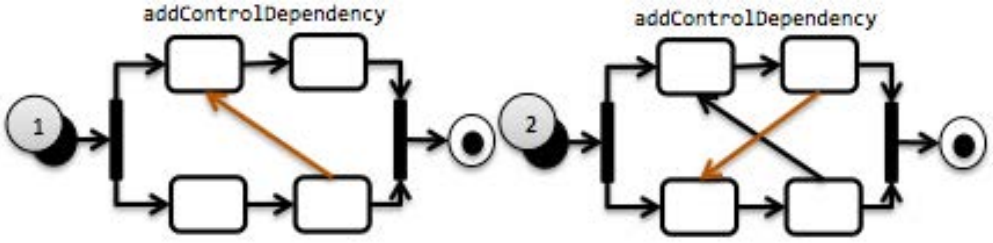


FIGURE 5.6 – Procédé sujet à un interblocage dû à l'application du *change pattern* `addControlDependency`

En revanche, l'algorithme ne peut s'assurer de la sûreté comportementale. Par exemple, la figure 5.6 montre l'application du *change pattern* `addControlDependency` impliquant un interblocage sur le procédé. Les deux `Action` après le `ForkNode` ont besoin d'un jeton sur tous leurs `ControlFlow` d'entrée afin de démarrer, ce qui n'est pas possible avec ces deux `ControlFlow` ajoutés.

Le problème avec la sûreté comportementale est que, à moins d'explorer complètement l'espace d'états du procédé, il n'est pas possible d'en assurer une évidence. Malheureusement, il n'est pas possible de s'offrir ces vérifications dans chacune des étapes de la génération du procédé (ex. après chaque *mutation*, vérifier qu'aucune erreur n'a été introduite sur le candidat) du fait qu'explorer entièrement l'espace d'états est notoirement connu pour être un problème exponentiel, qui échoue en temps de vérification pour de gros modèles [81].

Cependant, notre but en générant des procédés est de simuler la façon dont un modélisateur aurait pu créer un procédé. Ainsi, les procédés générés peuvent contenir des anomalies comportementales de la même façon qu'un modélisateur pourrait construire un procédé avec des anomalies comportementales. De plus, générer des procédés avec ce type d'anomalies est un point important pour tester des approches de vérification formelle, telles que présentées dans le chapitre 4.

5.6 Prototype

Le prototype que nous avons développé est actuellement fourni comme un plug-in Eclipse EMF (Eclipse Modeling Framework). Nous utilisons le *"Watchmaker Framework"* [68] pour implémenter l'algorithme génétique multi objectif. Ce framework fournit une API orientée-objet, extensible et haute performance (multi threadée) pour implémenter des algorithmes génétiques en Java.

La figure 5.7 montre ce prototype. Du fait que notre solution est indépendante du langage de modélisation, et afin de favoriser son adoption, nous avons intégré dans l'outil la possibilité de générer des procédés basés sur UML AD 2.0, mais aussi des modèles BPMN.

Concernant l'utilisation de l'outil, il existe trois types de réglages :

1. configuration générale :
   - Etiquette 1 : sélection de l'emplacement de génération.
   - Etiquette 2 : bouton pour démarrer et arrêter la génération.
   - Etiquette 3 : choix du type de modèle généré (UML AD ou BPMN).
2. Réglages des objectifs :
   - Etiquette 4 : choix de la taille des procédés générés (en terme de noeud et de marge autorisée).
   - Etiquette 5 : choix du nombre d'éléments présents du méta-modèle.
   - Etiquette 6 : choix des *workflow patterns* présents.
   - Etiquette 7 : contraintes syntaxiques (exprimées en utilisant le langage OCL).
3. Réglages du GA
   - Etiquette 8 : taille de la population et procédé initial (par défaut, un procédé avec simplement un noeud de début et de fin est utilisé).
   - Etiquette 9 : choix du nombre de candidats sujet à l'élitisme et de la stratégie de sélection (ex. Roulette Wheel Selection).
   - Etiquette 10 : choix des *change patterns* utilisés pour les mutations ainsi que leurs probabilités d'être appliqués.
   - Etiquette 11 : choix des conditions de terminaisons.
   - Etiquette 12 : choix des poids associé aux objectifs, afin de prioriser ceux-ci.

Des valeurs par défaut existent pour chacun des paramètres d'entrées de la configuration, de manière à ce que l'utilisateur ait simplement à spécifier le répertoire de destination afin de pouvoir commencer à générer des procédés.

5.6.1 Détails techniques d'implémentation

Afin d'implémenter l'algorithme génétique, les deux principaux concepts à implémenter correspondent aux *opérations génétiques* et à la *fonction d'évaluation*.

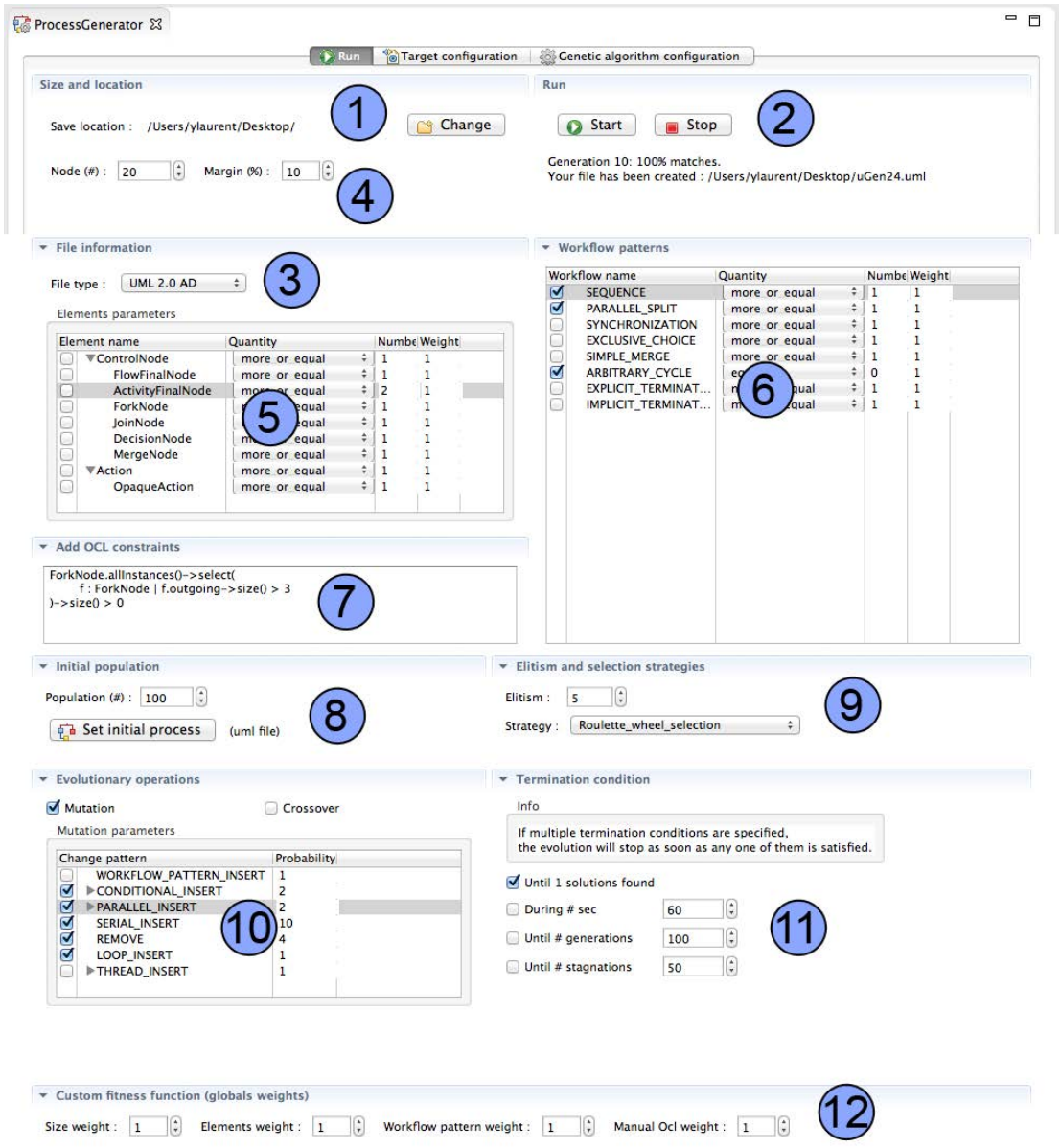


FIGURE 5.7 – Prototype du générateur de procédé intégré dans Eclipse

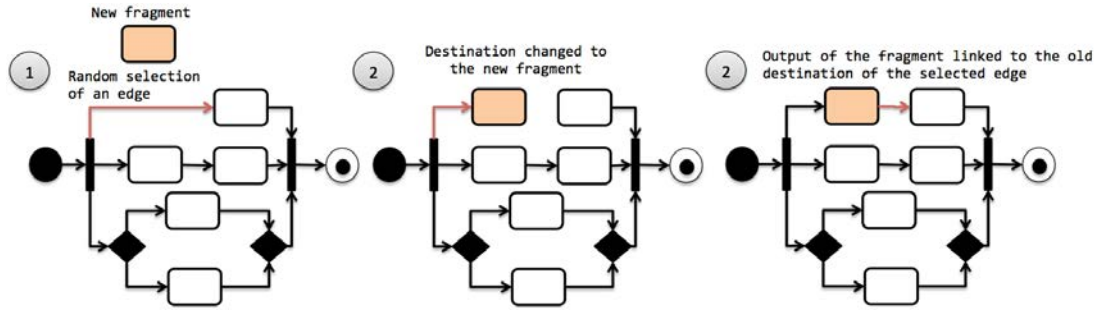


FIGURE 5.8 – Procédure pour appliquer aléatoirement le *change pattern* `serialInsert`

Opérations génétiques. La façon dont les *mutations* sont appliquées sur un candidat est faite de manière aléatoire. Prenons en exemple l'application du *change pattern* `serialInsert`. Les trois étapes concernant son application sont visibles sur la figure 5.8 : (1) un arc du procédé est choisi aléatoirement ; (2) la destination de cet arc est changée vers l'entrée du nouveau fragment à insérer ; (3) la sortie du fragment est reliée à l'ancienne destination de cet arc. Chacun des *change patterns* possède sa propre liste d'actions à effectuer afin d'être appliqué sur un procédé. La figure 5.9 montre la classe `UmlSerialInsert` implémentant le *change pattern* `serialInsert` pour appliquer cette mutation sur un procédé basé sur les diagrammes d'activité UML. L'algorithme suit exactement les étapes de la figure 5.8.

Fonction d'évaluation. Afin de calculer le *score* attribué aux candidats durant la phase d'évaluation, il est nécessaire d'évaluer chacun des objectifs isolément pour savoir si ceux-ci sont satisfaits par le candidat. La taille du procédé et le nombre d'éléments du méta-modèle correspondent simplement à parcourir la structure du procédé en comptant les différents éléments et en les comparant avec la valeur spécifiée par l'utilisateur. Nous utilisons l'API EMF d'eclipse pour parcourir la structure du modèle. Concernant la présence de *workflow pattern*, de la même manière, il faut explorer le modèle pour trouver la présence de certaines structures. La plupart des *workflow pattern* sont analysés en utilisant des interrogations OCL. Par exemple, le *workflow pattern* `ParallelSplit` correspond à vérifier que le modèle possède au moins un noeud de routage parallèle (ex. `ForkNode`) ayant au moins deux arcs de sortie. L'API EMF fournit aussi le framework pour analyser des requêtes OCL contre un modèle EMF. En revanche, certains *workflow patterns* tel que la présence de cycles arbitraires (ou boucles) nécessitent de parcourir le graphe du procédé et n'est pas exprimable en utilisant OCL. Pour ce type de *workflow pattern*, nous utilisons l'API EMF pour parcourir "à la main" la structure du modèle et vérifier leur présence. Finalement, l'objectif correspondant aux contraintes syntaxiques est spécifié par l'utilisateur directement en utilisant le langage OCL. De la même manière, il suffit simplement d'utiliser l'API EMF pour vérifier ces objectifs. Ainsi, selon les résultats de chacune des évaluations des objectifs, le *score* est calculé en utilisant la fonction 5.1.

5.7 Évaluation

L'efficacité du prototype, en terme de temps de génération, dépend grandement de la manière dont il est configuré. D'une manière générale, plus les objectifs de générations sont précis, plus le temps de génération est important. Par ailleurs, déterminer les *meilleurs* paramètres pour configurer l'algorithme génétique (ex. taille de la population), en fonction des objectifs de génération, est loin d'être trivial. Par *meilleurs* paramètres, nous entendons les paramètres

```
public class UmlSerialInsert extends AbstractChangePattern<UmlProcess> {
    @Override
    public UmlProcess apply(UmlProcess oldProcess,
            Random rng,
            List<StructuralConstraintChecker> workflowsConstraints) {

        // create a copy
        UmlProcess process = new UmlProcess(oldProcess);

        // --- (1) find a random arcs
        ActivityEdge edge = null;
        try {
            edge = UmlChangePatternHelper.instance.getRandomActivityEdge(process, rng);
        } catch (UmlException e) {
            // Can't not find a random arc, return the process without mutation
            return process;
        }
        // keep a reference to the old target of the edge
        ActivityNode oldTarget = edge.getTarget();
        // creation of a random node (or fragment)
        ActivityNode node = process.buildAction();

        // --- (2) change destination of the edge to the new node
        edge.setTarget(node);

        // --- (3) create a new control flow to link the old destination
        process.buildControlFlow(node, oldTarget);

        return process;
    }
}
```

FIGURE 5.9 – Classe implémentant la mutation `serialInsert` pour les diagrammes d'activités UML

optimisant le temps de génération afin de pouvoir converger vers une solution le plus rapidement possible. Par exemple, une population importante implique une diversité plus grande sur laquelle les solutions seront recherchées et ainsi améliore les chances d'obtenir, sur peu de générations, directement une solution. En revanche, le temps pour faire évoluer cette "grande" population à travers les opérations génétiques sera plus important.

En outre, les algorithmes génétiques étant basés sur l'application d'opérations génétiques et de sélection *aléatoire*, pour les mêmes objectifs, une solution peut être trouvée en 20 générations comme en 80. Cela dépend, en partie, de la chance de tomber rapidement (ou non) sur des candidats satisfaisant les objectifs. Ainsi, il est difficile d'évaluer, de manière *précise*, l'efficacité de notre prototype pour générer un procédé. Nous proposons donc deux évaluations : (1) le temps pour générer un procédé selon les mêmes objectifs, mais en faisant évoluer la taille de la population, et (2) le temps pour générer un procédé en faisant évoluer la complexité des objectifs. Toutes les mesures suivantes ont été faites sur un MacBook Air 2011 avec un processeur Intel Core i5 avec 4 GB de RAM. Nous avons fait la moyenne sur 100 exécutions afin de combattre le côté aléatoire de l'algorithme.

Nous réglons l'outil pour générer des diagrammes d'activités UML et initialisons les poids tel que $W_s = 1$, $W_e = 1$, $W_w = 1$, $W_c = 1$, et $\delta = 0$ (vu que $\delta = 0$, les poids importent peu étant donné que tous les objectifs doivent être remplis afin qu'une solution soit considéré comme valide)

Nous initialisons les objectifs tels que : $C_s = 30$ (taille des modèles générés en nombre de noeuds), $C_m = 10\%$ (marge pour la taille), $C_e = \{\{ForkNode \geq 2\}, \{JoinNode \geq 2\}\}$ (nombre d'éléments du méta-modèle), $C_w = \{\{ExclusiveChoice \geq 1\}, \{ArbitraryCycle > 2\}\}$ (présence

```
ForkNode.allInstances()->select(
        f : ForkNode | f.outgoing->size() > 3
)->size() >= 2
```

Listing 5.1 – Contrainte OCL exprimant qu'"au moins deux `ForkNode` possèdent plus de trois arcs sortants"

de workflow pattern), C_c = *"il doit y avoir au moins deux `ForkNode` possédant plus de trois arcs sortants"* (contrainte syntaxique, exprimée avec OCL et visible sur la figure 5.1).

Concernant les paramètres du GA, soit E_p l'ensemble associant à chacun des *change patterns* une probabilité d'application, E_e la portion de la population sujette à l'élitisme, E_c la condition de terminaison, E_s la taille de la population, et E_i le procédé pour initialiser la population initiale. Nous initialisons ces paramètres comme suit : $E_p = \{ conditionalInsert \rightarrow 2, parallelInsert \rightarrow 2, serialInsert \rightarrow 10, remove \rightarrow 4, loopInsert \rightarrow 1\}$ (probabilité sur chacun des *change patterns*, $E_e = 5$ (élitisme), $E_c =$ "jusqu'à la première solution trouvée", et E_i correspond au procédé par défaut avec un simple noeud initial et un noeud final.

Taille de la population (E_s)	Nombre de générations	Temps de génération
50	128	5.8s
100	54	**4.6s**
200	41	6.4s
400	36	9.3s
800	33	15.0s

TABLE 5.1 – Temps pour générer un procédé en fonction de la taille de la population

(1) Taille de population. Le tableau 5.1 montre le temps pour générer un procédé en fonction de la taille de la population. Les résultats montrent qu'une taille importante de la population permet de réduire le nombre de générations afin de trouver une solution. En revanche, le temps pour trouver une solution devient plus important (si $E_s \geq 100$) du fait qu'il est nécessaire d'appliquer les *opérations génétiques* sur l'ensemble de la population. Dans le cas d'une population trop faible (E_s) ≤ 50), beaucoup plus de générations sont nécessaires afin de, finalement, trouver une solution. Bien que le temps pour faire évoluer une population de petite taille soit plus faible, le temps global de génération est augmenté du fait que plus de générations sont nécessaire pour arriver à la solution. Selon ces mesures, il semblerait qu'une population de 100 candidats soit la plus optimale pour avoir un temps de génération le plus petit. Cependant, il est important de noter que cette valeur est vraie dans ce cas-ci seulement. En effet, dans un autre contexte avec d'autres objectifs de générations, il est possible qu'une population plus importante ou plus faible soit plus optimale en terme de temps pour arriver à une solution.

(2) Complexité des objectifs. Le tableau 5.2 montre le temps pour générer un procédé en fonction de la complexité des objectifs (la première ligne correspond au même test que la seconde ligne du tableau 5.1). La taille de la population est fixe et a pour valeur 100. Chacune des valeurs numériques des objectifs est multipliée par un facteur pour augmenter la complexité (ex. la taille désirée (C_s) passe de 30 à 60, le nombre de `ForkNode` requis passe de 2 à 4, etc). L'expérimentation montre qu'il est possible de générer relativement rapidement des procédés de grande taille. Le temps de génération est linéaire en fonction de la taille des procédés générés.

Complexité des objectifs	Nombre de générations	Temps de génération
$\times 1$ ($C_s = 30, \ldots$)	54	4.6s
$\times 2$ ($C_s = 60, \ldots$)	96	9.1s
$\times 4$ ($C_s = 120, \ldots$)	188	17.7s
$\times 8$ ($C_s = 240, \ldots$)	366	34.5s

TABLE 5.2 – Temps pour générer un procédé selon la complexité des objectifs

5.8 Travaux connexes

Alloy [177] étant un *"model-finder"*, il est intrinsèquement capable de générer des modèles. En effet, à partir d'une formule logique exprimée avec le langage Alloy, l'`Alloy Analyzer` essaye de trouver un modèle de façon à ce que la formule soit valide. Cependant, du fait que Alloy est basé sur un SAT solver, la complexité pour trouver un modèle appartient à la class NP-complet, et ainsi il n'est pas capable de produire de grands modèles. De plus, il n'est pas possible d'influencer la manière dont le SAT solver résoud les contraintes limitant le réalisme des procédés générés. Il est seulement possible de configurer les bornes supérieures des éléments composant le modèle.

Mougenot *et al.* [180] proposent un générateur aléatoire et uniforme de grosses instances de méta-modèle. L'approche repose sur la méthode de l'échantillonnage aléatoire de Boltzmann afin de générer, de façon scalable, des instances d'un méta-modèle donné de n'importe quelle taille. En plus, l'approche est capable d'influencer la génération en ajoutant des pondérations sur les éléments. Cependant, cette approche ne supporte pas l'addition de contraintes sur la syntaxe du méta-modèle et se limite à produire des modèles seulement valides envers son méta-modèle.

