View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by OpenGrey Repository

Ecole Doctorale : Sciences et Ingénierie

Thèse présentée pour l'obtention du diplôme de

DOCTEUR DE L'INSTITUT TELECOM SUDPARIS

Doctorat délivré conjointement par

l'Université d'Evry Val d'Essonne et l'Université de Sherbrooke

Spécialité : INFORMATIQUE

Par : Fama Diagne

Numéro 2013TELE0018

Preuve de Propriétés Dynamiques en B

Soutenue le 26 septembre 2013 devant le jury composé de :

Directeurs de thèse :

M. Marc Frappier	Professeur, Université de Sherbrooke (Canada)
Mme. Amel Mammar	MCF (HDR), Telecom SudParis (France)

Rapporteurs

M. Jérémie Christian Attiogbe	Professeur, Université de Nantes (France)
M. Vincent Poirriez	Professeur, Université de Valenciennes (France)

Examinateurs

Mme. Régine Laleau	Professeure, Université Paris-Est Créteil (France)
M. Richard St-Denis	Professeur, Université de Sherbrooke (Canada)
M. Samir Tata	Professeur, Telecom SudParis (France)

Remerciements

Je tiens à exprimer ma profonde gratitude et reconnaissance à toutes les personnes qui ont permis la réalisation et l'aboutissement de cette thèse.

Mes premiers remerciements sont adressés à mes directeurs de thèse Mme. Amel Mammar (Télécom Sud Paris) et M.Marc Frappier (Université de Sherbrooke) qui m'ont encadrée et soutenue pendant toutes ces années. Je remercie également M.Jérémie Christian Attiogbe et M.Vincent Poirriez, rapporteurs de cette thèse, pour la minutieuse relecture du manuscrit, les corrections et commentaires. Mes remerciements vont ensuite à Mme. Régine Laleau, M.Richard St-Denis et M.Samir Tata pour avoir accepté de participer au jury de thèse.

Merci à mes collégues Jose Pablo Escobedo, Zahra Movahedi, Rami Sellami, Nour Assy, Olfa Bouchaala, Emna Hachicha, Mohamed Mohamed ainsi que tous les membres de l'équipe SIMBAD de Télécom Sud Paris.

Je retourne toute ma gratitude à mes merveilleux parents Lamine et Yacine Diagne qui tiennent une place immense dans mon coeur. Papa, tu es une vraie école de la vie, je ne cesse d'apprendre tous les jours avec toi. Maman, une femme aussi brave et adorable que toi je n'en connais pas, tu as toujours été présente pour nous (tes enfants), et à aucun moment tu n'as cessé de nous couvrir de ta tendresse. Merci infiniment à vous deux...

Je remercie également ma marraine Aminata Diagne et mon deuxième papa Alioune Diagne qui m'ont accueillie chez eux à Paris comme leur propre fille. Ils ont tout fait pour que je réussisse dans mes études et je leurs dis un grand merci tout en souhaitant une pleine réussite à leur adorable fils Moustapha.

Merci à mes soeurs chéries Ndeye Astou Lamine Diagne, Sénéba Lamine Diagne, Seynabou Lamine Diagne et Mame Diarra Lamine Diagne sans oublier l'adorable petit dernier, l'âme de la famille Madiagne Lamine Diagne.

Merci à mes soeurs et amies Yaye Astou Diagne, Mame Diarra Faye, Yacine Diagne, Ndeye Oumy Thiam, Mame penda Sow, Astou Fall qui ne cessent de m'encourager et de m'appuyer dans toutes mes entreprises. Je vous aime tellement.

A mes fréres et amis Macodou Thiam ,Tachy Laye Diop et Gora Beye qui m'ont soutenue de si prés.

A mon meilleur ami, confident et mari Thierno Diagne qui n'a cessé de m'encourager

durant cette thèse, je te dis merci du fond du coeur. Votre soutien a été et continu d'être une des choses les plus précieuses pour moi.

Je dédie cette thèse à ma chère grand-mére Seynabou Thiam que j'aime plus que tout au monde. Pendant que j'étais en deuxiéme année de thèse à Sherbrooke, elle s'est éteinte sans que je puisse lui dire au revoir. Que le bon DIEU l'accueille dans son paradis le plus exceptionnel.

A la mémoire de ma grand-mére Seynabou Thiam

Table des matières

Table des figures

Chapitre 1

Introduction générale

Sommaire

1.1 Contexte

Un système d'information est un ensemble organisé de ressources permettant d'acquérir, de stocker et de traiter des informations sous forme de données au sein d'une ou plusieurs organisations [25]. Un tel système peut contenir des failles qui conduiraient à des pertes considérables comme des pertes matérielles très importantes. De telles failles peuvent être très pénalisantes surtout dans le cas des systèmes critiques qui ne tolèrent aucun dysfonctionnement, car des vies humaines sont mises en jeu. Ainsi pour avoir la garantie qu'un système ne se comportera pas de façon inattendue ou incorrecte, des vérifications doivent être réalisées dès ses premières phases de conception pour réduire le coût souvent élevé de la correction des bugs détectés tardivement.

Depuis quelques années, la nécessité d'utiliser des méthodes formelles s'est fait ressentir dans le développement des logiciels car ces dernières, fondées sur des notions mathématiques et rigoureuses, permettent de garantir la correction du logiciel ainsi obtenu. Cependant, et bien qu'elles permettent d'assurer l'absence de bugs dans les logiciels, l'utilisation de ces méthodes est souvent jugée coûteuse en termes de ressources matérielles et humaines. De ce fait, les méthodes formelles sont plus utilisées dans le cadre du développement des systèmes critiques. Le présent travail a pour but de contribuer à l'utilisation des méthodes formelles dans le développement de systèmes d'information sûrs.

Dans la littérature, les méthodes formelles de vérification les plus utilisées sont principalement le model checking et la preuve. Le model checking (appelé aussi exploration de modèles), une technique le plus souvent automatique, effectue une énumération exhaustive des différents états du système [14] et par conséquent devient inadapté pour les systèmes à nombre d'états élevé ou infini en raison de l'explosion combinatoire de l'espace des états. Par opposition au model checking, la preuve formelle a l'avantage d'être indépendante de la taille de l'espace des états du système à vérifier. Cependant, l'activité de preuve n'est généralement supportée que par des assistants à la preuve qui peuvent échouer sur des preuves trop complexes. Dans ce cas, l'intervention humaine devient nécessaire pour indiquer à l'assistant les bonnes règles d'inférence et de réécriture à appliquer.

1.2 Problématique

Jusqu'à présent, l'utilisation des méthodes formelles dans le cadre de la conception des systèmes d'information considère essentiellement les propriétés statiques qu'on désigne par le terme d'*invariant*. Un *invariant* est une propriété qui doit être toujours vérifiée sur tous les états possibles du système. De plus, de telles propriétés dépendent d'un seul état du système, *i.e.*, les valeurs des variables du système sont toutes prises au même instant. La vérification de ces propriétés est réalisée sur chaque état du système sans considérer ni son passé ni son futur. D'un point de vue pratique, pour vérifier un invariant, il suffit de s'assurer que ce dernier est satisfait par l'état initial du système et est maintenu par chacune de ses opérations.

Cependant, garantir la qualité et le bon fonctionnement d'un système d'information ne peut pas se restreindre à la vérification de propriétés statiques ; d'autres types de propriétés, dites *dynamiques* ou *temporelles*, doivent être considérées. Contrairement aux propriétés statiques, les propriétés dynamiques dépendent de plusieurs états du système pris à des instants différents et qui doivent donc être considérés lors de la vérification. De telles propriétés permettent par exemple de s'assurer que le système de gestion de prêts d'une bibliothèque offre à un lecteur inscrit la possibilité d'emprunter un livre et de le restituer. De même, un système de vente de billets ne doit pas envoyer le billet à un voyageur avant que ce dernier ait payé sa facture.

Les propriétés dynamiques peuvent être classées selon plusieurs critères. La classification qui suit est celle proposée dans [27] :

1. *Propriété de sûreté* : la sûreté est souvent liée à un ensemble de normes qui garantissent la qualité et l'absence de vices dans un système. Les propriétés de sûreté ont ainsi pour but de garantir que, sous certaines conditions, une situation redoutée ne se produit jamais dans un système. Un exemple de propriété de sûreté sous conditions est la suivante : *tant que l'on ne met pas la clef de contact, la voiture ne démarre pas*. Cette propriété peut être aussi considérée comme une propriété d'absence [3].
2. *Propriété d'équité* : une propriété d'équité permet de spécifier que, sous certaines conditions, un événement aura lieu ou n'aura pas lieu indéfiniment souvent. À titre d'exemples, dans un système à choix non déterministes, une telle propriété assure qu'aucun de ces choix ne sera délaissé indéfiniment. De même dans un système de

traitement de processus par un ordonnanceur, une propriété d'équité permettrait d'exprimer qu'aucun processus ne serait ignoré de manière infinie évitant ainsi le phénomène de famine. Cette notion d'équité sera approfondie dans le Chapitre 2.

3. *Propriété de vivacité* : une propriété de vivacité énonce que, sous certaines conditions, le système se trouvera dans une situation souhaitée. Ce type de propriété permettrait d'exprimer par exemple qu'un programme termine. De même, sur le protocole classique du bit alterné composé d'un émetteur A et d'un récepteur B reliés par une ligne AB, on souhaiterait s'assurer que tout message émis finira par être reçu.

4. *Propriété d'atteignabilité* : une propriété d'atteignabilité permet d'exprimer qu'un état désiré peut être atteint par le système à partir d'un état source donné [12]. Contrairement à la propriété de vivacité, cette propriété n'exige pas que tous les chemins d'exécution possibles du système mènent à cet état. En effet pour que la propriété soit vérifiée, il suffit qu'il existe un chemin qui mène à l'état souhaité.

Notons qu'une propriété de sûreté peut être vue comme une propriété de non atteignabilité. Par conséquent pour prouver qu'une propriété de sûreté P est violée, il suffit de montrer que la propriété d'atteignabilité $\neg P$ est vérifiée en exhibant un chemin qui mènerait à un tel état.

L'objectif de notre travail de thèse est de proposer des approches systématiques de vérification de propriétés dynamiques sur les applications systèmes d'information. En effet, non seulement les approches existantes pour la vérification de telles propriétés ne sont pas adaptées pour le domaine des systèmes d'information mais elles sont aussi très complexes à mettre en pratique. Nos propositions sont fondées sur l'utilisation de la méthode B et de son prouveur associé. L'idée clé est de définir un ensemble d'obligations de preuve nécessaires et suffisantes qui permettent de vérifier de telles propriétés. Nous avons choisi une approche orientée preuve car celle-ci nous semble la plus adaptée pour les applications systèmes d'information connues pour leurs volumes très importants de données reliées par des contraintes d'intégrité de natures diverses. D'autre part, la méthode B, basée essentiellement sur la logique du premier ordre et la théorie des ensembles, est une méthode formelle assez accessible aux utilisateurs et qui est dotée d'outils de preuve assez avancés et surtout gratuits.

1.3 Contributions

Les contributions présentées dans cette thèse sont les suivantes :

- Une approche systématique de génération d'obligations de preuve B permettant de prouver qu'un chemin donné exécuté sur l'état source fait bien évoluer le système pour atteindre l'état cible désiré. Une preuve de correction de ces obligations de preuve a été réalisée (Chapitre 3).
- Une approche de preuve de propriétés d'atteignabilité fondée sur le raffinement de spécification introduit par Morgan [21]. L'idée clé de cette approche est d'exhiber un chemin par une série de raffinements allant de l'état source à l'état cible souhaité (Chapitre 4).
- Une approche systématique de génération d'obligations de preuve B permettant de prouver la propriété de précédence. Cette propriété exprime qu'un état s' ne peut être atteint tant que le système n'a pas atteint auparavant un état donné s. Pour cette approche, une preuve de correction est également fournie (Chapitre 5).
- Un outil développé sous eclipse permettant d'automatiser ces différentes approches et qui facilite donc la mise en pratique de nos contributions (Chapitre 6).

La suite de ce manuscrit décrit, en détails, ces différentes contributions qui sont précédées par un chapitre présentant les travaux existants dans le domaine de la vérification de propriétés dynamiques ainsi qu'une analyse critique qui permet de dégager leurs avantages et limites (Chapitre 2).

Chapitre 2

État de l'art

Sommaire

Dans ce chapitre, nous présentons les principales approches existantes pour la vérification des propriétés dynamiques. Rappelons que les propriétés dynamiques désignent les propriétés dépendant de plusieurs états du système pris à des instants différents. La plupart des travaux existants utilisent les logiques linéaire LTL [22] et arborescente CTL [5]. Nous allons décrire essentiellement quatre approches : les trois premières utilisent les langages formels B et Event B, enfin la dernière approche porte sur la vérification de propriétés dynamiques sur un système FDS (*Fair Discrete System*). Une étude comparative de ces approches pour en tirer les avantages et les limites est également présentée.

2.1 La logique temporelle

La logique temporelle ou modale trouve de nombreuses applications en informatique : traitement du langage naturel, représentation des connaissances [16], spécification de systèmes [19], etc. Cette logique étend la logique conventionnelle (propositionnelle) en introduisant de nouveaux opérateurs qui permettent de référencer plusieurs instants différents de l'exécution d'un système. La logique temporelle permet ainsi d'exprimer des propriétés dynamiques portant sur plusieurs états des traces d'exécution d'un système. Dans la littérature, nous distinguons les deux classes principales de logique temporelle suivantes :

- La logique linéaire comme LTL (*Linear Temporal Logic*) permettant d'exprimer des propriétés portant sur des chemins individuels issus de l'état initial du programme.
- La logique arborescente comme CTL (*Computation Tree Logic*) permettant d'exprimer des propriétés portant sur les arbres d'exécution issus de l'état initial du programme.

Il existe aussi la logique CTL* qui permet de décrire à la fois des propriétés linéaires et arborescentes. La logique CTL* est souvent considérée comme une fusion des logiques LTL et CTL puisque les formules LTL et CTL sont des sous-ensembles des formules CTL*.

2.1.1 La logique LTL

LTL est définie sur un ensemble de variables propositionnelles $\{p, q, \dots\}$, de connecteurs logiques propositionnels habituels $\neg, \wedge, \vee, \rightarrow$ et de connecteurs temporels ($\mathbf{G}$, $\mathbf{F}$, $\mathbf{U}$ et $\mathbf{X}$). La grammaire associée à la logique LTL est comme suit :

– les formules atomiques :

$$\psi, \phi := p \mid q \mid \ldots \mid \mathit{true} \mid \mathit{false}$$

– les formules construites à partir de connecteurs propositionnels :

$$\psi := \phi \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi$$

– les formules construites à partir de connecteurs temporels :
  – $\mathbf{G}\phi$: (**G** comme Globally) spécifie que ϕ est toujours vrai
  – $\mathbf{F}\phi$: (**F** comme Futur) indique que ϕ sera vrai dans le futur
  – $\phi\mathbf{U}\psi$: (**U** comme Until) spécifie que ϕ reste vrai jusqu'à ce que ψ devienne vrai. Cet opérateur impose que le système atteigne un état où ψ devient vrai.
  – $\mathbf{X}\phi$: (**X** comme Next) spécifie que ϕ sera vrai à l'étape suivante de l'exécution

2.1.2 La logique CTL

Contrairement à la logique LTL qui considère une vue linéaire du système, la logique CTL (*Computation Tree Logic*) permet de prendre en compte plusieurs futurs possibles à partir d'un état donné du système. La logique CTL a donc une vision arborescente de l'évolution du système. Elle étend la grammaire de LTL en incluant deux quantificateurs de chemins : l'un universel (A) et un second existentiel (E).

– **A** signifie : "pour tous les chemins possibles de calcul à partir de l'état courant"
– **E** signifie : "pour un certain chemin de calcul à partir de l'état courant"

Les formules CTL sont donc toutes précédées des quantificateurs **A** ou **E** et sont définies par la grammaire suivante :

$$\phi, \psi := \mathbf{EG}\phi \mid \mathbf{EF}\phi \mid \mathbf{E}\phi\mathbf{U}\psi \mid \mathbf{EX}\phi \mid \mathbf{AG}\phi \mid \mathbf{AF}\phi \mid \mathbf{A}\phi\ \mathbf{U}\ \psi \mid \mathbf{AX}\phi$$

avec ϕ, ψ désignant des formules LTL. La logique CTL introduit également quelques notions d'équité afin de ne considérer que les chemins où aucun événement du système ne prend indéfiniment le contrôle. Par exemple, dans un système d'allocation de ressources partagées par plusieurs utilisateurs, on ne voudrait pas s'intéresser aux chemins où un utilisateur garde la ressource indéfiniment. Pour ce faire, CTL introduit deux opérateurs $\mathbf{A_f}$ et $\mathbf{E_f}$ pour préciser qu'on ne considère que les chemins jugés équitables.

2.1.3 La logique CTL*

La logique CTL* est la logique temporelle la plus générale. Comme en LTL et CTL, une formule CTL* est donc construite à partir de propositions atomiques, de connecteurs propositionnels, de connecteurs temporels et de quantificateurs de chemins. Les opérateurs propositionnels et les quantificateurs de chemins restent les mêmes que dans CTL. Ajouté à ces différents connecteurs, CTL* introduit un cinquième connecteur temporel noté **W** et dont la sémantique est comme suit : étant donné deux prédicats p et q, $p\mathbf{W}q$ signifie que p doit rester vrai jusqu'à ce que q soit établi. Contrairement au connecteur **U**, le connecteur **W** n'impose pas que le prédicat q devienne vrai durant l'exécution du programme. En CTL*, on distingue les formules d'états de celles de chemins :

- Une formule d'état ϕ est définie comme suit :
$$\phi := p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid A\psi \mid E\psi$$
avec ψ désignant une formule de chemin.
- Une formule de chemin ψ est quant à elle définie par :
$$\psi := \phi \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathbf{X}\psi \mid \mathbf{F}\psi \mid \mathbf{G}\psi \mid \psi\mathbf{U}\psi \mid \psi\mathbf{W}\psi$$

Notons, qu'une nouvelle notation des connecteurs temporels **F** et **G**, très utilisée dans la littérature, a été introduite dans la logique CTL*. Les connecteurs **F** et **G** sont notés $\Diamond$ et $\Box$ respectivement. Ces nouvelles notations s'expriment comme suit :

- $\Diamond p = \top \mathbf{U} p$ signifie que p sera vrai dans le futur.
- $\Box p = \neg\Diamond\neg p$ signifie que p est toujours vrai. Par conséquent, l'opposé de p ne sera jamais vrai dans le futur.

2.2 Vérification de propriétés de vivacité d'UNITY avec Event B

Dans [4], les auteurs proposent une méthode de vérification de propriétés de vivacité sur un système spécifié en Event B [2]. Pour ce faire, ils se basent sur l'opérateur *LeadsTo* défini dans la logique UNITY [18] en distinguant deux classes de propriétés :

1. Propriétés basiques : étant donnés deux prédicats p et q, la propriété p *ensures* q signifie que le prédicat p reste vérifié tant que le prédicat q ne l'est pas encore. De

plus, elle impose que q soit vérifié dans le futur. Cette propriété se décline en deux cas :

(a) Propriété basique avec progrès minimum : dans ce cas, on exige que le prédicat q soit vérifié immédiatement sur l'état suivant l'état où le prédicat p est vrai. Cette propriété exige également la possibilité d'évolution du système en tout état qui satisfait p, *i.e.*, il doit toujours exister un événement dont la précondition est vraie. Formellement :

$$p \; ensures \; q \equiv (\forall t.(p \wedge \neg q \wedge grd.t \Rightarrow wp.t.q)) \wedge (\exists s.(p \wedge \neg q \Rightarrow grd.s))$$

avec :
- t et s deux événements quelconques du système.
- $grd.t$ et $grd.s$ les gardes de t et s respectivement.
- $wp.t.q$ représente la plus faible précondition où t termine dans q.

(b) Propriété basique avec équité faible : dans ce cas, on n'exige pas que q soit vrai immédiatement sur l'état suivant l'état vérifiant p. Cependant, tous les événements potentiellement exécutables (dont la précondition est vraie) doivent établir q ou préserver p. De plus, il doit toujours exister un événement exécutable qui permet d'établir q. Formellement :

$$p \; ensures \; q \equiv (\forall t.(p \wedge \neg q \Rightarrow wp.t.(p \vee q))) \wedge (\exists s.(p \wedge \neg q \Rightarrow wp.s.q \wedge grd.s))$$

2. Propriétés générales : contrairement aux propriétés de base, les propriétés générales n'imposent pas que le prédicat p reste vrai tant que le prédicat q n'est pas vérifié. La preuve d'une propriété de vivacité générale *"LeadsTo"*, notée $p \rightsquigarrow q$, doit être établie en applicant les règles suivantes :
   - Règle basique (BRL) : si la propriété basique p *ensures* q est vérifiée alors, nous pouvons déduire que la propriété $p \rightsquigarrow q$ est aussi vérifiée :

$$\frac{p \; ensures \; q}{p \rightsquigarrow q}$$

   - Règle de transitivité (TRA) : une propriété générale $p \rightsquigarrow q$ est vérifiée si cette dernière peut être déduite à partir de plusieurs propriétés plus élémentaires en applicant la transitivité :

$$\frac{p \leadsto r \quad r \leadsto q}{p \leadsto q}$$

Afin donc de pouvoir prouver ces mêmes propriétés en Event B, les auteurs proposent une ré-expression des différentes règles suscitées en termes d'obligations de preuve B. La présentation de ces obligations de preuve fait l'objet des sections suivantes.

2.2.1 Preuve des propriétés basiques de vivacité

Pour prouver les propriétés basiques de vivacité, les auteurs ont défini quatre obligations de preuve qui correspondent à la définition formelle, vue précédemment, de ces propriétés. Pour cela, soient I et E l'invariant et l'ensemble des événements du système exprimé en Event B. On considère également la substitution de choix non déterministe F définie par :

$$F \mathrel{\widehat{=}} []_{e \in E} \ e$$

Les obligations de preuve permettant d'établir une propriété basique de vivacité p *ensures* q sont définies comme suit :

- Avec progrès minimum : il suffit d'établir les deux obligations de preuve suivantes :
  $(MP0)$ $I \wedge p \wedge \neg q \Rightarrow [F]q$ [1]
  $(MP1)$ $I \wedge p \wedge \neg q \Rightarrow grd(F)$ [2]
  L'obligation de preuve $(MP0)$ permet de vérifier que pour tout état potentiel du système (vérifiant l'invariant I) et satisfaisant le prédicat $(p \wedge \neg q)$, alors l'exécution de chaque événement, dont la garde est vraie, mène bien à un état qui vérifie q. L'obligation de preuve $(MP1)$, quand à elle, assure que tout état potentiel du système vérifiant le prédicat $(p \wedge \neg q)$ satisfait une des gardes des événements de l'ensemble E. Ceci permet de garantir qu'un des événements du système est exécutable. On en déduit donc que pour tous états σ_i et σ_{i+1} d'une trace d'exécution du système, si $(p \wedge \neg q)$ est vrai à l'état σ_i alors q devient vrai à l'état σ_{i+1}.
- Avec équité faible : il suffit d'établir les deux obligations de preuve suivantes :
  $(WF0)$ $I \wedge p \wedge \neg q \Rightarrow [F](p \vee q))$
  $(WF1)$ $I \wedge p \wedge \neg q \Rightarrow \neg[F]\neg q$

1. $[F]q \mathrel{\widehat{=}} [e_1[] \ldots []e_n]q = \forall e.(e \in E \wedge grd(e) \Rightarrow [e]q)$
2. $grd(F) \mathrel{\widehat{=}} grd(e_1[] \ldots []e_n) = \bigvee_{e \in E} grd(e)$

L'obligation de preuve ($WF0$) assure que pour tout état potentiel du système (vérifiant l'invariant I) et satisfaisant le prédicat $(p \wedge \neg q)$, tout événement exécutable préserve p ou établit q. De même, l'obligation de preuve ($WF1$) vérifie qu'il existe au moins un événement exécutable qui établit q. Ces deux obligations de preuve assurent donc que si le système est dans un état qui satisfait p alors il évoluera pour atteindre un état où q devient vrai. En termes de séquence d'exécution, si un état σ_i satisfait $(p \wedge \neg q)$, alors il existe un état σ_j, avec $(j > i)$, qui vérifie q. De plus, pour tout état σ_k, avec $(i \leq k \leq j)$, le prédicat $(p \wedge \neg q)$ est vrai et la garde d'un événement, au moins, qui établit q est vraie.