Brottier *et al.* [31] présentent un formalisme pour générer aléatoirement des modèles avec des contraintes, afin de tester des approches de transformations de modèles. L'approche consiste à dériver un ensemble de modèles d'exemple d'entrée en une instance aléatoire en utilisant un algorithme "fait maison". Un problème majeur de l'algorithme est qu'il nécessite une instance d'un modèle pour en générer d'autres. Par conséquent, le modèle de sortie peut avoir beaucoup de similitudes avec celui d'entrée.

Ehrig *et al.* [74] présentent un algorithme qui peut générer des instances d'un méta-modèle. Le méta-modèle est transformé en un ensemble de règles de spécification de graphes, puis ces règles sont appliquées aléatoirement pour effectuer la génération.

Pietsch *et al.* [202] présentent un générateur de modèles de tests pour les outils de traitement de modèles. Ils utilisent un contrôleur stochastique pour appliquer des opérations de bas niveau (créer, supprimer, mettre à jour, déplacer) et des opérations plus complexes (composées de ces opérations bas niveau) sur les éléments du modèle. Les éléments sont choisis en utilisant une méthode de sélection inspirée de celle utilisée par les algorithmes génétiques.

Malheureusement, [177, 180, 74, 202, 31] ne sont pas adaptés pour la génération de modèles *comportementaux* (ex. un procédé) et peuvent produire des modèles irréalistes du fait que la possibilité d'influencer la génération (en utilisant, par exemple, des pondérations) est soit non disponible, soit gérée seulement sur un élément donné (et non sur un ensemble d'éléments comme notre approche avec les *change patterns*). Ce problème provient du fait que ces approches ont pour but de générer des modèles *statiques* et n'ont pas été adaptées à l'égard de la génération de modèles de procédés.

5.9 Conclusion

Ce chapitre présente un algorithme génétique multi objectif pour générer des procédés. Le générateur construit à partir de cet algorithme possède 3 particularités intéressantes. Premièrement, il passe a l'échelle, la complexité de l'algorithme de génération est linéaire en fonction de la taille des procédés générés. Il permet aussi de générer des procédés selon différents objectifs, permettant de générer des procédés réellement personnalisés selon les besoins de l'utilisateur. Dans le cas où les objectifs ne sont pas réalisables, l'algorithme est toujours capable de retourner un procédé qui s'en rapproche le plus, tout en prenant en compte l'importance attribuée à chacun des objectifs par l'intermédiaire de leurs poids associés. Finalement, la génération permet de s'assurer de la validité syntaxique du modèle à travers la séquence d'application des *change patterns* et simule la façon dont un modélisateur aurait pu modéliser son procédé. Cela permet de s'assurer de générer des procédés réalistes tout en simulant les erreurs qu'un modélisateur aurait pu avoir fait.

L'algorithme peut être facilement étendu avec de nouveaux objectifs de génération en modifiant la fonction d'évaluation. Les objectifs présentés ici se concentrent principalement sur les aspects syntaxiques des procédés. Cependant, nous pouvons aussi imaginer l'ajout d'objectifs comportementaux (ex. trois actions peuvent s'exécuter simultanément). Par ailleurs, il serait aussi intéressant d'explorer d'autres *opérations génétiques* basées sur le principe du *croisement* afin de combiner directement entre eux des candidats (ou des partis de ceux-ci) dans le but de converger plus rapidement vers les objectifs définis.

La génération se concentre sur l'aspect structurel du procédé (c'est-à-dire le flux de contrôle). Une extension intéressante pourrait être de générer aussi les informations organisationnelles (telles que les ressources, les contraintes temporelles, etc) associées au procédé. Cependant, ces informations sont souvent dépendantes du domaine et il peut être difficile de trouver une solution générique qui s'adapte à tous les types de procédés. De plus, un inconvénient de l'ensemble des *change patterns* proposés par Weber *et al.* [275] provient du fait qu'ils ont été définis seulement pour le flux de contrôle. Un ensemble de *change patterns* incluant les éléments de données doit être établi afin de générer le flux de contrôle et de données de manière unifiée.

Finalement, la technique de génération proposée dans ce chapitre ouvre la voie à des applications plus larges que la simple génération des procédés. Par exemple, en initialisant la population avec un procédé donné et en réglant le bon objectif dans la fonction d'évaluation, il serait possible de chercher automatiquement pour des *dérivations* de ce procédé, qui répondraient aux besoins souhaités (ex. vers la correction automatique des erreurs comportementales, similairement aux travaux de Gambini *et al.* [95]).

Chapitre 6

Evaluation

Dans ce chapitre, nous présentons l'évaluation de nos contributions. La section 6.1 a pour but d'évaluer si nos objectifs de recherche ont été accomplis. La section 6.2 évalue les performances de notre approche en analysant trois cas d'études : (1) un procédé de la spécification UML, (2) un procédé fourni par un partenaire du projet MERgE, et (3) un ensemble de procédés générés.

Il est important de noter qu'une partie du travail réalisé concernant l'implémentation des différents outils constitue déjà en soi une partie de l'évaluation, en ayant montré comment les outils répondaient aux problématiques soulevées dans ce document. Ainsi, la majeure partie de ce chapitre se concentre sur la vérification de propriétés sur un modèle de procédé.

6.1 Accomplissement des objectifs

Comme cela a été présenté dans l'introduction et souligné dans l'état de l'art, les approches de la littérature souffrent non seulement d'un problème concernant leurs formalisation (par exemple, manque de perspectives supportées) (Ⓐa1), d'un support limité de type de propriétés (Ⓐa2), mais aussi d'un manque d'outillage pour automatiser le processus de vérification (Ⓑb1) et pour exprimer des propriétés métiers (Ⓑb2).

Dans la suite, nous reprenons ces problèmes en expliquant comment nos contributions se positionnent par rapport à ces manques.

6.1.1 Formalisation

La formalisation présentée dans le chapitre 3 utilise la logique de premier ordre pour définir le système afin d'utiliser un langage mathématique de haut niveau et se détacher des choix sémantiques mis en place par les langages utilisés par les *model-checker*. Comme expliquée dans la section 3.2, notre formalisation est capable de prendre en compte toutes les perspectives :

- le **flux de contrôle et de données**, à travers l'utilisation des structures proposées par les diagrammes d'activités UML et la sémantique fUML basée sur le passement de jetons ;
- les **ressources**, à travers la possibilité d'affecter des ressources aux actions, en spécifiant leurs capacités d'utilisations. Les prédicats d'activation pour démarrer un noeud prennent en compte la disponibilité des ressources ;
- le **temps**, à travers l'utilisation d'horloges globales et locales aux actions pour exprimer l'écoulement du temps dans le système. Encore une fois, les prédicats d'activation prennent en compte ces concepts afin de s'assurer que le temps écoulé pendant l'exécution d'une action est suffisant.

Le tableau 2.2 de l'état de l'art résume l'ensemble des perspectives supportées par les approches de la littérature et souligne le fait que, seule, notre approche les prend toutes en compte. En effet, comme exposé dans la section 3.3, l'ensemble des propriétés présentées dans la section 2.2 peut être exprimé en utilisant (P)LTL du fait que le modèle formel expose toutes les propositions atomiques nécessaires, et que sa sémantique de temps linéaire s'est révélée être parfaitement adaptée dans le contexte de la vérification de procédés métiers.

Dans le chapitre 4, nous avons choisi d'implémenter la formalisation avec le langage Alloy. Alloy étant un langage de modélisation orienté-objet et basé sur la logique de premier ordre, nous avons pu implémenter le modèle formel de manière élégante et directe. Chacun des concepts a pu être implémenté de manière modulaire. En effet, un module est présent pour représenter la syntaxe, la sémantique, la librairie de propriétés, le modèle de procédé à vérifier et l'ensemble des propriétés à analyser. Cette implémentation modulaire permet, dans le futur, de pouvoir facilement maintenir et mettre à jour l'implémentation. De plus, Alloy s'est montré assez expressif et versatile pour exprimer tous les concepts formalisés de manière élégante, avec pour seul compromis, la borne sur chacune des signatures. En effet, les analyses effectuées par l'Alloy Analyzer s'apparentent à du *model-checking* borné. Cependant, comme montrée dans la section 4.2.5, la possibilité de calculer le nombre d'états nécessaire pour vérifier "complètement" le procédé permet d'enlever les restrictions apportées par le *model-checking* borné [25]. Par restrictions apportées, nous parlons de la limite d'analyse du système sur une profondeur bornée k.

La traduction de l'ensemble des propriétés exprimées en PLTL peut être exprimée de manière presque directe en logique de premier ordre en suivant la traduction proposée par [47] (LTL vers Alloy). Par ailleurs, certaines propriétés demandant de pouvoir comparer les propositions atomiques de plusieurs états entre eux sont naturellement exprimables en utilisant Alloy. Ceci est dû aux *"traces pattern"* où chacun des états est modélisé explicitement. Ce type de propriété n'est pas typiquement exprimable en utilisant un *model-checker* classique supportant seulement la LTL. Dans ce contexte, il est nécessaire d'augmenter le système avec les variables nécessaires sur chacun des états. Cependant, cet ajout au système entraîne un accroissement de la complexité, et donc du temps de l'analyse.

Par ailleurs, l'utilisation de prédicats pour définir les propriétés (cf. `Library.als` section 4.2.3) permet d'exprimer de manière naturelle des propriétés paramétrables afin de représenter notre bibliothèque de propriétés métiers. Chacun des paramètres du prédicat représente un paramètre nécessaire pour utiliser la propriété.

En somme, Alloy s'est montré parfaitement adapté pour exprimer à la fois le système et les propriétés sur celui-ci répondant aux problèmes d'*inadéquation du formalisme* (a1) et du support *limité de type de propriétés* souligné dans l'introduction (a2).

6.1.2 Mise en application

Dans le chapitre 2, nous avons catégorisé, identifié et présenté l'ensemble des propriétés importantes à vérifier sur un procédé. Dans cette catégorisation, nous avons présenté l'ensemble des propriétés métiers essentielles (tableau 2.1 page 28) exprimables sur un procédé. Toutes ces propriétés ont été formalisées en (P)LTL puis implémentées dans le langage Alloy.

En utilisant l'outil associé à ALLOY4PV dans la section 4.3, il est non seulement possible de vérifier très facilement l'ensemble de ces propriétés en cochant celles désirées dans l'interface graphique, mais aussi d'exprimer des propriétés métiers en sélectionnant les éléments du procédé sur lesquels la propriété s'applique. Pour les utilisateurs plus expérimentés, il est possible de *composer* l'ensemble des propriétés métiers en utilisant les opérateurs de la logique ($\neg, \wedge, \vee, \implies$) à l'aide de l'interface graphique. Ainsi, l'utilisateur n'aura jamais à exprimer lui même la propriété en utilisant, par exemple, une logique temporelle. L'outil traduit automatiquement le modèle de procédé (`ProcessToAlloy`) et les propriétés choisies et exprimées dans l'interface graphique

(`PropertiesToAlloy`) dans sa représentation Alloy (cf. `ProcessModel.als` et `Properties.als` page 98).

De plus, afin de faciliter la compréhension des résultats d'analyse (`AlloyToProcess`), nous avons mis plusieurs éléments en place :

1. le résultat de l'analyse est affiché graphiquement sur le procédé, en coloriant en vert ou en rouge le chemin d'exécution satisfaisant ou invalidant (contre-exemple) la propriété ;
2. un message sous forme de `CommentNode` est inséré pour notifier à l'utilisateur un problème concernant la modélisation effectuée ;
3. les différentes perspectives sont vérifiées en isolation en suivant l'ordre établi dans la section 4.3.1. En effet, comme souligné dans cette section, un modèle peut exhiber le même contre-exemple pour deux propriétés différentes. Ainsi, l'utilisation de l'ordre de vérification permet de cibler et comprendre plus facilement la *cause* du problème.

L'approche proposée réduit le travail du modélisateur à la spécification de son procédé, sans qu'il ne se préoccupe de la représentation formelle. Le *model-checker* (Alloy Analyzer) est complètement caché à l'utilisateur, et les résultats renvoyés par celui-ci sont analysés et traités par l'outil afin d'être affichés graphiquement. Aucune connaissance particulière en méthodes formelles n'est nécessaire pour utiliser l'outil, permettant à un modélisateur lambda de tirer profit de la vérification formelle.

Ainsi, l'outil associé à Alloy4PV permet de répondre à la problématique concernant *la difficulté d'expression des propriétés métiers* (b1), mais aussi au *manque d'outillage* (b2) concernant l'automatisation du processus de vérification à travers les traductions automatiques, la vérification des perspectives en différentes étapes, et un affichage graphique des résultats.

6.2 Performance

Dans cette section, nous donnons les performances concernant l'analyse de procédé avec l'outil Alloy4PV. Nous proposons d'utiliser l'outil sur trois cas d'étude :

1. le procédé tiré de la spécification UML pour gérer les commandes reçus par une l'entreprise,
2. un procédé fourni par un partenaire du projet MERgE, et
3. un ensemble de procédés générés avec l'outil présenté dans le chapitre 5.

Probleme SAT. Afin d'effectuer la vérification, l'Alloy Analyzer réduit la spécification Alloy en un problème SAT. Il le présente à un SAT solveur dans le format Conjunctive Normal Form (CNF). Une formule CNF est une conjonction de clauses. Chacune des clauses représente une disjonction de variables binaires. Une solution d'un problème SAT consiste en une affectation de variables booléennes de façon à ce que toutes les clauses soient satisfaites. Généralement, la complexité d'un problème SAT est mesurée par son nombre de clauses et de variables.

Cadre de l'analyse. Toutes les analyses de cette section ont été effectuées sur un MacBook Air 2011 avec un processeur Intel Core i5 basse tension et 4GB de RAM. Le système d'exploitation utilisé est OS X *"Yosemite"* (version 10.10). Le SAT solveur utilisé est à MiniSat [71], un de ceux fournis par défaut avec l'Alloy Analyzer. Chacun des résultats donnés dans les sous-sections suivantes correspond à une moyenne effectuée sur 10 analyses.

6.2.1 Procédé de la spécification UML

La figure 6.1 montre le procédé `ProcessOrder` tiré de la spécification UML [190], modélisé comme un diagramme d'activité UML dans l'outil Alloy4PV grâce à Obeo Designer. Ce

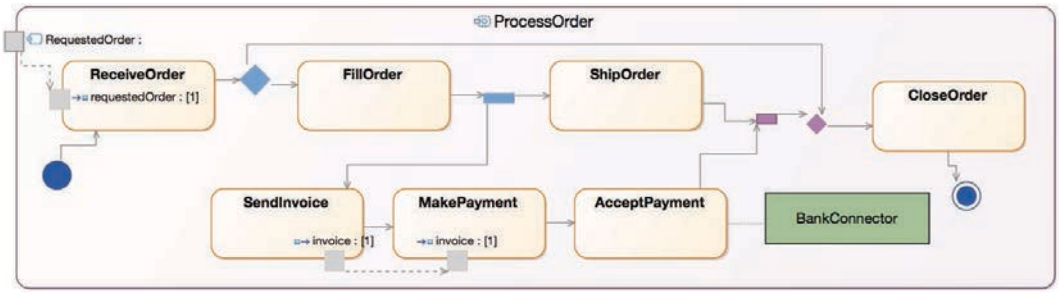


FIGURE 6.1 – Procédé `ProcessOrder` de la spécification UML [190] modélisée avec Obeo Designer

procédé est le même que celui présenté sur la figure 1.3 de l'introduction. Le temps affecté aux actions n'est pas visible graphiquement, mais seulement comme attribut du noeud dans la vue `Properties` d'eclipse. Ainsi, les temps correspondent à :

$$\begin{aligned} Timing = \{&(ReceiveOrder, 1), (FillOrder, 2), (SendInvoice, 1) \\ &(MakePayment, 1), (AcceptPayment, 2), (ShipOrder, 3) \\ &(CloseOrder, 3), \ldots\} \end{aligned}$$

Le tableau 6.1 résume les résultats obtenus pour analyser les propriétés présentées dans la section 2.2.2. La colonne 1 représente la variante de la propriété utilisée (`run` ou `check`). La colonne 2 représente le nom de la propriété analysée. La colonne 3 définit la version du module `Semantics.als` utilisé (voir section 4.3.1). Lorsque le module complet est utilisé (c'est-à-dire la version incluant les ressources et les horloges globales et locales pour gérer l'écoulement du temps), nous utilisons l'icône . Les colonnes 4 et 5 représentent respectivement le nombre de variables et de clauses. Les colonnes 6 et 7 représentent respectivement le temps pour générer la CNF et le temps pour résoudre le problème SAT. La colonne 8 spécifie le résultat de la vérification : si une instance a été trouvée lors de l'utilisation de la commande `run` (**Instance**), ou si un contre-exemple est présent lors de l'utilisation de la commande `check` (***CE***). Dans le cas où aucun de ces deux résultats n'est disponible, nous affichons "-". La longueur de trace, comme expliquée dans la section 4.2.5, a été calculée à 27 `State`s pour la sémantique sans le temps, et 38 `State`s pour la sémantique avec le temps.

Il est important de noter que si aucun contre-exemple n'est trouvé lors de la vérification d'une propriété forte (`check`), la vérification de son homologue faible trouvera forcément au moins une instance satisfaisant la propriété. Par exemple, si aucun contre-exemple n'est trouvé de sorte que le procédé entre en situation d'interblocage (`check Completion`), alors trouver une instance telle que le procédé puisse terminer est trivialement vrai (`run Completion`). En effet, de manière général, l'Alloy Analyzer est simplement capable de trouver des instances satisfaisant une formule (`run F`). Afin de générer des contre-exemples lors de la vérification (`check F`), l'Alloy Analyzer tente simplement de trouver des instances de la négation de la formule (c'est-à-dire que `run F` ≡ `check ¬F`). En règle générale, il est intéressant d'évaluer une propriété faible seulement lorsque la propriété forte a exhibé un contre-exemple.

Flux de contrôle. Le procédé ne souffre pas d'interblocages, se termine toujours proprement, et les 3 actions testées ne sont pas mortes (elles sont toutes exécutables par au moins *une* exécution, telle qu'attestée par la présence d'un contre-exemple).

Flux de données. Le procédé ne souffre ni de données manquantes ni redondantes.