2.2.2 Preuve de propriétés générales de vivacité

Rappelons qu'une propriété générale de vivacité p *LeadsTo* q, notée $p \rightsquigarrow q$, signifie qu'à partir de tout état s vérifiant p, il est possible d'atteindre un état s' qui vérifie q sans exiger pour autant que p demeure vrai tant qu'on n'a pas atteint s'. Pour prouver une telle propriété, les auteurs ont repris le théorème suivant, défini dans la logique UNITY, et l'ont adapté à Event B :

$$\frac{\forall v.(p \wedge V = v \rightsquigarrow (p \wedge (V < v)) \vee q)}{p \rightsquigarrow q}$$

Ce théorème signifie que pour prouver la propriété générale de vivacité $p \rightsquigarrow q$ est vraie, il suffit d'exhiber une expression entière décroissante et strictement positive V appelée variant et de prouver que chaque transition du système décroît ce variant, tout en préservant p, ou établit q. Pour prouver la prémisse de ce théorème, on utilise la propriété basique de vivacité $((p \wedge V = v)\ ensures((p \wedge (V < v)) \vee q))$ qu'on prouve grâce aux différentes règles présentées dans la section précédente.

2.2.3 Exemple illustratif

Dans cette section, nous allons illustrer l'approche proposée sur un système de gestion de processus demandant une ressource critique qui est le processeur. Nous supposons que le nombre de processeurs est inférieur au nombre de processus en attente. à la création, un processus se trouve dans l'état `NT` (Nouvelle Tâche) et doit donc s'insérer dans une file, à

taille limitée, si cette dernière n'est pas pleine. Dans ce cas, le processus passe dans l'état `EF` (En File) en attendant qu'un processeur se libère. Dès qu'un processeur devient libre, ce dernier est alloué, par l'ordonnanceur, à un processus en attente et qui passera donc à l'état `EL` (Élu). Sur ce système de gestion de processus, on souhaiterait s'assurer qu'un processus en attente dans la file finira toujours par être élu.

Afin de modéliser ce système en Event B, on considère les ensembles et variables suivants :

- l'ensemble abstrait `PROCS` pour représenter l'ensemble de tous les processus possibles, et l'ensemble énuméré `ETATS` pour modéliser les trois états possibles d'un processus :

  $\texttt{PROCS}$;

  $\texttt{ETATS} = \{\texttt{NT}, \texttt{EF}, \texttt{EL}\}$
- les variables `Procs` et `Etat` représentant respectivement l'ensemble des processus existants à tout instant et l'état de chaque processus :

  $$\texttt{Procs} \subseteq \texttt{PROCS}$$

  $$\texttt{Etat} \in \texttt{Procs} \rightarrow \texttt{ETATS}$$
- la variable `File` représentant, à tout instant, l'état de la file :

  $$\texttt{File} \in \mathbf{seq}(\texttt{Procs})$$

Pour faciliter l'écriture de la spécification Event B du système, nous considérons aussi les définitions suivantes :

- `New` pour représenter l'ensemble des processus nouveaux :

  $$\texttt{New} \mathrel{\widehat{=}} \texttt{Etat}^{-1}[\{\texttt{NT}\}]$$
- `EnFile` pour représenter l'ensemble des processus présents dans la file :

  $$\texttt{EnFile} \mathrel{\widehat{=}} \texttt{Etat}^{-1}[\{\texttt{EF}\}]$$

  `Elu` pour représenter l'ensemble des processus élus :

  $$\texttt{Elu} \mathrel{\widehat{=}} \texttt{Etat}^{-1}[\{\texttt{EL}\}]$$

Le système de gestion de processus est défini en Event B comme suit :

MACHINE

GestionProcessus

DEFINITIONS

$\texttt{EnFile} \mathrel{\widehat{=}} \texttt{Etat}^{-1}[\{\texttt{EF}\}]$;

$\texttt{New} \mathrel{\widehat{=}} \texttt{Etat}^{-1}[\{\texttt{NT}\}]$;

$\texttt{Elu} \mathrel{\widehat{=}} \texttt{Etat}^{-1}[\{\texttt{EL}\}]$

SETS

$\texttt{PROCS}$;

$\texttt{ETATS} = \{\texttt{NT}, \texttt{EF}, \texttt{EL}\}$

CONCRETE_CONSTANTS

$\texttt{TAIL}$

PROPERTIES

$\texttt{TAIL} \in \texttt{NAT}$

ABSTRACT_VARIABLES

$\texttt{Procs}, \texttt{Etat}, \texttt{File}$

INVARIANT

$\texttt{Procs} \subseteq \texttt{PROCS} \wedge$

$\texttt{Etat} \in \texttt{Procs} \rightarrow \texttt{ETATS} \wedge$

$\texttt{File} \in \mathbf{seq}(\texttt{Procs})$

INITIALISATION

$\texttt{Etat} := \emptyset \;||\; \texttt{Procs} :(\texttt{Procs} \subseteq \texttt{PROCS}) \;||\; \texttt{File} := \emptyset$

EVENTS

$\texttt{eluEven} \mathrel{\widehat{=}}$

ANY $\texttt{pr}$ **WHERE** $\texttt{pr} \in \texttt{Procs} \wedge \texttt{pr} \in \texttt{EnFile} \wedge \texttt{File}(1) = \texttt{pr}$

THEN

$\texttt{Etat}(\texttt{pr}) := \texttt{EL} \;||$

$\texttt{File} := \texttt{file} \downarrow 1$

END

END;

$\texttt{fileEven} \mathrel{\widehat{=}}$

ANY $\texttt{pr}$ **WHERE** $\texttt{pr} \in \texttt{procs} \wedge \texttt{pr} \in \texttt{New} \wedge \mathbf{size}(\texttt{File}) < \texttt{TAIL}$

THEN

$\texttt{Etat}(\texttt{pr}) := \texttt{EF} \;||$

$\texttt{File} := \texttt{File} \leftarrow \texttt{pr}$

END

END;

$\texttt{supEven} \mathrel{\widehat{=}}$

ANY pr **WHERE** pr $\in$ Procs $\wedge$ pr $\in$ Elu
THEN
Procs := Procs - {pr} ||
Etat := {pr} $\lhd$ Etat
END
END
END

Sur cette spécification Event B, nous allons illustrer l'approche présentée pour prouver deux propriétés, l'une sous l'hypothèse du progrès minimum et une seconde sous l'équité.

2.2.3.1 Vérification sous l'hypothèse du progrès minimum

Sur le système de gestion de processus précédent, nous souhaitons vérifier qu'un processus se trouvant en tête de la file se verra allouer un processeur. Une telle propriété est exprimée comme suit :

$$(p \in \texttt{EnFile} \wedge \texttt{File}(1) = p) \rightsquigarrow p \in \texttt{Elu} \tag{2.1}$$

En appliquant la règle BRL, prouver la propriété (2.1) revient à établir la propriété suivante :

$$(p \in \texttt{EnFile} \wedge \texttt{File}(1) = p)\ \mathit{ensures}\ (p \in \texttt{Elu}) \tag{2.2}$$

Nous devons donc prouver les deux obligations de preuve suivantes :

$MP0$:

$$(\texttt{p} \in \texttt{EnFile}) \wedge (\texttt{File}(1)=\texttt{p}) \wedge (\neg\ \texttt{p} \in \texttt{Elu})$$
$$\Rightarrow$$
$$[\texttt{eluEven}\ []\ \texttt{fileEven}\ []\ \texttt{supEven}]\ (\texttt{p} \in \texttt{Elu})$$

$MP1$:

$$(\texttt{p} \in \texttt{EnFile}) \wedge (\texttt{File}(1)=\texttt{p}) \wedge (\neg\ \texttt{p} \in \texttt{Elu})$$
$$\Rightarrow$$
$$\texttt{grd}(\texttt{eluEven}\ []\ \texttt{fileEven}[]\ \texttt{supEven})$$

L'obligation de preuve ($MP0$) spécifie que si un processus p est en tête de la file, alors l'exécution de l'un des événements de la spécification rend ce processus élu. Ceci est vrai

pour l'événement `eluEven` qui permet donc de prouver ($MP0$).
Prouvons à présent l'obligation de preuve ($MP1$) qui exprime qu'il doit exister au moins un événement du système qui soit exécutable. Cette obligation de preuve est évidente pour l'événement `eluEven` dont la garde est vraie et qui prouve aussi ($MP0$).
à présent, nous allons prouver la propriété générale de vivacité suivante :

$$\texttt{p} \in \texttt{EnFile} \leadsto \texttt{p} \in \texttt{Elu} \tag{2.3}$$

Cette propriété garantit que chaque processus en attente dans la file finira par être élu. Pour établir une telle propriété, considérons *V(p)* une variable qui donne la position du processus `p` dans la file. Cette variable est définie par :

$$\texttt{V(p)}=\texttt{File}^{-1}\texttt{(p)}$$

En applicant la règle citée précédemment, nous devons établir :

$$\forall \texttt{n}.((\texttt{p} \in \texttt{EnFile} \wedge \texttt{V(p)=n}) \leadsto ((\texttt{p} \in \texttt{EnFile} \wedge \texttt{V(p)<\ n}) \vee \texttt{p} \in \texttt{Elu})) \tag{2.4}$$

En utilisation la règle BRL, cela revient à prouver :

$$(\texttt{p} \in \texttt{EnFile} \wedge \texttt{V(p)=n})\ \textit{ensures}\ (((\texttt{p} \in \texttt{EnFile}) \wedge (\texttt{V(p)<\ n})) \vee (\texttt{p} \in \texttt{Elu})) \tag{2.5}$$

pour chaque processus `p` et entier n. Pour cela, nous utilisons les obligations de preuve $MP0$ et $MP1$ suivantes :

$MP0$:

$$\texttt{p} \in \texttt{EnFile} \wedge \texttt{V(p)=n}$$
$$\Rightarrow$$
$$[\texttt{eluEven}][]\texttt{fileEven}[]\texttt{supEven}]((\texttt{p}\in \texttt{EnFile} \wedge \texttt{V(p)<\ n}) \vee \texttt{p}\in \texttt{Elu})$$

$MP1$:

$$\texttt{p} \in \texttt{EnFile} \wedge \texttt{V(p)=n}$$
$$\Rightarrow$$
$$\texttt{grd(eluEven)} \vee \texttt{grd(supEven)} \vee \texttt{grd(fileEven)}$$

Pour prouver $MP0$, il suffit de montrer qu'après chaque extraction du premier élément de la file, les positions des processus dans la file décroissent. Ce qui se déduit de la définition de la variable `File` qui est une séquence de processus et par conséquent $MP0$ est prouvée. Pour prouver $MP1$, l'invariant suivant a été défini :

$$\textbf{size}(\texttt{File}) \neq 0 \Rightarrow \exists\ \texttt{p}.(\texttt{p} \in \texttt{EnFile} \wedge \texttt{File(1)=p})$$

Cet invariant exprime que si la file n'est pas vide, alors il existe toujours un processus qui est en tête. Dans ce cas, l'événement `eluEvent` devient donc exécutable. Par conséquent, tant qu'on a (`V(p)=n`) avec $n > 0$ (ce qui veut dire que la file n'est pas vide), il existe un événement dont la précondition est toujours vérifiée si `(V(p)=n)`. L'obligation de preuve $MP1$ est donc vraie.

2.2.3.2 Vérification sous l'hypothèse d'équité faible

Considérons toujours le même système de gestion de processus défini précédemment sous l'hypothèse d'équité faible. L'objectif est de montrer comment une telle hypothèse suffit pour prouver la propriété :

$$(p \in EnFile) \rightsquigarrow (p \in Elu) \tag{2.6}$$

Pour ce faire, il suffit de prouver :

$$(\texttt{p} \in \texttt{EnFile})\ \textit{ensures}\ (\texttt{p} \in \texttt{Elu}) \tag{2.7}$$

en appliquant la règle basique `BRL`. Pour prouver la propriété (2.7), il faut démontrer que les deux obligations de preuve $WF0$ et $WF1$ suivantes sont vérifiées :

$WF0$:

$(\texttt{p} \in \texttt{EnFile} \wedge \neg\ \texttt{p} \in \texttt{Elu}) \Rightarrow [\texttt{fileEven}\ []\ \texttt{eluEven}[]\ \texttt{supEven}](\texttt{p} \in \texttt{EnFile} \vee \texttt{p} \in \texttt{Elu})$

$WF1$:

$(\texttt{p} \in \texttt{EnFile} \wedge \neg\ \texttt{p} \in \texttt{Elu}) \Rightarrow \neg[\ \texttt{fileEven}\ []\ \texttt{eluEven}\ []\ \texttt{supEven}\ ]\ \neg(\texttt{p} \in \texttt{Elu})$

D'après ($MP0$), on sait que ($\texttt{p} \in \texttt{EnFile} \Rightarrow [\texttt{fileEven}\ []\ \texttt{eluEven}\ []\ \texttt{supEven}]\ (\texttt{p} \in \texttt{EnFile} \vee \texttt{p} \in \texttt{Elu})$). Nous en déduisons donc que l'obligation de preuve ($WF0$) est vérifiée. Pour prouver ($WF1$), il suffit de prouver que ($\texttt{p} \in \texttt{EnFile} \Rightarrow [\ \texttt{eluEven}\ ]\ \texttt{p} \in \texttt{Elu}$). Ce prédicat est équivalent à :

$$\mathtt{p} \in \mathtt{EnFile} \Rightarrow \exists\, \mathtt{x}.(\mathtt{x} \in \mathtt{Procs} \wedge \mathtt{x} \in \mathtt{EnFile} \wedge \mathtt{p} \in ((\mathtt{Etat} \lhd\!+ \{\mathtt{x} \mapsto \mathtt{EL}\})^{-1}[\{\mathtt{EL}\}]))$$

Ce prédicat est vrai pour (x=p), l'obligation de preuve ($WF1$) est donc vérifiée.

2.3 Vérification de propriétés dynamiques en Event B

Dans [28], les auteurs proposent un ensemble de règles de vérification de propriétés dynamiques sur un système exprimé en Event B, et plus précisément des propriétés d'existence et de progression. Pour ce faire, ils ont défini un ensemble de règles d'inférence pour prouver des propriétés plus élémentaires comme la convergence, la divergence et le non-blocage.

2.3.1 Preuve de propriétés élémentaires

Dans cette section, nous décrivons les trois propriétés élémentaires qui ont été définies et qui permettent la preuve de propriétés plus élaborées comme l'existence et la progression.

2.3.1.1 Propriété LeadsTo

Soient P_1 et P_2 deux prédicats quelconques. Un système S vérifie la propriété "P_1 $LeadsTo$[3] P_2" ssi tout état du système qui satisfait P_1 est suivi d'un état qui vérifie P_2. Formellement, pour tout événement du système dont la garde et la postcondition sont G et $Post_{v,v'}$ respectivement :

$$P_1 \wedge G \wedge Post_{v,v'} \Rightarrow P_2$$

avec les prédicats P_1 et P_2 dépendant des variables v et v' respectivement. La notation : $S \vdash P_1 \curvearrowright P_2$ représente l'expression P_1 $LeadsTo$ P_2.

2.3.1.2 Propriété de convergence

La convergence d'un système vers un prédicat P signifie qu'il n'existe pas de traces d'exécution qui se terminerait avec une suite infinie d'états qui vérifie le prédicat P. Pour

3. Notons que cette propriété, bien que portant portant le même nom, est différente de celle présentée à la section 2.2.

montrer qu'un système converge vers le prédicat P, il suffit d'exhiber une expression entière $V(v)$, dite variant et définie sur les variables v du système, et d'établir les obligations de preuve suivantes pour tout événement du système dont la garde et la postcondition sont G et $Post_{v,v'}$ respectivement :

1. l'expression $V(v)$ désigne toujours une expression entière :

$$P \wedge G \Rightarrow V(v) \in \mathbb{N}$$

2. l'exécution de tout événement décroît le variant :

$$P \wedge G \wedge Post_{v,v'} \Rightarrow V(v') < V(v)$$

La notation : $S \vdash\downarrow P$ indique que le système converge vers P.

2.3.1.3 Propriété de divergence

La divergence d'un système sur un prédicat P signifie que toutes ses traces d'exécution se terminent avec une suite infinie d'états qui vérifie le prédicat P. Pour montrer qu'un système diverge sur le prédicat P, il suffit d'exhiber une expression entière $V(v)$, dite variant et définie sur les variables v du système, et d'établir les obligations de preuve suivantes pour tout événement du système dont la garde et la postcondition sont G et $Post_{v,v'}$ respectivement :

1. l'expression $V(v)$ désigne toujours une expression entière :

$$\neg P \wedge G \Rightarrow V(v) \in \mathbb{N}$$

2. l'exécution de tout événement sur un état qui vérifie $\neg P$ décroît le variant :

$$\neg P \wedge G \wedge Post_{v,v'} \Rightarrow V(v') < V(v)$$

3. l'exécution de tout événement sur un état qui vérifie P ne fait pas accroitre le variant :

$$P \wedge G \wedge Post_{v,v'} \wedge V(v) \in \mathbb{N} \Rightarrow V(v') \leq V(v)$$

En effet, comme le variant doit toujours rester positif et décroît à chaque exécution d'événement, alors on est sûr d'atteindre un état où P est vérifié. La notation : $S \vdash\nearrow P$ indique que le système diverge vers P.

2.3.1.4 Propriété de non-blocage

Le non-blocage d'un système sur un prédicat P signifie qu'il existe toujours un événement du système dont la garde est vraie :

$$P \Rightarrow \bigvee_i G_i$$

avec G_i désignant la garde d'un événement e_i du système. La notation : $S \vdash_{\circlearrowleft} P$ indique que le système n'est pas bloqué sur P.

2.3.2 Propriété d'existence

Une propriété d'existence spécifie qu'une situation, désignée par le prédicat P, se produira toujours dans le futur. Cette propriété est notée $(\Box\Diamond P)$. L'idée intuitive pour établir une telle propriété est de montrer que les traces du système S ne se terminent pas par une séquence infinie d'états vérifiant $\neg P$. Formellement

$$\frac{S \vdash_{\downarrow} \neg P \qquad S \vdash_{\circlearrowleft} \neg P}{S \vdash \Box\Diamond P}$$

2.3.3 Propriété de progression

Cette propriété exprime que si le système S est dans un état satisfaisant un prédicat P_1, alors il évoluera pour atteindre un état qui vérifie un prédicat P_2 : $S \vdash \Box(P_1 \Rightarrow \Diamond P_2)$. Pour établir une telle propriété, les auteurs ont défini une règle intermédiaire qui permet de prouver la propriété $S \vdash \Box(P_1 \Rightarrow P_1 \mathbf{U} P_2)$ qui exprime que si le système est dans un état qui satisfait P_1 alors ce prédicat reste vrai tant qu'un état satisfaisant P_2 n'est pas encore atteint. De plus, cette propriété exige qu'un tel état soit atteignable. Pour prouver cette propriété, la règle suivante a été définie :

$$\frac{S \vdash (P_1 \wedge \neg P_2) \curvearrowright (P_1 \vee P_2) \qquad S \vdash \Box\Diamond(\neg P_1 \vee P_2)}{S \vdash \Box(P_1 \Rightarrow (P_1 \mathbf{U} P_2))}$$

Cette règle est ensuite utilisée pour prouver la propriété de progression :

$$\frac{M \vdash \Box(P_1 \wedge \neg P_2) \Rightarrow P_3) \qquad M \vdash \Box(P_3 \Rightarrow P_3 \mathbf{U} P_2))}{M \vdash \Box(P_1 \Rightarrow \Diamond P_2)}$$

Cette dernière règle signifie qu'on doit exhiber un prédicat P_3 (invariant) qui doit rester vrai tant que P_2 n'est pas encore vérifié.

2.3.4 Application à un cas d'étude

Pour illustrer cette approche, soit le modèle du lecteur/rédacteur représenté par deux variables l (pour lecteur) et r (pour rédacteur). Le tampon partagé est modélisé par l'intervalle $[l+1,\ r]$ et dénote les données écrites mais pas encore lues. On pose comme hypothèse que la taille du tampon est de 3 :

$$0 \leq r - l \leq 3$$

nous obtenons alors la spécification B suivante :

MACHINE
LectRedac
ABSTRACT_VARIABLES
`l`, `r`
INVARIANT
`l` $\in$ `NAT` $\wedge$ `r` $\in$ `NAT` $\wedge$
`r-l` $\geq 0 \wedge$ `r-l` ≤ 3
EVENTS
`Lire` $\widehat{=}$
WHEN `l < r` **THEN**
`l` := `l+1`
END ;
`Ecrire` $\widehat{=}$
WHEN `r < l+3` **THEN**
`r := r+1`
END
END

Sur ce système on se propose de prouver une propriété d'existence et une seconde de progression.

Propriété d'existence On souhaiterait prouver sur le système *LectRedac* précédent la propriété d'existence suivante : $\Box\Diamond(l \geq L)$ avec L désignant un entier naturel positif. Pour ce faire, nous devons donc établir :

$$\begin{array}{ll} \mathit{LectRedac} \vdash_{\downarrow} (l < L) & \text{(a)} \\ \mathit{LectRedac} \vdash_{\circlearrowleft} (l < L) & \text{(b)} \end{array}$$

Preuve de (a) : pour établir (a), il suffit de choisir comme variant $V = (L - l) + (L + 3 - r)$. En effet, ce variant décroît à chaque lecture ou écriture et est toujours positif ou nul.

Preuve de (b) : on doit prouver la formule $(l < L \Rightarrow l < r \vee r < l + 3)$. Cette formule se déduit directement de l'invariant du système.

Propriété de progression On se propose de prouver que toute donnée écrite sera lue dans le futur $\Box(r = L \Rightarrow \Diamond l = L)$. Pour ce faire, nous devons établir :

$$\begin{array}{ll} \mathit{LectRedac} \vdash \Box((r = L \wedge l \neq L) \Rightarrow l < L) & \textbf{(c)} \\ \mathit{LectRedac} \vdash \Box(l < L \Rightarrow ((l < L)\mathbf{U}(l = L))) & \textbf{(d)} \end{array}$$

Preuve de (c) : pour prouver cette formule, la règle suivante est utilisée avec $(J = r - l \geq 0)$ et I désignant l'invariant de la machine B :

$$\frac{\vdash J \Rightarrow I \qquad S \vdash \Box J}{S \vdash \Box I}$$

Preuve de (d) : pour établir cette formule, nous devons prouver :

$$\begin{array}{ll} \mathit{LectRedac} \vdash \Box(l < L \wedge l \neq L) \curvearrowright (l < L \vee l = L) & \textbf{(d.1)} \\ \mathit{LectRedac} \vdash \Box\Diamond(l \geq L \vee l = L) & \textbf{(d.2)} \end{array}$$

Preuve de $(d.1)$: cette formule est équivalente à $\mathit{LectRedac} \vdash \Box(l < L) \curvearrowright (l \leq L)$. Il faut donc prouver que les événements d'écriture et de lecture conduisent d'un état satisfaisant $(r < L)$ vers un état qui satisfait $(r \leqslant L)$, ce qui est trivialement prouvé.

Preuve de $(d.2)$: cette formule est équivalente à $\mathit{LectRedac} \vdash \Box\Diamond(l \geq L)$ qu'on a déjà prouvée en tant que propriété d'existence.

2.4 Vérification de propriétés dynamiques en Event B étendue avec PLTL

La majorité des propriétés des systèmes réactifs sont dynamiques. Ainsi, quelques travaux ont porté sur l'extension de la méthode B dans le but d'exprimer et de vérifier ce type de propriétés. Dans [26], les auteurs proposent une approche de spécification et de vérification de propriétés dynamiques des systèmes réactifs qui utilise la méthode B étendue par la logique temporelle linéaire PLTL (Past Linear Temporal Logic, version antérieure de LTL). La logique temporelle PLTL définit les opérateurs temporels du passé comme *Previous*, *Since*, *Always* et *Eventually* que nous introduisons dans la suite.

Dans un premier temps, les auteurs montrent comment les propriétés PLTL sont préservées par le raffinement B, puis proposent un processus de reformulation, en termes de modèles, qui consiste à définir de nouvelles propriétés à la suite du raffinement et de vérifier celles-ci en utilisant un ensemble de théorèmes qu'ils proposent également. Ainsi, chaque modèle relie une propriété abstraite à une propriété définie sur une spécification plus concrète. Cette redéfinition a pour but de mieux prendre en compte les détails du système introduits durant les différentes phases de raffinement de la spécification initiale.

Les propriétés dynamiques auxquelles s'intéressent les auteurs portent sur le passé du système. En effet, les propriétés dynamiques ne se limitent pas à ce qui peut se produire dans le futur d'une exécution mais peuvent également concerner des événements survenus dans le passé. à titre d'exemple, la propriété $\Box(ack \Rightarrow \ominus\ mess)$ exprime qu'un système ne transmet un acquittement que s'il a reçu un message précédemment. Pour un prédicat p, l'expression $\ominus\ p$ signifie que p était vrai précédemment. Avec un opérateur exprimant le futur comme *Until*(U), cette propriété peut également s'exprimer par : $\Box\neg ack \vee (\neg ack) U mess$. Comme l'opérateur **U** exige qu'un message soit reçu dans le futur, la première partie de cette expression exprime le cas où un message n'est jamais reçu. Dans ce cas, aucun acquittement ne doit être envoyé. La seconde partie spécifie qu'un acquittement n'est envoyé que si un message a été déjà reçu. Nous pouvons remarquer que ce type de propriétés est plus simple quand il est exprimé avec les opérateurs du passé comme l'opérateur *previous* ($\ominus$). D'autres opérateurs faisant référence au passé seront présentés dans la suite de cette section.

2.4.1 Reformulation de propriétés

En B, le mécanisme de raffinement permet de passer d'un modèle abstrait vers un modèle plus concret. Dans cette approche, les auteurs proposent un raffinement conjoint de la spécification abstraite et des propriétés. Le raffinement des propriétés, appelé ici reformulation, est représenté par un modèle $\sharp\sharp'$ avec $P1 : \Box(p_1 \Rightarrow \sharp q_1)$ représentant la propriété abstraite, $P2 : \Box(p_2 \Rightarrow \sharp' q_2)$ la propriété reformulée et $\sharp, \sharp' \in \{\ominus, \diamond, S\}$. Par conséquent, pour prouver que le raffinement est correct, on doit s'assurer que : si une propriété P_1 est vérifiée sur la spécification abstraite d'un système TS_1 alors la propriété P_2 qui raffine P_1 est aussi vérifiée sur le système TS_2 qui raffine TS_1. Pour ce faire, les auteurs introduisent un ensemble de définitions :

Définition 1 *(Propositions atomiques) Soit $V = \{x_1, \ldots, x_n\}$ l'ensemble des variables du système. L'ensemble des propositions atomiques sur V, noté AP_V, est défini comme suit :*

$$AP_V = \{ap_0, ap_1, \ldots, ap_n\}$$

avec ap_k désignant une expression de la forme $(x_i = d_j)$ où d_j appartient au domaine de x_i, $Dom(x_i)$.

Définition 2 *(Propositions d'état) Soit $V = \{x_1, \ldots, x_n\}$ l'ensemble des variables du système. L'ensemble des propositions d'état sur V, noté SP_V, est défini comme suit :*

$$SP_V = \{sp_0, sp_1, \ldots, sp_n\}$$

avec sp_k désignant l'une des expressions suivantes :

$$sp ::= ap \mid sp \vee sp \mid sp \wedge sp \mid \neg sp$$

et $ap \in AP_V$.

Définition 3 *(Système de transitions) Un système de transitions est un quintuplet $TS = (Q, Q_0, L, T, l)$ avec*

- *Q représentant l'ensemble des états du système.*
- *Q_0 représentant l'ensemble des états initiaux : $Q_0 \subseteq Q$.*
- *$L = \{a_0, a_1, \ldots, a_n\}$ est l'ensemble des étiquettes appelées actions.*
- *$T \in P(Q \times L \times Q)$ dénotant la relation de transition.*

– $l : Q \rightarrow P(SP_V)$ *est une fonction qui associe à chaque état* s *une proposition d'état de la forme :*

$$l(s) = \bigwedge_{i=1}^{n}(x_i = d_j) \text{ avec } d_j \in Dom(x_i)$$

Dans la suite, une transition (s_1, a, s_2) est notée $s_1 \xrightarrow{a} s_2$.

Définition 4 *(Satisfaction d'une proposition d'état) Soit un système de transitions* $TS1 = (Q, Q_0, L, T, l)$. *Un état* s *satisfait une proposition d'état* sp *ssi* $l(s) \Rightarrow sp$.

Définition 5 *(Notion de chemin) Soient un système de transitions* $TS = (Q, Q_0, L, T, l)$ *et* $\sigma = (s_0, s_1, \ldots, s_n)$ *une séquence d'états de ce système.* σ *est un chemin ssi :*

$$\forall i.(i \geq 0 \Rightarrow \exists a.(a \in L \wedge s_i \xrightarrow{a} s_{i+1} \in T))$$

Intuitivement, il doit exister une action capable de faire passer le système d'un état s_i à l'état suivant s_{i+1}.

Définition 6 *(Invariant de collage)*
Soient deux systèmes de transitions $TS_1 = (Q_1, Q_{01}, L_1, T_1, l_l)$ *et* $TS_2 = (Q_2, Q_{02}, L_2, T_2, l_2)$ *tels que* TS_2 *raffine* TS_1. *L'état* s_2, *de* TS_2, *est relié à l'état* s_1, *de* TS_1, *par l'invariant de collage* $I_{1,2}$ *ssi* $(l_2(s_2) \wedge I_{1,2}) \Rightarrow l_1(s_1)$.

Définition 7 *(Opérateurs temporels du passé) Soient* σ *une séquence d'exécution et* ϕ, ψ *deux formules PLTL. Les quatre opérateurs temporels de la logique PLTL sont définis comme suit où la notation* $((\sigma, i) \models \phi)$ *spécifie que la propriété* ϕ *est satisfaite à l'état* i *de la séquence* σ *:*

– *Previous noté* $\ominus$ *et défini par :*

$$(\sigma, i) \models \ominus \phi$$
$$\Leftrightarrow$$
$$i > 0 \wedge (\sigma, i-1) \models \phi$$

Cet opérateur signifie que ϕ *était vrai à l'état précédent* i *c'est à dire à l'état* $(i-1)$.

– *Since noté* S *et défini par :*

$$(\sigma, i) \models \phi S \psi$$
$$\Leftrightarrow$$
$$\exists j.(j \leq i \wedge (\sigma, j) \models \psi \wedge \forall k.(j < k \leq i \Rightarrow (\sigma, k) \models \phi))$$

Cet opérateur signifie que depuis l'état où ψ est vrai, la formule ϕ est restée vraie.

– *Always noté $\boxminus$ et défini par :*

$$(\sigma, i) \models \boxminus \phi$$
$$\Leftrightarrow$$
$$\forall j.(j \leqslant i \wedge (\sigma, j) \models \phi)$$

Cet opérateur signifie que ϕ a toujours été vrai dans le passé jusqu'à l'état i.

– *Eventually noté $\diamond$ et défini par :*

$$(\sigma, i) \models \diamond \phi$$
$$\Leftrightarrow$$
$$\exists j.(j \leq i \wedge (\sigma, j) \models \phi)$$

Cet opérateur signifie que ϕ a été vrai précédemment. Autrement dit, il a existé un état précédent l'état actuel pour lequel ϕ était satisfaite.

Les auteurs ont prouvé également que les propriétés temporelles PLTL exprimées avec ces opérateurs sont préservées par le raffinement B.

2.4.2 Vérification d'une propriété reformulée

De manière générale, pour vérifier une propriété de sécurité sur un modèle, il suffit de la prouver sur le modèle abstrait et de garantir que cette propriété est préservée par le mécanisme de raffinement. En outre, étant donné que le raffinement ajoute au système de nouveaux détails (événements, variables, etc.), les variables du niveau abstrait et celles du niveau raffiné doivent être reliées par un invariant de collage. L'approche présentée dans cette section consiste à définir de nouvelles propriétés à partir des propriétés abstraites et à vérifier le respect de ces nouvelles propriétés sur la spécification raffinée à l'aide des théorèmes qu'ils ont eux-mêmes définis. La reformulation des propriétés abstraites a pour but de générer de nouvelles propriétés qui sont plus faciles à prouver sur la spécification concrète.

Pour vérifier les propriétés reformulées, des patterns de reformulation ont été proposés. Pour chaque pattern, des théorèmes sont définis afin d'assurer la correction de la reformulation appliquée. Cette section présente quelques uns de ces patterns ainsi que les théorèmes qui leur sont associés. Pour tous ces théorèmes, on considère TS_1 et TS_2 deux systèmes de transitions tels que TS_2 raffine TS_1 par l'invariant de collage $I_{1,2}$ (noté $TS_1 \sqsubseteq TS_2$), et ϕ_1

et ϕ_2 deux formules PLTL qui doivent être vérifiées respectivement sur les systèmes TS_1 et TS_2.

Théorème 1 *(Pattern $\ominus S$) Si $TS_1 \models \phi_1$ et les formules ϕ_1 et ϕ_2 sont telles que :*

$$\phi_1 = \Box(p_1 \Rightarrow \ominus q_1)$$
$$\phi_2 = \Box(p_2 \Rightarrow r_2 S q_2)$$

alors l'établissement des propriétés $\{p_2 \wedge I_{1,2} \Rightarrow p_1, p_1 \wedge I_{1,2} \Rightarrow r_2, q_1 \wedge I_{1,2} \Rightarrow q_2\}$ garantit $TS_2 \models \phi_2$.

Théorème 2 *(Pattern $\ominus \diamond$) Si $TS_1 \models \phi_1$ et les formules ϕ_1 et ϕ_2 sont telles que :*

$$\phi_1 = \Box(p_1 \Rightarrow \ominus q_1)$$
$$\phi_2 = \Box(p_2 \Rightarrow \diamond\, q_2)$$

alors l'établissement des propriétés $\{p_2 \wedge I_{1,2} \Rightarrow p_1, q_1 \wedge I_{1,2} \Rightarrow q_2\}$ garantit $TS_2 \models \phi_2$.

Théorème 3 *(Pattern $\diamond\, \diamond$) Si $TS_1 \models \phi_1$ et les formules ϕ_1 et ϕ_2 sont telles que :*

$$\phi_1 = \Box(p_1 \Rightarrow \diamond\, q_1)$$
$$\phi_2 = \Box(p_2 \Rightarrow \diamond\, q_2)$$

alors l'établissement des propriétés $\{p_2 \wedge I_{1,2} \Rightarrow p_1, q_1 \wedge I_{1,2} \Rightarrow q_2\}$ garantit $TS_2 \models \phi_2$.

Théorème 4 *(Pattern SS) Si $TS_1 \models \phi_1$ et les formules ϕ_1 et ϕ_2 sont telles que :*

$$\phi_1 = \Box(p_1 \Rightarrow r_1 S q_1)$$
$$\phi_2 = \Box(p_2 \Rightarrow r_2 S q_2)$$

alors l'établissement des propriétés $\{p_2 \wedge I_{1,2} \Rightarrow p_1, r_1 \wedge I_{1,2} \Rightarrow r_2,\ q_1 \wedge I_{1,2} \Rightarrow q_2\}$ garantit $TS_2 \models \phi_2$.

Théorème 5 *(Pattern $S\diamond$) Si $TS_1 \models \phi_1$ et les formules ϕ_1 et ϕ_2 sont telles que :*

$$\phi_1 = \Box(p_1 \Rightarrow r_1 S q_1)$$
$$\phi_2 = \Box(p_2 \Rightarrow \diamond\, q_2)$$

alors l'établissement des propriétés $\{p_2 \wedge I_{1,2} \Rightarrow p_1, q_1 \wedge I_{1,2} \Rightarrow q_2\}$ garantit $TS_2 \models \phi_2$.

Dans la section suivante, nous illustrons cette approche de vérification par reformulation de propriétés sur un cas d'étude.

2.4.3 Application à un cas d'étude

Considérons un système de transitions d'états formé de deux états $\{S1, S2\}$ avec l'état $S1$ comme état initial. L'événement *Transition1* permet au système de passer de l'état $S1$ à l'état $S2$. La machine suivante décrit la spécification B correspondant à ce système :

MACHINE

$\texttt{System}_1$

SETS

`States`

CONSTANTS

`S1, S2`

PROPERTIES

`S1` $\in$ `STATES` $\wedge$

`S2` $\in$ `STATES` $\wedge$

VARIABLES

$\texttt{ss}_1$

INVARIANT

$\text{ss}_1 \in \text{STATES}$

INITIALISATION

$\texttt{ss}_1$`:=S1`

EVENTS

Transition$_1$=

PRE $\texttt{ss}_1$`=S1` **THEN**

$\texttt{ss}_1$`:=S2`

END

END

Sur ce système, la propriété P_1 suivante est vérifiée : *si le système se trouve dans l'état S2, alors il a été précédemment dans l'état S1.* Cette propriété est exprimée par :

$$\Box((s1 = S2) \Rightarrow \ominus(s1 = S1))$$

Supposons que le système précédent soit raffiné par l'ajout d'un état intermédiaire SI et un événement supplémentaire qui fait passer le système de l'état intermédiaire SI vers l'état

S2. La machine correspondant à ce raffinement est la suivante :

REFINEMENT

$\texttt{System}_2$

REFINES

$\texttt{System}_1$

CONSTANTS

S12, SI, S22

PROPERTIES

S12=S1 $\wedge$

S22=S2 $\wedge$

SI $\in$ STATES $\wedge$

SI=S2

VARIABLES

$\texttt{ss}_2$

INVARIANT

$\texttt{ss}_2$ $\in$ STATES $\wedge$

$\texttt{ss}_2$=$\texttt{ss}_1$

INITIALISATION

$\texttt{ss}_2$:=S12

EVENTS

Transition$_1$=

PRE $\texttt{ss}_2$=S12 **THEN**

$\texttt{ss}_2$:=SI

END ;

Transition$_2$=

PRE $\texttt{ss}_2$=SI **THEN**

$\texttt{ss}_2$:=S22

END

END

En reformulant la propriété P_1, on obtient la propriété P_2 suivante :

$$\Box((ss_2 = S22) \Rightarrow \diamond\, (ss_2 = S12))$$

Cette propriété spécifie que si le système se trouve dans l'état $S22$ (raffinement de l'état S2), alors il s'est trouvé dans le passé dans l'état $S12$ (raffinement de l'état $S1$). Pour prouver cette propriété, nous devons utiliser le pattern $\ominus\diamond$ présenté précédemment. Par conséquent, nous devons établir les preuves suivantes :

- $(ss_2 = S22 \wedge S22 = S2 \wedge ss_2 = ss_1) \Rightarrow ss_1 = S2$
- $(ss_1 = S1 \wedge ss_2 = ss_1 \wedge S12 = S1) \Rightarrow ss_2 = S12$

Ces deux preuves sont déchargées automatiquement à partir de la partie gauche de chaque implication. On en déduit ainsi que la propriété reformulée P_2 est vérifiée sur le raffinement du système.

2.5 Vérification de propriétés dynamiques sur un système FDS

Dans cette section, nous présentons deux approches, très proches, de vérification de propriétés dynamiques sur un système FDS (*Fair Discrete System*) doté d'un ensemble d'hypothèses de faible et forte équité. La première approche [24] propose une règle pour vérifier des propriétés dynamiques de la forme $p \Rightarrow \Diamond q$ sous l'hypothèse de forte équité. Cette propriété signifie que si un prédicat p est vrai durant l'exécution d'un système alors on atteindra un état où le prédicat q sera vrai. La seconde approche [23] propose quand à elle un ensemble de règles de vérification de propriétés CTL* de la forme $p \Rightarrow A_f \Diamond q$. Ces propriétés expriment que si un prédicat p est vrai, alors pour toute exécution possible et juste du système, à partir d'un état qui satisfait p, on aura le prédicat q qui sera vrai dans le futur. Le quantificateur A_f a été introduit dans la section 2.1.2.

Tout d'abord, nous allons montrer la façon dont sont représentées les hypothèses d'équité, ensuite nous définissons ce que c'est un système FDS, puis enfin nous présentons les règles de vérification proposées par les deux approches.

2.5.1 Représentation des hypothèses d'équité

Dans [24] et [23], deux types d'hypothèses sont considérées : les hypothèses de faible et de forte équité. Ces deux hypothèses sont représentées comme suit :

– Une hypothèse de faible équité est représentée par une assertion J. Une exécution respecte une hypothèse de faible équité J si elle contient un nombre infini d'états qui satisfont l'assertion J.
– Une hypothèse de forte équité (appelée aussi de compassion) C est représentée par un couple d'assertions noté $\langle p, q\rangle$. Une exécution respecte l'hypothèse de forte équité C si elle contient soit un nombre fini d'états satisfaisant p soit un nombre infini d'états satisfaisant q.

2.5.2 Système FDS

Un système FDS est un *quintuplet* $D = \langle V, \Theta, \rho, J, C\rangle$ tel que :
– $V = \{u_1, ..., u_n\}$ représente l'ensemble des variables du système D.
– Θ est la condition initiale du système. Par conséquent, l'état initial s_0 de toute exécution ($\sigma = \{s_0, s_1, ...\}$) du système D doit satisfaire la condition : $s_0 \models \Theta$.
– $\rho(V, V')$ est une assertion reliant V et V', avec V et V' étant respectivement les valeurs des variables du système aux états s et s'.
– J=$\{J_1, ..., J_k\}$ est l'ensemble des assertions de faible équité.
– C=$\{\langle p_1, q_1\rangle, ..., \langle p_n, q_n\rangle\}$ est l'ensemble des assertions de forte équité.

Un système FDS qui ne comprend que des hypothèses de forte équité est appelé *CDS* (pour *Compassion Discrete System*). De même, un système FDS qui ne considère que des hypothèses de faible équité est appelé *JDS*(pour *Justice Discrete System*).

2.5.3 Preuve de propriétés de sécurité sous hypothèse de forte équité

Dans [24], des règles de vérification de propriétés de sécurité, qui ne reposent que sur des hypothèses de forte équité, ont été proposées. Pour ce faire, les auteurs ont ré-exprimé les hypothèses de faible équité en termes d'hypothèses de forte équité.

Soit *En(T)* et *Taken(T)* deux assertions définies comme suit :
– *En(T)* est vraie pour tous les états où la transition T est active. Cette assertion indique ainsi si la transition T est franchissable.
– *Taken(T)* est vraie pour tous les états où la transition T est réellement franchie.

L'assertion J_T suivante :

$$J_T = (\neg En(T) \vee Taken(T))$$

indique que si une transition peut être franchie alors elle l'est. Une exécution respecte l'hypothèse de faible équité correspondant à J_T s'il existe un nombre infini d'états où J_T est satisfaite comme le montre la figure ci dessous :

$$\underset{J_T}{\bigcirc} \longrightarrow \bigcirc \longrightarrow ... \underset{J_T}{\bigcirc} \longrightarrow \underset{J_T}{\bigcirc} \longrightarrow \bigcirc \longrightarrow ...$$

Les auteurs ont montré que l'hypothèse de faible équité J_T est équivalente à celle de forte équité $\langle true, J_T \rangle$ car l'assertion *true* reste toujours vérifiée. Ainsi toute hypothèse de faible équité peut être exprimée sous la forme d'une hypothèse de forte équité. En se basant sur des hypothèses de forte équité, les auteurs ont proposé un ensemble de règles pour la vérification de propriétés CTL* sur des systèmes réactifs. Parmi ces règles, nous présentons ci-après la règle *RESPONSE*.