Type	Propriété	Semantic	Variables	Clauses	CNF	SAT	Résultat
	Flux de contrôle						
run	`Completion`	-	397k	1094k	17s	0.3s	Instance
check	`Completion`	-	393k	1093k	16s	0.2s	-
run	`CompletionClean`	-	395k	1098k	17s	0.4s	Instance
check	`CompletionClean`	-	397k	1099k	16s	3s	-
check	`DeadTransition[ReceiveOrder]`	-	397k	1093k	17s	0.2s	CE
check	`DeadTransition[FillOrder]`	-	397k	1093k	17s	1.8s	CE
check	`DeadTransition[SendInvoice]`	-	396k	1098k	21s	1.3s	CE
	Flux de données						
run	`NoMissingData`	-	397k	1094k	20s	2.1s	Instance
check	`NoMissingData`	-	407k	1127k	16s	5s	-
run	`NoRedondantData`	-	395k	1097k	17s	0.5s	Instance
check	`NoRedondantData`	-	401k	1095k	16s	4s	-
	Ressources						
run	`LackOfResource`		611k	1701k	24s	2.5s	Instance
check	`LackOfResource`		621k	1725k	22s	33s	-
run	`InefficientRessourceUseA[BankConnector]`		614k	1704k	22s	31s	Instance
check	`InefficientRessourceUseA[BankConnector]`		609k	1697k	22s	3.2s	CE
run	`InefficientRessourceUseB[BankConnector]`		614k	1705k	22s	31s	Instance
check	`InefficientRessourceUseB[BankConnector]`		611k	1702k	23s	3.3s	CE
	Temps						
run	`TotalTime[12]`		2329k	7908k	110s	22s	Instance
check	`TotalTime[12]`		2307k	7899k	114s	128s	-
run	`TotalTime[6]`		2329k	7908k	116s	12s	Instance
check	`TotalTime[6]`		2307k	7899k	112s	85s	CE
	Métiers						
check	`range[ReceiveOrder,0,1]`		609k	1697k	24s	20s	-
check	`range[ReceiveOrder,2,2]`		609k	1697k	34s	2.3s	CE
check	`relation_before[SendInvoice,MakePayment]`		614k	1698k	30s	3.1s	-
check	`relation_before[MakePayment,SendInvoice]`		614k	1698k	30s	3.1s	CE
check	`relation_excl[ReceiveOrder,CloseOrder]`		614k	1698k	25s	12.5s	-
check	`relation_excl[ShipOrder,AcceptPayment]`		614k	1698k	26s	13s	CE
check	`relation_between[MakePayment,SendInvoice,AcceptPayment]`		614k	1699k	25s	27s	-
check	`existence_data[OutputPinInvoice]`		616k	1718k	25s	7s	CE
check	`time_activity_before[ShipOrder,2]`		2329k	7908k	122s	171s	CE
check	`time_activity_before[ShipOrder,12]`		2329k	7908k	117s	209s	-
check	`time_data_between[OutputPinInvoice,1,4]`		2350k	7987k	112s	115s	CE
check	`time_data_between[OutputPinInvoice,1,10]`		2350k	7987k	114s	196s	-

Table 6.1 – Métrique de l'Alloy Analyzer pour analyser le procédé de la figure 6.1

Ressources. Le procédé ne souffre pas d'un manque de ressources. En revanche, il existe des exécutions telles que des ressources allouées aux procédés (`BankConnector`) ne sont pas toujours utilisées, bien qu'allouées sur le procédé ($\{Initial, ReceiveOrder, CloserOrder, Final\}$). De la même manière, la version plus stricte de la propriété vérifiant l'utilisation des ressources montre que la capacité totale d'utilisation de la ressource n'est pas toujours atteinte (`check InefficientRessourceUseB`). Ce résultat est attendu parce que que la version moins stricte (`InefficientRessourceUseA`) possédait déjà un contre-exemple. Par ailleurs, il existe des exécutions telles que ces deux propriétés soient valides (`run InefficientRessourceUseA` et `InefficientRessourceUseB`) comme l'exécution : $\{Initial, FillOrder, SendInvoice, MakePayment, AcceptPayment, ShipOrder, CloseOrder, Final\}$.

Temps. La propriété pour vérifier si le procédé termine à temps (`TotalTime`) est analysée avec 2 valeurs : 6 et 12 unités de temps. Les résultats montrent que la variante faible de ces propriétés est satisfaite : il est possible de terminer le procédé en moins de 6 ou 12 unités de temps (`run TotalTime[12]` et `TotalTime[6]`). En revanche, bien que le procédé puisse *toujours* (variante forte) se terminer en moins de 12 unités de temps (`check TotalTime[12]`), ce n'est pas vrai pour seulement 6 unités de temps (`check TotalTime[6]`).

Métiers. La première catégorie montre que l'action `ReceiveOrder` est toujours exécutée entre 0 et 1 fois, mais ne peut pas l'être 2 fois. La seconde catégorie explore différentes propriétés sur l'ordre d'exécution des actions. Par exemple, le résultat de la propriété `"relation_before[SendInvoice,MakePayment]"` montre que l'action `SendInvoice` est toujours exécutée avant `MakePayment`, et donc, de toute évi-

dence, son contraire `"relation_before[SendInvoice,MakePayment]"` exhibe un contre-exemple. La propriété de la troisième catégorie (`existence_data[OutputPinInvoice]`) montre que la donnée "invoice" n'est pas toujours disponible, comme le montre le contre-exemple : $\{Initial, ReceiveOrder, CloserOrder, Final\}$. Finalement, la dernière catégorie de propriété métiers vérifie certaines contraintes temporelles sur le procédé. Par exemple, l'action `ShipOrder` n'est pas toujours exécutée avant 2 unités de temps (`time_activity_before[ShipOrder,2]`, mais l'est toujours avant 12 unités (`time_activity_before[ShipOrder,12]`). De la même manière, les propriétés `"time_data_between[OutputPinInvoice,1,X]"` montrent que lorsque cette donnée est présente, elle ne l'est pas toujours entre 1 et 4 unités de temps, mais entre 1 et 10 unités de temps.

Globalement, la vérification des propriétés sans la sémantique du temps est plus rapide (flux de contrôle et de données). Ceci provient du fait que la profondeur sur laquelle le procédé est analysé est plus faible (27 contre 38 `States`). En effet, il n'est pas nécessaire de calculer les états intermédiaires pour faire progresser le temps dans le système. Ces propriétés sont toutes analysées autour de 20 secondes (environ 400k variables et 1100k clauses). Concernant l'ensemble des propriétés reposant sur la sémantique du temps, les états supplémentaires impliquent une augmentation du temps de vérification (plus ou moins 30 secondes, pour 620k variables et 1700k clauses).

En revanche, toutes les propriétés ayant recours à la proposition atomique comparant l'horloge globale (`State.gc`) à un entier (c'est-à-dire les propriétés `TotalTime` et `time_*`) ont une complexité beaucoup plus grande (2300k variables et 7900k clauses). La raison principale provient du besoin d'utiliser un *bitwidth* plus important pour encoder les entiers dans le système. Le *bitwidth* correspond aux bornes minimales et maximales utilisées pour encoder les entiers. L'horloge globale du système devant contenir le temps *total* écoulé depuis le début de l'exécution, il est nécessaire d'utiliser une borne plus grande pour "encoder" cet entier.

Lors de la transformation de la spécification Alloy en problème SAT, les entiers sont encodés sous forme booléenne. Une plus grande borne des entiers entraîne donc des formules beaucoup plus complexes. Plus particulièrement, l'utilisation d'opérations arithmétiques sur les entiers (telles que les additions utilisées pour incrémenter les horloges, ou l'ajout et suppression de jetons sur les noeuds et arcs) entraîne une complexité du système croissante lors de l'utilisation d'un plus grand *bidwidth*. En effet, les opérations sont effectuées bit à bit dans un problème SAT. Ainsi, les propriétés temps réel ont un temps de vérification supérieur dû à une longueur de trace et un *bitwidth* plus important pour prendre en compte l'horloge globale.

L'ensemble des modules de ALLOY4PV pour vérifier un procédé disponible dans l'annexe A est basé sur ce cas d'étude (pour les deux modules dynamiques `ProcessModel.als` et `Properties.als`).

6.2.2 Procédé d'un partenaire industriel

Dans le cadre du projet MERgE, nous avons travaillé sur des procédés de différents domaines venant de nos partenaires industriels : aérospatiale, automobile, système de contrôle industriel, et communications. Dans cette section, nous utilisons notre outil sur un de ces procédés. La figure 6.4 montre ce procédé modélisé avec les diagrammes d'activités UML par un de nos partenaires. Ce procédé représente une procédure de haut niveau pour délivrer des services au sein de l'entreprise. Pour des raisons de confidentialité, nous n'avons pas le droit de publier ce procédé dans ce document. Ainsi, nous avons rendu anonyme le procédé en remplaçant le nom des noeuds par les lettres de l'alphabet et nous ne donnons pas de détails sur leurs descriptions. Il est important de noter que le modèle de procédé qui nous a été fourni ne contient pas les informations organisationnelles telles que le temps alloué pour les actions et les ressources. Ce procédé contient

Type	Propriété	Semantic	Variables	Clauses	CNF	SAT	Résultat
			Flux de contrôle				
run	`Completion`	-	8590k	25528k	17min	13min	**Instance**
check	`Completion`	-	8552k	25526k	17min	12min	-
run	`CompletionClean`	-	8557k	25540k	18min	13min	**Instance**
check	`CompletionClean`	-	8590k	25566k	17min	15min	-
check	`DeadTransition[N]`	-	8590k	25528k	16min	13min	**CE**
check	`DeadTransition[S]`	-	8590k	25528k	17min	14min	**CE**
check	`DeadTransition[R]`	-	8590k	25528k	17min	13min	**CE**
			Flux de données				
run	`NoMissingData`	-	8564k	25551k	21min	16min	**Instance**
check	`NoMissingData`	-	8653k	25761k	22min	35min	-
run	`NoRedondantData`	-	8557k	25539k	16min	14min	**Instance**
check	`NoRedondantData`	-	8622k	25558k	16min	15min	**CE**

Table 6.2 – Métrique de l'Alloy Analyzer pour analyser le procédé de la figure 6.4

19 `Actions`, 30 `ObjectNodes` dont 28 `Pins` et 2 `ActivityParameterNodes`, 15 `ControlNodes` dont 3 `MergeNodes`, 3 `DecisionNodes`, 4 `ForkNodes` et 5 `JoinNodes` ; 14 `ObjectFlows`, 40 `ControlFlows`, et 4 `Swimlanes`. Les `Swimlanes` représentent une façon de partitionner le diagramme pour regrouper les actions effectuées par la même entité. Ces structures ne font pas partie du sous-ensemble de fUML (et donc de notre formalisation) du fait qu'elles ne possèdent pas de sémantique d'exécution : elles ajoutent une information sur le diagramme, mais n'influent pas sur l'exécution de celle-ci. Ainsi, le procédé contient (sans compter les swimlanes) 54 `ActivityEdges` et 64 `ActivityNodes` pour un total de 118 éléments UML dans le modèle.

Le tableau 6.2 présente le résultat pour l'évaluation du flux de contrôle et de données du procédé. La longueur de trace est calculée à 70 `States`. Les colonnes du tableau sont les mêmes que le tableau 6.1 présenté précédemment.

Flux de contrôle. Le procédé ne possède pas d'erreurs sur son flux de contrôle. Il peut toujours se terminer, et, quand il se termine, aucune autre action n'est toujours en cours d'exécution. De plus, les noeuds testés (`N`, `S` et `R`) ne sont pas morts : il existe toujours au moins une exécution de façon à ce que le noeud soit exécuté, tel qu'attesté par la présence d'un contre-exemple.

Flux de données. Le flux de données ne souffre pas de données manquantes. En revanche, la propriété vérifiant les données redondantes exhibe un contre-exemple. En effet, l'exécution peut se terminer, alors que des offres sont toujours présentes dans les flux de donnée `OutN` $\mapsto$ `ParamOutA` et `OutS` $\mapsto$ `ParamOutB`. La propriété "`run NoRedondantData`" démontre que ce problème n'est pas *toujours* présent. En fait, une condition de compétition (*race condition*) est présente dans le procédé : lorsque `N` et `S` sont exécutés, des offres sont présentes sur les arcs sortants de leurs pins de sortie. Cependant, il n'existe aucune restriction forçant les noeuds de sortie d'activité `ParamOutA` et `ParamOutB` à s'exécuter directement après l'exécution de `N` et `S`. Après leurs exécutions, 3 noeuds sont prêts à être exécutés : `JoinNS`, `ParamOutA` et `ParamOutB`. Si `JoinNS` est exécuté, les 3 noeuds qui peuvent être exécutés deviennent : `Final`, `ParamOutA` et `ParamOutB`. Ainsi, dans le cas où le prochain noeud exécuté est `Final`, les noeuds `ParamOutA` et `ParamOutB` ne sont pas exécutés, et l'exécution d'un noeud `ActivityFinalNode` (`Final`) termine l'exécution de l'activité, sans les données sur les noeuds de sortie d'activité. La propriété "`check NoRedondantData`" est donc détectée invalide du fait que des données ont été produites, mais non utilisées avant la fin de l'activité. Nous avons effectué des tests d'exécution en utilisant notre outil de débogage basé sur l'implémentation de référence de fUML [155]. Comme expliquée dans la section 3.1.3 (page 54), l'implémentation de référence utilise seulement un seul thread pour exécuter les modèles.

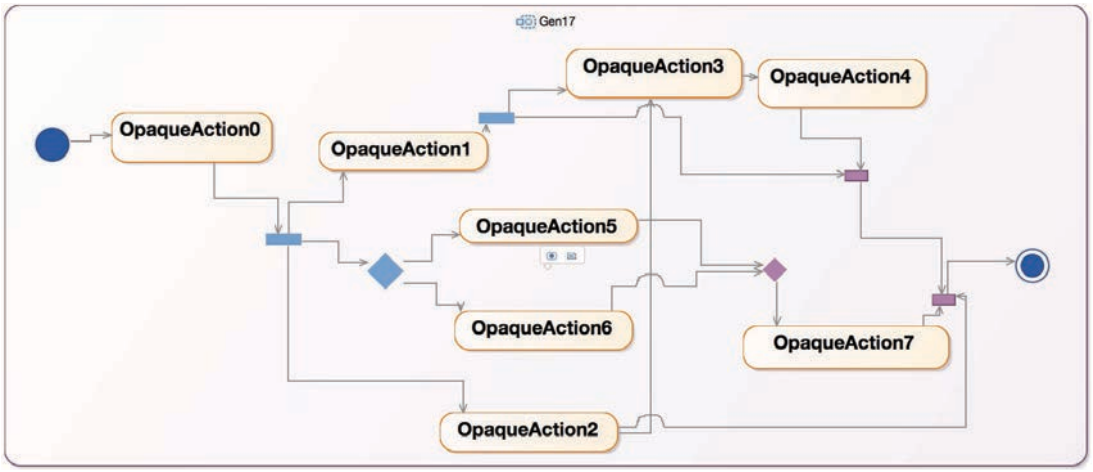


FIGURE 6.2 – Exemple d'un procédé généré de 35 éléments UML (19 `ControlFlow` et 16 `ActivityNode`) , en utilisant notre outil de génération aléatoire de procédés

> Lorsqu'une action est exécutée, les jetons sont directement propagés sur les arcs de sorties des pins en exécutant, si possible, le noeud de destination. Ainsi, l'implémentation de référence n'exhibe pas ce problème dû à son déterminisme venant de son unique thread. Une façon de corriger le modèle consiste à forcer l'exécution des noeuds `ParamOutA` et `ParamOutB` avant le noeud `Final` en ajoutant, par exemple, des flux de contrôle de `ParamOutA`/`ParamOutB` vers `Final`.

La taille du procédé étant beaucoup plus grande que celle du procédé analysé précédemment, l'analyse du procédé est de toute évidence beaucoup plus longue. La réduction de la spécification Alloy produit un problème SAT d'environ 8 millions de variables et 25 millions de clauses pour l'ensemble des propriétés. La vérification met en moyenne *environ* 30 minutes sur l'ensemble des propriétés, dont 16 minutes pour générer le problème CNF et 14 minutes pour le résoudre. La propriété `NoMissingData` prend plus de temps dû à la complexité beaucoup plus grande de la propriété. En effet, cette propriété utilise les prédicats *pNode* et *pAction* (page 63 du chapitre 3) inspectant les jetons des flux de contrôle et de données entrant ainsi que les multiplicités des noeuds de données.

La prochaine section sur la vérification de procédés générés discute plus en détail du temps d'analyse par rapport aux tailles des procédés.

6.2.3 Procédés générés aléatoirement

Dans cette section, nous évaluons notre approche sur des procédés que nous avons générés avec l'outil de génération de procédés présenté dans le chapitre 5.

Nous avons généré des procédés aléatoires entre 10 et 100 éléments UML (noeuds et arcs). Ces procédés contiennent seulement du flux de contrôle, sans aucune boucle, et contiennent du routage séquentiel (`ControlFlow`), des actions à effectuer (`OpaqueAction`), du routage parallèle (`ForkNode`), des synchronisations (`JoinNode`), du routage conditionnel (`DecisionNode`) et des fusions de flux alternatifs (`MergeNode`). Bien que ces procédés aient été générés, ils ressemblent grandement à des procédés qui auraient pu être modélisés par un humain. La figure 6.2 montre un exemple de procédé généré. Chacun de ces procédés est vérifié sur la propriété `Completion`, et, seuls les modèles sans contre-exemple sont retenus. En effet, seul le temps de vérification le

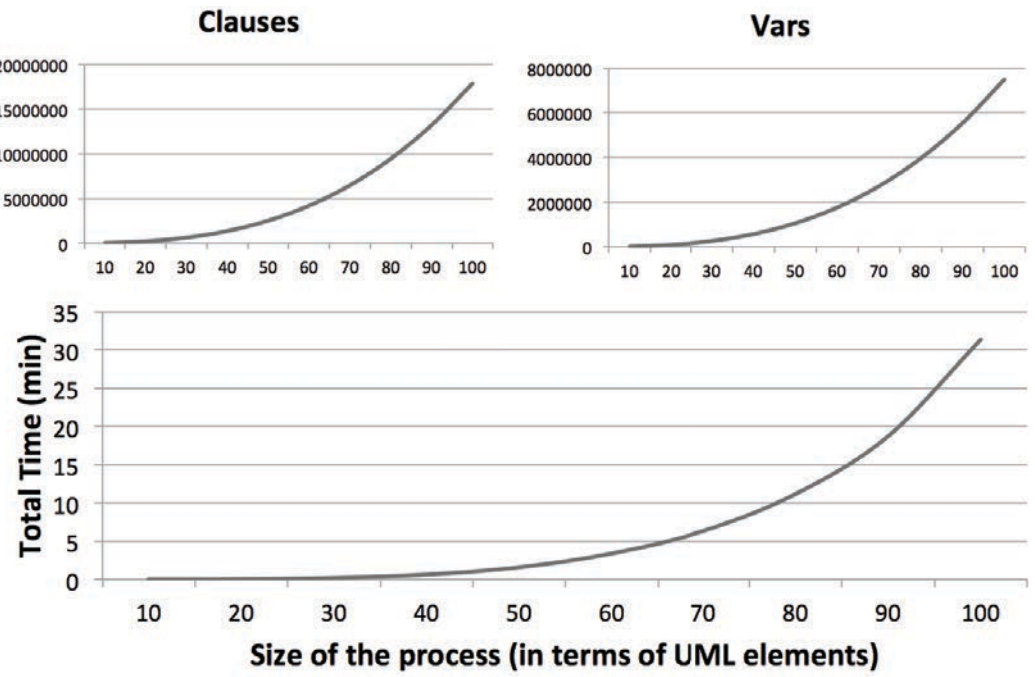


FIGURE 6.3 – Nombre de variables, de clauses, et temps total (génération CNF et résolution SAT) pour vérifier la propriété `check Completion`, selon la taille du procédé

plus large a de l'intérêt (la présence d'un contre-exemple étant toujours plus rapide à trouver que son absence).

La figure 6.3 montre les courbes représentant le nombre de variables, clauses, et le temps total pour vérifier la propriété, en fonction de la taille du procédé généré. Les résultats montrent que le temps de vérification est raisonnable par rapport à la complexité des procédés générés. Les courbes sont exponentielles du fait que les problèmes SAT appartiennent à la catégorie des problèmes NP-complets. Le problème SAT généré pour les modèles de 100 éléments contient presque 18 millions de clauses et 8 millions de variables, et les vérifier prend en moyenne 31 minutes ce qui met en évidence le fait que nos problèmes SAT appartiennent à une catégorie facile à vérifier.

6.3 Conclusion

L'évaluation met en évidence l'accomplissement des objectifs définis dans l'introduction. La formalisation, son implémentation, et l'outil associé permettent de vérifier toutes les perspectives des procédés métiers de manière automatique.

Dans l'ensemble, les temps de vérification sont encourageants en partant du principe que nous n'avons pas appliqué d'optimisations particulières telles que des techniques de *slicing* [276] ou de réduction de graphe [181]. Nous discutons de façon plus détaillée de l'ensemble des techniques d'optimisation applicables dans les perspectives de notre chapitre conclusion.

Un des principaux inconvénients de l'Alloy Analyzer provient de l'impossibilité de spécifier le *bitwidth* pour *chacun* des entiers déclarés. En effet, le *bitwidth* s'applique à l'ensemble des entiers du système. Ainsi, lors de l'analyse des propriétés temps réel, il est nécessaire de spécifier un *bitwidth* plus important pour gérer l'horloge globale. Cependant, tous les entiers, incluant

ceux spécifiant le nombre de jetons que possèdent les noeuds et arcs, sont impactés. Ainsi, toutes les opérations d'ajout et de suppression de jetons entre les différents éléments du diagramme d'activités sont encodées de manières plus complexes dans le problème SAT. La possibilité de spécifier le *bitwidth* pour chacun des entiers permettrait d'améliorer la vitesse d'analyses des propriétés temps réel de manière significative.