La règle *RESPONSE* permet de vérifier les propriétés temporelles de la forme $(p \Rightarrow \Diamond q)$, sur un système CDS, sous des hypothèses de forte équité. La règle *RESPONSE* est définie comme suit :

Règle Response

Hypothèses

Pour un système FDS D avec une relation de transition ρ

Sur le domaine bien fondé $A : (W, \succ)$, on considère :

Σ l'ensemble des états du système

les assertions $p, q, \varphi_1, \ldots, \varphi_n$,

les hypothèses de compassion $\langle p_1, q_1 \rangle, ..., \langle p_n, q_n \rangle$,

les fonctions de classement $\Delta_1, \ldots, \Delta_n$ tels que $\Delta_i : \Sigma \mapsto W$

R1. $p \Rightarrow q \vee \bigvee_{j=1}^{n} (p_j \wedge \varphi_j)$

Pour chaque $i = 1, \ldots, n$,

R2. $p_i \wedge \varphi_i \wedge \rho \Rightarrow q' \vee \bigvee_{j=1}^{n} (p'_j \wedge \varphi'_j)$

R3. $\varphi_i \wedge \rho \Rightarrow q' \vee (\varphi'_i \wedge \Delta_i = \Delta'_i) \vee \bigvee_{j=1}^{n} (p'_j \wedge \varphi'_j \wedge \Delta_i \succ \Delta'_j)$

R4. $\varphi_i \Rightarrow \neg q_i$

alors

$D \models (p \Rightarrow \Diamond q)$

Cette règle permet de vérifier qu'en partant d'un état où un prédicat p est vrai, le système évolue pour atteindre un état où un prédicat q devient vrai. Elle est définie sous le domaine $A : (W, \succ)$, sachant que l'ensemble $(W, \succ)$ est bien fondé, c.à.d., il est impossible de former une suite infinie strictement décroissante avec les éléments de W. En outre, cette règle utilise un ensemble d'hypothèses de forte équité $\langle p_i, q_i \rangle$. à chacune de ces hypothèses correspond une assertion φ_i et une fonction Δ_i qui détermine la distance entre l'état actuel du système et l'état où q sera satisfait. Les différentes hypothèses de cette règle sont comme suit :

- R1 : elle spécifie que si le prédicat p est vrai, alors soit q est vrai ou il existe une assertions φ_i qui est vraie. Donc, tant qu'on n'a pas atteint l'état où q est établi, il existe toujours une assertion intermédiaire φ_i qui est vraie.
- R2 : elle spécifie qu'à chaque fois qu'une assertion φ_i est vraie, alors après une transition ρ soit le prédicat q devient vrai ou il existe une autre assertion qui s'établit.
- R3 : elle spécifie qu'à chaque fois qu'une assertion φ_i est vraie, alors à la suite d'une transition ρ on se retrouve dans l'un des trois cas suivants :
  - le prédicat q devient vrai.
  - l'assertion φ_i reste vraie et la distance entre l'état actuel du système et l'état satisfaisant q reste inchangé.
  - une autre assertion φ_j devient vraie avec diminution de la distance entre l'état présent et l'état satisfaisant q.
- R4 : elle spécifie que s'il y'a une assertion φ_i qui est vraie alors la deuxième composante q_i de l'hypothèse de compassion qui lui est associée est fausse.

à partir de l'hypothèse $R1$, on peut déduire que si le prédicat q n'est pas vrai initialement, alors il existe une assertion φ_i qui est vérifiée. En plus, la première composante p_i de l'hypothèse de compassion correspondant à cette assertion intermédiaire est également vraie. L'hypothèse $R2$ effectue la même vérification que $R1$ à chaque étape d'exécution, ce qui nous assure qu'il est impossible que le système se trouve dans un état où aucune assertion n'est vraie. $R3$ vérifie qu'après chaque transition, soit l'assertion qui était vraie à l'état précédent le reste et, dans ce cas, la distance entre l'état actuel et l'état satisfaisant q reste inchangée, soit une autre assertion avec un rang plus petit devient vraie. à ce stade, on n'est pas sûr d'atteindre l'état où q est vrai. Cette règle ne serait pas applicable si une assertion φ_i et le p_i qui lui est associé restent indéfiniment vrais. Ce problème est résolu par

l'hypothèse $R4$ qui assure que si une assertion φ_i est vraie, alors la deuxième composante de l'hypothèse de compassion associée est fausse. Ceci implique que la première composante p_i ne peut pas rester indéfiniment vraie. Par conséquent, la règle termine puisqu'une autre assertion avec un rang plus petit deviendra vraie.

Nous décrivons dans la section suivante une seconde approche de vérification très similaire à celle que nous venons de présenter mais basée, cette fois-ci, sur des hypothèses de faible équité ou de justice.

2.5.4 Un système de preuve déductive pour CTL*

Dans [23], les auteurs proposent un système de preuve pour la vérification de propriétés CTL* sur les systèmes réactifs. Pour simplifier la vérification des propriétés CTL*, un principe de réduction a été introduit. Ce principe consiste à ramener la preuve d'une propriété CTL* à celle d'un ensemble d'expressions de la forme $p_i \Rightarrow \beta_i$, avec p_i désignant une assertion et β_i une formule basique de CTL*. Une formule basique CTL* contient uniquement un opérateur temporel principal et est de la forme $\varrho\varphi$, avec ϱ représentant l'opérateur principal de la propriété et φ une formule de chemin selon la terminologie CTL*. En supposant que $f(\beta)$ soit une formule CTL* composée d'une ou plusieurs occurrences de la formule basique β, et en utilisant ce principe de réduction pour vérifier la formule CTL* $f(\beta)$ sur un système, il suffit de montrer que $p \Rightarrow \beta$ et $f(p)$ sont vrais, avec $f(p)$ désignant l'expression qu'on obtient en remplaçant toutes les occurrences de β par p dans $f(\beta)$.

Plus particulièrement, les auteurs proposent une approche de vérification de propriété de vivacité de la forme $p \Rightarrow A_f \Diamond q$. Cette propriété signifie que si p est vrai alors pour toute exécution équitable possible, q sera vrai dans le futur. L'approche consiste en la règle de déduction suivante :

Règle de vérification

Hypothèses

Pour un système FDS D avec une relation de transition ρ

Σ l'ensemble des états du système

des hypothèses de justice $J = J_1, \ldots, J_m$,

des assertions $p, q, \varphi_1, \ldots, \varphi_m$,

un ensemble bien fondé $(A, \succ)$

des fonctions de classement $\Delta_1, ..., \Delta_m : \Sigma \mapsto A$

W1. $p \Rightarrow q \vee \bigvee_{j=1}^{m}(\varphi_j)$

W2.Pour $i = 1, \ldots, m$

$\varphi_i \wedge \rho \Rightarrow q' \vee (\neg J'_i \wedge \varphi'_i \wedge (\bigvee_{j=1}^{m}(\varphi'_j) \wedge \Delta_i = \Delta'_i) \vee (\Delta_i \succ \Delta'_j))$ **alors**

$p \Rightarrow A_f \Diamond q$

Cette règle est définie sur l'ensemble bien fondé (A,$\succ$). Par conséquent, il est impossible de former avec les éléments de A une séquence infinie strictement décroissante. Cette règle utilise un ensemble d'assertions intermédiaires $\varphi_1, ..., \varphi_m$, chacune étant associée à une fonction Δ_i qui détermine la distance entre l'état actuel du système et l'état auquel q sera satisfait. Cette règle est constituée de deux éléments importants qui sont $W1$ et $W2$:

- $W1$ spécifie que si le système se trouve dans un état où p est vrai alors soit q est vrai ou il existe une assertion φ_i qui est vraie.
- $W2$ spécifie que si une assertion φ_i est vraie, alors après une transition, on sera dans l'un des cas suivants :
  - q devient vrai.
  - l'assertion φ_i reste vraie, l'hypothèse J_i est fausse et la distance entre cet état et l'état satisfaisant q reste inchangée.
  - une autre assertion φ_j devient vraie avec diminution de la distance entre l'état présent et l'état satisfaisant q.

à partir de *W1*, on peut déduire que si le prédicat q n'est pas vrai initialement, alors il existe une assertion intermédiaire qui est vraie. La règle $W2$ comprend cependant trois alternatives. La première correspond au cas où q est vrai à la suite d'une transition, auquel cas l'état voulu est atteint. Dans le second cas, l'assertion qui était vraie avant la transition reste vérifiée, mais l'hypothèse de faible équité J_i qui lui est associée est fausse. Dans une telle situation, la distance entre ce nouvel état et l'état satisfaisant q ne change pas. Enfin,

la troisième partie correspond au cas où il existe une nouvelle assertion intermédiaire qui s'établit avec un rang plus petit que l'assertion précédente. Sachant qu'une hypothèse de justice est indéfiniment souvent vérifiée, nous sommes sûrs d'atteindre l'état où q sera vrai.

2.5.5 Application à un cas d'étude

Dans cette section, nous présentons un exemple qui illustre le fonctionnement de la règle *RESPONSE* avec comme relation d'ordre $\succ$ l'ordre lexicographique. Considérons le programme suivant qui assure le traitement des éléments d'une file d'attente. L'opération $first\_elt(f)$ renvoie l'élément en tête de la file f tout en le supprimant de la file. L'opération *trait* applique un traitement sur l'élément qu'elle reçoit en argument :

$f : file$
l1 : **while** $f \neq \varnothing$ **do**
l2 : $\quad trait(first\_elt(f))$
l3 : **end**

Sur ce programme, nous souhaitons s'assurer qu'un élément x ne reste pas indéfiniment dans la file sans être traité par le programme. Comme les insertions (resp. extractions) se font par la queue (resp. par la tête), un élément x présent dans une file à une position p_x verra sa position décroître à chaque extraction. Pour garantir donc que n'importe quel élément x sera traité, nous devons nous assurer que le programme ne reste pas bloqué sur une ligne donnée. L'assertion $assert\_l_i$ indique que le programme se trouve à la ligne l_i du programme et l'assertion $C_i : \langle true, \neg assert\_l_i \rangle$ garantit que le programme visitera un nombre infini de fois les autres lignes (différentes de l_i). Nous voulons donc montrer que si un élément x est dans la file et si le programme démarre alors cet élément sera traité. Cette propriété s'exprime en CTL* comme suit :

$$(assert\_l_1 \wedge x \in f) \Rightarrow \Diamond(x \notin f)$$

Pour prouver cette propriété, nous allons appliquer la règle *RESPONSE* en utilisant les hypothèses définies précédemment. En notant C_i l'hypothèse de compassion à la ligne i du programme, le tableau suivant résume les différentes hypothèses de la règles *RESPONSE* :

i	C_i	φ_i	Δ_i
1	$\langle true, \neg assert\_l_1 \rangle$	$assert\_l_1 \wedge x \in f$	(p_x,2)
2	$\langle true, \neg assert\_l_2 \rangle$	$assert\_l_2 \wedge x \in f \wedge p_x \neq 1$	(p_x,1)
3	$\langle true, false \rangle$	$assert\_l_2 \wedge x \in f \wedge p_x = 1$	(0,0)

D'après le tableau ci-dessus, nous remarquons qu'à chaque étape d'exécution du programme, il y'a une assertion qui est vraie mais également la fonction Δ_i décroît après chaque transition. Au début de l'exécution du programme, φ_1 est établie puisque le programme se trouve à la ligne l_1 et que $(x \in f)$ d'après l'hypothèse de départ. Après la première transition, soit l'élément x se retrouve en tête de file, auquel cas φ_3 est vraie, ou l'assertion φ_2 devient vraie. Si φ_3 est vraie, nous savons qu'après une transition l'élément x sera traité et supprimé de la file, la propriété est donc vérifiée. Par contre si φ_2 est vrai, on aura $\Delta_2 = (p_x, 1)$ qui est inférieur à $\Delta_1 = (p_x, 2)$. Le programme va donc refaire un deuxième tour de boucle et tester si la file est vide, ce qui n'est pas le cas puisque x est dans la file. à ce point d'exécution, on a φ_1 qui redevient vraie et on a également $\Delta_1 = (p_x, 2)$ qui est inférieur à $\Delta_2 = (p_x, 1)$ puisque p_x a diminué lors du traitement de la tête de file. Sachant qu'à chaque tour de boucle, la tête de file est traitée, alors nous sommes sûrs que ce système atteindra un état où l'élément x sera traité.

2.6 Commentaires généraux

Cette section est consacrée à une étude comparative des travaux présentés dans les sections précédentes. Cette étude comparative permet une meilleure perception de notre contribution à la vérification formelle de propriétés dynamiques. Les approches proposées dans [4, 28] ont l'avantage de définir des règles précises et faciles à mettre en œuvre ; néanmoins elles requièrent la définition de variants qui peut représenter une vraie difficulté.

Les travaux présentés dans [23,24] sur la vérification de propriétés sur des systèmes FDS sont basés sur la définition d'assertions intermédiaires et des fonctions de distance pour prouver une sorte de convergence du système vers la propriété souhaitée. Ces méthodes ne sont pas aisées à appliquer, car aucune indication n'est fournie sur comment calculer de telles assertions et fonctions de distance.

Enfin, les travaux proposés dans [26] portent sur la vérification de propriétés dyna-

miques en utilisant la logique PLTL. Bien que les patterns définis pour la preuve soient faciles à appliquer, ils ne considèrent que les propriétés faisant référence au passé. Cependant, en pratique, on s'intéresse plus à l'évolution de l'exécution d'un système dans le futur. Ainsi, la majorité des propriétés dynamiques se rapportent au futur du système. Par conséquent, cette approche reste intéressante mais limitée à un sous-ensemble assez restreint des propriétés dynamiques.

2.7 Conclusion

Dans ce chapitre, nous avons présenté des travaux réalisés dans le cadre de la vérification des propriétés dynamiques. Comme nous l'avons fait remarquer, les approches existantes sont soit difficiles à mettre en œuvre, soit restreintes aux propriétés faisant référence au passé. De plus, ces différentes méthodes ne sont pas du tout outillées. Nos travaux de recherche s'intéressent à d'autres types de propriétés dynamiques faisant référence à la fois au passé et au futur du système, en proposant des approches systématiques de preuve fondées sur l'utilisation de la méthode B. Ces propriétés sont particulièrement importantes pour les systèmes d'information sans être limitées à ce domaine ; nos techniques de preuve sont applicables à tout type de système, incluant les systèmes d'information. De plus, l'ensemble des approches que nous proposons sont outillées et sont donc faciles à mettre en œuvre.

Chapitre 3

Preuve de propriété d'atteignabilité par génération formelle d'assertions

Sommaire

Dans ce chapitre est présentée une approche de vérification de propriété d'atteignabilité basée sur la génération systématique d'obligations de preuve B pour un ensemble de chemins qui permettent de satisfaire la propriété. Rappelons que ce type de propriété permet de s'assurer qu'il est possible d'atteindre un état cible à partir d'un état de départ. En représentant ces deux états par les prédicats ϕ et ψ respectivement, la propriété d'atteignabilité précédente peut être exprimée en CTL par $\mathsf{AG}(\psi \Rightarrow \mathsf{EF}\phi)$. L'idée de base de l'approche présentée dans ce chapitre est d'exhiber un ensemble de chemins et de générer un ensemble d'obligations de preuve B qui permettent de s'assurer que ces chemins mènent bien de ψ à ϕ. Pour ce faire, nous avons développé un algorithme de génération d'obligations de preuve dont la correction est démontrée à l'annexe B. Dans ce présent chapitre, nous présenterons deux versions de cet algorithme en montrant l'avantage et la limite de chacun d'eux. Nous proposons également une alternative de ces algorithmes basée sur le calcul de la plus faible précondition telle que introduite par Dijkstra [7]. Ces travaux ont été publiés dans [17].

3.1 Notions et définitions

Dans cette section, nous présentons les notions et définitions nécessaires à la compréhension des différents algorithmes proposés pour la preuve de la propriété d'atteignabilité. L'approche que nous préconisons est basée sur la notion de chemin représenté par une séquence d'actions qui peut être gardé par une condition. En effet dans l'état initial et en fonction des valeurs des différentes variables, plusieurs chemins sont possibles. Pour cela, nous considérons des chemins gardés de la forme $(cond \rightsquigarrow (act_1; \ldots; act_n))$. Un chemin gardé n'est exécuté que si sa garde *cond* est vérifiée. Nous donnons ci-après les définitions relatives aux actions ainsi que les différentes notions qui leurs sont associées.

Définition 1 *(Action) Une action d'un chemin est définie par la syntaxe BNF suivante (E désigne un ensemble quelconque d'éléments) :*

$action \quad ::=$

/*exécution de l'opération op*/

$op(param_1, \ldots, param_n)$ |

/*exécution de l'opération op pour n'importe quelle valeur de x appartenant à E*/

$|_{x \in E}\ op(\ldots, x, \ldots)$ |

/*exécution de l'opération op pour des valeurs x de E tant que C est vrai*/

$\mathsf{LOOP}(C, |_{x \in E} op(\ldots, x, \ldots), V)$ |

/*exécution de l'action si la condition cond est vraie, sinon ne rien faire[1]*/

$cond \rightarrow action$

L'action $\mathsf{LOOP}(C, |_{x \in E} op(\ldots, x, \ldots), V)$ signifie que l'opération *op* est exécutée en choisissant un ensemble d'éléments de E tant que la condition C est vérifiée. Le terme entier naturel décroissant V, appelé variant, permet d'assurer la terminaison de la boucle.

Définition 2 *(Précondition d'une action) Soit op une opération de la forme* ***PRE*** P ***THEN*** S ***END****. La précondition d'une action est définie par le tableau suivant :*

Action	***Pre***
$op(param_1, \ldots, param_n)$	P
$\|_{x \in E}\ op(\ldots, x, \ldots)$	$\exists x.(x \in E \wedge P)$
$\mathsf{LOOP}(C, \|_{x \in E} op(\ldots, x, \ldots), V)$	$C \Rightarrow \exists x.(x \in E \wedge P)$
$cond \rightarrow act$	$cond \Rightarrow Pre(act)$

Définition 3 *(Postcondition d'une action) Soient act une action comme définie précédemment, V_{Spec} l'ensemble des variables de la spécification et V_M le sous-ensemble de V_{Spec} représentant les variables modifiées par l'action act. En posant comme hypothèse que chaque opération soit de la forme,* **PRE** P **THEN** S **END***, la postcondition d'une action, notée $Post_{x_i, x_{i'}}(action)$, est définie comme suit avec $Post_{x,x'}(S)$ représentant la relation valeur-avant, valeur-après de la variable de x suite à l'exécution de la substitution S :*

1. ($cond \rightarrow Act$) est du sucre syntaxique de (**IF** *cond* **THEN** *Act* **END**)

Action	$Post_{x_i,x_i'}$
$op(param_1, \ldots, param_n)$	$P \wedge Post_{x_i,x_i'}(S)_{x_i \in V_M} \wedge (x_i' = x_i)_{x_i \in (V_{spec} - V_M)}$
$Loop(C, \vert_{x \in E}\ op(\ldots, x, \ldots), V)$	$not(C) \wedge (x_i' = x_i)_{x_i \in (V_{spec} - V_M)}$
$\vert_{x \in E}\ op(\ldots, x, \ldots)$	$Post_{x_i,x_i'}(op(\ldots, v, \ldots))\ avec\ v \in E$
$cond \rightarrow Action$	$(cond \Rightarrow Post_{x_i,x_i'}(Action)) \wedge (\neg cond \Rightarrow (x_i' = x_i)_{x_i \in V_{spec}})$

Dans la section qui suit, nous présentons l'approche de génération des obligations permettant la preuve d'une propriété d'atteignabilité.

3.2 Algorithme de génération d'obligations de preuve : version simplifiée

3.2.1 Méthode de preuve

L'approche que nous proposons pour prouver une propriété d'atteignabilité ($\mathsf{AG}(\psi \Rightarrow \mathsf{EF}\ \phi)$) consiste à démontrer qu'il existe au moins un chemin, $(cond \rightsquigarrow (act_1; \ldots; act_n))$, qui mène de ψ à ϕ. Pour ce faire, nous devons établir que :

1. l'exécution de chaque chemin mène à ϕ :

$$\psi \wedge cond \Rightarrow [(act_1; \ldots; act_n)]\phi \tag{3.1}$$

2. pour un ensemble de chemins $\{ch_1, \ldots, ch_m\}$, au moins un des chemins ch_i menant à ϕ peut être exécuté :

$$\psi \Rightarrow (\bigvee cond_i) \tag{3.2}$$

avec $cond_i$ représentant la garde du chemin ch_i. L'utilisation de l'AtelierB pour la preuve de ces formules est évidemment possible. Néanmoins, les preuves générées sont très nombreuses et assez complexes à décharger (voir section 3.4). C'est pour cette raison que nous avons développé une autre approche de preuve qui permet d'avoir des obligations beaucoup plus simples à établir. L'idée clé de notre approche peut être résumée par les trois points suivants :

1. Dans l'état s où ψ et $cond$ sont vrais, l'action act_1 est exécutable, c.à.d., sa précondition est satisfaite,

2. à tout point du chemin i $(1 \leq i \leq (n-1))$, après l'exécution de chaque action act_i, les valeurs des variables vérifient la précondition de l'action suivante act_{i+1},

3. Après exécution de la dernière action act_n, les valeurs des variables satisfont ϕ.

Pour faciliter la présentation de l'algorithme de génération d'obligations de preuve, nous avons défini les fonctions γ, $Ajout$ et $Cons\_PO$ comme suit :

– la fonction γ_j fait le renommage, dans une action act, de chaque variable x par la variable x^{j-1} :

$$\gamma_j(act) = [\|\vec{x} := \vec{x}^{j-1}]act$$

avec $\vec{x}^{j-1}$ désignant les valeurs $({x_1}^{j-1}, \ldots, {x_n}^{j-1})$ des variables de la spécification à l'étape $(j-1)$ de l'exécution du chemin.

– la fonction $Ajout$ mémorise, à chaque étape, les valeurs intermédiaires des variables locales (utilisées dans les actions de la forme $|_{x \in E}$), représentées par P, et celles de la spécification :

$$Ajout(H, \forall X.P) = \begin{cases} \forall (Y, X).(P \wedge Q), & \text{si } (H = \forall Y.Q) \\ \forall X.P, & \text{sinon} \end{cases}$$

– La fonction $Cons\_PO$ définit, à chaque étape, les formules à prouver :

$$Cons\_PO(H, P) = \begin{cases} \forall X.(Q \Rightarrow P), & \text{si } (H = \forall X.Q) \\ P, & \text{sinon} \end{cases}$$

En utilisant ces différentes fonctions, notre algorithme procède en trois étapes pour chaque chemin :

1. Initialisation des variables locales relatives aux objets utilisés par les actions de la forme $(|_{oo \in E}\ op(\ldots, oo, \ldots))$, aux valeurs des variables à chaque étape (variable Hyp), et aux formules à prouver (variable For).

2. Génération de l'obligation de preuve pour la première action act_1 : cette obligation de preuve assure qu'il est possible d'exécuter act_1, i.e., sa précondition $Pre(act_1)$ doit être vraie quand ψ et $cond$ sont vérifiés.

$$\psi \wedge cond \Rightarrow Pre(act_1)$$

3. Pour chaque action act_k ($k \in 2..n$), génération d'une obligation de preuve de la forme :

$$\forall(\vec{x}^1, \ldots, \vec{x}^{k-1}).(\bigwedge_{j=1}^{k-1} Post_{{x_i}^{j-1},{x_i}^j}(\gamma_j(act_j)) \Rightarrow Pre(\gamma_k(act_k)))$$

avec $\vec{x}^l$ désignant les valeurs $({x_1}^l, \ldots, {x_n}^l)$ de la variable x_i à l'étape l de l'exécution du chemin.

4. Enfin, une obligation de preuve pour s'assurer que ϕ est établie par la dernière action act_n est générée :

$$\forall(\vec{x}^1, \ldots, \vec{x}^n).(Post_{x_i,{x_i}^1}(a_1) \wedge \bigwedge_{j=2}^{n} Post_{{x_i}^{j-1},{x_i}^j}(\gamma_j(act_j)) \Rightarrow \phi)$$

Ces principes sont détaillés dans l'algorithme 1 qui utilise les notations suivantes :

- à chaque étape k de l'algorithme, Hyp représente un prédicat sur les valeurs ${x_i}^k$ de chaque variable x_i, mais également sur les variables locales utilisées dans les actions de la forme $|_{x \in E}\ op(\ldots)$.
- Pour chaque action de la boucle $LOOP(C, |_{oo \in E} op(\ldots, oo, \ldots), V)$ où op est définie par (**PRE** P **THEN** S **END**), nous devons prouver que le variant V est un entier naturel décroissant afin d'assurer la terminaison de la boucle : $(P \Rightarrow (V \in N \wedge [n := V][S](V < n)))$. Quand la boucle se termine, les valeurs des variables (la postcondition de la boucle) sont telles que C soit fausse.

3.2.2 Application à un cas d'étude

Cette section présente un cas d'étude simplifié de gestion d'un club vidéo qui permet d'illustrer l'approche proposée. Nous considérons donc des clients qui peuvent emprunter des cassettes ou les réserver si ces dernières ne sont pas disponibles. On fait l'hypothèse qu'un client ne puisse pas emprunter plus de deux cassettes et que les réservations sur une cassette donnée sont stockées dans une file. Nous obtenons la spécification suivante avec

Algorithm 1 Génération des obligations de preuve : version 1

Entrées: La propriété d'atteignabilité $F = (\mathsf{AG}\ (\psi \Rightarrow \mathsf{EF}\ \phi))$ et un ensemble de chemins $chs = \{ch_1, \ldots, ch_m\}$

1: Soit $POs \leftarrow \emptyset$ l'ensemble des obligations de preuve à établir
2: **pour chaque** $(ch \in chs \wedge ch \mathrel{\widehat{=}} (cond \rightsquigarrow (act_1; \ldots; act_n)))$ **faire**
3: Soit $V_{Spec} \leftarrow \{x_1, \ldots, x_k\}$ l'ensemble des variables de la spécification /*Initialisation des variables Hyp and For*/
4: Soit $Hyp \leftarrow True$ les hypothèses locales
5: Soit $For \leftarrow \emptyset$ l'ensemble des formules à prouver /*$Traitement\ de\ l'action\ act_1$*/
6: $For \leftarrow For \cup \{Pre(act_1)\}$
7: **si** $(act_1 = \mathsf{LOOP}(C, |_{oo \in E}\ op(oo, \ldots), V))$ **alors**
8: $For \leftarrow For \cup \{Pre(op) \Rightarrow (V \in N \wedge [n := V][S](V < n))\}$
/*Hyp doit mémoriser la variable locale x utilisée dans $|_{x \in E}\ op(\ldots)$*/
9: **sinon si** $(act_1 = |_{x \in E}\ \_)$ **alors**
10: $Hyp \leftarrow \forall x.(x \in E)$
11: **finsi**
12: $Hyp \leftarrow Ajout(Hyp, \forall\ \vec{x_i}^1\ .Post_{x_i{}^0, x_i{}^1}(S_{act_1}))$ /*$Traitement\ des\ actions\ \{a_2, \ldots, a_n\}$*/
13: **pour chaque** (action act_j telle que $(j \in 2..n)$) **faire**
14: $act_j \leftarrow \gamma_j(act_j)$
15: $For \leftarrow For \cup \{Cons\_PO(Hyp, Pre(act_j))\}$
16: **si** $(act_j = \mathsf{LOOP}(C, |_{oo \in E}\ op(oo, \ldots), V))$ **alors**
17: $For \leftarrow For \cup \{Cons\_PO(Hyp, Pre(op) \Rightarrow (V \in N \wedge [n := V][S](V < n)))\}$
18: **sinon si** $(act_j = |_{x \in E}\ \_)$ **alors**
19: $Hyp \leftarrow Ajout(Hyp, \forall x.(x \in E))$
20: **finsi**
21: $Hyp \leftarrow Ajout(Hyp, \forall \vec{x_i}^{j-1}.Post_{x_i{}^{j-1}, x_i{}^j}(S_{act_j}))$
22: **fin pour**
23: $For \leftarrow For \cup \{Cons\_PO(Hyp, \gamma_{n+1}(\phi)\}$
24: Soit V_F l'ensemble des variables de F
25: Soit $V_F - V_{Spec} = \{y_1, \ldots, y_l\}$ l'ensemble des variables de F qui n'appartiennent pas à V_{Spec}. Ces variables sont libres et sont implicitement quantifiées universellement.
26: L'obligation de preuve du chemin ch est $PO(ch) \leftarrow \forall(y_1, \ldots, y_l).(\psi \wedge cond \Rightarrow (\bigwedge_{for \in For} for))$
27: **fin pour**/*une obligation de preuve additionnelle est générée pour assurer que l'ensembles des chemins couvre, dans l'état s (où ψ est vrai), toutes les valeurs possibles des variables V_{Spec}.*/
28: $POs \leftarrow \bigcup_{i=1,m} PO(ch_i) \cup \{\forall(y_1, \ldots, y_l).(\psi \Rightarrow \bigvee_{i=1..m} cond_i)\}$

les opérations associées :

MACHINE

ClubVideo

SETS

Clients, Cassette

VARIABLES

Emprunt, Reservation

INVARIANT

Emprunt $\in$ Cassettes $\nrightarrow$ Clients $\wedge$

Reservation $\in$ Cassettes $\rightarrow$ **iseq**(Clients) $\wedge$

$\forall$ cl. (cl $\in$ Clients $\Rightarrow$ **card**(Emprunt^{-1}[{cl}] ≤ 2)

DEFINITIONS

/*$Indice(bo, me)$ renvoie le rang d'un client cl dans

la file de réservation d'une cassette ca*/

Indice(ca, cl) $\widehat{=}$ (Reservation(ca))$^{-1}$(cl) ;

OPERATIONS

Emprunter(cl, ca)$\widehat{=}$

PRE

cl $\in$ Clients $\wedge$ ca $\in$ Cassettes $\wedge$ ca $\notin$ dom(Emprunt) $\wedge$

(Reservation(ca)$=$ [] $\vee$ (**first** (Reservation(ca)) = cl)) $\wedge$

card(Emprunt^{-1}[{cl}]) < 2

THEN

Emprunt:=Emprunt $\leftarrow$ {ca $\mapsto$ cl}

END ;

Reserver(cl, ca)$\widehat{=}$

PRE

cl $\in$ Clients $\wedge$ ca $\in$ Cassettes $\wedge$ cl $\notin$ **ran**(Reservation(ca)) $\wedge$

ca $\mapsto$ cl $\notin$ Emprunt $\wedge$ ((ca $\in$ dom(Emprunt)) $\vee$ (Reservation(ca) $\neq$ []))

THEN

Reservation := Reservation $\leftarrow$ {ca $\mapsto$ ((Reservation(ca) $\leftarrow$ cl)}

END ;

```
Rendre(ca)≙
PRE
    ca ∈ Cassettes ∧ ca ∈ dom (Emprunt)
THEN
        Emprunt := {ca} ⩤ Emprunt
END ;
Annuler_Reservation(cl, ca)≙
PRE
    cl ∈ Clients ∧ ca ∈ Cassettes ∧ cl ∈ ran(Reservation(ca))
THEN
    Reservation:=Reservation <+ {ca ↦ ((Reservation(ca) ↑
        (Indice(ca, cl) - 1))⌢ (reservation (ca) ↓ Index(ca, cl)))}
END
END
```

Sur cette spécification, on se propose de prouver qu'il est possible pour un client cl d'emprunter une cassette ca. Cette propriété est exprimée en CTL par :

$$\mathsf{AG}((ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt) \Rightarrow \mathsf{EF}(ca \mapsto cl \in Emprunt))$$

Pour prouver cette propriété, deux chemins sont possibles :

- ch_1 : ce chemin est possible quand il n'y a aucune réservation sur la cassette ca. Il est décrit par

$$\begin{aligned} ch_1 \mathrel{\widehat{=}}\ & Reservation(ca) = [] \rightsquigarrow \\ & (\\ & (ca \in dom(Emprunt) \rightarrow Rendre(ca)); \\ & ((card(Emprunt^{-1}[\{cl\}]) = 2) \rightarrow |_{ca' \in Emprunt^{-1}[\{cl\}]} Rendre(ca')); \\ & Emprunter(cl, ca) \\ &) \end{aligned}$$

- ch_2 : ce chemin est possible quand il y a des réservations sur la cassette ca. Il est

décrit par :

$$
\begin{aligned}
ch_2 \mathrel{\widehat{=}}\ & Reservation(ca) \neq [] \rightsquigarrow \\
& (\\
& (cl \notin ran(Reservation(ca)) \rightarrow Reserver(cl, ca)); \\
& \mathsf{LOOP}(Reservation(ca) \neq [cl], \\
& \qquad |_{oo \in ran(Reservation(ca)) - \{cl\}} Annuler\_Reservation(oo, ca), \\
& \qquad \mathbf{size}(ran(reservation(ca))); \\
& \qquad)(ca \in dom(Emprunt) \rightarrow Rendre(ca)); \\
& card(Emprunt^{-1}[\{cl\}]) = 2 \rightarrow |_{ca' \in Emprunt^{-1}[\{cl\}]} Rendre(ca'); \\
& Emprunter(cl, ca) \\
&)
\end{aligned}
$$

En appliquant l'algorithme proposé sur le chemin ch_1, nous obtenons les résultats suivants :

- $act_1 = (ca \in dom(Emprunt) \rightarrow Rendre(ca))$

$$
For \leftarrow \ \{ca \in dom(Emprunt) \Rightarrow (ca \in Cassettes \wedge ca \in dom(Emprunt))\}
$$

$$
\begin{aligned}
Hyp \leftarrow\ & \forall(,_{v_1 \in V_{Spec}}).(((ca \in dom(Emprunt) \wedge Emprunt_1 = \{ca\} \lhd Emprunt) \vee \\
& (ca \notin dom(Emprunt) \wedge Emprunt_1 = Emprunt)) \wedge \textstyle\bigwedge_{v \in V_{Spec} - \{Emprunt\}} (v_1 = v))
\end{aligned}
$$

- $act_2 = ((card(Emprunt^{-1}[\{cl\}]) = 2) \rightarrow |_{ca' \in Emprunt^{-1}[\{cl\}]} Rendre(ca'))$

$$
\begin{aligned}
For \leftarrow\ & For \cup \{\forall(,_{v_1 \in V_{Spec}}).(\\
& ((ca \in dom(Emprunt) \wedge Emprunt_1 = \{ca\} \lhd Emprunt) \vee \\
& (ca \notin dom(Emprunt) \wedge Emprunt_1 = Emprunt)) \wedge \\
& \textstyle\bigwedge_{v \in V_{Spec} - \{Emprunt\}} (v_1 = v) \Rightarrow \\
& (card({Emprunt_1}^{-1}[\{cl\}]) = 2 \Rightarrow \exists ca'.(ca' \in {Emprunt_1}^{-1}[cl]] \wedge \\
& \qquad ca' \in Cassettes_1 \wedge ca' \in dom(Emprunt_1))))\}
\end{aligned}
$$

$$
\begin{aligned}
Hyp \leftarrow\ & \forall(,_{v_1 \in V_{Spec}},\ ,_{v_2 \in V_{Spec}}, ca'\ ).(\\
& ((ca \in dom(Emprunt) \wedge Emprunt_1 = \{ca\} \lhd Emprunt) \vee \\
& (ca \notin dom(Emprunt) \wedge Emprunt_1 = Emprunt)) \wedge \textstyle\bigwedge_{v \in V_{Spec} - \{Emprunt\}} (v_1 = v) \wedge \\
& ((card({Emprunt_1}^{-1}[\{cl\}]) = 2 \wedge Emprunt_2 = \{ca'\} \lhd Emprunt_1) \vee \\
& (card({Emprunt_1}^{-1}[\{cl\}]) \neq 2 \wedge Emprunt_2 = Emprunt_1)) \wedge \\
& \textstyle\bigwedge_{v \in V_{Spec} - \{Emprunt\}} (v_2 = v_1) \wedge (ca' \in {Emprunt_1}^{-1}[\{cl\}] \wedge \\
& ca' \in Cassettes_1 \wedge ca' \in dom(Emprunt_1)))
\end{aligned}
$$

– $act_3 = Emprunter(cl, ca)$

$$
\begin{aligned}
For \leftarrow\ & For \cup \{\forall(,_{v_1 \in V_{Spec}},\ ,_{v_2 \in V_{Spec}}, ca').(\\
& ((ca \in dom(Emprunt) \wedge Emprunt_1 = \{ca\} \lhd Emprunt) \vee \\
& (ca \notin dom(Emprunt) \wedge Emprunt_1 = Emprunt)) \wedge \\
& \textstyle\bigwedge_{v \in V_{Spec} - \{Emprunt\}} (v_1 = v) \wedge ((card({Emprunt_1}^{-1}[\{cl\}]) = 2 \wedge \\
& Emprunt_2 = \{ca'\} \lhd Emprunt_1) \vee \\
& (card({Emprunt_1}^{-1}[\{cl\}]) \neq 2 \wedge Emprunt_2 = Emprunt_1)) \wedge \\
& \textstyle\bigwedge_{v \in V_{Spec} - \{Emprunt\}} (v_2 = v_1) \wedge \\
& (ca' \in {Emprunt_1}^{-1}[\{cl\}] \wedge ca' \in Cassettes_1 \wedge \\
& ca' \in dom(Emprunt_1)) \Rightarrow (cl \in Clients_2 \wedge ca \in Cassettes_2 \wedge \\
& ca \notin dom(Emprunt_2) \wedge \\
& (Reservation_2(ca) = [] \ \vee (\mathbf{first}(Reservation_2(ca)) = cl)) \wedge \\
& card({Emprunt_2}^{-1}[\{cl\}]) < 2)\}
\end{aligned}
$$

$$
\begin{aligned}
Hyp \leftarrow\ & \forall(,_{v_1 `\in V_{Spec}}, ,_{v_2 \in V_{Spec}}, ca',\ ,_{v_3 \in V_{Spec}}).(\\
& ((ca \in dom(Emprunt) \wedge Emprunt_1 = \{ca\} \lhd Emprunt) \vee \\
& (ca \notin dom(Emprunt) \wedge Emprunt_1 = Emprunt)) \wedge \\
& \bigwedge_{v \in V_{Spec} - \{Emprunt\}} (v_1 = v) \wedge \\
& ((card(Emprunt_1\ [cl]) = 2 \wedge Emprunt_2 = \{ca'\} \lhd Emprunt_1) \vee \\
& \quad (card(Emprunt_1\ [cl]) \neq 2 \wedge Emprunt_2 = Emprunt_1)) \wedge \\
& \bigwedge_{v \in V_{Spec} - \{Emprunt\}} (v_2 = v_1) \wedge (ca' \in {Emprunt_1}^{-1}[\{cl\}] \wedge ca' \in Cassettes_1 \wedge \\
& ca' \in dom(Emprunt_1)) Emprunt_3 = Emprunt_2 \cup \{ca \mapsto cl\} \\
& \wedge \bigwedge_{v \in V_{Spec} - \{Emprunt\}} (v_3 = v_2))
\end{aligned}
$$

En posant ($Hyp \mathrel{\widehat{=}} \forall(,_{v_1 `\in V_{Spec}}, ,_{v_2 \in V_{Spec}}, ca',\ ,_{v_3 \in V_{Spec}}).P$), on obtient :

$$
For \leftarrow For\ \cup \{\forall(,_{v_1 `\in V_{Spec}}, ,_{v_2 \in V_{Spec}}, ca',\ ,_{v_3 \in V_{Spec}}).(P \Rightarrow (ca \mapsto cl \in Emprunt_3)\}
$$

Enfin pour prouver que ch_1 permet d'atteindre un état où ($ca \mapsto cl \in Emprunt$), on doit montrer :

$$
\begin{aligned}
\forall(ca, cl).(ca \in Cassettes \wedge cl \in Clients \wedge\ & ca \mapsto cl \notin \\
& Emprunt \wedge Reservation(ca) = [] \Rightarrow \bigwedge_{for \in For} for)
\end{aligned}
$$

Sous l'AtelierB, ces assertions ont généré 41 obligations de preuve dont six ont nécessité l'intervention de l'utilisateur pour être prouvées.

Comme nous pouvons le remarquer, les obligations de preuve générées comportent des parties répétitives relatives aux valeurs intermédiaires des variables à chaque point d'exécution du chemin. En effet, chaque obligation de preuve, générée à l'étape i, inclut toutes les hypothèses concernant les valeurs intermédiaires des variables aux étapes 1 jusqu'à $(i-1)$. Pour pallier cet inconvénient, une nouvelle version de cet algorithme est présentée dans la section suivante.

3.3 Algorithme de génération d'obligations de preuve : version améliorée

3.3.1 Méthode de preuve

L'idée clé de l'amélioration de l'algorithme proposé dans la section précédente est d'imbriquer les obligations de preuve de sorte que les hypothèses portant sur les valeurs intermédiaires des variables aux différentes étapes n'apparaissent qu'une seule fois. Pour ce faire, il est nécessaire de redéfinir la fonction *Ajout* comme suit :

$$Ajout(\forall X.((P \wedge Q) \Rightarrow R), N) = \begin{cases} \forall X.((P \wedge Q) \Rightarrow (U \wedge Ajout(V, N))), \\ \quad si\ R = U \wedge V\ et\ V = \forall X'.((P' \wedge Q') \Rightarrow R') \\ \forall X.((P \wedge Q) \Rightarrow (R \wedge N)),\ sinon \end{cases}$$

La fonction *Ajout* étant redéfinie, l'obligation de preuve générée à l'étape k est de la forme suivante :

$$\begin{array}{l} Pre(act_1) \wedge \\ \forall x_i^1.\ (Post_{x_i,x_i{}^1}(act_1) \Rightarrow \\ \quad (Pre(\gamma_2(act_2)) \wedge \forall x_i^2.(Post_{x_i{}^1,x_i{}^2}(\gamma_2(act_2)) \Rightarrow \\ \qquad (Pre(\gamma_3(act_3)) \wedge \forall x_i^3.(Post_{x_i{}^2,x_i{}^3}(\gamma_3(act_3)) \Rightarrow \\ \qquad\qquad \vdots \\ \qquad\qquad (Post_{x_i{}^{k-1},x_i{}^k}(\gamma_k(act_k)) \Rightarrow \ldots) \end{array}$$

L'algorithme 2 illustre les différentes étapes de génération de ces obligations de preuve.

3.3.2 Application à un cas d'étude

Afin de comparer les deux versions de l'algorithme proposé, nous avons appliqué l'algorithme que nous venons de présenter sur le même cas d'étude et la même propriété. Nous avons donc généré les obligations de preuve suivantes :

Algorithm 2 Génération des obligations de preuve : version 2

Entrées: La propriété d'atteignabilité $F = (\mathsf{AG}\ (\psi \Rightarrow \mathsf{EF}\ \phi))$ et un ensemble de chemins $chs = \{ch_1, \ldots, ch_m\}$

1: Soit $POs \leftarrow \emptyset$ l'ensemble des obligations de preuve à établir
2: **pour chaque** $(ch \in chs \wedge ch \mathrel{\widehat{=}} (cond \rightsquigarrow (act_1; \ldots; act_n)))$ **faire**
3: Soit $V_{Spec} \leftarrow \{x_1, \ldots, x_k\}$ l'ensemble des variables de la spécification
/*Initialisation de la variable Hyp et For*/
4: Soit $Hyp \leftarrow True$ les hypothèses locales

/*Traitement de l'action act_1*/
5: $For \leftarrow Pre(act_1)$
6: **si** $(act_1 = \mathsf{LOOP}(C, |_{oo \in E}\ op(oo, \ldots), V))$ **alors**
7: $For \leftarrow For \wedge \forall oo.(oo \in E \wedge Pre(op) \Rightarrow (V \in N \wedge [n := V][S](V < n)))$
8: **finsi**

/*Traitement de l'action act_2*/
9: **si** $(n > 1)$ **alors**
10: $act_2 \leftarrow \gamma_2(act_2)$
11: **si** $(act_2 = \mathsf{LOOP}(C, |_{oo \in E}\ op(oo, \ldots), V))$ **alors**
12: $Hyp \leftarrow \forall x_i^1.(Post_{x_i{}^0, x_i{}^1}(act_1) \Rightarrow (Pre(act_2) \wedge \forall oo.(oo \in E \wedge Pre(op) \Rightarrow (V \in N \wedge [n := V][S](V < n)))))$
13: **sinon**
14: $Hyp \leftarrow \forall x_i^1.(Post_{x_i{}^0, x_i{}^1}(act_1) \Rightarrow Pre(act_2))$
15: **finsi**
16: **finsi**/*Traitement des actions $\{act_3, \ldots, act_n\}$*/
17: **pour** $j = 3$ jusqu'à n **faire**
18: $act_j \leftarrow \gamma_j(act_j)$
19: **si** $(act_j = \mathsf{LOOP}(C, |_{oo \in E}\ op(oo, \ldots), V))$ **alors**
20: $Hyp \leftarrow Ajout(Hyp, \forall x_i^{j-1}.(Post_{x_i{}^{j-2}, x_i{}^{j-1}}(act_{j-1}) \Rightarrow (Pre(act_j) \wedge (\forall oo.(oo \in E \wedge Pre(op) \Rightarrow (V \in N \wedge [n := V][S](V < n)))))))$
21: **sinon**
22: $Hyp \leftarrow Ajout(Hyp, \forall x_i^{j-1}.(Post_{x_i{}^{j-2}, x_i{}^{j-1}}(act_{j-1}) \Rightarrow Pre(act_j)))$
23: **finsi**
24: **fin pour**
25: $Hyp \leftarrow Ajout(Hyp, \forall x_i^n.(Post_{x_i{}^{n-1}, x_i{}^n}(act_n) \Rightarrow \gamma_n(\phi)))$
/*Obligation de preuve du chemin*/
$PO(ch_i) \leftarrow \forall(y_1, \ldots, y_l).(\psi \wedge cond \Rightarrow (For \wedge Hyp))$
26: **fin pour** $POs \longleftarrow \cup_{i=1..m} PO(ch_i) \cup \forall(y_1, \ldots, y_l).(\psi \Rightarrow \bigvee_{i=1..m} cond_i)$

– $act_1 = (ca \in dom(Emprunt) \rightarrow Rendre(ca))$

$$For \leftarrow \{ca \in dom(Emprunt) \Rightarrow (ca \in Cassettes \wedge ca \in dom(Emprunt))\}$$

– $act_2 = ((card(Emprunt^{-1}[\{cl\}]) = 2) \rightarrow |_{ca' \in Emprunt^{-1}[\{cl\}]} Return(ca'))$

$$\begin{aligned} Hyp \leftarrow\ & \forall(x_i^1 \in V_{spec}).((((ca \in dom(Emprunt) \wedge Emprunt_1 = \{ca\} \lhd Emprunt) \vee \\ & (ca \notin dom(Emprunt) \wedge Emprunt_1 = Emprunt)) \wedge \textstyle\bigwedge_{x_i \in V_{spec} - \{Emprunt\}} (x_i^1 = x_i))) \\ & \Rightarrow \\ & (card(Emprunt_1^{-1}[\{cl\}]) = 2 \Rightarrow \exists ca'(ca' \in Emprunt_1^{-1}[\{cl\}] \wedge \\ & ca' \in Cassettes \wedge ca' \in dom(Emprunt_1))) \end{aligned}$$