Le prochain chapitre conclut cette thèse en résumant nos contributions et en présentant les perspectives à court, moyen et long termes.

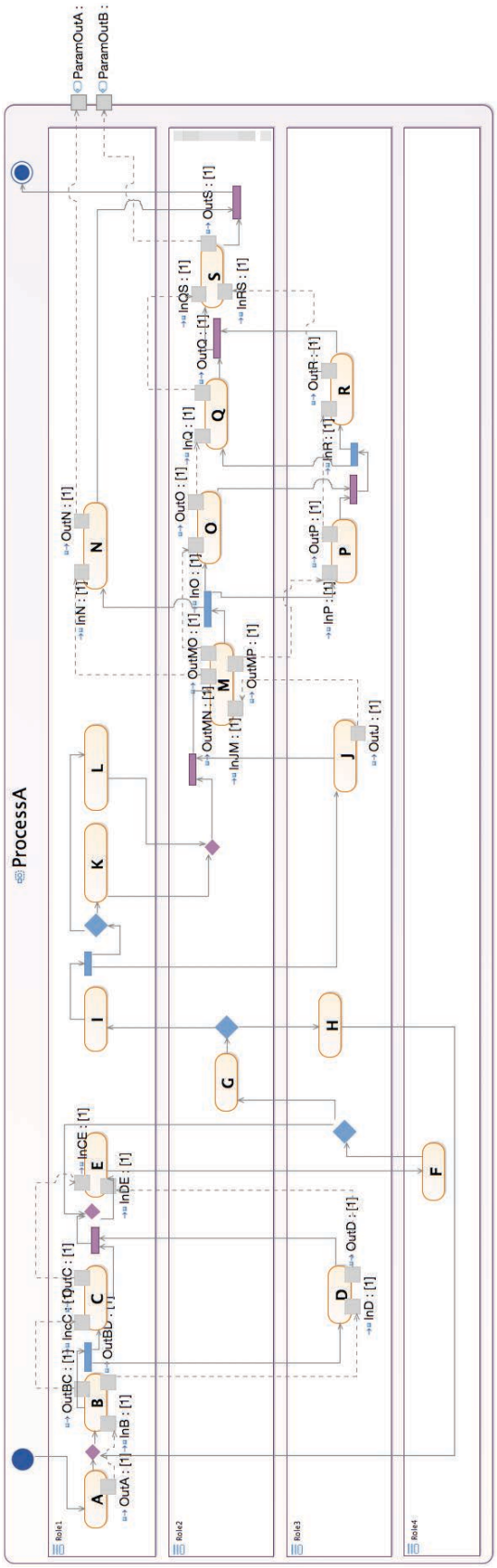


FIGURE 6.4 – Procédé d'un partenaire du projet MERgE modélisé avec Obeo Designer

Chapitre 7

Conclusion

Dans cette thèse, nous avons présenté notre framework pour vérifier de manière automatique toutes les perspectives d'un modèle de procédé, en se basant sur des techniques de *model-checking*.

7.1 Contributions

Nos contributions sont constituées des points suivants :

Identification des propriétés. Nous avons identifié les propriétés importantes à vérifier afin de valider la conformité d'un modèle de procédé. Nous avons effectué une étude de la littérature dans les différents domaines de procédé métiers afin d'identifier et de catégoriser les propriétés essentielles à vérifier. Nous avons classé ces propriétés en 5 catégories : **(1) "flux de contrôle"** pour s'assurer que le procédé n'a pas d'erreurs sur ses structures de flux de contrôle telles que les impossibilités de complétion et autres transitions mortes ; **(2) "flux de données"** pour s'assurer que le procédé n'a pas d'erreurs sur ses structures de flux de données telles qu'un manque de données pour effectuer une activité ; **(3) "ressources"** pour s'assurer que les ressources sont correctement utilisées sur le procédé ; **(4) "temps"** pour s'assurer que les contraintes temporelles du procédé sont respectées ; et **(5) "métiers"** pour s'assurer que des propriétés contextuelles et spécifiques à un procédé donné soient respectées.

Librairie de propriétés métiers. Nous avons défini une librairie paramétrable de propriétés afin de pouvoir exprimer de manière simplifiée des propriétés métiers sur un procédé, sans avoir besoin de manipuler une logique temporelle.

Formalisation de fUML. Nous avons formalisé fUML pour la vérification de procédés en prenant en compte les différentes perspectives (flux de contrôle, données, ressources et temps). Dans la littérature, les formalisations existantes de la sémantique des diagrammes d'activités UML ne sont pas basées sur la sémantique officielle fUML. Dans la littérature, les auteurs ont dû faire beaucoup d'hypothèses sur la spécification faite en langage naturel dans le standard UML. Notre modèle formel de fUML constitue le premier travail pour vérifier des procédés basés sur les diagrammes d'activités UML associées à la sémantique fUML. Cette formalisation a donné lieu à une publication dans la *"International Conference on Advanced Information Systems Engineering"* (CAiSE'14) à Thessalonique en Grèce [153].

Formalisation des propriétés. Nous avons formalisé l'ensemble des propriétés importantes à vérifier pour valider un procédé, mais aussi une librairie paramétrable de propriétés métiers en logique PLTL. Le modèle formel de fUML défini est assez expressif pour supporter l'ensemble de ces propriétés.

Alloy4PV. Nous avons implémenté la formalisation de fUML ainsi que les propriétés en utilisant le langage Alloy. L'implémentation est basée sur différents modules Alloy faisant une distinction entre tous les concepts formalisés et vérifiés. En effet, contrairement à beaucoup de langages pris en entrée par un *model-checker*, le langage Alloy a permis d'implémenter la formalisation de manière modulaire en faisant une séparation entre la syntaxe, la sémantique, le modèle de procédé, et les propriétés à vérifier. Bien que Alloy effectue du *model-checking* borné, nous sommes capables de calculer la longueur de trace nécessaire pour analyser toutes les exécutions du modèle. A titre d'exemple, l'ensemble des modules de ALLOY4PV (`Syntax.als`, `Semantic.als` et `Library.als`), dont ceux générés dynamiquement pour le premier cas d'étude de l'évaluation (`Process.als` et `Properties.als`), fait environ 1100 lignes de code Alloy.

Outil graphique intégré à Eclipse. Nous avons créé un outil graphique afin d'automatiser tout le processus de vérification d'un procédé basé sur le framework ALLOY4PV. L'outil, basé sur Eclipse EMF, Obeo Designer et l'API Alloy, permet de supporter la chaîne complète de la spécification du modèle de procédé à la production des résultats de vérification. Les résultats sont affichés graphiquement à l'utilisateur en coloriant sur le modèle le chemin satisfaisant la propriété ou le contre-exemple. Il permet aussi de spécifier des propriétés métiers, sous forme d'annotation, en utilisant la librairie paramétrable de propriétés métiers que nous avons définie. Un des points importants concernant l'outil est qu'il ne demande aucune connaissance d'un formalisme particulier pour exprimer le système ou la propriété afin d'être utilisé. Tout est automatisé à travers l'interface graphique. Finalement, il guide l'utilisateur lors de la vérification du procédé en analysant les différentes perspectives du procédé lors de différentes étapes. L'outil est composé d'environ 10000 lignes de code Java qui ont été intégrées et déployées dans la platforme MERgE. Il a été utilisé au sein du projet afin de vérifier l'ensemble des procédés des différents partenaires industriels. Le framework ALLOY4PV et son outil ont donné lieu à une publication dans la *"European Conference on Modelling Foundations and Applications"* à York au Royaume-Uni [152].

Une approche de génération de procédés. Nous avons proposé une approche et un outil basés sur un algorithme génétique multi objectifs pour la génération de modèles de procédés. Le procédé est généré au travers d'une séquence d'opérations de haut niveau (*change patterns*) inspirée par la façon dont un modélisateur aurait pu modéliser un procédé, favorisant le réalisme des procédés générés. À notre connaissance, c'est une des premières approches se confrontant à la génération de modèles comportementaux, et non seulement syntaxiques. L'outil implémenté contient presque 8000 lignes de code Java et permet de générer des instances de modèles de procédés dans le format UML AD ou BPMN, selon les objectifs définis par l'utilisateur. Cette approche a donné lieu à une publication dans la *"International Conference on Software and Systems Process"* (ICSSP'13) à San Francisco aux États-Unis.

Travaux complémentaires. Parallèlement à nos travaux sur la vérification de procédés métiers, nous avons non seulement proposé une extension au standard fUML pour pallier certaines de ses lacunes, mais aussi développé une approche permettant la planification de procédés *déclaratifs*.

Extension du standard fUML. Dans le cadre du projet MERgE, il était aussi nécessaire de développer un outil permettant d'*exécuter* des procédés logiciels modélisés avec UML AD. La façon la plus naturelle pour exécuter ces procédés consiste à utiliser la machine virtuelle

fUML définie par le standard [189]. Cependant, le modèle d'exécution utilisé par la machine virtuelle n'est pas adapté pour exécuter des procédés. En effet, le modèle d'exécution est défini pour exécuter les modèles de manière "atomique" et ne remplit pas les conditions pour "enacter" le procédé. Il n'est pas possible de *contrôler* l'exécution ni de l'*observer*. Nous avons proposé une extension du standard afin de pallier ces inconvénients et nous avons développé un débogueur de modèle UML en utilisant cette extension. Ce débogueur étend l'état de l'art concernant les fonctionnalités de débogage des autres approches [139, 62, 92, 211], mais surtout propose les fonctionnalités essentielles permettant d'"enacter" un procédé (mettre en pause, reprendre, exécuter pas à pas, revenir en arrière (rollback), inspecter et modifier les données instanciées, etc.). Cette extension du standard pour exécuter et déboguer un modèle UML a donné lieu à une publication dans la conférence *"Symposium on Applied Computing"* (SAC'13) à Coimbra au Portugal [155].

Planification de procédés déclaratifs. Récemment, la modélisation de procédé déclaratif a gagné un grand intérêt dans le monde de la recherche et de l'industrie [257]. Ces langages sont basés sur le fait de déclarer les différents éléments du procédé (activités, ressources, données, etc.) et d'appliquer des contraintes sur ces éléments qui doivent être respectées pendant l'exécution du procédé. Dans ce contexte, l'exécution du procédé est dirigée par les contraintes : tout ce qui ne viole pas une contrainte peut être exécuté. L'engouement pour les langages déclaratifs pour la modélisation de procédés provient de leur flexibilité. En effet, il a été prouvé que ces langages étaient appropriés pour accomplir un haut degré de flexibilité du fait qu'ils ne nécessitent pas une spécification explicite de chacune des exécutions possibles, mais permettent une spécification implicite de ces alternatives [198]. La différence clef entre un modèle de procédé déclaratif et impératif est que, dans le premier, tout est autorisé, à moins qu'explicitement interdit, tandis que dans le second, tout est interdit, à moins qu'explicitement spécifié. Bien que ces langages permettent un haut degré de flexibilité, cette liberté mène à des problèmes de compréhension : avoir une représentation mentale des chemins d'exécution possible du modèle devient très rapidement beaucoup trop compliqué pour être géré par un humain, au fur et à mesure que le nombre de contraintes augmente sur le modèle [198]. Nous avons proposé une nouvelle approche formelle pour générer automatiquement des plans d'exécution d'un procédé déclaratif. Au moment de la modélisation, les plans d'exécutions permettent d'augmenter la compréhension du modèle en fournissant une expérience directe avec celui-ci. Au moment de l'exécution, un module de planification est primordial pour donner des traces d'exécution satisfaisant les objectifs désirés. Ce travail a donné lieu à une publication dans la conférence *"Symposium on Applied Computing"* (SAC'14) à Gyeongju en Corée [154].

Ces deux contributions n'étant pas directement liées à la *vérification* de procédés, nous ne les avons pas présentées en détail dans ce document.

En plus d'une collaboration fructueuse dans le cadre du projet européen MERgE incluant différents partenaires académiques et industriels, l'ensemble de nos travaux a donc donné lieu à 5 publications dans des conférences internationales incluant des conférences de premier plan.

7.2 Perspectives

Ces travaux ouvrent différentes perspectives à court, moyen et long termes.

7.2.1 À court terme

Sous-ensemble de fUML formalisé. Un des points importants que nous souhaitons améliorer concerne le sous-ensemble de fUML formalisé. Bien que la formalisation couvre les points les plus importants de UML AD pour la définition d'un procédé, certains éléments tels que les `AcceptEventAction` et `SendSignalAction` (pour gérer les événements) et les `StructuredActivity` (noeud de haut niveau pour représenter des boucles et autres noeuds conditionnels...) pourraient être intéressants à inclure dans un premier temps.

Un des points non traités de la formalisation concerne l'aspect contenu des données (*data-aware*). Actuellement, le contenu des jetons à l'intérieur des `ObjectNode`s ne sont pas pris en compte. Cela empêche, par exemple, d'exprimer des gardes (*guard*) sur les arcs pour déterminer si l'arc peut être traversé. Certaines formalisations ont pris ces concepts en compte [79]. Cependant, beaucoup de simplifications ont été faites par rapport à la façon dont fUML opère et seulement les entiers ont été considérés.

Dans fUML, chaque jeton peut avoir un type primitif (*integer, string, natural, boolean*) ou un type plus complexe (un `Classifier`) défini dans un diagramme de classes, représentant ainsi une instance d'objet. Ces jetons sont manipulés en utilisant les noeuds d'actions définies dans le package `IntermediateActions`. Ce package définit les actions classiques pour créer, lire, supprimer ou modifier les jetons pendant l'exécution au sein d'un AD. Formaliser l'ensemble de ces concepts est une tâche non triviale et pourrait être aussi intéressant à inclure dans un second temps.

7.2.2 À moyen terme

Amélioration de l'outil de génération de procédés. Il pourrait être intéressant de définir un ensemble de *change patterns* prenant en compte le flux de données, et d'étendre l'approche pour générer les informations organisationnelles associées au procédé. Ceci permettrait d'avoir un échantillon de modèle de procédé plus complet afin d'avoir une évaluation plus poussée de notre approche de vérification.

Par ailleurs, il serait aussi intéressant d'étendre l'approche à tous les types de modèles *comportementaux*. Un procédé étant seulement *un* type de modèle comportemental, une approche de plus haut niveau et générique, basée sur le même type d'algorithme génétique, permettrait un champ d'application beaucoup plus large, et également de toucher des communautés autres que celles liées aux procédés métiers.

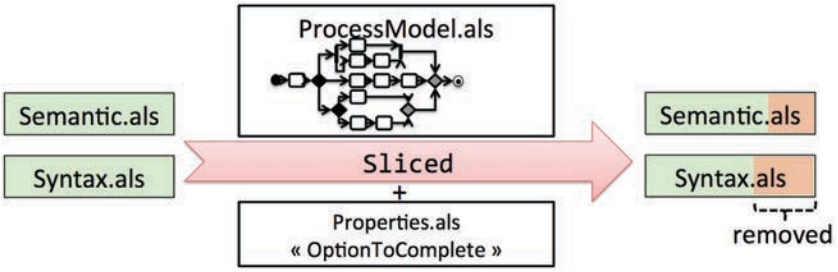
Module de planification. Actuellement, les propriétés *faibles* correspondent à des propriétés de satisfiabilité : trouver une exécution telle qu'une condition (la propriété) est satisfaite. Faire de la planification comme un problème de satisfiabilité n'est pas nouveau [136]. Cependant, dans notre cas, définir un module de planification basé sur notre framework ALLOY4PV correspondrait à générer *un* ou *plusieurs* plans qui satisfont le ou les objectifs définis (ex. `ActiviteA` et `ActiviteB` sont exécutées), sans optimisation quelconque. Moolloy [127], une extension d'Alloy implémentant un algorithme d'amélioration guidée permet de résoudre des problèmes d'optimisation multi objectifs (MOO, pour *multi-objective optimization*). Moolloy fonctionne en ajustant à maintes reprises le problème SAT pour trouver des solutions de mieux en mieux jusqu'à ce qu'aucune meilleure solution n'existe, explorant ainsi les frontières du Pareto Front. Moolloy étend légèrement la syntaxe du langage Alloy en permettant de définir différentes fonctions à optimiser (minimiser ou maximiser). Ainsi, en utilisant Moolloy, il serait possible de générer le plan *optimal* pour les objectifs donnés. Des exemples de fonctions à optimiser pourraient être : la minimisation du nombre d'états nécessaires pour atteindre les objectifs, la minimisation du temps (horloge globale) pour atteindre les objectifs, la maximisation d'utilisation des ressources, etc. Ainsi, en intégrant

Moolloy à notre prototype, et en définissant ces fonctions à optimiser, il serait, dès lors, possible de générer des plans optimisés en respectant la sémantique de fUML, pour atteindre des objectifs définis. Les premiers tests effectués "manuellement" ont déjà validé la possibilité d'intégration de Moolloy dans nos outils.

7.2.3 À long terme

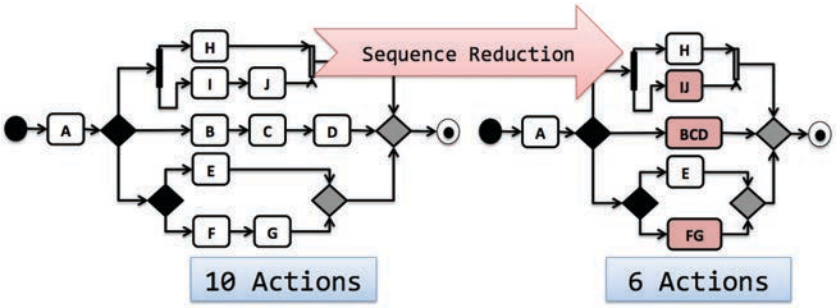
Techniques d'optimisation. Bien que les outils développés et l'évaluation aient prouvé la faisabilité de notre approche, il reste beaucoup de points qui pourraient être optimisés :

1. Utiliser des techniques de *slicing* [276], c'est-à-dire en générant partiellement les modules Alloy selon les besoins de la propriété :



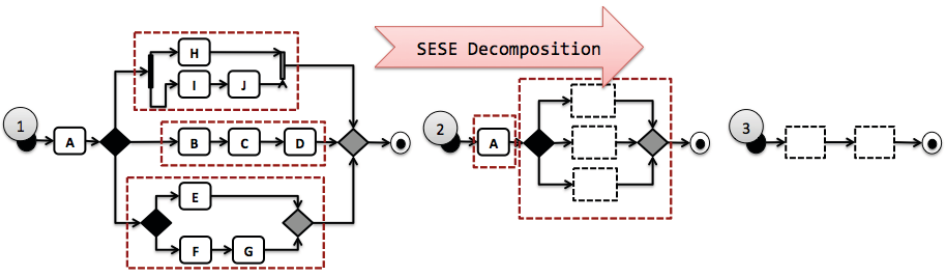
   En réalité, nous avons déjà appliqué cette technique manuellement en implémentant les modules Alloy avec et sans les horloges globales et locales pour évaluer plus efficacement les propriétés ne nécessitant pas le temps réel. Cependant, l'idée, ici, est d'aller plus loin et d'être encore plus modulaire.

2. Utiliser des règles de réduction de graphes pour réduire la taille du procédé en s'inspirant des règles de réduction utilisées sur les réseaux de Petri [181] :



   De toute évidence, les règles de réductions ne peuvent pas être appliquées indifféremment des propriétés analysées. Par rapport à cet exemple, la réduction appliquée ne pose pas de problème pour vérifier si le procédé est sujet aux interblocages, mais aurait trop abstrait le procédé dans le cas où l'on aurait souhaité vérifier $relation\_before(C, F)$. Ainsi, les règles de réduction appliquées dépendent en partie des propositions atomiques de la propriété analysée. Comme expliqué dans l'évaluation, analyser des procédés de taille réduite permet un gain de temps de vérification non négligeable.