– $act_3 = Emprunter(cl, ca)$

$$\begin{aligned} Hyp \leftarrow\ & \forall(x_i^1 \in V_{spec}).((\\ & ((ca \in dom(Emprunt) \wedge Emprunt_1 = \{ca\} \lhd Emprunt) \vee \\ & (ca \notin dom(Emprunt) \wedge Emprunt_1 = Emprunt)) \wedge \textstyle\bigwedge_{x_i \in V_{spec} - \{Emprunt\}} (x_i^1 = x_i))) \\ & \Rightarrow \\ & ((card(Emprunt_1^{-1}[\{cl\}]) = 2 \Rightarrow \exists ca'(ca' \in Emprunt_1^{-1}[\{cl\}] \wedge \\ & ca' \in Cassettes \wedge ca' \in dom(Emprunt_1))) \\ & \wedge \\ & \forall(x_i^2 \in V_{spec}).((\\ & \quad ((card(Emprunt_1^{-1}[\{me\}]) = 2 \wedge \forall ca'(ca' \in Emprunt_1^{-1}[\{cl\}] \\ & \quad \Rightarrow Emprunt_2 = \{ca'\} \lhd Emprunt_1 \wedge \textstyle\bigwedge_{x_i \in V_{spec} - \{Emprunt\}} (x_i^2 = x_i^1))) \\ & \quad \vee \\ & \quad (card(Emprunt_1^{-1}[\{cl\}]) \neq 2 \wedge \textstyle\bigwedge_{x_i \in V_{spec}} (x_i^2 = x_i^1)))) \\ & \quad \Rightarrow \\ & \quad (cl \in Clients \wedge ca \in Cassettes \wedge ca \in dom(Emprunt_2) \wedge \\ & \quad (Reservation_2(ca) = [] \vee first(Reservation(ca)) = cl) \\ & \quad \wedge card(Emprunt_2^{-1}[\{cl\}]) < 2))) \end{aligned}$$

– Sortie de boucle

$$
\begin{aligned}
Hyp \leftarrow\ & \forall(x_i^1 \in V_{spec}).((\\
& ((ca \in dom(Emprunt) \wedge Emprunt_1 = \{ca\} \lhd Emprunt) \vee \\
& (ca \notin dom(Emprunt) \wedge Emprunt_1 = Emprunt)) \wedge \\
& \qquad \bigwedge_{x_i \in V_{spec} - \{Emprunt\}} (x_i^1 = x_i))) \\
& \qquad \Rightarrow \\
& ((card(Emprunt_1^{-1}[\{cl\}]) = 2 \Rightarrow \exists ca'(ca' \in Emprunt_1^{-1}[\{cl\}] \wedge \\
& ca' \in Cassettes \wedge ca' \in dom(Emprunt_1))) \\
& \wedge \\
& \forall(x_i^2 \in V_{spec}).((\\
& \qquad ((card(Emprunt_1^{-1}[\{cl\}]) = 2 \wedge \forall ca'(ca' \in Emprunt_1^{-1}[\{cl\}] \\
& \qquad \Rightarrow Emprunt_2 = \{ca'\} \lhd Emprunt_1 \wedge \bigwedge_{x_i \in V_{spec} - \{Emprunt\}} (x_i^2 = x_i^1))) \\
& \qquad \vee \\
& \qquad (card(Emprunt_1^{-1}[\{cl\}]) \neq 2 \wedge \bigwedge_{x_i \in V_{spec}} (x_i^2 = x_i^1)))) \\
& \qquad \qquad \Rightarrow \\
& \qquad (cl \in members_2 \wedge ca \in books_2 \wedge ca \in dom(Emprunt_2) \wedge \\
& \qquad (Reservation_2(ca) = [] \vee first(Reservation(ca)) = cl) \\
& \qquad \wedge card(Emprunt_2^{-1}[\{cl\}]) < 2) \\
& \qquad \wedge \\
& \qquad \forall(x_i^3 \in V_{spec}).((Emprunt_3 = (Emprunt_2 \cup \{ca \mapsto cl\}) \wedge \\
& \qquad \qquad \bigwedge_{x_i \in V_{spec} - \{Emprunt\}} (x_i^3 = x_i^2)) \\
& \qquad \qquad \Rightarrow \\
& \qquad \qquad (ca \mapsto cl \in Emprunt_3))))
\end{aligned}
$$

L'obligation de preuve finale générée pour le chemin ch_1 est alors la suivante :

$$
\begin{aligned}
\forall(ca, cl).(& ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt \wedge Reservation(ca) = [] \\
& \Rightarrow \\
& For \wedge Hyp)
\end{aligned}
$$

Sous l'AtelierB, ces assertions ont généré 41 obligations de preuve dont six ont nécessité l'intervention de l'utilisateurs pour être prouvées.

3.4 Preuve de propriété d'atteignabilité par la plus faible précondition

Dans cette section, nous décrivons une seconde approche de vérification de la propriété d'atteignabilité $\mathsf{AG}(\psi \Rightarrow \mathsf{EF}\phi)$ en utilisant la notion de la plus faible précondition que nous rappelons ci-après.

3.4.1 Méthode de preuve

Introduite par Dijkstra [7], la notion de la plus faible précondition $wp(S, R)$ désigne la plus large condition sous laquelle une substitution S peut être exécutée pour satisfaire une postcondition R. En posant comme hypothèse que chaque opération soit de la forme, **PRE** P **THEN** S **END**, la plus faible précondition d'une substitution S par rapport à la postcondition R, notée $wp(S, R)$ est donnée par le tableau suivant :

S	$wp(S, R)$
Skip	R
$x := E$	$R[x/E]$
$op(param_1, \ldots, param_n)$	$P \wedge wp(S, R)$
IF P **THEN** S **ELSE** T **END**	$(P \Rightarrow wp(S, R)) \wedge (\neg P \Rightarrow wp(T, R))$
$S; T$	$wp(S, wp(T, R))$
$\|_{v \in E}\ op(v, \ldots)$	$\forall v.(v \in E \Rightarrow wp(op(v, \ldots), R))$
WHILE C **DO** S **END**	il existe un prédicat I et une expression V tels que : $I \wedge$ $C \wedge I \Rightarrow wp(S, I) \wedge$ $C \wedge I \wedge n = V \Rightarrow wp(S, V < n) \wedge$ $C \wedge I \Rightarrow V \geq 0 \wedge$ $\neg C \wedge I \Rightarrow R$

Dans le tableau ci-dessus, la plus faible précondition de la boucle est le prédicat P à l'itération k qui correspond à la terminaison de la boucle. La notion de la plus faible précondition étant définie, prouver qu'un chemin $(cond \rightsquigarrow (act_1; \ldots; act_n))$ permet de mener de l'état ψ à l'état ϕ revient à établir que :

$$(\psi \wedge cond) \Rightarrow wp(act_1; \ldots; act_n, \phi)$$

3.4.2 Application à un cas d'étude

Afin de comparer l'approche présentée précédemment avec les deux premières méthodes que nous avons proposées, nous l'avons appliquée sur la même propriété d'atteignabilité suivante :

$$\mathsf{AG}((ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt) \Rightarrow \mathsf{EF}(ca \mapsto cl \in Emprunt))$$

nous obtenons alors l'obligation de preuve qui suit :

$$
\begin{array}{l}
\forall(ca, cl).(ca \in Cassettes \wedge cl \in Clients \wedge Reservation[\{ca\}] = \emptyset \Rightarrow \\
\quad (\\
\qquad ca \in dom(Emprunt) \Rightarrow \\
\qquad\quad ca \in Cassettes \wedge ca \in dom(Emprunt) \wedge \\
\qquad (\\
\qquad (card((\{ca\} \lhd Emprunt)\ [\{cl\}]) = 2 \Rightarrow \\
\qquad\quad (\forall to.((to \in (\{ca\} \lhd Emprunt)\ [\{cl\}]) \Rightarrow \\
\qquad\qquad (to \in Cassette \wedge to \in dom((\{ca\} \lhd Emprunt)) \wedge cl \in Clients \wedge \\
\qquad\qquad\quad cl \in Clients \wedge ca \in Cassettes \wedge ca \in dom(\{to\} \lhd (\{ca\} \lhd Emprunt))) \wedge \\
\qquad\qquad\quad ca \in dom(Reservation) \wedge card((\{to\} \lhd (\{ca\} \lhd Emprunt)) \rhd \{cl\}) < 2 \wedge \\
\qquad\qquad\quad (ca \mapsto cl) \in (\{to\} \lhd (\{ca\} \lhd Emprunt)) \cup \{(ca \mapsto cl)\}))) \\
\qquad\qquad\quad \wedge \\
\qquad\qquad\quad (card((\{ca\} \lhd Emprunt)\ [\{cl\}]) \neq 2 \Rightarrow (cl \in Clients \wedge \\
\qquad\qquad\quad ca \in Cassettes \wedge ca \in dom((\{ca\} \lhd Emprunt)) \wedge \\
\qquad\qquad\quad ca \in dom(Reservation) \wedge card((\{ca\} \lhd Emprunt) \rhd \{me\}) < 2 \wedge \\
\qquad\qquad\quad (ca \mapsto cl) \in (\{ca\} \lhd Emprunt) \cup \{(ca \mapsto cl)\})) \\
\qquad) \\
\quad) \\
\wedge \\
\quad (\\
\qquad ca \notin dom(Emprunt) \Rightarrow \\
\qquad (\\
\qquad\quad (card(Emprunt\ [\{me\}]) = 2 \Rightarrow (\forall to.((to \in Emprunt\ [\{me\}]) \Rightarrow (to \in Cassettes \wedge \\
\qquad\quad to \in dom(Emprunt) \wedge cl \in Clients \wedge ca \in BOOKID \wedge ca \in Cassettes
\end{array}
$$

$$
\begin{array}{l}
\wedge ca \in dom(\{to\} \lhd Emprunt)) \wedge ca \in dom(Reservation) \\
\wedge card((\{to\} \lhd Emprunt) \rhd \{cl\}) < 2 \wedge \\
(ca \mapsto cl) \in (\{to\} \lhd Emprunt) \cup \{(ca \mapsto cl)\})))
\end{array}
$$

$$
\begin{array}{l}
\wedge \\
(card(Emprunt\ [\{cl\}]) \neq 2 \Rightarrow (cl \in Clients \wedge ca \in Cassettes \\
\wedge ca \in Cassettes \wedge ca \in dom(Emprunt) \wedge ca \in dom(Reservation) \wedge \\
card(Emprunt \rhd \{cl\}) < 2 \wedge (ca \mapsto cl) \in Emprunt \cup \{(ca \mapsto cl)\}))
\end{array}
$$

$$
)
$$

$$
)
$$

Sous l'AtelierB, cette assertion a généré 34 obligations de preuve dont huit ont nécessité l'intervention de l'utilisateur pour être prouvées.

3.5 Comparaison des différentes approches

Dans ce chapitre, différentes méthodes de vérification des propriétés d'atteignabilité sur une spécification B ont été présentées. Ces approches consistent à générer des obligations de preuve pour s'assurer que les chemins exhibés permettent bien d'atteindre un état cible ψ à partir d'un état source ϕ. Pour les prouver, ces obligations de preuve sont ajoutées comme des assertions à la spécification B initiale. Le tableau suivant donne une synthèse des résultats obtenus durant la phase de preuve en utilisant le prouveur de l'AtelierB, version 4.0.

Approche	OP générées	OP automatiques	OP interactives
Algorithme 1	41	35	6
Algorithme 2	41	35	6
Méthode WP	34	26	8

D'après le tableau, les deux versions de l'algorithme proposé génèrent le même nombre d'obligations de preuve car l'idée de base d'obtention des assertions reste identique, seule la manière dont sont présentées les obligations de preuve diffère. D'après l'analyse des obligations de preuve générées à partir de ces assertions, on se rend compte que le générateur d'obligations de preuve de l'AtelierB se ramène aux mêmes obligations de preuve dans les deux cas. Néanmoins, la présentation des assertions sous forme imbriquée traduit mieux

l'intuition qu'on a du processus de preuve. Par contre, les assertions générées par le premier algorithme, bien qu'elles contiennent des parties répétitives, sont plus faciles à lire car une assertion séparée est produite à chaque point d'exécution du chemin.

Enfin, bien que la méthode de preuve par le calcul de la plus faible précondition génère un nombre total d'obligations de preuve plus faible, le nombre d'obligations de preuve interactives est plus important d'autant plus que ces dernières sont plus complexes à prouver. Cette complexité est due aux substitutions successives réalisées sur les prédicats intermédiaires à chaque point d'exécution du chemin. On en déduit donc que, sur le cas courant, nos approches sont beaucoup plus avantageuses.

3.6 Conclusion

Dans ce chapitre, nous avons présenté trois approches de vérification de propriétés d'atteignabilité sur une spécification B. Ces approches ont en commun d'être basées sur le même principe de génération d'assertions à partir d'un ensemble de chemins proposés pour satisfaire la propriété. L'idée clé étant de prouver que ces chemins, exécutés sur l'état source de la propriété, permettent d'atteindre l'état cible. Pour chacune de ces approches, nous avons mis en évidence les avantages et les inconvénients. Les résultats présentés dans ce chapitre ont été publiés dans [17]. Une propriété d'atteignabilité $\mathbf{LIVE}_{Progress}$, assez similaire à la propriété présentée dans ce chapitre, a été traitée dans [13]. Exprimée en LTL par $\mathsf{G}(p \Rightarrow \mathsf{F}q)$, cette propriété signifie qu'un état vérifiant un prédicat q sera toujours atteint à partir d'un état vérifiant un prédicat p. Bien évidement, cette propriété n'est pas très adaptée pour les systèmes d'information car, en effet, les traces d'exécution du système sont définies par les actions des utilisateurs extérieurs qu'on ne peut pas contrôler. On ne peut pas, par exemple, obliger un client à emprunter une cassette ou même à la restituer.

Dans le chapitre suivant, nous allons présenter une autre méthode de vérification de propriétés d'atteignabilité basée aussi sur la notion de chemin d'exécution mais également sur le raffinement de substitutions comme technique de construction incrémentale de programmes sûrs.

Chapitre 4

Preuve de propriété d'atteignabilité par raffinement

Sommaire

Dans ce chapitre, nous présentons une approche de vérification de propriétés d'atteignabilité utilisant le principe de raffinement de substitutions. Rappelons qu'une propriété d'atteignabilité, exprimée en CTL par $\mathsf{AG}(\psi \Rightarrow \mathsf{EF}\phi)$, signifie qu'il existe toujours au moins un chemin qui mène d'un état satisfaisant ψ á un état où ϕ est vrai. Cette approche est assez similaire á celle décrite dans le chapitre précédent en reposant sur la même notion de chemin représenté par une séquence d'exécution d'opérations du système. Suivant cette approche et la notion de spécification présentée par Morgan dans [21], une propriété d'atteignabilité est vue comme une spécification et le chemin comme étant un programme qui la raffine ou la réalise. Notre approche consiste donc á utiliser les règles de raffinement introduites par Morgan afin de générer un tel programme á partir de la spécification associée á la propriété d'atteignabilité. Ces travaux ont été publiés dans [9].

4.1 Le raffinement B

Le raffinement constitue un principe fondamental de développement d'applications suivant la méthode B. Il permet une construction graduelle et progressive d'un système depuis sa spécification abstraite initiale jusqu'á la génération de code exécutable en introduisant, étape par étape, les structures de données et de contrôle du langage cible choisi. En B, on distingue les deux types de raffinement suivants :

- **Raffinement de données** : il consiste á remplacer une donnée abstraite D par une donnée plus concrète D'. Dans ce cas, un prédicat, appelé invariant de collage établissant la relation entre D et D', est défini dans la clause **INVARIANT** du composant raffiné.
- **Raffinement de substitutions** : il consiste á remplacer une substitution S par une substitution plus concrète S'. Notons qu'un raffinement de données donne lieu au raffinement de substitutions afin que celles-ci soient ré-exprimées en fonction du nouvel espace de données.

Dans le cadre de notre travail, nous utilisons exclusivement le raffinement de substitutions car l'espace des données est toujours conservé.

Le développement progressif de système par raffinement est basé sur la propriété de transitivité suivante :

Définition 1 *(Propriété de transitivité) Soient S_1, S_2 et S_3 trois substitutions. Si S_2 raffine S_1 et S_3 raffine S_2 alors S_1 est raffinée par S_3 :*

$$S_1 \sqsubseteq S_2 \wedge S_2 \sqsubseteq S_3 \Rightarrow S_1 \sqsubseteq S_3$$

Comme nous le verrons plus loin, la correction de chaque étape de raffinement est vérifiée par la génération d'obligations de preuve qui ont pour but de montrer que la spécification concrète ne contredit pas la spécification abstraite et préserve donc toutes ses propriétés. De proche en proche, nous prouvons ainsi la correction du code obtenu.

Dans [21], Morgan a proposé un ensemble de règles de raffinement de spécifications permettant la génération progressive de programmes á partir d'une spécification initiale. Selon Morgan, une spécification, représentée par $w : [pre, post]$, désigne tout programme s'exécutant depuis un état initial vérifiant *pre* et terminant dans un état vérifiant *post* tout en modifiant les variables w. Dans le but d'alléger la notation de Morgan, nous représentons une spécification par SPEC(ψ, ϕ) qui agit implicitement sur l'ensemble des variables de la spécification.

Le paragraphe suivant introduit les règles de raffinement définies par Morgan pour la génération de programmes á partir de spécifications.

4.2 Règles de raffinement de Morgan

Dans cette section, nous présentons trois règles de raffinement introduites par Morgan. Nous restreignons notre présentation á ces règles utiles pour la preuve de propriété d'atteignabilité objectif de ce chapitre.

4.2.1 Raffinement par une substitution

Cette règle permet de raffiner une spécification SPEC(*pre*, *post*) par une substitution S :

$$Spec(pre, post) \sqsubseteq S \tag{4.1}$$

Pour prouver un tel raffinement, nous devons montrer que si la substitution S s'exécute á

partir d'un état qui satisfait *pre* alors elle terminera dans un état qui vérifie *post*. Formellement :

$$Spec(pre, post) \sqsubseteq S \Leftrightarrow (pre \Rightarrow [S]post) \tag{4.2}$$

4.2.2 Raffinement par décomposition séquentielle

Cette règle raffine une spécification par la composition séquentielle de deux sous-spécifications permettant ainsi de mieux gérer la complexité de la spécification initiale. Une spécification SPEC($pre,\ post$) est donc raffinée par deux sous-spécifications SPEC($pre,\ P_1$) et SPEC($P_2,\ post$) mises en séquence comme suit :

$$Spec(pre, post) \sqsubseteq Spec(pre, P_1); Spec(P_2, post) \tag{4.3}$$

Pour prouver un tel raffinement, nous utilisons la notion de la plus faible précondition et montrons que :

$$\begin{aligned} &Spec(pre, post) \sqsubseteq Spec(pre, P_1); Spec(P_2, post) \\ &\qquad\qquad \Leftrightarrow \\ &pre \Rightarrow [Spec(pre, P_1)]([Spec(P_2, post)]post) \end{aligned} \tag{4.4}$$

Un cas particulier de la règle de raffinement par décomposition séquentielle (4.3) et très utilisé pour la construction de programmes par raffinement a été défini par Morgan. En effet, lorsque les prédicats P_1 et P_2 sont égaux et représentent donc un seul prédicat intermédiaire noté *mid*, la règle de raffinement (4.3) devient donc :

$$Spec(pre, post) \sqsubseteq Spec(pre, mid); Spec(mid, post) \tag{4.5}$$

Intuitivement, cette règle signifie que la construction d'un programme qui relie un état *pre* á un état *post* peut être obtenue par la mise en séquence de deux sous-programmes : le premier reliant l'état *pre* á un état intermédiaire *mid*, le second liant ce même état intermédiaire á l'état cible *post*. Une telle decomposition permet un raffinement modulaire et indépendant des deux sous-spécifications générées. Notons que par construction, ce raffinement est correct et ne génère donc aucune obligation de preuve additionnelle. Ceci peut

être évidement établi en utilisant la définition de la correction du raffinement (4.4).

La condition intermédiaire *mid* doit être choisie de manière minutieuse. En effet, bien qu'une condition trop forte (resp. faible) facilite le raffinement de la seconde (resp. première) sous-spécification, elle complexifie le raffinement de la première (resp. seconde).

4.2.3 Raffinement par une boucle

Le raffinement d'une spécification par une boucle est un cas particulier de la première règle de raffinement par une substitution. Dans cette section, nous présentons le cas général du raffinement d'une spécification par une boucle mais aussi un cas particulier défini également par Morgan et très utile pour la génération de programmes par raffinement. Pour raffiner une specification SPEC(pre, $post$) par une boucle, nous devons exhiber les trois éléments principaux d'une boucle, á savoir :

1. *Condition d'arrêt de la boucle* C : permet de limiter le nombre d'itérations de la boucle. La boucle se termine quand la condition C devient vraie.
2. *Invariant* Inv : désigne un prédicat qui doit être vrai au début de la boucle, á chaque itération de la boucle mais également á la fin de boucle. Ce prédicat permet principalement de prouver la correction du raffinement d'une spécification par une boucle.
3. *Variant* V : dénote une expression arithmétique de type entier naturel qui décroit strictement á chaque itération de la boucle. Cette notion permet de prouver la terminaison de la boucle.

La règle de raffinement d'une spécification par une boucle s'exprime donc comme suit :

$$\text{SPEC}(pre,\ post) \sqsubseteq \textbf{WHILE } \neg C \textbf{ DO } S \textbf{ INVARIANT } Inv \textbf{ VARIANT } V \textbf{ END} \tag{4.6}$$

Pour prouver un tel raffinement, nous utilisons donc la formule générale de correction du raffinement d'une spécification par une substitution (4.2) ainsi que les règles de calcul de la plus faible précondition pour une boucle. Nous obtenons alors les obligations de preuve suivantes :

$$pre \Rightarrow Inv \tag{4.7}$$

$$Inv \wedge \neg C \Rightarrow [S]Inv \tag{4.8}$$

$$Inv \wedge \neg C \Rightarrow [n := V][S](V < n) \tag{4.9}$$

$$Inv \wedge \neg C \Rightarrow V \in \mathbb{N} \tag{4.10}$$

$$Inv \wedge C \Rightarrow post \tag{4.11}$$

- L'obligation de preuve (4.7) assure que l'invariant est vérifié au début de la boucle.
- L'obligation de preuve (4.8) assure que l'invariant est vérifié á chaque itération de la boucle.
- Les obligations de preuve (4.9) et (4.10) assurent que le variant est une expression naturelle et qu'elle décroît á chaque itération de la boucle. Ces deux obligations de preuve permettent de garantir la terminaison de la boucle.
- L'obligation de preuve (4.11) vérifie que le prédicat *post* est vrai á la fin de la boucle.

Une forme très particulière du raffinement d'une substitution par une boucle est tel que :

$$pre \triangleq Inv$$
$$post \triangleq Inv \wedge C$$

La règle de raffinement d'une spécification par une boucle s'écrit alors :

$$\text{SPEC}(Inv,\ Inv \wedge C) \sqsubseteq \textbf{WHILE } \neg C \textbf{ DO } S \textbf{ INVARIANT } Inv \textbf{ VARIANT } V \textbf{ END} \tag{4.12}$$

La preuve de ce raffinement particulier se restreint donc aux obligations de preuves (4.8), (4.9) et (4.10) ; les deux autres deviennent naturellement triviales.

4.3 Preuve de la propriété d'atteignabilité

4.3.1 Principe de preuve

Rappelons que le but de ce chapitre est de prouver la propriété d'atteignabilité en utilisant le principe de raffinement. Prouver une propriété d'atteignabilité "atteindre ϕ á partir de ψ", exprimée en CTL par $\mathsf{AG}(\psi \Rightarrow \mathsf{EF}\phi)$, revient á exhiber un programme P tel que :

$$\psi \Rightarrow [P]\phi \tag{4.13}$$

Or d'après l'équation (4.2), cela revient á monter que le programme P raffine la spécification $\text{SPEC}(\psi,\ \phi)$ qui représenterait alors la propriété d'atteignabilité. Notre principe de preuve est donc le suivant. Le programme P doit être généré de manière progressive á partir de la spécification initiale $\text{SPEC}(\psi,\ \phi)$ en applicant les différentes règles de raffinement présentées á la section précédente. Ce processus de raffinement est réitéré jusqu'á l'obtention de substitutions finales correspondant á celle du langage B. á chaque étape de ce raffinement, des obligations de preuve sont générées afin de s'assurer de la correction des transformations effectuées. La section suivante illustre ce principe de preuve sur le même cas d'étude et la même propriété d'atteignabilité introduits dans le chapitre précédent.

4.3.2 Application á un cas d'étude

Reprenons donc le cas d'étude de la gestion d'un club vidéo sur lequel nous souhaitons prouver la propriété d'atteignabilité "tout client *cl* a la possibilité d'emprunter une cassette *ca*". Cette propriété est exprimée en CTL par $\mathsf{AG}(\psi \Rightarrow \mathsf{EF}\phi)$ avec :

$$\psi \triangleq \left(\begin{array}{c} ca \in Cassettes \\ \wedge \\ cl \in Clients \\ \wedge \\ ca \mapsto cl \notin Emprunt \end{array} \right)$$

et

$$\phi \triangleq \Big(\; ca \mapsto cl \in Emprunt \; \Big)$$

Un chemin possible pour établir cette propriété d'atteignabilité serait le suivant :

1. si le client cl a atteint le nombre limite d'emprunts, alors il rend une des cassettes qu'il a déjá empruntées ;
2. si la cassette ca n'est pas disponible, cette dernière doit être restituée ;
3. si la file de réservation n'est pas vide, annuler toutes les réservations qui s'y trouvent ;
4. Enfin, emprunter la cassette ca pour le client cl.

Ce chemin peut bien évidement sembler drastique comme les réservations des autres clients sont tout simplement annulées. Un chemin plus réaliste serait d'effectuer une réservation de la cassette ca par le client cl et d'attendre que les autres clients aient réalisé leurs emprunts et retours. Nous choisissons néanmoins le premier chemin afin de faciliter la présentation de notre approche. Un tel chemin est représenté par le programme suivant :

IF $\neg(\mathbf{card}(Emprunt^{-1}[\{cl\}]) < MaxNbLoans)$ **THEN**
  ANY ca' **WHERE** $ca' \in Emprunt^{-1}[\{cl\}]$
  THEN $Rendre(ca')$ **END**
END;
IF $\neg(ca \notin dom(Emprunt))$ **THEN**
  $Rendre(ca)$
END;
WHILE $\neg(Reservation(ca) = [])$ **DO**
  ANY cl' **WHERE** $cl' \in ran(Reservation(ca))$
  THEN $Annuler\_Reservation(cl', ca)$ **END**
INVARIANT $\psi \wedge (\mathbf{card}(Emprunt^{-1}[\{cl\}]) < MaxNbLoans) \wedge (ca \notin dom(Emprunt))$
VARIANT $\mathbf{size}(Reservation(ca))$
END;
$Emprunter(cl, ca)$

á présent, nous allons montrer comment ce programme peut être dérivé de la spécification initiale $\text{SPEC}(\psi,\ \phi)$ en utilisant les règles de raffinement. Nous commençons par

décomposer $\text{SPEC}(\psi,\ \phi)$ en deux sous-spécifications :

$$\text{SPEC}(\psi,\ \phi) \sqsubseteq S_1; S_2 \tag{4.14}$$

avec les spécifications S_1 et S_2 définies comme suit :

$$\begin{aligned}
&S_1 \triangleq Spec(\psi, \psi \wedge C_1)\\
&S_2 \triangleq Spec(\psi \wedge C_1, \phi)\\
&C_1 \triangleq C_2 \wedge C_3 \wedge C_4\\
&C_2 \triangleq \mathbf{card}(Emprunt^{-1}[\{cl\}]) < MaxNbLoans\\
&C_3 \triangleq ca \notin dom(Emprunt)\\
&C_4 \triangleq Reservation(ca) = []
\end{aligned}$$

D'après l'équation (4.5), ce raffinement est correct et ne génère aucune obligation de preuve. Il faut á présent appliquer le même processus de raffinement sur les spécifications S_1 et S_2. La spécification S_2 se raffine en une seule étape en applicant la règle de raffinement (4.1) comme suit :

$$S_2 \sqsubseteq Emprunter(cl, ca) \tag{4.15}$$

D'après la règle (4.2), ce raffinement est validé par l'obligation de preuve suivante :

$$\psi \wedge C_1 \Rightarrow [Emprunter(cl, ca)]\phi$$

nous devons donc établir :

$$
\begin{array}{l}
\forall(ca, cl).(\psi \wedge C_1 \Rightarrow \\
\qquad cl \in Clients \wedge \\
\qquad ca \in Cassettes \wedge \\
\qquad ca \notin dom(Emprunt) \wedge \\
\qquad (Reservation(ca) = [] \vee (\mathbf{first}(Reservation(ca) = cl))) \wedge \\
\qquad \mathbf{card}(Emprunt^{-1}[\{cl\}]) < MaxNbLoans \wedge \\
\qquad ca \mapsto cl \in Emprunt \leftarrow\!\!+ \{ca \mapsto cl\})
\end{array}
$$

Raffinons maintenant la spécification S_1 par la règle (4.5) et décomposons la en deux sous-spécifications S_3 et S_4 :

$$S_1 \sqsubseteq S_3; S_4 \tag{4.16}$$

avec S_3 et S_4 définies par :

$$
\begin{array}{l}
S_3 \triangleq Spec(\psi, \psi \wedge C_2) \\
S_4 \triangleq Spec(\psi \wedge C_2, \psi \wedge C_1)
\end{array}
$$

De la même manière, la spécification S_3 est raffinée par la substitution suivante en utilisant la règle de raffinement (4.1) :

$$
\begin{array}{lll}
S_3 \sqsubseteq & \textbf{IF } \neg C_2 \textbf{ THEN} & (4.17) \\
& \quad \textbf{ANY } ca' \textbf{ WHERE } ca' \in Emprunt^{-1}[\{cl\}] \textbf{ THEN} & \\
& \quad\quad Rendre(ca') & \\
& \quad \textbf{END} & \\
& \textbf{END} &
\end{array}
$$

Ce raffinement est prouvé par la génération de l'obligation de preuve suivante (règle (4.1)) :

$$\psi \Rightarrow \quad [\textbf{IF } \neg C_2 \textbf{ THEN}$$
$$\textbf{ANY } ca' \textbf{ WHERE } ca' \in Emprunt^{-1}[\{cl\}] \textbf{ THEN}$$
$$Rendre(ca')$$
$$\textbf{END}$$
$$\textbf{END}](\psi \wedge C_2)$$

ce qui revient á prouver :

$$\forall(ca, cl).\left(\begin{array}{c} \\ \neg C_2 \Rightarrow \left(\begin{array}{c} \\ \forall ca'.\left(\begin{array}{c} \\ \left(\begin{array}{c} \psi \\ \Rightarrow \\ ca' \in Emprunt^{-1}[\{cl\}] \\ \Rightarrow \\ \left(\begin{array}{c} ca' \in Cassette \wedge ca' \in dom(Emprunt) \\ \wedge \\ cl \in Clients \wedge \\ ca' \mapsto cl \notin \{ca'\} \lhd Emprunt \\ \wedge \\ card((\{ca'\} \lhd Emprunt)^{-1}[\{cl\}]) < MaxNbLoans \end{array}\right) \end{array}\right) \\ \\ \end{array}\right) \\ \\ \end{array}\right) \\ \\ \left(\begin{array}{c} C_2 \Rightarrow (\psi \wedge C_2) \end{array}\right) \\ \\ \end{array}\right)$$

et

$$\forall(ca, cl).\left(\begin{array}{c} \psi \wedge \neg C_2 \\ \Rightarrow \\ \exists ca'.\left(\begin{array}{c} ca' \in Emprunt^{-1}[\{cl\}] \end{array}\right) \end{array}\right)$$

Similairement, la spécification S_4 est raffinée en la décomposant en deux sous-spécifications S_5 et S_6 :

$$S_4 \sqsubseteq S_5; S_6 \tag{4.18}$$

avec les deux spécifications S_5 et S_6 définies par :

$$S_5 \triangleq \text{SPEC}(\psi \wedge C_2,\ \psi \wedge C_2 \wedge C_3)$$
$$S_6 \triangleq \text{SPEC}(\psi \wedge C_2 \wedge C_3,\ \psi \wedge C_1)$$

La spécification S_5 se raffine directement par une substitution en applicant la règle (4.1) :

$$S_5 \sqsubseteq \textbf{IF } \neg C_3 \textbf{ THEN } Rendre(ca) \textbf{ END} \tag{4.19}$$

et génère l'obligation de preuve suivante :

$$\begin{array}{l}
\forall(ca, cl).(\psi \wedge C_2 \Rightarrow \\
\qquad \neg C_3 \Rightarrow \\
\qquad\quad (ca \in Cassette \wedge ca \in dom(Emprunt) \wedge cl \in client \wedge ca \mapsto cl \notin Emprunt \wedge \\
\qquad\quad (card((ca \lhd Emprunt)^{-1}[\{cl\}]) < MaxNbLoans) \wedge ca \notin dom(\{ca\} \lhd Emprunt))) \\
\qquad \wedge \\
\qquad (C_3 \Rightarrow (\psi \wedge C_2 \wedge C_3))))
\end{array}$$

Finalement, la spécification S_6 est raffinée par une boucle conformément á la règle de raffinement (4.12) :

$$\begin{array}{lll}
S_6 \sqsubseteq & \textbf{WHILE } \neg C_4 \textbf{ DO} & \\
& \quad S_7 & \\
& \quad \textbf{INVARIANT } (\psi \wedge C_2 \wedge C_3) & \\
& \quad \textbf{VARIANT size}(Reservation(ca)) & \\
& \textbf{END} &
\end{array} \tag{4.20}$$

avec la substitution S_7 définie par :

ANY cl' **WHERE** $cl' \in ran(Reservation(ca))$ **THEN**
$Annuler\_Reservation(cl', ca)$
END

Pour prouver le raffinement (4.3.2), nous instancions les équations (4.8)–(4.10) relatives au raffinement d'une spécification par une boucle :

1. le corps de la boucle préserve l'invariant :

$$\psi \wedge C_2 \wedge C_3 \wedge \neg C_4 \Rightarrow [S_7]\psi \wedge C_2 \wedge C_3$$

2. le variant est un entier naturel :

$$\psi \wedge C_2 \wedge C_3 \wedge \neg C_4 \Rightarrow V \in \mathbb{N}$$

3. le variant décroît á chaque itération de la boucle :

$$\psi \wedge C_2 \wedge C_3 \wedge \neg C_4 \Rightarrow [n := V][S_7](V < n)$$

Notons que la première obligation de preuve est triviale puisque la substitution S_7 ne modifie aucune variable de l'invariant. La seconde obligation de preuve est également simple á établir. Enfin pour la troisième, nous devons prouver :

$$\begin{array}{l}
\forall(ca, cl).((\psi \wedge C_2 \wedge C_3 \wedge \neg C_4) \Rightarrow \\
\quad (\\
\quad\quad \forall cl'.(cl' \in ran(Reservation(ca)) \Rightarrow \\
\quad\quad (\\
\quad\quad\quad cl \in Clients \wedge ca \in Cassettes \wedge cl \in ran(Reservation(ca)) \wedge \\
\quad\quad\quad size((Reservation(ca) \uparrow Index(ca, cl) - 1)) \frown \\
\quad\quad\quad\quad\quad\quad (Reservation(ca) \downarrow Index(ca, cl)) \\
\quad\quad\quad < \\
\quad\quad\quad size(Reservation(ca)) \\
\quad\quad) \\
\quad) \\
))
\end{array}$$

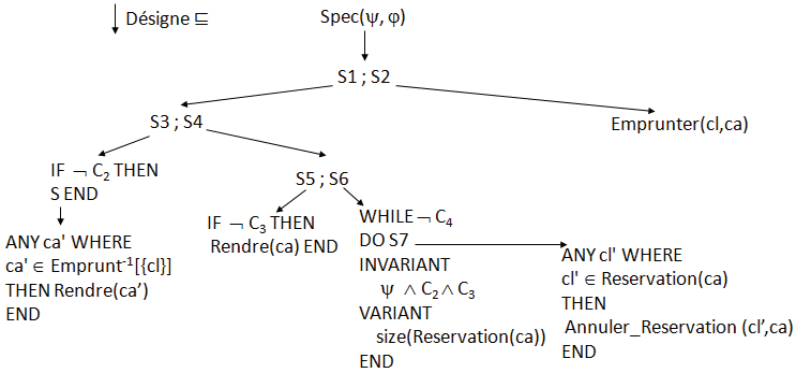

FIGURE 4.1 – Arbre de raffinement de la propriéte d'atteignabilité

La figure 4.1 illustre l'arbre de raffinement de la spécification initiale en vue de la génération du programme qui vérifie la propriété d'atteignabilité. Le programme produit s'obtient en mettant en séquence les actions, de gauche á droite, présentes aux feuilles de cet arbre

L'ensemble des obligations de preuve générées tout au long de ce processus de raffinement a été ajouté á la machine abstraite représentant la spécification du club vidéo (Voir section 3.2.2, page 58). Ajoutées comme des assertions (clause **ASSERTIONS** de B), ces formules ont généré 14 obligations de preuve dont dix ont été déchargées automatiquement par le prouveur, quatre ont nécessité l'utilisation de l'interface interactive afin de guider le prouveur dans le processus de preuve en lui indiquant les règles adéquates á appliquer.

4.4 Conclusion

Dans ce chapitre, nous avons proposé une méthode de vérification de propriété d'atteignabilité, exprimée en CTL par $\mathsf{AG}(\psi \Rightarrow \mathsf{EF}\phi)$, fondée sur la notion de raffinement de substitutions. L'idée de base de cette approche est d'exhiber un programme P dont l'exécution á partir d'un état satisfaisant ψ se termine dans un état qui vérifie ϕ. Pour ce faire, nous utilisons les règles de raffinement, introduites par Morgan dans [21], afin de générer de manière formelle et sûre un tel programme á partir de la spécification associée á la propriété d'atteignabilité considérée. L'utilisation de la technique de raffinement permet de prouver la correction de chaque étape de transformation et, de proche en proche, on assure que le programme généré réalise bien la spécification initiale. Enfin, notons que pour le

même cas d'étude et le même chemin d'exécution, cette approche produit moins d'obligations de preuve que celle présentée dans le chapitre précédent. Néanmoins, cette approche nécessite de trouver des prédicats intermédiaires lors du raffinement d'une spécification par une séquence ou aussi une boucle.

Chapitre 5

Preuve de propriété de précédence

Sommaire

Ce chapitre présente une approche formelle de vérification de propriétés de précédence de la forme : $\mathsf{Prec}(P_1, P_2)$. Introduite dans [8], ce type de propriété exprime que tout état satisfaisant un prédicat P_2 est précédé par un état satisfaisant P_1. Sur un système de réservation de billets d'avions, on aimerait s'assurer que le client ne reçoit de billet avant d'avoir réglé sa facture. De même, un processeur ne devrait pas allouer une ressource à un processus si une demande n'a pas été faite auparavant. L'idée clé de cette approche de vérification consiste à exhiber l'ensemble des chemins possibles menant de l'état initial jusqu'à un état satisfaisant P_2 et de démontrer que, sur ces chemins, il existe toujours un état qui satisfait P_1. Pour ce faire, nous définissons les conditions nécessaires et suffisantes pour la preuve de propriétés de précédence.

5.1 Définition formelle de $\mathsf{Prec}(P_1, P_2)$

Afin de définir la sémantique formelle du pattern de précédence $\mathsf{Prec}(P_1, P_2)$, introduisons les notations suivantes :

1. Soit Σ l'ensemble des états du système M, $op(\sigma)$ désigne les états atteignables à partir de l'état σ après exécution de l'opération op. L'expression $\sigma \models P$ signifie que l'état σ vérifie le prédicat P.

2. Soit Chs l'ensemble des chemins possibles, à partir de l'état initial σ_0, en exécutant les différentes opérations Ops du système :

$$Chs = \{(\sigma_0, \sigma_1, \ldots) \mid \forall i.(i \geq 1 \Rightarrow \exists op.(op \in Ops \wedge \sigma_i \in op(\sigma_{i-1})))\}$$

3. Étant donnés un chemin $ch = (\sigma_0, \sigma_1, \ldots, \sigma_i, \ldots)$ et un rang i $(0 \leq i)$, $ch[i]$ représente l'état σ_i du chemin ch.

4. Pour tous prédicats Q_1, Q_2 et opération op, nous définissons l'expression $(Q_1 \Rightarrow [S]Q_2)$ par :

$$\forall ch. \left(\begin{array}{c} ch \in Chs \\ \Rightarrow \\ \forall i. \left(\begin{array}{c} i \geq 0 \wedge ch[i] \models Q_1 \wedge ch[i] \models pre(op) \\ \Rightarrow \\ \left(\; ch[i+1] \in op(ch[i]) \Rightarrow ch[i+1] \models Q_2 \; \right) \end{array} \right) \end{array} \right) \tag{5.1}$$

Un système M vérifie la propriété de précédence :

$$\mathsf{Prec}(P_1, P_2) \tag{5.2}$$

ssi :

$$\forall ch. \left(\begin{array}{c} ch \in Chs \\ \Rightarrow \\ \forall i. \left(\begin{array}{c} \left(\begin{array}{c} i \geq 0 \\ \wedge \\ ch[i] \models P_2 \end{array} \right) \\ \Rightarrow \\ \exists j. \left(\begin{array}{c} j \geq 0 \wedge j \leq i \\ \wedge \\ ch[j] \models P_1 \end{array} \right) \end{array} \right) \end{array} \right) \tag{5.3}$$

Cette définition exprime que, pour tout chemin ch, si un état, de rang i, satisfait le prédicat P_2, alors il existe un rang j ($j \leq i$) sur le chemin ch tel que l'état associé vérifie le prédicat P_1. La section suivante présente la génération des conditions nécessaires et suffisantes à la preuve d'une telle formule.

5.2 Définition des obligations de preuve

Afin de faciliter la présentation de notre approche pour la preuve de la propriété (5.3), procédons en deux étapes complémentaires. La première étape concerne le cas de base et

correspond au cas où l'état satisfaisant le prédicat P_1 précède immédiatement l'état vérifiant le prédicat P_2. Dans la suite de ce manuscrit, nous désignons ce cas par : $\mathsf{Precl}(P_1,\ P_2)$. La seconde étape correspond au cas général et nous montrons qu'il est possible d'établir la propriété (5.3) en décomposant le pattern de précédence en plusieurs cas de base.

5.2.1 Preuve du cas de base $\mathsf{Precl}(P_1,\ P_2)$

Pour prouver qu'un état vérifiant le prédicat P_2 est immédiatement précédé d'un état où P_1 est vrai, nous devons établir que le système se comporte comme suit : à partir d'un état vérifiant $(\neg P_1 \wedge \neg P_2)$ et après exécution d'une opération *op*, le système ne devrait pas atteindre un état où P_2 serait vérifié alors que le prédicat P_1 est toujours faux, i.e., le système doit atteindre un état qui vérifie P_1 suivi d'un état satisfaisant P_2 ou demeurer dans le même état. Aussi, nous admettons un état où les prédicats P_1 et P_2 seraient vrais en même temps. Dans la figure 5.1, la flèche en pointillés désigne une transition invalide dont l'état cible vérifie P_2 et non P_1 :

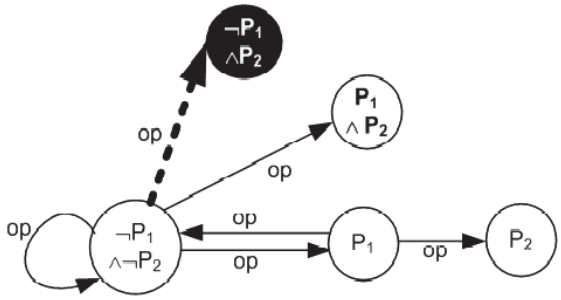

FIGURE 5.1 – Cas de base de la propriété de précédence immédiate $\mathsf{Precl}(P_1,\ P_2)$

Formellement, la précédence immédiate $\mathsf{Precl}(P_1,\ P_2)$ est définie par :

$$\mathsf{Precl}(P_1, P_2) \Leftrightarrow \forall ch. \left(\begin{array}{c} ch \in Chs \\ \Rightarrow \\ \forall i. \left(\left(\begin{array}{c} i > 0 \\ \wedge \\ ch[i] \models P_2 \end{array} \right) \Rightarrow \left(\begin{array}{c} ch[i] \models P_1 \\ \vee \\ ch[i-1] \models P_1 \end{array} \right) \right) \end{array} \right) \tag{5.4}$$

Pour établir cette formule, nous définissons le théorème suivant :

Théorème 1 Soient P_1 et P_2 deux prédicats. La propriété de précédence immédiate $\mathsf{PrecI}(P_1, P_2)$ est vérifiée ssi l'obligation de preuve suivante est établie pour chaque opération *op* du système :

$$\forall(\vec{x}, \vec{y}, \vec{v}).((\neg P_1 \wedge \neg P_2) \wedge pre(op) \Rightarrow [S_{op}](P_1 \vee \neg P_2))$$

avec $\vec{x}$ désignant les variable du système, $(x_1, \ldots, x_n)$, $\vec{y}$ sont les variables, distinctes des variables $\vec{x}$, utilisées pour l'expression des prédicats P_1 et P_2, et $\vec{v}$ représentant les paramètres d'entrée de l'opération *op*.