3. Utiliser des techniques de décomposition du procédé, telles que la décomposition SESE (Single Entry Single Exit) mise en place par Vanhatalo *et al.* dans [269] :



Encore une fois, les décompositions doivent être adaptées aux propriétés analysées selon ses propositions atomiques. Dans la littérature, cette approche a, pour le moment, seulement été mise en place pour l'analyse des propriétés de flux de contrôle.

Bibliographie

[1] Islam Abdelhalim, Steve Schneider, and Helen Treharne. Towards a practical approach to check uml/fuml models consistency using csp. In *Formal Methods and Software Engineering*, pages 33–48. Springer, 2011. 36, 39, 42

[2] Islam Abdelhalim, Steve Schneider, and Helen Treharne. An optimization approach for effective formalized fuml model checking. In *Software Engineering and Formal Methods*, pages 248–262. Springer, 2012. 36, 39, 42

[3] Islam Abdelhalim, Steve Schneider, and Helen Treharne. An integrated framework for checking the behaviour of fuml models using csp. *International Journal on Software Tools for Technology Transfer*, 15(4) :375–396, 2013. 36, 39, 42

[4] Islam Abdelhalim, James Sharp, Steve Schneider, and Helen Treharne. Formal verification of tokeneer behaviours modelled in fuml using csp. In *Formal Methods and Software Engineering*, pages 371–387. Springer, 2010. 36, 39, 42

[5] J-R Abrial, Matthew KO Lee, DS Neilson, PN Scharbach, and Ib Holm Sørensen. The b-method. In *VDM'91 Formal Software Development Methods*, pages 398–405. Springer, 1991. 19

[6] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, et al. The key tool. *Software & Systems Modeling*, 4(1) :32–54, 2005. 89

[7] DH Akehurst. Validating bpel specifications using ocl. 2004. 22

[8] Rajeev Alur, Thomas A Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5) :672–713, 2002. 36

[9] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy : A challenging model transformation. In *Model Driven Engineering Languages and Systems*, pages 436–450. Springer, 2007. 89

[10] Pallas Athena. Protos user manual. *Pallas Athena BV, Plasmolen, The Netherlands*, 1997. 32

[11] Darren C Atkinson, Daniel C Weeks, and John Noll. Tool support for iterative software process modeling. *Information and Software Technology*, 49(5) :493–514, 2007. 20

[12] Ahmed Awad. Bpmn-q : A language to query business processes. In *EMISA*, volume 119, pages 115–128, 2007. 5

[13] Ahmed Awad, Gero Decker, and Mathias Weske. Efficient compliance checking using bpmn-q and temporal logic. In *Business Process Management*, pages 326–341. Springer, 2008. 5, 32, 41, 43, 46

[14] Ahmed Awad, Matthias Weidlich, and Mathias Weske. Specification, verification and explanation of violation for data aware compliance rules. In *Service-Oriented Computing*, pages 500–515. Springer, 2009. 5, 32, 41, 43, 46

[15] Thomas Back. *Evolutionary algorithms in theory and practice*. Oxford Univ. Press, 1996. 117

[16] Jos CM Baeten, Twan Basten, Twan Basten, and MA Reniers. *Process algebra : equational theories of communicating processes*, volume 50. Cambridge university press, 2010. 5, 29

[17] AA Basten. In terms of nets : system design with petri nets and process algebra. 1998. 29, 36

[18] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor : A successful application of b in a large project. In *FM'99—Formal Methods*, pages 369–387. Springer, 1999. 19

[19] Reda Bendraou and M-P Gervais. A framework for classifying and comparing process technology domains. In *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, pages 5–5. IEEE, 2007. 4

[20] Reda Bendraou, Marie-Pierre Gervais, and Xavier Blanc. Uml4spm : A uml2. 0-based metamodel for software process modelling. In *Model Driven Engineering Languages and Systems*, pages 17–38. Springer, 2005. 6, 17, 27, 58

[21] Reda Bendraou, J Jezequel, M-P Gervais, and Xavier Blanc. A comparison of six uml-based languages for software process modeling. *Software Engineering, IEEE Transactions on*, 36(5) :662–675, 2010. 6, 18

[22] Jan A Bergstra, Alban Ponse, and Scott A Smolka. *Handbook of process algebra*. Elsevier, 2001. 35

[23] Gérard Berthelot. Transformations and decompositions of nets. In *Petri Nets : Central models and their properties*, pages 359–376. Springer, 1987. 34

[24] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE transactions on software engineering*, 17(3) :259–273, 1991. 34

[25] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999. 58, 71, 95, 96, 128

[26] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58 :117–148, 2003. 101

[27] Jonathan Billington, Søren Christensen, Kees Van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. *The Petri net markup language : concepts, technology, and tools*. Springer, 2003. 32

[28] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Computer Networks and ISDN systems*, 14(1) :25–59, 1987. 35

[29] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory with applications*, volume 6. Macmillan London, 1976. 15, 58

[30] Grady Booch. *Object Oriented Analysis & Design with Application*. Pearson Education India, 2006. 15

[31] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based test generation for model transformations : an algorithm and a tool. In *Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on*, pages 85–94. IEEE, 2006. 9, 112, 124

[32] Aaron G Cass, AS Lerner, Eric K McCall, Leon J Osterweil, Stanley M Sutton, and Alexander Wise. Little-jil/juliette : a process definition language and interpreter. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 754–757. IEEE, 2000. 37

[33] Saoussen Cheikhrouhou, Slim Kallel, Nawal Guermouche, Mohamed Jmaiel, et al. A survey on time-aware business process modeling. In *Proceedings of the International Conference on Enterprise Information Systems (ICEIS)*, 2013. 24

[34] Bin Chen, George S Avrunin, Elizabeth A Henneman, Lori A Clarke, Leon J Osterweil, and Philip L Henneman. Analyzing medical processes. In *Proceedings of the 30th international conference on Software engineering*, pages 623–632. ACM, 2008. 2, 37, 42

[35] Mingsong Chen, Prabhat Mishra, and Dhrubajyoti Kalita. Coverage-driven automatic test generation for uml activity diagrams. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 139–142. ACM, 2008. 38

[36] Mary Beth Chrissis, Mike Konrad, and Sandy Shrum. *CMMI Guidlines for Process Integration and Product Improvement*. Addison-Wesley Longman Publishing Co., Inc., 2003. 2, 3

[37] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2 : An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002. 37

[38] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv : A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499. Springer, 1999. 61

[39] Edmund M Clarke and I Anca Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 428–437. Springer, 1989. 30

[40] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite state concurrent system using temporal logic specifications : a practical approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126. ACM, 1983. 4, 19

[41] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2) :244–263, 1986. 5, 31

[42] Edmund M Clarke, Orna Grumberg, and Doron A Peled. *Model checking*. MIT press, 1999. 4, 19, 37, 58, 61

[43] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification*, pages 1–30. Springer, 2012. 20

[44] Michelle L Crane and Juergen Dingel. On the semantics of uml state machines : Categorization and comparision. In *In Technical Report 2005-501, School of Computing, Queen's*. Citeseer, 2005. 39

[45] Michelle L Crane and Juergen Dingel. Towards a uml virtual machine : implementing an interpreter for uml 2 actions and activities. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research : meeting of minds*, page 8. ACM, 2008. 38, 39

[46] Gianpaolo Cugola. Tolerating deviations in process support systems via flexible enactment of process models. *Software Engineering, IEEE Transactions on*, 24(11) :982–1001, 1998. 2

[47] Alcino Cunha. Bounded model checking of temporal formulas with alloy. *arXiv preprint arXiv :1207.2746*, 2012. 8, 88, 89, 95, 128

[48] David Cyrluk, Sreeranga Rajan, Natarajan Shankar, and Mandayam K Srivas. Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design*, pages 203–222. Springer, 1995. 19

[49] Marcos Aurelio Almeida da Silva, Xavier Blanc, and Reda Bendraou. Deviation management during process execution. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 528–531. IEEE, 2011. 2

[50] Peter Dadam and Manfred Reichert. The adept project : a decade of research and development for robust and flexible process support. *Computer Science-Research and Development*, 23(2) :81–97, 2009. 2

[51] Thomas H Davenport. *Process innovation : reengineering work through information technology*. Harvard Business Press, 2013. 13, 14

[52] Thomas H Davenport and James E Short. Information technology and business process redesign. *Operations management : critical perspectives on business and management*, 1 :1–27, 2003. 3

[53] L. Davis et al. *Handbook of genetic algorithms*, volume 115. Van Nostrand Reinhold New York, 1991. 114

[54] Leonardo De Moura, Sam Owre, Harald Rueß, John Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. Sal 2. In *Computer aided verification*, pages 496–500. Springer, 2004. 37

[55] Birgit Demuth. The dresden ocl toolkit and its role in information systems development. In *Proc. of the 13th International Conference on Information Systems Development (ISD'2004)*, 2004. 7

[56] Greg Dennis, Robert Seater, Derek Rayside, and Daniel Jackson. Automating commutativity analysis at the design level. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 165–174. ACM, 2004. 8

[57] Remco M Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Information and Software Technology*, 50(12) :1281–1294, 2008. 6, 32, 41

[58] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. *Notes on structured programming*. Technological University Eindhoven Netherlands, 1970. 38

[59] David L Dill, Andreas J Drexler, Alan J Hu, and C Han Yang. Protocol verification as a hardware design aid. In *ICCD*, volume 92, pages 522–525. Citeseer, 1992. 61

[60] Wei Dong, Ji Wang, Xuan Qi, and Zhi-Chang Qi. Model checking uml statecharts. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 363–370. IEEE, 2001. 38

[61] Yang Dong and Zhang ShenSheng. Using π-calculus to formalize uml activity diagram for business process modeling. In *Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the*, pages 47–54. IEEE, 2003. 39

[62] D. Dotan and A. Kirshin. Debugging and testing behavioral UML models. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 838–839. ACM, 2007. 141

[63] Doron Drusinsky. Visual formal specification using (n) tlcharts : Statechart automata with temporal logic and natural language conditioned transitions. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 268. IEEE, 2004. 11, 44, 46

[64] Marlon Dumas and Arthur HM Ter Hofstede. Uml activity diagrams as a workflow specification language. In *"UML" 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 76–90. Springer, 2001. 6

[65] Marlon Dumas, Wil M Van der Aalst, and Arthur H Ter Hofstede. *Process-aware information systems : bridging people and software through process technology*. Wiley-Interscience, 2005. 11, 2

[66] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420. IEEE, 1999. 44

[67] Matthew B Dwyer, Lori A Clarke, Jamieson M Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(4) :359–430, 2004. 38

[68] Daniel W. Dyer. Watchmaker framework for evolutionary computation. http://watchmaker.uncommons.org/. 119

[69] Johann Eder, Euthimios Panagos, Heinz Pozewaunig, and Michael Rabinovich. Time management in workflow systems. In *BIS'99*, pages 265–280. Springer, 1999. 2

[70] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004. 82

[71] Niklas Een and Niklas Sörensson. Minisat : A sat solver with conflict-clause minimization. *Sat*, 5, 2005. 129

[72] Henk Eertink, Wil Janssen, Paul Oude Luttighuis, Wouter Teeuw, and Chris Vissers. A business process design language. In *FM'99—Formal Methods*, pages 76–95. Springer, 1999. 37

[73] Alexander Egyed. Instant consistency checking for the uml. In *Proceedings of the 28th international conference on Software engineering*, pages 381–390. ACM, 2006. 22

[74] Karsten Ehrig, Jochen M Küster, Gabriele Taentzer, and Jessica Winkelmann. Generating instance models from meta models. In *Formal Methods for Open Object-Based Distributed Systems*, pages 156–170. Springer, 2006. 9, 112, 124

[75] Ralf Ellner, Samir Al-Hilank, Johannes Drexler, Martin Jung, Detlef Kips, and Michael Philippsen. espem–a spem extension for enactable behavior modeling. In *Modelling Foundations and Applications*, pages 116–131. Springer, 2010. 6, 17

[76] E Allen Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, 995 :1072, 1990. 5, 70

[77] Wolfgang Emmerich and Volker Gruhn. Funsoft nets : a petri-net based software process modeling language. In *Proceedings of the 6th international workshop on Software specification and design*, pages 175–184. IEEE Computer Society Press, 1991. 32

[78] Hendrik Eshuis. Semantics and verification of uml activity diagrams for workflow modelling. 2002. 5, 37, 40, 42

[79] Rik Eshuis. Symbolic model checking of uml activity diagrams. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1) :1–38, 2006. 39, 102, 142

[80] Dirk Fahland. Translating uml2 activity diagrams to petri nets, 2008. 33, 39, 41

[81] Dirk Fahland, Cédric Favre, Barbara Jobstmann, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Instantaneous soundness checking of industrial business process models. In *Business Process Management*, pages 278–293. Springer, 2009. 33, 119

[82] Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. Specification and validation of the business process execution language for web services. In *Abstract State Machines 2004. Advances in Theory and Practice*, pages 78–94. Springer, 2004. 29

[83] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem-Paul de Roever. 29 new unclarities in the semantics of uml 2.0 state machines. In *Formal Methods and Software Engineering*, pages 52–65. Springer, 2005. 49

[84] Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Multi Agent Systems, 1998. Proceedings. International Conference on*, pages 128–135. IEEE, 1998. 2

[85] Alexander Foerster, Gregor Engels, and Tim Schattkowsky. Activity diagram patterns for modeling quality constraints in business processes. In *Model Driven Engineering Languages and Systems*, pages 2–16. Springer, 2005. 43, 46

[86] Alexander Forster, Gregor Engels, Tim Schattkowsky, and Ragnhild Van Der Straeten. A pattern-driven development process for quality standard-conforming business process models. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 135–142. IEEE, 2006. 43, 46

[87] Alexander Forster, Gregor Engels, Tim Schattkowsky, and Ragnhild Van Der Straeten. Verification of business process quality constraints based on visual process patterns. In *Theoretical Aspects of Software Engineering, 2007. TASE'07. First Joint IEEE/IFIP Symposium on*, pages 197–208. IEEE, 2007. 39, 43, 46

[88] Martin Fowler. *UML distilled : a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004. 15

[89] Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of model checking tools for information systems. In *Formal Methods and Software Engineering*, pages 581–596. Springer, 2010. 30

[90] Marcelo F Frias, Juan P Galeotti, Carlos G López Pombo, and Nazareno M Aguirre. Dynalloy : upgrading alloy with actions. In *Proceedings of the 27th international conference on Software engineering*, pages 442–451. ACM, 2005. 86

[91] Marcelo F Frias, Carlos G López Pombo, Gabriel A Baum, Nazareno M Aguirre, and Thomas SE Maibaum. Reasoning about static and dynamic properties in alloy : A purely relational approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(4) :478–526, 2005. 89

[92] L. Fuentes, J. Manrique, and P. Sánchez. Execution and simulation of (profiled) UML models using Pópulo. In *Proceedings of the 2008 international workshop on Models in software engineering*. ACM, 2008. 141

[93] Dov Gabbay. The declarative past and imperative future. In *Temporal logic in specification*, pages 409–448. Springer, 1989. 71

[94] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173. ACM, 1980. 71

[95] Mauro Gambini, Marcello La Rosa, Sara Migliorini, and Arthur HM Ter Hofstede. Automated error correction of business process models. In *Business Process Management*, pages 148–165. Springer, 2011. 110, 125

[96] E. Gamma. *Design patterns : elements of reusable object-oriented software*. 1995. 52

[97] Geri Georg, Jores Bieman, and Robert B France. Using alloy and uml/ocl to specify run-time configuration management : A case study. *pUML*, 7 :128–141, 2001. 8

[98] Diimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management : From process modeling to workflow automation infrastructure. *Distributed and parallel Databases*, 3(2) :119–153, 1995. 13, 14

[99] Rohit Gheyi, Tiago Massoni, and Paulo Borba. Formally introducing alloy idioms. In *Brazilian Symposium on Formal Methods*, 2007. 86

[100] Software-Ley GmbH. Cosa 2.0 user manual, 1998. 32

[101] Eugene Goldberg and Yakov Novikov. Berkmin : A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12) :1549–1561, 2007. 82

[102] Michael Goldsmith, Bill Roscoe, and Philip Armstrong. Failures-divergence refinement-fdr2 user manual, 2005. 30, 35

[103] Volker Gruhn and Ralf Laue. Complexity metrics for business process models. In *9th international conference on business information systems (BIS 2006)*, volume 85, pages 1–12, 2006. 2, 4

[104] Volker Gruhn and Ralf Laue. What business process modelers can learn from programmers. *Science of Computer Programming*, 65(1) :4–13, 2007. 3

[105] Nicolas Guelfi and Amel Mammar. A formal semantics of timed activity diagrams and its promela translation. In *Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific*, pages 8–pp. IEEE, 2005. 5, 37, 39, 42

[106] John V Guttag, James J Horning, Stephen J Garland, Kevin D Jones, Andres Modet, and Jeannette M Wing. Larch : languages and tools for formal specification. In *Texts and Monographs in Computer Science*. Citeseer, 1993. 62

[107] Thilo Hafer and Wolfgang Thomas. Computation tree logic ctl* and path quantifiers in the monadic theory of the binary tree. In *Automata, Languages and Programming*, pages 269–279. Springer, 1987. 5, 19, 70

[108] Michael Hammer and James Champy. Reengineering the corporation : A manifesto for business revolution. *Business Horizons*, 36(5) :90–91, 1993. 13

[109] JB Hill, M Pezzini, and YV Natis. Findings : confusion remains regarding bpm terminologies. *Gartner Research*, 501(G00155817), 2008. 14

[110] Sebastian Hinz. *Implementierung einer Petrinetz-semantik für BPEL*. PhD thesis, Diplomarbeit, Humboldt-Universität zu Berlin, 2005. 33

[111] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. In *Business Process Management*, pages 220–235. Springer, 2005. 33, 41

[112] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969. 19, 62

[113] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, 1978. 19

[114] Charles Antony Richard Hoare. *Communicating sequential processes*, volume 178. Prentice-hall Englewood Cliffs, 1985. 35

[115] Gerard J Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5) :279–295, 1997. 30, 37, 38, 61

[116] Gerard J Holzmann. Landing a spacecraft on mars. *IEEE software*, 30(2) :83–86, 2013. 19

[117] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979. 37

[118] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Addison Wesley, 2007. 5

[119] Nien-Lin Hsueh, Wen-Hsiang Shen, Zhi-Wei Yang, and Don-Lin Yang. Applying uml and software simulation for process definition, verification, and validation. *Information and Software Technology*, 50(9) :897–911, 2008. 7, 22, 38

[120] Wenjia Huai, Xudong Liu, and Hailong Sun. Towards trustworthy composite service through business process model verification. In *Ubiquitous Intelligence & Computing and 7th International Conference on Autonomic & Trusted Computing (UIC/ATC), 2010 7th International Conference on*, pages 422–427. IEEE, 2010. 34, 41

[121] IBM. Ibm websphere business modeler. ibm.com/software/products/en/modeler-advanced. 3, 33

[122] Radu Iosif, Matthew B Dwyer, and John Hatcliff. Translating java for multiple model checkers : The bandera back-end. *Formal Methods in System Design*, 26(2) :137–180, 2005. 38

[123] ITEA. Merge project, safety & security. http://www.merge-project.eu/. 6

[124] Daniel Jackson. Automating first-order relational logic. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 130–139. ACM, 2000. 8, 83

[125] Daniel Jackson. Lightweight formal methods. In *FME 2001 : Formal Methods for Increasing Software Productivity*, pages 1–1. Springer, 2001. 81

[126] Daniel Jackson. *Software Abstractions : logic, language, and analysis*. MIT press, 2012. 8, 81, 82, 86, 91, 92, 100

[127] Daniel Jackson, H.-Christian Estler, and Derek Rayside. The guided improvement algorithm for exact, general-purpose, many-objective combinatorial optimization. Technical report, MIT Computer Science and Artificial Intelligence Laboratory, 2009. 89, 142

[128] Ivar Jacobson. Object oriented software engineering : a use case driven approach. 1992. 15

[129] Dirk Jäger, Ansgar Schleicher, and Bernhard Westfechtel. Using uml for software process modeling. In *Software Engineering—ESEC/FSE'99*, pages 91–108. Springer, 1999. 18

[130] Monique Jansen-Vullers and Mariska Netjes. Business process simulation–a tool survey. In *Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark*, 2006. 38

[131] Kurt Jensen. *Coloured petri nets*. Springer, 1987. 5, 53, 58

[132] Cliff B Jones. *Systematic software development using VDM*, volume 2. Prentice Hall Englewood Cliffs, 1990. 62

[133] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, et al. Web services business process execution language version 2.0. *OASIS Standard*, 11, 2007. 1

[134] Hyo Taeg Jung and Sang Hyun Joo. Transformation of an activity model into a colored petri net model. In *TISC*, pages 32–37, 2010. 33, 39, 41

[135] Mohammed Kabbaj, Redouane Lbath, and Bernard Coulette. A deviation management system for handling software process enactment evolution. In *Making Globally Distributed Software Development a Success Story*, pages 186–197. Springer, 2008. 2

[136] Henry A Kautz, Bart Selman, et al. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363, 1992. 142

[137] Marc I Kellner, Raymond J Madachy, and David M Raffo. Software process simulation modeling : why ? what ? how ? *Journal of Systems and Software*, 46(2) :91–105, 1999. 38

[138] Yonit Kesten, Zohar Manna, and Amir Pnueli. Verification of clocked and hybrid systems. *Acta Informatica*, 36(11) :837–912, 2000. 37, 62

[139] A. Kirshin, D. Dotan, and A. Hartman. A UML simulator based on a generic model execution engine. *Lecture notes in computer science*, 4364 :324, 2007. 141

[140] Andrei Kirshin, Dolev Dotan, and Alan Hartman. A uml simulator based on a generic model execution engine. In *MoDELS Workshops*, volume 4364, pages 324–326, 2006. 39

[141] Anneke G Kleppe, Jos Warmer, Wim Bast, and MDA Explained. The model driven architecture : practice and promise, 2003. 8, 20

[142] William Calvert Kneale. The development of logic. 1962. 15

[143] Andres Knöpfel, Bernhard Gröne, and Peter Tabeling. *Fundamental modeling concepts*. Wiley, West Sussex UK, 2005. 33

[144] David Knuplesch, Linh Ly, Stefanie Rinderle-Ma, Holger Pfeifer, and Peter Dadam. On enabling data-aware compliance checking of business process models. *Conceptual Modeling–ER 2010*, pages 332–346, 2010. 5, 37, 42

[145] Ryan KL Ko, Stephen SG Lee, and Eng Wah Lee. Business process management (bpm) standards : a survey. *Business Process Management Journal*, 15(5) :744–791, 2009. 3, 14

[146] Birgit Korherr and Beate List. *Extending the UML 2 activity diagram with business process goals and performance measures and the mapping to BPEL*. Springer, 2006. 39

[147] Manuel Laguna and Johan Marklund. *Business process modeling, simulation and design*. CRC Press, 2013. 38

[148] Filippo Lanubile and Giuseppe Visaggio. Evaluating defect detection techniques for software requirements inspections. *ISERN Report no. 00-08*, 2000. 2

[149] Andreas Lanz, Barbara Weber, and Manfred Reichert. Workflow time patterns for process-aware information systems. In *Enterprise, Business-Process and Information Systems Modeling*, pages 94–107. Springer, 2010. 2

[150] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1) :134–152, 1997. 37

[151] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. *Formal aspects of computing*, 11(6) :637–664, 1999. 38

[152] Yoann Laurent, Reda Bendraou, Souheib Baarir, and Marie-Pierre Gervais. Alloy4spv : A formal framework for software process verification. In *Modelling Foundations and Applications*, pages 83–100. Springer, 2014. 11, 105, 140

[153] Yoann Laurent, Reda Bendraou, Souheib Baarir, and Marie-Pierre Gervais. Formalization of fuml : An application to process verification. In *Advanced Information Systems Engineering*, pages 347–363. Springer, 2014. 10, 105, 139

[154] Yoann Laurent, Reda Bendraou, Souheib Baarir, and Marie-Pierre Gervais. Planning for declarative processes. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1126–1133. ACM, 2014. 10, 26, 141

[155] Yoann Laurent, Reda Bendraou, and Marie-Pierre Gervais. Executing and debugging uml models : an fuml extension. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1095–1102. ACM, 2013. 10, 105, 133, 141

[156] Yoann Laurent, Reda Bendraou, and Marie-Pierre Gervais. Generation of process using multi-objective genetic algorithm. In *Proceedings of the 2013 International Conference on Software and System Process*, pages 161–165. ACM, 2013. 12, 9, 10, 116

[157] Peter Lawrence. *Workflow handbook 1997*. John Wiley & Sons, Inc., 1997. 14

[158] Daniel Le Berre, Anne Parrain, et al. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7 :59–64, 2010. 82

[159] Johan Lilius and I Porres Paltor. vuml : A tool for verifying uml models. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 255–258. IEEE, 1999. 38

[160] Johan Lilius and Ivan Porres Paltor. Formalising uml state machines for model checking. In *«UML»'99—The Unified Modeling Language*, pages 430–444. Springer, 1999. 38

[161] Sea Ling and Heinz Schmidt. Time petri nets for workflow modelling and analysis. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 4, pages 3039–3044. IEEE, 2000. 34

[162] Ying Liu, Samuel Muller, and Ke Xu. A static compliance-checking framework for business process models. *IBM Systems Journal*, 46(2) :335–361, 2007. 5, 35, 36, 42, 43, 46, 70

[163] Ruopeng Lu and Shazia Sadiq. A survey of comparative business process modeling approaches. In *Business information systems*, pages 82–94. Springer, 2007. 15, 29

[164] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for ws-bpel. *The Journal of Logic and Algebraic Programming*, 70(1) :96–118, 2007. 29

[165] Daniel Lucrédio, Renata P de M Fortes, and Jon Whittle. Moogle : A model search engine. In *Model Driven Engineering Languages and Systems*, pages 296–310. Springer, 2008. 112

[166] Linh Thao Ly, Stefanie Rinderle-Ma, and Peter Dadam. Design and verification of instantiable compliance rule graphs in process-aware information systems. In *Advanced Information Systems Engineering*, pages 9–23. Springer, 2010. 11, 43, 46

[167] Zohar Manna and Amir Pnueli. *Clocked transition systems*. Department of Computer Science, Stanford University, 1996. 37, 62

[168] Olivera Marjanovic and Maria E Orlowska. On modeling and verification of temporal constraints in production workflows. *Knowledge and Information Systems*, 1(2) :157–192, 1999. 25

[169] S Mauw, R Mateescu, and W Janssen. Verifying business processes using spin. In *Proceedings of the International SPIN Workshop*, pages 21–36, 1998. 37, 42

[170] Jan Mendling. *Detection and prediction of errors in EPC business process models*. PhD thesis, Wirtschaftsuniversität Wien, 2007. 32

[171] Jan Mendling. Empirical studies in process model verification. In *Transactions on Petri Nets and Other Models of Concurrency II*, pages 208–224. Springer, 2009. 3, 4

[172] Jan Mendling, Michael Moser, Gustaf Neumann, HMW Verbeek, Boudewijn F van Dongen, and Wil MP van der Aalst. Faulty epcs in the sap reference model. In *Business Process Management*, pages 451–457. Springer, 2006. 3, 7

[173] Jan Mendling, Gustaf Neumann, and Wil Van Der Aalst. Understanding the occurrence of errors in process models based on metrics. In *On the Move to Meaningful Internet Systems 2007 : CoopIS, DOA, ODBASE, GADA, and IS*, pages 113–130. Springer, 2007. 3

[174] Erich Mikk, Yassine Lakhnechi, and Michael Siegel. Hierarchical automata as model for statecharts. In *Advances in Computing Science—ASIAN'97*, pages 181–196. Springer, 1997. 38

[175] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989. 35

[176] Robin Milner. *Communicating and mobile systems : the pi calculus*. Cambridge university press, 1999. 35

[177] MIT. Alloy : a language and tool for relational models. http://alloy.mit.edu/alloy/. 79, 112, 124

[178] MIT. Alloy tutorial. http://alloy.mit.edu/alloy/tutorials/day-course/. 11, 87

[179] Shoichi Morimoto. A survey of formal verification for business process modeling. In *Computational Science–ICCS 2008*, pages 514–522. Springer, 2008. 4, 26

[180] Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michèle Soria. Uniform random generation of huge metamodel instances. In *Model Driven Architecture-Foundations and Applications*, pages 130–145. Springer, 2009. 9, 112, 124

[181] Tadao Murata. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989. 15, 34, 135, 143

[182] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011. 3

[183] Muan Yong Ng and Michael Butler. Tool support for visualizing csp in uml. In *Formal Methods and Software Engineering*, pages 287–298. Springer, 2002. 39

[184] Muan Yong Ng and Michael Butler. Towards formalizing uml state diagrams in csp. In *Software Engineering and Formal Methods, 2003. Proceedings. First International Conference on*, pages 138–147. IEEE, 2003. 38

[185] Markus Nüttgens and Frank J Rump. Syntax und semantik ereignisgesteuerter prozessketten (epk). In *Promise*, volume 2, pages 64–77, 2002. 29

[186] OMG. Object management group (omg). http://www.omg.org/. 15

[187] OMG. Object constraint language (ocl) version 2.0. http://www.omg.org/spec/OCL/, 2006. 20

[188] OMG. Software & systems process engineering metamodel specification (spem) version 2.0. http://www.omg.org/spec/SPEM/2.0/, 2008. 1, 17, 29

[189] OMG. Semantics of a foundational subset for executable uml models (fuml) version 1.0. http://www.omg.org/spec/FUML/, 2011. 11, 6, 22, 49, 50, 54, 56, 61, 141

[190] OMG. Unified modeling language (uml) version 2.4.1. http://www.omg.org/spec/UML/, 2011. 11, 12, 1, 4, 6, 15, 17, 18, 29, 49, 129, 130

[191] OMG. Concrete syntax for a uml action language : Action language for foundational uml (alf). http://www.omg.org/spec/ALF/, 2013. 50

[192] Object Management Group (OMG). Business process model and notation (bpmn) version 2.0. Technical report, jan 2011. 1, 7, 15, 29

[193] Chun Ouyang, Eric Verbeek, Wil MP van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur HM ter Hofstede. Wofbpel : A tool for automated analysis of bpel processes. In *Service-Oriented Computing-ICSOC 2005*, pages 484–489. Springer, 2005. 32, 41

[194] Chun Ouyang, Eric Verbeek, Wil MP Van Der Aalst, Stephan Breutel, Marlon Dumas, and Arthur HM Ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Science of Computer Programming*, 67(2) :162–198, 2007. 32, 41

[195] Lawrence C Paulson. *Isabelle : A generic theorem prover*, volume 828. Springer, 1994. 19

[196] Eliana B Pereira, Ricardo M Bastos, Toacy C Oliveira, and Michael C Móra. A set of well-formedness rules to checking the consistency of the software processes based on spem 2.0. In *Enterprise Information Systems*, pages 284–299. Springer, 2012. 22

[197] Maja Pesic. Constraint-based workflow management systems : shifting control to users. 2008. 11, 3

[198] Maja Pesic, Helen Schonenberg, and Wil MP van der Aalst. Declare : Full support for loosely-structured processes. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pages 287–287. IEEE, 2007. 15, 141

[199] Maja Pesic, MH Schonenberg, Natalia Sidorova, and Wil MP van der Aalst. Constraint-based workflow models : Change made easy. In *On the Move to Meaningful Internet Systems 2007 : CoopIS, DOA, ODBASE, GADA, and IS*, pages 77–94. Springer, 2007. 2

[200] James L Peterson. Petri net theory and the modeling of systems. 1981. 5, 29

[201] Carl Adam Petri. Introduction to general net theory. In *Net theory and applications*, pages 1–19. Springer, 1980. 19, 34

[202] Pit Pietsch, Hamed Shariat Yazdi, and Udo Kelter. Generating realistic test models for model processing tools. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 620–623. IEEE, 2011. 9, 112, 124

[203] Staffware Plc. Staffware gwd procedure definer's guide, version 8, issue 2, 1999. 32

[204] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977. 5, 19, 70, 71

[205] Louchka Popova-Zeugmann. *Time Petri Nets*. Springer, 2013. 34

[206] Mukul R Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2) :156–173, 2005. 83

[207] Gianna Reggio and Roel Wieringa. Thirty one problems in the semantics of uml 1.3 dynamics. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)–Workshop" Rigorous Modelling and Analysis of the UML : Challenges and Limitations*. Citeseer, 1999. 49

[208] Manfred Reichert and Peter Dadam. Adeptflex—supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10(2) :93–129, 1998. 2

[209] Wolfgang Reisig. *Petri nets : an introduction*. Springer-Verlag New York, Inc., 1985. 32

[210] Mark Richters and Martin Gogolla. Ocl : Syntax, semantics, and tools. In *Object Modeling with the OCL*, pages 42–68. Springer, 2002. 20

[211] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The architecture of a UML virtual machine. *ACM SIGPLAN Notices*, 2001. 141

[212] Asbjørn Rolstadås. *Performance management : A business process benchmarking approach*. Springer, 1995. 13

[213] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William E. Lorensen, et al. *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991. 15

[214] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004. 16

[215] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004. 49

[216] Nicholas Russell, Arthur HM Ter Hofstede, and Wil M van der Aalst. newyawl : specifying a workflow reference language using coloured petri nets. 2007. 2

[217] Nick Russell, Arthur HM Ter Hofstede, David Edmond, and Wil MP van der Aalst. Workflow resource patterns. Technical report, BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, 2004. 2

[218] Nick Russell, Arthur HM Ter Hofstede, and Nataliya Mulyar. Workflow controlflow patterns : A revised view. 2006. 2, 6

[219] Nick Russell, Wil MP van der Aalst, Arthur HM ter Hofstede, and David Edmond. Workflow resource patterns : Identification, representation and tool support. In *Advanced Information Systems Engineering*, pages 216–232. Springer, 2005. 2

[220] Nick Russell, Wil MP van der Aalst, Arthur HM Ter Hofstede, and Petia Wohed. On the suitability of uml 2.0 activity diagrams for business process modelling. In *Proceedings of the 3rd Asia-Pacific conference on Conceptual modelling-Volume 53*, pages 95–104. Australian Computer Society, Inc., 2006. 6, 17

[221] Shazia Sadiq, Maria Orlowska, Wasim Sadiq, and Cameron Foulger. Data flow and validation in workflow modelling. In *Proceedings of the 15th Australasian database conference-Volume 27*, pages 207–214. Australian Computer Society, Inc., 2004. 23

[222] Wasim Sadiq and Maria E Orlowska. Analyzing process models using graph reduction techniques. *Information systems*, 25(2) :117–134, 2000. 38

[223] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking uml state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3) :357–369, 2001. 38

[224] Karsten Schmidt. Lola a low level analyser. In *Application and Theory of Petri Nets 2000*, pages 465–474. Springer, 2000. 30, 32, 33

[225] Karsten Schmidt and Christian Stahl. A petri net semantic for bpel4ws-validation and application. In *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets*, pages 1–6, 2004. 33, 41

[226] Claus Schröter, Stefan Schwoon, and Javier Esparza. The model-checking kit. In *Applications and Theory of Petri Nets 2003*, pages 463–472. Springer, 2003. 30, 34

[227] Wuwei Shen, Kevin Compton, and James Huggins. A validation method for uml model based on abstract state machines. In *Proceeding of EUROCAST*, pages 220–223, 2001. 38

[228] Wuwei Shen, Kevin Compton, and James Huggins. A toolset for supporting uml static and dynamic model checking. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 147–152. IEEE, 2002. 38, 39

[229] A Sheth, K Kochut, and J Miller. Large scale distributed information systems (lsdis) laboratory, meteor project page, 2001. 32

[230] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 94–105. IEEE, 2003. 83

[231] Avraham Shtub and Reuven Karni. Business process improvement. In *ERP*, pages 217–254. Springer, 2010. 3

[232] David Simchi-Levi, Philip Kaminsky, and Edith Simchi-Levi. Design and managing the supply chain : Concepts, strategies, and case studies, 2000. 13

[233] Anthony JH Simons and Ian Graham. 30 things that go wrong in object modelling with uml 1.3. In *Behavioral Specifications of Businesses and Systems*, pages 237–257. Springer, 1999. 49

[234] Howard Smith and Peter Fingar. *Business process management : the third wave*, volume 1. Meghan-Kiffer Press Tampa, 2003. 29

[235] Rachel L Smith, George S Avrunin, Lori A Clarke, and Leon J Osterweil. Propel : an approach supporting property elucidation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 11–21. ACM, 2002. 5, 37, 44, 45, 46

[236] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992. 62, 81

[237] Christian Stahl. *Transformation von BPEL4WS in Petrinetze*. PhD thesis, Master's thesis, Humboldt University, Berlin, Germany, 2004. 33

[238] Tony Spiteri Staines. Intuitive mapping of uml 2 activity diagrams into fundamental modeling concept petri net diagrams and colored petri nets. In *ECBS*, pages 191–200. IEEE, 2008. 33, 39, 41

[239] Mark Strembeck and Jan Mendling. Modeling process-related rbac models with extended uml activity models. *Information and Software Technology*, 53(5) :456–483, 2011. 18

[240] Mana Taghdiri and Daniel Jackson. *A lightweight formal analysis of a multicast key management scheme*. Springer, 2003. 8

[241] Alfred Tarski and Steven R Givant. *A formalization of set theory without variables*, volume 41. American Mathematical Soc., 1987. 82

[242] The ModelDriven Community. Foundational UML reference implementation. http://portal.modeldriven.org/project/foundationalUML, 2007. 55

[243] Yann Thierry-Mieg and Lom-Messan Hillah. Uml behavioral consistency checking using instantiable petri nets. *Innovations in systems and software engineering*, 4(3) :293–300, 2008. 33, 39, 41

[244] CPN Tools. Computer tool for coloured petri nets, 2009. 30, 33, 34

[245] Emina Torlak and Daniel Jackson. Kodkod : A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007. 82

[246] Nikola Trcka, Wil van der Aalst, and Natalia Sidorova. Analyzing control-flow and data-flow in workflow processes in a unified way. *Computer Science Report*, (08-31), 2008. 23, 34, 41

[247] Nikola Trčka, Wil MP Van der Aalst, and Natalia Sidorova. Data-flow anti-patterns : Discovering data-flow errors in workflows. In *Advanced Information Systems Engineering*, pages 425–439. Springer, 2009. 5, 23, 34, 41

[248] Mattias Ulbrich, Ulrich Geilmann, Aboubakr Achraf El Ghazi, and Mana Taghdiri. A proof assistant for alloy specifications. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 422–436. Springer, 2012. 89

[249] Amirhossein Vakili and Nancy A Day. Temporal logic model checking in alloy. In *Abstract State Machines, Alloy, B, VDM, and Z*, pages 150–163. Springer, 2012. 88, 89

[250] Antti Valmari. The state explosion problem. In *Lectures on Petri nets I : Basic models*, pages 429–528. Springer, 1998. 83

[251] Wil MP Van Der Aalst. Three good reasons for using a petri-net-based workflow management system. In *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96)*, pages 179–201. Cambridge, Massachusetts, 1996. 29, 32

[252] Wil MP van der Aalst. The application of petri nets to workflow management. *Journal of circuits, systems, and computers*, 8(01) :21–66, 1998. 29, 32, 34

[253] Wil MP van der Aalst. Formalization and verification of event-driven process chains. *Information and Software technology*, 41(10) :639–650, 1999. 6, 32, 41

[254] Wil MP Van Der Aalst. Workflow verification : Finding control-flow errors using petri-net-based techniques. In *Business Process Management*, pages 161–183. Springer, 2000. 34

[255] Wil MP van der Aalst. Pi calculus versus petri nets : Let us eat "humble pie" rather than further inflate the "pi hype". *BPTrends*, 3(5) :1–11, 2005. 29, 36

[256] Wil MP van der Aalst, Alexander Hirnschall, and HMW Verbeek. An alternative way to analyze workflow graphs. In *Advanced Information Systems Engineering*, pages 535–552. Springer, 2002. 38

[257] Wil MP Van Der Aalst and Maja Pesic. *DecSerFlow : Towards a truly declarative service flow language*. Springer, 2006. 27, 141

[258] Wil MP Van Der Aalst and Arthur HM Ter Hofstede. Yawl : yet another workflow language. *Information systems*, 30(4) :245–275, 2005. 32, 34

[259] Wil MP van Der Aalst, Arthur HM Ter Hofstede, Bartek Kiepuszewski, and Alistair P Barros. Workflow patterns. *Distributed and parallel databases*, 14(1) :5–51, 2003. 5, 6, 17, 34, 36, 112, 113

[260] Wil MP Van Der Aalst, Arthur HM Ter Hofstede, and Mathias Weske. *Business process management : A survey*. Springer, 2003. 1, 14, 29

[261] Wil MP van der Aalst, Kees M van Hee, Arthur HM ter Hofstede, Natalia Sidorova, HMW Verbeek, Marc Voorhoeve, and Moe Thandar Wynn. Soundness of workflow nets : classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3) :333–363, 2011. 4, 5, 7, 22, 34

[262] WMP Van Der Aalst. Challenges in business process management : Verification of business processes using petri nets. *Bulletin of the EATCS*, 80 :174–199, 2003. 32

[263] W.M.P. van der Aalst and A. H. M. Ter Hofstede. Workflow patterns : On the expressive power of (petri-net-based) workflow languages. In *of DAIMI, University of Aarhus*, pages 1–20, 2002. 5, 34, 58

[264] WMP Van der Aalst and KM Van Hee. Business process redesign : a petri-net-based approach. *Computers in industry*, 29(1) :15–26, 1996. 34

[265] B.F. van Dongen. *Process Mining and Verification*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, July 2007. 1

[266] Boudewijn F van Dongen, Ana Karla A de Medeiros, HMW Verbeek, AJMM Weijters, and Wil MP Van Der Aalst. The prom framework : A new era in process mining tool support. In *Applications and Theory of Petri Nets 2005*, pages 444–454. Springer, 2005. 32

[267] Boudewijn F van Dongen, Wil MP van der Aalst, and Henricus MW Verbeek. Verification of epcs : Using reduction rules and petri nets. In *Advanced Information Systems Engineering*, pages 372–386. Springer, 2005. 32

[268] Kees Max van Hee et al. *Workflow management : models, methods, and systems*. The MIT press, 2004. 2, 34, 58

[269] Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and more focused control-flow analysis for business process models through sese decomposition. In *Service-Oriented Computing–ICSOC 2007*, pages 43–55. Springer, 2007. 3, 33, 143

[270] E. Verbeek. *Verification of WF-nets*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, June 2004. 4

[271] Henricus MW Verbeek, Twan Basten, and Wil MP van der Aalst. Diagnosing workflow processes using woflan. *The Computer Journal*, 44(4) :246–279, 2001. 32, 33, 34, 41, 108

[272] Henricus MW Verbeek and Wil MP van der Aalst. Analyzing bpel processes using petri nets. In *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 59–78, 2005. 32, 41

[273] Valdis Vitolins and Audris Kalnins. Semantics of uml 2.0 activity diagram for business modeling by means of virtual machine. In *EDOC Enterprise Computing Conference, 2005 Ninth IEEE International*, pages 181–192. IEEE, 2005. 39

[274] Kenji Watahiki, Fuyuki Ishikawa, and Kunihiko Hiraishi. Formal verification of business processes with temporal and resource constraints. In *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, pages 1173–1180. IEEE, 2011. 5, 37, 42

[275] Barbara Weber, Manfred Reichert, and Stefanie Rinderle-Ma. Change patterns and change support features–enhancing flexibility in process-aware information systems. *Data & knowledge engineering*, 66(3) :438–466, 2008. 12, 9, 113, 114, 118, 125

[276] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981. 135, 143

[277] Mathias Weske. *Business process management : concepts, languages, architectures*. Springer, 2012. 1

[278] Dirk Wodtke and Gerhard Weikum. A formal foundation for distributed workflow execution based on state charts. In *Database Theory—ICDT'97*, pages 230–246. Springer, 1997. 29

[279] Peter Wong and Jeremy Gibbons. A process-algebraic approach to workflow specification and refinement. In *Software Composition*, pages 51–65. Springer, 2007. 6, 35, 42

[280] Peter YH Wong and Jeremy Gibbons. A process semantics for bpmn. In *Formal Methods and Software Engineering*, pages 355–374. Springer, 2008. 6, 35, 42

[281] Peter YH Wong and Jeremy Gibbons. A relative timed semantics for bpmn. *Electronic Notes in Theoretical Computer Science*, 229(2) :59–75, 2009. 6, 35, 42

[282] Jim Woodcock and Jim Davies. *Using Z : specification, refinement, and proof*. Prentice-Hall, Inc., 1996. 35

[283] Moe Thandar Wynn, HMW Verbeek, Wil MP van der Aalst, Arthur HM ter Hofstede, and David Edmond. Business process verification–finally a reality ! *Business Process Management Journal*, 15(1) :74–92, 2009. 34, 41

[284] Dong Xu, Huaikou Miao, and Nduwimfura Philbert. Model checking uml activity diagrams in fdr. In *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*, pages 1035–1040. IEEE, 2009. 35, 39, 42

[285] Wing Lok Yeung, Karl RPH Leung, Ji Wang, and Wei Dong. Modelling and model checking suspendible business processes via statechart diagrams and csp. *Science of Computer Programming*, 65(1) :14–29, 2007. 38

[286] Wing Lok Yeung, KRP Leung, Ji Wang, and Wei Dong. Improvements towards formalizing uml state diagrams in csp. In *Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific*, pages 7–pp. IEEE, 2005. 38

[287] Xiangpeng Zhao, Quan Long, and Zongyan Qiu. Model checking dynamic uml consistency. In *Formal Methods and Software Engineering*, pages 440–459. Springer, 2006. 38, 39

[288] Lenore Zuck. Past temporal logic. *Ann Arbor*, 1001 :48106–1346, 1987. 49, 71

Annexe A

Modules Alloy4PV

A.1 Modules statiques

A.1.1 Syntax.als

```
module Syntax

open util/integer
open util/boolean

abstract sig Object {}

--
-- fUML·Syntax·Classes·Kernel
--
abstract sig RedefinableElement extends NamedElement {}
abstract sig Element extends Object {}
abstract sig NamedElement extends  Element {}
abstract sig Class extends BehavioredClassifier {}
abstract sig Classifier extends Type {}
abstract sig Type extends Namespace {}
abstract sig Namespace extends PackageableElement {}
abstract sig PackageableElement extends NamedElement {}
abstract sig MultiplicityElement extends Element {
  upper : Int,
  lower : Int
}
abstract sig TypedElement extends NamedElement {}
abstract sig Parameter extends TypedElement {}

--
-- fUML·Syntax·Activities·IntermediateActivities
--
abstract sig ActivityEdge extends RedefinableElement {
  source : ActivityNode,
  target : ActivityNode,
}
abstract sig ActivityNode extends RedefinableElement {
  outgoing : set ActivityEdge,
  incoming : set ActivityEdge
}
abstract sig ControlNode extends ActivityNode {}
abstract sig FinalNode extends ControlNode {}
abstract sig ActivityFinalNode extends FinalNode {}
abstract sig FlowFinalNode extends FinalNode {}
abstract sig Activity extends Behavior {}
abstract sig ActivityParameterNode extends ObjectNode {
  parameterMultiplicityElement : MultiplicityElement,
  direction : ParameterDirectionKind
}
abstract sig ParameterDirectionKind extends Element {}
abstract sig ObjectNode extends ActivityNode {}
```

```
48 abstract sig ControlFlow extends ActivityEdge {}
49 abstract sig ObjectFlow extends ActivityEdge {}
50 abstract sig DecisionNode extends ControlNode {}
51 abstract sig MergeNode extends ControlNode {}
52 abstract sig ForkNode extends ControlNode {}
53 abstract sig JoinNode extends ControlNode {}
54 abstract sig InitialNode extends ControlNode {}
55
56 --
57 -- fUML·Syntax·CommonBehaviors·BasicBehaviors
58 --
59 abstract sig Behavior extends Class {}
60 abstract sig BehavioredClassifier extends Classifier {}
61
62 --
63 -- fUML·Syntax·Activities·ExtraStructuredActivities
64 --
65 abstract sig ExpansionKind extends Object {}
66 abstract sig EK_parallel, EK_iterative, EK_stream extends ExpansionKind {}
67 abstract sig ExpansionNode extends ObjectNode {}
68 abstract sig ExpansionRegion extends StructuredActivityNode {}
69
70 --
71 -- fUML·Syntax·Activities·CompleteStructuredActivities
72 --
73 abstract sig Clause extends Element {}
74 abstract sig ConditionalNode extends StructuredActivityNode {}
75 abstract sig ExecutableNode extends ActivityNode {}
76 abstract sig LoopNode extends StructuredActivityNode {}
77 abstract sig StructuredActivityNode extends Action {}
78
79 --
80 -- fUML·Syntax·Actions·BasicActions
81 --
82 abstract sig Action extends ExecutableNode {
83   output : set OutputPin,
84   input : set InputPin
85 }
86 abstract sig CallAction extends InvocationAction {}
87 abstract sig CallBehaviorAction extends CallAction {}
88 abstract sig CallOperationAction extends CallAction {}
89 abstract sig SendSignalAction extends InvocationAction {}
90 abstract sig OpaqueAction extends Action {}
91 abstract sig InputPin extends Pin {}
92 abstract sig OutputPin extends Pin {}
93 abstract sig Pin extends ObjectNode {
94   multiplicityElement : MultiplicityElement
95 }
96 abstract sig InvocationAction extends Action {}
97
98 --
99 -- sig - help to write simple process
100 --
101 one sig ParameterDirectionIn extends ParameterDirectionKind {}
102 one sig ParameterDirectionOut extends ParameterDirectionKind {}
103
104 one sig MultiZeroOne extends MultiplicityElement {} {
105   lower = 0
106   upper = 1
107 }
108 one sig MultiOneOne extends MultiplicityElement {} {
109   lower = 1
110   upper = 1
111 }
112 one sig MultiZeroStar extends MultiplicityElement {} {
113   lower = 0
114   upper = max[]
115 }
116 one sig MultiOneStar extends MultiplicityElement {} {
117   lower = 1
118   upper = max[]
119 }
120 one sig MultiTwoStar extends MultiplicityElement {} {
121   lower = 2
```

```
  upper = max[]
}
one sig MultiStarStar extends MultiplicityElement {} {
  lower = max[]
  upper = max[]
}

fact nodeAndEdgeConnectivity {
  all node : ActivityNode, edge : ActivityEdge | edge.source = node implies edge in node.outgoing
  and
  all node : ActivityNode, edge : ActivityEdge | edge.target = node implies edge in node.incoming
  and
  all node : ActivityNode, edge : ActivityEdge | edge in node.outgoing implies edge.source = node
  and
  all node : ActivityNode, edge : ActivityEdge | edge in node.incoming implies edge.target = node
}
```

A.1.2 Semantic.als

```
module Semantic

open util/ordering[State] as Ord
open util/integer
open util/boolean
open Syntax

--
-- organizational
--
abstract sig Resource extends Object {}

abstract sig Info {
  use_i      : ExecutableNode →set Resource,
  timing_i     : ExecutableNode →Int,
  capacity_i    : Resource →Int
}
fun Use      : ExecutableNode →set Resource   { Info.use_i }
fun Timing     : ExecutableNode →Int     { Info.timing_i }
fun Capacity     : Resource →Int        { Info.capacity_i }

--
-- State
--
abstract sig Status {}
one sig Running, Finished extends Status {}

sig State {
  v : ActivityNode →one Int, // tokens
  e : ActivityEdge →one Int, // offers

  lc : ExecutableNode →one Int, // local clock
  gc : Int, // global clock

  running : Status
}

--
-- Helper
--
// getters
pred State.hasOffers[edge:ActivityEdge]
  { this.e[edge] > 0 }
fun State.getTokens[node:ActivityNode] : Int
  { this.v[node] }
pred State.hasTokens[node:ActivityNode]
  { this.v[node] > 0 }
fun State.getOffers [edge:ActivityEdge] : Int
  { this.e[edge] }
fun State.getLocalClock[node:ExecutableNode] : Int
  { this.lc[node] }
// setters
pred State.setTokens[node:ActivityNode, i : Int]
  { this.v[node] = i}
pred State.setOffers [edge:ActivityEdge, i : Int]
```

```
56    { this·e[edge] = i}
57  pred State·setLocalClock[node:ExecutableNode, i : Int]
58    { this·lc[node] = i }
59  // block Edge and Node
60  pred freezeAll[s,s':State]
61    { freezeAllNode[s,s',ActivityNode] and freezeAllEdge[s,s',ActivityEdge]}
62  pred freezeAllEdge[s,s':State,edge: set ActivityEdge]
63    { all e : edge | freezeEdge[s,s',e] }
64  pred freezeAllNode[s,s':State,node: set ActivityNode]
65    { all n : node | freezeNode[s,s',n] }
66  pred freezeEdge[s,s':State, edge:ActivityEdge]
67    {s·getOffers[edge] = s'·getOffers[edge] }
68  pred freezeNode[s,s':State, node:ActivityNode]
69    { s·getTokens[node] = s'·getTokens[node] }
70  pred freezeTime[s,s':State]
71    { s'·lc = s·lc and s'·gc = s·gc }
72  pred freezeTime[s,s':State,r:ActivityNode] {
73    r in ExecutableNode implies {
74      all n:ExecutableNode-r | s'·setLocalClock[ n,s·getLocalClock[n] ]
75      s'·setLocalClock[ r, 0 ]
76      s'·gc = s·gc
77    } else {
78      freezeTime[s,s']
79    }
80  }
81
82  --
83  -- Transition Systems
84  --
85  pred init[ nodes:ActivityNode→Int, edges:ActivityEdge→Int, global : Int] {
86    first·v = (ActivityNode →0) ++ nodes
87    first·e = (ActivityEdge →0) ++ edges
88
89    first·gc = global
90    first·lc = (ExecutableNode →0)
91
92    first·running = Running
93  }
94  fact traces {
95    all s: State - last | let s' = s·next | {
96      s·running = Running implies {
97        transition[s,s']
98      } else {
99        // stuttering the last state
100       endLoop[s,s']
101     }
102   }
103 }
104 pred transition[s,s':State] {
105   some f:ActivityFinalNode | s·hasTokens[f] implies {
106     endLoop[s,s']
107   } else {
108     s'·running = Running
109     (stepTime[s,s'] or stepTokens[s,s'])
110   }
111
112 }
113 pred stepTime[s,s':State] {
114   s·enabledTime and fireTime[s,s']
115 }
116 pred stepTokens[s,s':State] {
117   one n:(ActivityNode-Pin) | {
118     (s·enabledStart[n] and fireStart[s,s',n])
119     or
120     (s·enabledFinish[n] and fireFinish[s,s',n])
121   }
122 }
123 pred endLoop[s,s':State] {
124   freezeAll[s,s']
125   freezeTime[s,s']
126   s'·running = Finished
127 }
128
129 --
```

```
130 -- Timing
131 --
132 pred State·enabledTime[] {
133   some en:ExecutableNode | this·hasTokens[en] and not this·getLocalClock[en]·gte[Timing[en]]
134 }
135
136 pred fireTime[s,s':State] {
137   freezeAll[s,s']
138   // update localClock
139   all n:ExecutableNode | s·hasTokens[n] implies {
140     // incremental localClock
141     s'·setLocalClock[n,s·getLocalClock[n]·add[1]]
142   } else {
143     s'·setLocalClock[n,s·getLocalClock[n]]
144   }
145   // update globalClock
146   s'·gc = s·gc·add[1]
147 }
148
149 --
150 -- Step Finish
151 --
152 pred State·enabledFinish[node:ActivityNode] {
153   // the node is already executing
154   this·hasTokens[node]
155   and
156   // the node owns outgoing edge
157   some node·outgoing
158   and
159   // enough time spent on this action?
160   node in Action implies {
161     this·getLocalClock[node]·gte[Timing[node]]
162   }
163 }
164 pred fireFinish[s , s' : State, node:ActivityNode] {
165   freezeTime[s,s',node] // old one using "node" in param
166   // Action
167   node in Action implies {
168     s'·setTokens[node, 0]
169     // add offers to both outgoing edge and outgoing edge of output
170     // the number of added offers corresponds to the number of tokens on the node
171     all edge : (node·outgoing+ node·output·outgoing) | {
172       s'·setOffers[edge, s·getOffers[edge]·add[s·getTokens[edge·source]]]
173     }
174     // reset input and output pin
175     all pin : (node·output+ node·input) | {
176       s'·setTokens[pin, 0]
177     }
178     freezeAllEdge[s,s',(ActivityEdge-(node·outgoing+ node·output·outgoing))]
179     freezeAllNode[s,s',(ActivityNode-(node+ node·input+ node·output))]
180   }
181
182   // DecisionNode
183   else node in DecisionNode implies {
184     s'·setTokens[node, 0]
185     // offer tokens on only one outgoing edge
186     one edge : node·outgoing | {
187       s'·setOffers[edge, s·getOffers[edge]·add[s·getTokens[edge·source]]]
188       freezeAllEdge[s,s',(ActivityEdge-edge)]
189     }
190     freezeAllNode[s,s',(ActivityNode-node)]
191   }
192
193   // FlowFinalNode
194   else node in FlowFinalNode implies {
195     s'·setTokens[node, 0]
196     freezeAllEdge[s,s',(ActivityEdge)]
197     freezeAllNode[s,s',(ActivityNode-node)]
198   }
199
200   // (default) ActivityNode
201   else node in ActivityNode implies {
202     s'·setTokens[node, 0]
203     all edge : node·outgoing | {
```

```
204       s'·setOffers[edge, s·getOffers[edge]·add[s·getTokens[edge·source]]]
205     }
206     freezeAllEdge[s,s',(ActivityEdge-node·outgoing)]
207     freezeAllNode[s,s',(ActivityNode-node)]
208   }
209 }
210
211 --
212 -- Step Start
213 --
214 pred State·enabledStart[node:ActivityNode] {
215   // the node is not already executing
216   not this·hasTokens[node]
217   and
218   // the node owns incoming edge
219   some node·incoming
220   and
221   // if the node is a MergeNode
222   // only 1 incoming edge with offers is required
223   (node in MergeNode implies {
224     some n:node·incoming | this·hasOffers[n]
225   }
226   // if the node is a Action
227   // check if all the incoming edge have offers
228   // check if all the input pin have enough offers depending of their multiplicity
229   else node in Action implies {
230     (all n:node·incoming | this·hasOffers[n])
231     and
232     (all n:node·input·incoming | this·getOffers[n] ≥ n·target·multiplicityElement·lower)
233     and
234     (all r:Use[node] | (
235       #{others:(ExecutableNode-node) | r in Use[others] and this·hasTokens[others] } < Capacity[r]
236     ))
237   }
238   // (default) If the node is a ActivityNode
239   // check if all the incoming edge have offers
240   else node in ActivityNode implies {
241     all n:node·incoming | this·hasOffers[n]
242   })
243 }
244 pred fireStart[s , s' : State, node : ActivityNode] {
245   freezeTime[s,s']
246   // Action
247   node in Action implies {
248     s'·setTokens[node, 1]
249     // set the number of consumed tokens in the input pin
250     // trying to consume all offered tokens
251     all iPin : node·input | {
252       iPin·multiplicityElement·upper ≥ s·getOffers[iPin·incoming]
253       implies
254         // consume all
255         s'·getTokens[iPin] = s·getOffers[iPin·incoming]
256       else
257         // consume up to upper
258         s'·getTokens[iPin] = iPin·multiplicityElement·upper
259     }
260     // set the number of tokens in the output pin
261     // nondeterministic between lower and upper multiplicity
262     all oPin : node·output | {
263       s'·getTokens[oPin] ≥ oPin·multiplicityElement·lower
264       and
265       s'·getTokens[oPin] < = oPin·multiplicityElement·upper
266     }
267     // remove offers from both incoming edge and incoming edge of input
268     // the number of removed offers corresponds to the new number of tokens on the node
269     all edge : (node·incoming+ node·input·incoming) | {
270       s'·setOffers[edge, s·getOffers[edge]·sub[s'·getTokens[edge·target]]]
271     }
272     freezeAllEdge[s,s',ActivityEdge-(node·incoming+ node·input·incoming) ]
273     freezeAllNode[s,s',ActivityNode-(node+ node·input+ node·output)]
274   }
275
276   // MergeNode
277   else node in MergeNode implies {
```

```
278     s'·setTokens[node, 1]
279     // choose one incoming edge
280     one edge : node·incoming | {
281       // the choosed edge need to have offers
282       s·hasOffers[edge]
283       // remove offers from the edge
284       s'·setOffers[edge, s·getOffers[edge]·sub[s'·getTokens[edge·target]]]
285       freezeAllEdge[s,s',(ActivityEdge-edge)]
286     }
287     freezeAllNode[s,s',(ActivityNode-node)]
288   }
289
290   // ActivityFinalNode
291   else node in ActivityFinalNode implies {
292     s'·setTokens[node, 1]
293     all edge : node·incoming | {
294       s'·setOffers[edge, s·getOffers[edge]·sub[s'·getTokens[edge·target]]]
295     }
296     freezeAllEdge[s,s',(ActivityEdge-node·incoming)]
297     freezeAllNode[s,s',(ActivityNode-node)]
298     s'·next·running = Finished
299   }
300
301   // (default) ActivityNode
302   else node in ActivityNode implies {
303     s'·setTokens[node, 1]
304     all edge : node·incoming | {
305       s'·setOffers[edge, s·getOffers[edge]·sub[s'·getTokens[edge·target]]]
306     }
307     freezeAllEdge[s,s',(ActivityEdge-node·incoming)]
308     freezeAllNode[s,s',(ActivityNode-node)]
309   }
310 }
```

A.1.3 Library.als

```
1  module Library
2
3  open Syntax
4  open Semantic
5
6  // Helper
7  pred State·start[n:ExecutableNode]
8    { not this·hasTokens[n] and this·next·hasTokens[n] }
9  pred State·end[n:ExecutableNode]
10   { not this·hasTokens[n] and this·prev·hasTokens[n] }
11 pred State·data[n:Pin]
12   { this·hasTokens[n] or this·hasOffers[n·outgoing] }
13 fun State·parallel[] : set ExecutableNode
14   { { n:ExecutableNode | this·hasTokens[n] } }
15
16 fun getNumberOfEdgeActivation[edge:ActivityEdge] : State {
17   { s : State | not s·hasOffers[edge] and s·next·hasOffers[edge] }
18 }
19 fun getNumberOfNodeActivation[node:ActivityNode] : State {
20   { s : State | not s·hasTokens[node] and s·next·hasTokens[node] }
21 }
22
23 --
24 -- Fairness
25 --
26 // (1)
27 pred fairness1 {
28   some s:State | s·running = Finished
29 }
30 // (2)
31 pred fairness2[n:Int] {
32   all decision:DecisionNode | fairDecision[decision,n]
33 }
34 pred fairDecision[node:DecisionNode,n:Int] {
35   all edge:node·outgoing |
36     #{getNumberOfEdgeActivation[edge]} < = n
37 }
38
```

```
39 --
40 -- Flux de controle
41 --
42 // (1.1) Impossibilite de completion
43 pred Completion {
44   some s:State, f:ActivityFinalNode | s.hasTokens[f]
45 }
46
47 // (1.2) Completion non propre
48 pred CompletionNotClean {
49   all s:State | some f:ActivityFinalNode | s.hasTokens[f] implies {
50     all o:ExecutableNode | not s.hasTokens[o]
51   }
52 }
53
54 // (1.3) Transition morte (accessibilite)
55 pred DeadTransition[n:ExecutableNode] {
56   all s:State | not s.hasTokens[n]
57 }
58
59 --
60 -- Data
61 --
62 pred State.pAll[node:ExecutableNode] {
63   not this.hasTokens[node]
64   and
65   some node.incoming
66 }
67 pred State.pNode[node:ExecutableNode] {
68   this.pAll[node]
69   and
70   node in ActivityNode implies {
71     all n:node.incoming | this.hasOffers[n]
72   }
73 }
74 pred State.pAction[node:ExecutableNode] {
75   this.pAll[node]
76   and
77   node in Action implies {
78     (all n:node.incoming | this.hasOffers[n])
79     and
80     (all n:node.input.incoming | this.getOffers[n] ≥ n.target.multiplicityElement.lower)
81     and
82     (all r:Use[node] | (
83       #{others:(ExecutableNode-node) | r in Use[others] and this.hasTokens[others] } < Capacity[r]
84     ))
85   }
86 }
87 // (2.1) Donnees manquantes
88 pred MissingData {
89   all s:State, n:ExecutableNode | s.pNode[n] implies {
90     some ss:(s+ s.nexts) | ss.pAction[n]
91   }
92 }
93 // (2.2) Donnees redondantes
94 pred RedondantData {
95   all s:State | some f:ActivityFinalNode | s.hasTokens[f] implies {
96     all o:ObjectFlow | not s.hasOffers[o]
97   }
98 }
99
100 --
101 -- Resources
102 --
103 // (3.1) LackOfResource
104 pred LackOfResource {
105   all s:State, en:ExecutableNode, r:Resource | s.enabledStart[en] and (r in Use[en]) implies {
106     #{ o : ExecutableNode | o ≠ en and r in Use[o] and s.hasTokens[o] } < Capacity[r]
107   }
108 }
109 // (3.2) Utilisation inefficace des ressources.
110 pred InefficientRessourceUse[r:Resource] {
111   some s:State, n:ExecutableNode | r in Use[n] and s.hasTokens[n]
112 }
```

```
pred InefficientRessourceUse2[r:Resource] {
  some s:State | #{n : s.parallel[] | r in Use[n]} = Capacity[r]
}

--
-- Time
--
// (4.1) Duree du procede.
pred TotalTime[t:Int] {
  some s:State, f:ActivityFinalNode | s.hasTokens[f] and s.gc.lt[t]
}
// (4.2) Contrainte de debut fin d activite.
pred TimeActivityStartLowerThan[t:Int, n:ExecutableNode] {
  all s:State | s.start[n] implies {
    s.gc.lt[t]
  }
}
pred TimeActivityStartHigherThan[t:Int, n:ExecutableNode] {
  all s:State | s.start[n] implies {
    s.gc.gt[t]
  }
}
pred TimeActivityEndLowerThan[t:Int, n:ExecutableNode] {
  all s:State | s.end[n] implies {
    s.gc.lt[t]
  }
}
pred TimeActivityEndHigherThan[t:Int, n:ExecutableNode] {
  all s:State | s.end[n] implies {
    s.gc.gt[t]
  }
}
// (4.3) Contrainte relative de debut/fin d'activite.
pred TimeRelativeStartLowerThan[a,b:ExecutableNode, t:Int]{
  all s1:State | s1.start[b] implies {
    some s2 : (s1+ s1.nexts) | s2.start[a] and s2.gc.lt[s1.gc.add[t]]
  }
}
pred TimeRelativeStartHigherThan[a,b:ExecutableNode, t:Int]{
  all s1:State | s1.start[b] implies {
    some s2 : (s1+ s1.nexts) | s2.start[a] and s2.gc.gt[s1.gc.add[t]]
  }
}
pred TimeRelativeEndLowerThan[a,b:ExecutableNode, t:Int]{
  all s1:State | s1.start[b] implies {
    some s2 : (s1+ s1.nexts) | s2.start[a] and s2.gc.lt[s1.gc.add[t]]
  }
}
pred TimeRelativeEndHigherThan[a,b:ExecutableNode, t:Int]{
  all s1:State | s1.start[b] implies {
    some s2 : (s1+ s1.nexts) | s2.start[a] and s2.gc.gt[s1.gc.add[t]]
  }
}

--
-- Business
--
// Existence
pred range[a:ExecutableNode, min,max:Int] {
  let n = #{getNumberOfNodeActivation[a]} | {
    n.gte[min] and n.lte[max]
  }
}
// Relation
pred relation_before[a,b:ExecutableNode] {
  all s:State | s.hasTokens[b] implies {
    some ss:(s+ s.prevs) | ss.hasTokens[a]
  }
}
pred relation_after[a,b:ExecutableNode] {
  all s:State | s.hasTokens[b] implies {
    some ss:(s+ s.nexts) | ss.hasTokens[a]
  }
}
```

```
187 pred relation_par[a,b:ExecutableNode] {
188   all s:State | s.hasTokens[a] implies {
189     some ss:(s+ s.nexts) | ss.hasTokens[a] and ss.hasTokens[b]
190   }
191 }
192 pred relation_excl[a,b:ExecutableNode] {
193   all s:State | s.hasTokens[a] implies {
194     not s.hasTokens[b]
195   }
196 }
197 pred relation_between[a,b,c:ExecutableNode] {
198   all s:State | s.hasTokens[a] implies {
199     (some ss:(s+ s.prevs) | ss.hasTokens[b])
200     and
201     (some ss:(s+ s.nexts) | ss.hasTokens[c])
202   }
203 }
204 // ExistenceData
205 pred existence_data[d:OutputPin] {
206   some s:State | s.data[d]
207 }
208 // RelationData
209 pred relation_data_before[da,db:OutputPin] {
210   all s:State | s.data[db] implies {
211     some ss:(s+ s.prevs) | ss.data[da]
212   }
213 }
214 pred relation_data_after[da,db:OutputPin] {
215   all s:State | s.data[db] implies {
216     some ss:(s+ s.nexts) | ss.data[da]
217   }
218 }
219 pred relation_data_par[da,db:OutputPin] {
220   all s:State | s.data[da] implies {
221     some ss:(s+ s.nexts) | ss.data[da] and ss.data[db]
222   }
223 }
224 pred relation_data_excl[da,db:OutputPin] {
225   all s:State | s.data[da] implies {
226     not s.data[db]
227   }
228 }
229 // RelationDataActivity
230 pred relation_data_act_before[d:OutputPin, a:ExecutableNode] {
231   all s:State | s.hasTokens[a] implies {
232     some ss:(s+ s.prevs) | ss.data[d]
233   }
234 }
235 pred relation_data_act_after[d:OutputPin, a:ExecutableNode] {
236   all s:State | s.hasTokens[a] implies {
237     some ss:(s+ s.nexts) | ss.data[d]
238   }
239 }
240 pred relation_data_act_during[d:OutputPin, a:ExecutableNode] {
241   all s:State | s.hasTokens[a] implies {
242     some ss:(s+ s.nexts) | ss.hasTokens[a] and ss.data[d]
243   }
244 }
245 pred relation_data_act_excl[d:OutputPin, a:ExecutableNode] {
246   all s:State | s.data[d] implies {
247     not s.hasTokens[a]
248   }
249 }
250 pred relation_data_act_between[d:OutputPin, a,b:ExecutableNode] {
251   all s:State | s.data[d] implies {
252     (some ss:(s+ s.prevs) | ss.hasTokens[a])
253     and
254     (some ss:(s+ s.nexts) | ss.hasTokens[b])
255   }
256 }
257 // ExistenceTimeActivity
258 pred time_activity_before[a:ExecutableNode, t:Int] {
259   all s:State | s.hasTokens[a] implies {
260     s.gc.lt[t]
```

```
  }
}
pred time_activity_after[a:ExecutableNode, t:Int] {
  all s:State | s.hasTokens[a] implies {
    s.gc.gt[t]
  }
}
pred time_activity_between[a:ExecutableNode, t1,t2:Int] {
  all s:State | s.hasTokens[a] implies {
    s.gc.gt[t1] and s.gc.lt[t2]
  }
}
// ExistenceTimeData
pred time_data_before[d:OutputPin, t:Int] {
  all s:State | s.data[d] implies {
    s.gc.lt[t]
  }
}
pred time_data_after[d:OutputPin, t:Int] {
  all s:State | s.data[d] implies {
    s.gc.gt[t]
  }
}
pred time_data_between[d:OutputPin, t1,t2:Int] {
  all s:State | s.data[d] implies {
    s.gc.gt[t1] and s.gc.lt[t2]
  }
}
// ExistenceTimeResource
pred time_resource_before[r:Resource, t:Int] {
  all s:State, a:ExecutableNode | (
    s.hasTokens[a] and r in Use[a]
    ) implies {
      s.gc.lt[t]
    }
}
pred time_resource_after[r:Resource, t:Int] {
  all s:State, a:ExecutableNode | (
    s.hasTokens[a] and r in Use[a]
    ) implies {
      s.gc.gt[t]
    }
}
pred time_resource_between[r:Resource, t1,t2:Int] {
  all s:State, a:ExecutableNode | (
    s.hasTokens[a] and r in Use[a]
    ) implies {
      s.gc.gt[t1] and s.gc.lt[t2]
    }
}
```

A.2 Modules dynamiques

A.2.1 Process.als

```
module Process

open Semantic
open Syntax

fact initTokens {
  init[
    Initial →1 + RequestedOrder →1 ,
    ActivityEdge →0,
    0
  ]
}

one sig BankConnector extends Resource {}{}

one sig InfoIPSW6 extends Info {} {
  use_i   = AcceptPayment →BankConnector
```

```
18   timing_i  = (Action →1) ++ (ReceiveOrder →1 + FillOrder →2 + SendInvoice →1 +
19            MakePayment →1 + AcceptPayment →2 + ShipOrder →3 +
20            CloseOrder →1)
21 }
22
23 one sig RequestedOrder extends ActivityParameterNode {} {
24   parameterMultiplicityElement = MultiOneOne
25   direction = ParameterDirectionIn
26 }
27 one sig Initial extends InitialNode {} {}
28 one sig ReceiveOrder extends OpaqueAction {}{
29   input = InputPinReceiveOrder
30   output = none
31 }
32 one sig DecisionOrder extends DecisionNode {} {}
33 one sig FillOrder extends OpaqueAction {}{
34   input = none
35   output = none
36 }
37 one sig Fork extends ForkNode {} {} //6
38 one sig SendInvoice extends OpaqueAction {}{
39   input = none
40   output = OutputPinInvoice
41 }
42 one sig MakePayment extends OpaqueAction {}{
43   input = InputPinInvoice
44   output = none
45 }
46 one sig AcceptPayment extends OpaqueAction {}{
47   input = none
48   output = none
49 }
50 one sig ShipOrder extends OpaqueAction {}{
51   input = none
52   output = none
53 }
54 one sig Join extends JoinNode {} {}
55 one sig Merge extends MergeNode {} {}
56 one sig CloseOrder extends OpaqueAction {}{
57   input = none
58   output = none
59 }
60 one sig Final extends ActivityFinalNode {} {}
61 one sig OutputPinInvoice extends OutputPin {}{
62   multiplicityElement = MultiOneOne
63 }
64 one sig InputPinInvoice extends InputPin {}{
65   multiplicityElement = MultiOneOne
66 }
67 one sig InputPinReceiveOrder extends InputPin {}{ //17
68   multiplicityElement = MultiOneOne
69 }
70 one sig f1 extends ObjectFlow {} {
71   source = RequestedOrder
72   target = InputPinReceiveOrder
73 }
74 one sig f2 extends ControlFlow {} {
75   source = Initial
76   target = ReceiveOrder
77 }
78 one sig f3 extends ControlFlow {} {
79   source = ReceiveOrder
80   target = DecisionOrder
81 }
82 one sig f4 extends ControlFlow {} {
83   source = DecisionOrder
84   target = FillOrder
85 }
86 one sig f5 extends ControlFlow {} {
87   source = FillOrder
88   target = Fork
89 }
90 one sig f6 extends ControlFlow {} {
91   source = Fork
```

```
  target = SendInvoice
}
one sig f7 extends ControlFlow {} {
  source = SendInvoice
  target = MakePayment
}
one sig f8 extends ObjectFlow {} {
  source = OutputPinInvoice
  target = InputPinInvoice
}
one sig f9 extends ControlFlow {} {
  source = MakePayment
  target = AcceptPayment
}
one sig f10 extends ControlFlow {} {
  source = AcceptPayment
  target = Join
}
one sig f11 extends ControlFlow {} {
  source  = ShipOrder
  target  = Join
}
one sig f12 extends ControlFlow {} {
  source = Join
  target = Merge
}
one sig f13 extends ControlFlow {} {
  source = DecisionOrder
  target = Merge
}
one sig f14 extends ControlFlow {} {
  source = Merge
  target = CloseOrder
}
one sig f15 extends ControlFlow {} {
  source = CloseOrder
  target = Final
}
one sig f16 extends ControlFlow {} {
  source  = Fork
  target  = ShipOrder
}
```

A.2.2 Properties.als

```
module Properties

open Process
open Library

fact {fairness1}

// (1·1) Impossibilite de completion
run Completion for 0 but 38 State, 3 Int expect 1
check {Completion} for 0 but 38 State, 3 Int expect 0
// (1·2) Completion non propre
run CompletionNotClean for 0 but 38 State, 3 Int expect 1
check {CompletionNotClean} for 0 but 38 State, 3 Int expect 0
// (1·3) Transition morte (accessibilite)
check {DeadTransition[ReceiveOrder]} for 0 but 38 State, 3 Int expect 1
check {DeadTransition[FillOrder]} for 0 but 38 State, 3 Int expect 1
check {DeadTransition[SendInvoice]} for 0 but 38 State, 3 Int expect 1
// (2·1) Donnees manquantes
run MissingData for 0 but 38 State, 3 Int expect 1
check {MissingData} for 0 but 38 State, 3 Int expect 0
// (2·2) Donnees redondantes
run RedondantData for 0 but 38 State, 3 Int expect 1
check {RedondantData} for 0 but 38 State, 3 Int expect 0
// (3·1) LackOfResource
run LackOfResource for 0 but 38 State, 3 Int expect 1
check {LackOfResource} for 0 but 38 State, 3 Int expect 0
// (3·2) Utilisation inefficace des ressources·
run {InefficientRessourceUse[BankConnector]} for 0 but 38 State, 3 Int expect 1
check {InefficientRessourceUse[BankConnector]} for 0 but 38 State, 3 Int expect 1
```

```
30 run {InefficientRessourceUse2[BankConnector]} for 0 but 38 State, 3 Int expect 1
31 check {InefficientRessourceUse2[BankConnector]} for 0 but 38 State, 3 Int expect 1
32 // (4·1) Duree du procede·
33 run {TotalTime[12]} for 0 but 38 State, 5 Int expect 1
34 check {TotalTime[12]} for 0 but 38 State, 5 Int expect 0
35 run {TotalTime[6]} for 0 but 38 State, 5 Int expect 1
36 check {TotalTime[6]} for 0 but 38 State, 5 Int expect 1
37 //--- Metiers
38 // Existence
39 check {range[ReceiveOrder,0,1]} for 0 but 38 State, 3 Int expect 0
40 check {range[ReceiveOrder,2,2]} for 0 but 38 State, 3 Int expect 1
41 // Relation
42 check {relation_before[SendInvoice,MakePayment]} for 0 but 38 State, 3 Int expect 0
43 check {relation_before[MakePayment,SendInvoice]} for 0 but 38 State, 3 Int expect 1
44 check {relation_excl[ReceiveOrder,CloseOrder]} for 0 but 38 State, 3 Int expect 0
45 check {relation_excl[ShipOrder,AcceptPayment]} for 0 but 38 State, 3 Int expect 1
46 check {relation_between[MakePayment,SendInvoice, AcceptPayment]} for 0 but 38 State, 3 Int expect 0
47 // RelationData
48 check {existence_data[OutputPinInvoice]} for 0 but 38 State, 3 Int expect 1 //
49 // ExistenceTimeActivity
50 check {time_activity_before[ShipOrder,2]} for 0 but 38 State, 5 Int expect 1
51 check {time_activity_before[ShipOrder,12]} for 0 but 38 State, 5 Int expect 0
52 check {time_data_between[OutputPinInvoice,1,4]} for 0 but 38 State, 5 Int expect 1
53 check {time_data_between[OutputPinInvoice,1,10]} for 0 but 38 State, 5 Int expect 0
```

Index