5.2.2 Preuve du cas général $\mathsf{Prec}(P_1,\ P_2)$

Rappelons que notre objectif est de prouver le pattern de précédence $\mathsf{Prec}(P_1, P_2)$. Remarquons que l'obligation de preuve du théorème (1) n'est pas applicable pour la preuve d'une telle propriété. En effet, la propriété $\mathsf{Prec}(P_1, P_2)$ peut être vérifiée alors que l'état précédant celui vérifiant le prédicat P_2 ne satisfait pas P_1 comme le montre la figure 5.2. Cette figure montre des états non successifs vérifiant les prédicats P_1 et P_2 alors que les états intermédiaires ne satisfont pas P_1. De plus, plusieurs chemins peuvent mener d'un état vérifiant P_1 à un état qui satisfait P_2. Comme le montre la figure 5.2, la propriété $\mathsf{Prec}(P_1,\ P_2)$ peut être décomposée en l'ensemble des propriétés suivantes :

$$\{\mathsf{Prec}(P_1,\ R_i)_{i\in 1..n},\ \mathsf{PrecI}(\textstyle\bigvee_{i\in 1..n} R_i,\ P_2)\}$$

avec les prédicats R_i modélisant les états précédant immédiatement ceux vérifiant P_2. Ainsi, le théorème 1 est appliqué sur la propriété $\mathsf{PrecI}(\bigvee_{i\in 1..n} R_i,\ P_2)$ et le processus de décomposition est répété sur les propriétés $\mathsf{Prec}(P_1,\ R_i)_{i\in 1..n}$ jusqu'à l'obtention de tous les états intermédiaires entre P_1 et P_2 ou la preuve que la propriété est fausse. Nous obtenons alors le théorème général suivant :

Théorème 2 Soient P_1 et P_2 deux prédicats. La propriété $\mathsf{Prec}(P_1, P_2)$ est vérifiée ssi il existe un ensemble de prédicats $\{R_1, \ldots, R_n\}$ tels que :

1. $\mathsf{PrecI}(\bigvee_{i\in 1..n} R_i,\ P_2)$

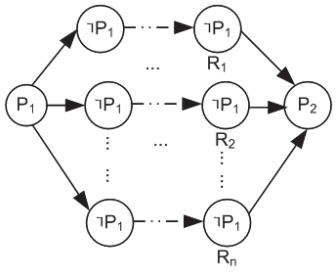

FIGURE 5.2 – Cas général de la précédence $\mathsf{Prec}(P_1,\ P_2)$

2. $\mathsf{Prec}(P_1,\ R_i)$ pour chaque $R_{i(i\in 1..n)}$

La section suivante décrit une heuristique qui permet de générer les différents prédicats R_i.

5.2.3 Définition des prédicats R_i

L'idée clé dans la définition des prédicats intermédiaires R_i pour la preuve de la propriété de précédence $\mathsf{Prec}(P_1,\ P_2)$ est la caractérisation des états intermédiaires situés entre ceux vérifiant P_1 et P_2. En effet, ces états vérifient les conditions suivantes :

1. les prédicats P_1 et P_2 sont faux sur ces états ;
2. il existe une opération qui fait évoluer le système vers un état qui satisfait P_2.

La définition des prédicats R_i consiste donc à trouver des contre-exemples de la propriété LTL suivante en utilisant le model checker ProB [15] :

$$(\neg P_2 \wedge \neg P_1) \Rightarrow X(\neg P_2 \vee P_1)$$

L'algorithme suivant décrit une heuristique de calcul des prédicats R_i basée sur cette dernière formule LTL.

Algorithm Heuristique de calcul des prédicats R_i

1: **Entrées :** prédicats P_1, P_2
2: **Sorties :** prédicats $\{R_1, \ldots, R_n\}$
3: Soit R un prédicat initialisé à faux $R = false$, $i = 1$.
4: Initialiser la spécification B à $(\neg P_2 \wedge \neg(P_1 \vee R))$ en utilisant le model checker ProB.
5: Établissement de formule LTL $X(\neg P_2 \vee (P_1 \vee R))$ en utilisant le model checker ProB.

6: **si** *aucun contre-exemple n'est trouvé* **alors**
7: La propriété de précédence $\mathsf{Prec}(P_1,\ P_2)$ est basique et peut être établie par le théorème 1.
8: **sinon**
9: la propriété est violée et un contre-exemple (valeurs des variables) est retourné.
10: /**vérifier si l'état s représentant le contre-exemple est atteignable à partir de l'état initial**/
11: Définir le prédicat P' qui représente l'état s et vérifier la formule CTL $\mathsf{EF}(P')$ avec l'initialisation réelle du système.
12: /*deux cas sont possibles*/
13: **si** (*Pas de chemin menant à l'état s*) **alors**
14: $\neg P'$ est un invariant qui est ajouté à l'invariant de la machine B, répéter le processus (Aller à l'étape 3)
15: **sinon**
16: /*vérifier si le prédicat P_1 est satisfait sur le chemin exhibé*/
17: **si** (P_1 *n'est pas vérifié*) **alors**
18: la propriété $\mathsf{Prec}(P_1,\ P_2)$ n'est pas vérifiée
19: **sinon**
20: $R_i = P'$
21: $R = R \vee R_i$
22: $i = i + 1$
23: Aller à l'étape 4.
24: **finsi**
25: **finsi**
26: **finsi**

5.3 Étude de cas

Afin d'illustrer notre approche de vérification de propriété de précédence, nous choisissons le cas du système scolaire français d'inscription aux études doctorales. En France, l'inscription en thèse est possible pour les étudiants de niveau Master 2 (niveau Bac+5), mais également pour ceux ayant suivi une formation d'ingénieur au terme de deux ans

de classes préparatoires aux grandes écoles. La figure 5.3 illustre ces deux options. La spécification B représentant ce système est comme suit.

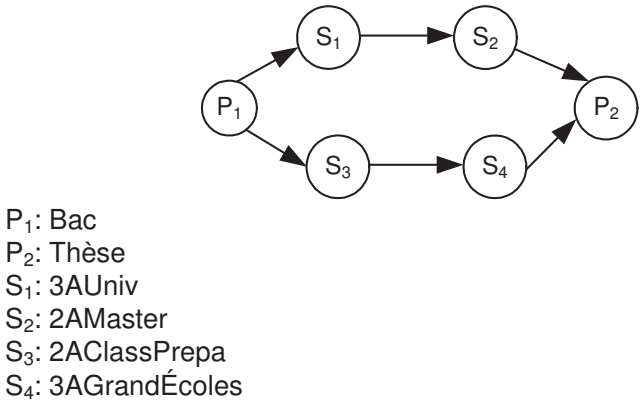

FIGURE 5.3 – Système éducatif français

Machine

$SystemeEducatifFrancais$

Sets

$Etudiants$;

$Niveaux$={*Bac, 3AUniv, 2AMaster, 2AClassPrepa, 3AGrandEcoles, These*}

Variables

$NiveauEt$

Invariant

$NiveauEt \in Etudiants \nrightarrow Niveaux$

Initialisation

$NiveauEt := \emptyset$

Operations

AdmissionBac(et)$\widehat{=}$

PRE

$et \in Etudiants \wedge et \notin dom(NiveauEt)$

THEN

$NiveauEt(et) := Bac$

END ;

Admission3AUniv(et)$\widehat{=}$

PRE

$et \in Etudiants \wedge NiveauEt(et) = Bac$

THEN

$NiveauEt(et) := 3AUniv$

END ;

Admission2AMaster(et)$\widehat{=}$

PRE

$et \in Etudiants \wedge NiveauEt(et) = 3AUniv$

THEN

$NiveauEt(et) := 2AMaster$

END ;

Admission2AClassPrepa(et)$\widehat{=}$

PRE

$et \in Etudiants \wedge NiveauEt(et) = Bac$

THEN

$NiveauEt(et) := 2AClassPrepa$

END ;

Admission3AGrandEcoles(et)$\widehat{=}$

PRE

$et \in Etudiants \wedge NiveauEt(et) = 2AClassPrepa$

THEN

$NiveauEt(et) := 3AGrandEcoles$

END ;

AdmissionThese(et)$\widehat{=}$

PRE

$et \in Etudiants \wedge$

$NiveauEt(et) \in \{2AMaster, 3AGrandEcoles\}$

THEN

$NiveauEt(et) := These$

END

END

Cette spécification a été validée en utilisant le prouveur de l'AtelierB [6]. Sept obligations de preuve ont été générées et prouvées pour s'assurer que les différentes opérations préservent l'invariant. Cependant, de telles obligations de preuve ne garantissent aucun ordre d'exécution sur les opérations du système. En effet, nous souhaitons vérifier que tout étudiant *et* inscrit en thèse est titulaire du diplôme du baccalauréat. Une telle propriété peut être exprimée, en utilisant le pattern de précédence, comme suit :

$$\begin{array}{l} \mathsf{Prec}(\\ \quad NiveauEt(et) = Bac, \\ \quad NiveauEt(et) = These \\ \quad) \end{array} \tag{5.5}$$

La première itération de l'algorithme avec la formule LTL suivante :

$$X(NiveauEt(et) \neq These \vee NiveauEt(et) = Bac)$$

en prenant comme état initial du système :

$$(NiveauEt(et) \neq These \wedge NiveauEt(et) \neq Bac)$$

produit un contre-exemple où l'étudiant *et* est de niveau `2AMaster` puis s'inscrit en thèse en exécutant l'opération **AdmissionThese**. Donc, le prédicat P' est égal à $(NiveauEt(et) = 2AMaster)$. En cherchant un chemin menant de l'état initial à un état vérifiant un tel prédicat, ProB renvoie le chemin suivant :

$$\textbf{Init} \bullet \textbf{AdmissionBac}(et) \bullet \textbf{Admission3AUniv}(et) \bullet \textbf{Admission2AMaster}(et)$$

Comme nous pouvons le remarquer ce chemin contient un état où $(NiveauEt(et) = Bac)$. Posons donc $(R_1 \mathrel{\widehat{=}} NiveauEt(et) = 2AMaster)$ et réitérons le processus avec $(R \mathrel{\widehat{=}} NiveauEt(et) = 2AMaster)$. La seconde itération de l'algorithme avec la formule LTL suivante :

i	R	**Initialisation**	**LTL Formula**	**Contre-exemple(R_i)**
1	$false$	$\neg P_2 \wedge \neg P_1$	$X(\neg P_2 \vee P_1)$	$NiveauEt(et) = 2AMaster$
2	$\bigvee_{k=1..i-1} R_k$	$\neg P_2 \wedge \neg(P_1 \vee R)$	$X(\neg P_2 \vee (P_1 \vee R))$	$NiveauEt(et) = 3AGrandEcoles$
3	$\bigvee_{k=1..i-1} R_k$	$\neg P_2 \wedge \neg(P_1 \vee R)$	$X(\neg P_2 \vee (P_1 \vee R))$	-

TABLE 5.1 – Définition des prédicats intérmediares R_i

$$X\left(\begin{array}{c} NiveauEt(et) \neq These \\ \vee \\ \left(\begin{array}{c} NiveauEt(et) = Bac \\ \vee \\ NiveauEt(et) = 2AMaster \end{array}\right) \end{array}\right)$$

avec l'état initial

$$\left(\begin{array}{c} NiveauEt(et) \neq These) \\ \wedge \\ NiveauEt(et) \neq Bac \\ \wedge \\ NiveauEt(et) \neq 2AMaster \end{array}\right)$$

donne un contre-exemple avec un état vérifiant ($NiveauEt(et) = 3AGrandEcoles$) et atteignable à partir de l'état initial par un chemin sur lequel le prédicat ($NiveauEt(et) = Bac$) est également vérifié. Nous réitérons alors le processus avec $R_2 = R_1 \vee (NiveauEt(et) = 3AGrandEcoles)$. Le tableau 5.1 donne les résultats que nous obtenons sur l'étude de cas. où les prédicats P_1 et P_2 sont définis comme suit :

$$P_1 \triangleq NiveauEt(et) = Bac$$
$$P_2 \triangleq NiveauEt(et) = These$$

D'après le tableau ci-dessus, prouver la propriété (5.5) revient à établir :

1. $\mathsf{Precl}(\bigvee_{1..2} R_i,\ P_2)$
2. $\mathsf{Prec}(P_1,\ R_1)$
3. $\mathsf{Prec}(P_1,\ R_2)$

En réitérant le processus sur les deux dernières propriétés, nous obtenons les cinq assertions suivantes :

1. $\mathsf{Precl}(NiveauEt(et) = 2AMaster \vee$
   $NiveauEt(et) = 3AGrandEcoles, NiveauEt(et) = These)$
2. $\mathsf{Precl}(NiveauEt(et) = 3AUniv,\ NiveauEt(et) = 2AMaster)$
3. $\mathsf{Precl}(NiveauEt(et) = Bac, NiveauEt(et) = 3AUniv)$
4. $\mathsf{Precl}(NiveauEt(et) = 2AClassPrepa,\ NiveauEt(et) = 3AGrandEcoles)$
5. $\mathsf{Precl}(NiveauEt(et) = Bac,\ NiveauEt(et) = 2AClassPrepa)$

Pour établir ces différentes assertions, nous les avons ajoutées à la clause **ASSERTIONS** de la machine B donnée à la page 94. Sur les 25 obligations de preuve générées, 22 ont été prouvées automatiquement ; les trois autres ont nécessité l'utilisation du prouveur interactif mais restent quand même faciles à décharger.

5.4 Preuve de correction

Dans cette section, nous donnons la preuve de correction de l'approche proposée pour la vérification de la propriété de précédence. L'objectif est de montrer que les conditions définies par les théorèmes 1 et 2 sont suffisantes et nécessaires pour la preuve de la formule (5.3).

5.4.1 Les conditions sont-elles suffisantes ?

Supposons que :

(i) $\forall(\vec{x}, \vec{y}, \vec{v}).((\neg P_1 \wedge \neg P_2) \wedge pre(op) \Rightarrow [S_{op}](P_1 \vee \neg P_2))$

ou qu'il existe un ensemble de prédicats $\{R_1, \ldots, R_n\}$ tels que :

(ii) $\mathsf{Precl}(\bigvee_{i\in 1..n} R_i,\ P_2)$

(iii) $\mathsf{Prec}(P_1,\ R_i)$ pour chaque prédicat $R_{i(i\in 1..n)}$

et prouvons que la formule (5.3) est vraie. Pour ce faire, considérons un chemin pa tel que $pa[k] \models P_2$ pour un rang k. Nous devons prouver qu'il existe un rang j tel que :

$$j \geq 0 \wedge j \leq k \wedge pa[j] \models P_1 \qquad (G_1)$$

Pour cela, soit k_1 le premier rang (état), de pa, qui vérifie le prédicat P_2 et supposons que l'opération op' est exécutée pour faire évoluer le système de l'état $pa[k_1 - 1]$ à l'état $pa[k_1]$.

$$k_1 > 0 \wedge pa[k_1 - 1] \models \neg P_2 \wedge pa[k_1 - 1] \models pre(op') \wedge \\ pa[k_1] \in op'(pa[k_1 - 1]) \wedge pa[k_1] \models P_2 \tag{H$_1$}$$

Deux cas sont à distinguer :

1. La condition (i) est vérifiée : en utilisant la formule (5.1), cette condition est réécrite en :

$$\forall ch. \left(\begin{array}{c} ch \in Chs \Rightarrow \forall i. \left(\begin{array}{c} \left(\begin{array}{c} i \geq 0 \\ \wedge \\ ch[i] \models (\neg P_1 \wedge \neg P_2) \\ \wedge \\ ch[i] \models pre(op) \end{array} \right) \\ \Rightarrow \\ \left(\begin{array}{c} ch[i+1] \in op(ch[i]) \\ \Rightarrow \\ ch[i+1] \models (P_1 \vee \neg P_2) \end{array} \right) \end{array} \right) \end{array} \right) \tag{H$_2$}$$

   Si $(pa[k_1 - 1] \models P_1)$, le but (G$_1$) est vérifié pour $j = k_1 - 1$. Sinon, l'instantiation de (H$_2$) avec pa, $(k_1 - 1)$ et op' donne :

$$\left(\begin{array}{c} k_1 - 1 \geq 0 \\ \wedge \\ pa[(k_1 - 1)] \models (\neg P_1 \wedge \neg P_2) \\ \wedge \\ pa[(k_1 - 1)] \models pre(op) \end{array} \right) \Rightarrow \left(\begin{array}{c} pa[k_1] \in op'(pa[(k_1 - 1)]) \\ \Rightarrow \\ pa[k_1] \models (P_1 \vee \neg P_2) \end{array} \right) \tag{H$_3$}$$

   (H$_2$)+(H$_3$)+Modus ponens donne :

$$pa[k_1] \models (P_1 \vee \neg P_2)$$

   Donc, nous concluons que le but (G$_1$) est vérifié pour $j = k_1$ car $(pa[k_1] \models P_2)$.

2. Les conditions (ii) et (iii) sont vérifiées : la condition (ii) se réécrit en :

$$\bigvee_{\mathsf{i}\in\mathsf{1..n}} \mathsf{Precl}(R_i,\ P_2)$$

Soit R' le prédicat associé au rang k_1-1 du chemin pa : $pa[k_1-1] \models R'$. La condition (iii) et la définition (5.3) impliquent :

$$\forall ch. \left(\begin{array}{c} ch \in Chs \\ \Rightarrow \\ \forall i. \left(\left(\begin{array}{c} i \geq 0 \\ \wedge \\ ch[i] \models R' \end{array} \right) \Rightarrow \exists l. \left(\begin{array}{c} l \geq 0 \wedge l \leq i \\ \wedge \\ ch[l] \models P_1 \end{array} \right) \right) \end{array} \right) \qquad (\mathrm{H}_4)$$

L'instantiation de (H_4) par pa et $k_1 - 1$ donne :

$$\left(\begin{array}{c} (k_1-1) \geq 0 \\ \wedge \\ pa[(k_1-1)] \models R' \end{array} \right) \Rightarrow \exists l. \left(\begin{array}{c} l \geq 0 \wedge l \leq (k_1-1) \\ \wedge \\ pa[l] \models P_1 \end{array} \right) \qquad (\mathrm{H}_5)$$

(H_5)+Modus ponens donne :

$$\exists l. \left(\begin{array}{c} l \geq 0 \wedge l \leq (k_1-1) \\ \wedge \\ pa[l] \models P_1 \end{array} \right) \qquad (\mathrm{H}_6)$$

Soit l_0 la valeur de l qui vérifie (H_6) :

$$l_0 \geq 0 \wedge l_0 \leq (k_1-1) \wedge pa[l_0] \models P_1$$

Finalement, le but (G_1) est vérifié pour $j = l_0$.

5.4.2 Les conditions sont-elles nécessaires ?

Pour montrer que l'une des conditions (i) et (ii) ou (iii) sont nécessaires, nous devons établir que celles-ci sont vérifiées si la propriété (5.3) est satisfaite. Supposons donc que (5.3) est vérifiée et montrons que les conditions (i) ou $((ii)$ et $(iii))$ sont satisfaites. Pour ce faire, raisonnons par l'absurde et supposons l'existence d'une opération op' pour laquelle il existe un chemin pa tel que pour un rang donné i_0, nous avons :

$$\left(\begin{array}{c} pa[i_0] \models (\neg P_1 \wedge \neg P_2) \\ \wedge \\ pa[i_0] \models pre(op') \\ \wedge \\ op'(pa[i_0]) \models (P_2 \wedge \neg P_1) \end{array}\right) \tag{H$_7$}$$

$$\neg Prec(P_1, Pred(pa[i_0])) \tag{H$_8$}$$

avec $Pred(\sigma)$ désignant le prédicat associé à l'état σ. Instancions la formule (5.3) avec pa et $i_0 + 1$:

$$\left(\begin{array}{c} i_0 + 1 \geq 0 \\ \wedge \\ pa[i_0 + 1] \models P_2 \end{array}\right) \tag{H$_9$}$$

$$\exists j. \left(\begin{array}{c} j \geq 0 \wedge j \leq i_0 + 1 \\ \wedge \\ pa[j] \models P_1 \end{array}\right) \tag{H$_{10}$}$$

(H$_9$) et (H$_{10}$) donne : $Prec(P_1, Pred(pa[i_0])$. Ceci contredit l'hypothèse (H$_8$).

5.5 Conclusion

Dans ce chapitre, nous avons présenté une approche formelle, fondée sur la preuve, pour la vérification de propriété d'absence. Exprimée en CTL par $\mathsf{Prec}(P_1, P_2)$, cette propriété permet de spécifier qu'un état vérifiant un prédicat P_2 ne peut être atteint avant un autre état satisfaisant P_1. C'est une façon d'exprimer un ordre sur l'apparition des événements du système. Pour ce faire, nous avons défini un ensemble de conditions suffisantes et nécessaires qui permettent de prouver une telle propriété. Nous avons également prouvé la correction de ces conditions.

Une propriété de précédence $\mathsf{Prec}(P_1, P_2)$ peut également s'exprimer comme une propriété d'absence $\mathsf{Abs}(P_2,\ \mathsf{From}\ Init\ \mathsf{Until}\ P_1)$ qui spécifie qu'un état vérifiant P_2 n'est pas atteignable, de l'état initial du système, jusqu'à ce que le prédicat P_1 soit vérifié. La preuve d'une propriété d'absence de la forme $\mathsf{Abs}(Q_2,\ \mathsf{From}\ Q_1\ \mathsf{Until}\ Q_3)$ a été explorée dans [10] pour laquelle deux conditions nécessaires et suffisantes ont été définies pour l'établissement d'une telle propriété avec l'idée clé de caractériser les états intermédiaires entre l'état Q_1 et Q_3 en définissant un prédicat Q' tel que pour chaque opération op, les deux assertions suivantes sont vérifiées :

$(i)\ \ (Q_1 \Rightarrow (\neg(Q_2 \vee Q') \vee Q_3))$

$(ii)\ \ (\neg(Q_2 \vee Q') \wedge pre(op) \Rightarrow [S_{op}](\neg(Q_2 \vee Q') \vee Q_3))$

En instanciant ces deux assertions sur notre étude de cas avec :

$Q_1 = (NiveauEt = \emptyset)$
$Q_2 = (NiveauEt(et) = PhD)$
$Q_3 = (NiveauEt(et) = Bac)$

$$Q' = \left(\begin{array}{c} NiveauEt(et) = 2YMaster \\ \vee \\ NiveauEt(et) = 3YUniv \\ \vee \\ NiveauEt(et) = Bac \\ \vee \\ NiveauEt(et) = 3YGrandEcoles \\ \vee \\ NiveauEt(et) = 2YPostSec \end{array} \right)$$

nous obtenons sept obligations de preuve dont une seule est prouvée automatiquement, les six restantes ont nécessité l'utilisation du prouveur interactif pour guider le prouveur dans son choix des règles à appliquer. De plus, ces preuves sont assez complexes car elles contiennent plusieurs termes. L'approche proposée dans ce chapitre est donc meilleure puisqu'elle produit des obligations de preuves plus simples que le prouveur de l'AtelierB décharge automatiquement.

Enfin dans [20], des règles de preuve sont proposées pour la preuve de formules LTL. La règle la plus proche de la propriété de précédence est la règle WAIT suivante :

$$\frac{P_1 \Rightarrow P_3 \vee \phi, \quad \phi \Rightarrow \neg P_2, \quad Pre(op) \wedge \phi \Rightarrow [op](P_3 \vee \phi)}{P_1 \Rightarrow (\neg P_2 \mathsf{W} P_3)}$$

En effet, la propriété de précédence $\mathsf{Prec}(P_1, P_2)$ peut s'exprimer en LTL par $(Init \Rightarrow (\neg P_2 \mathsf{W} P_1))$. La règle ci-dessus devient donc applicable. Cependant, la preuve de la prémisse $(\phi \Rightarrow \neg P_2)$ peut nécessiter l'ajout d'invariants qui doivent être prouvés sur toutes les opérations du système. Le processus de preuve devient alors long et fastidieux.

Chapitre 6

Implémentation

Sommaire

Dans le but d'automatiser les approches de vérification de propriétés dynamiques proposées dans ce manuscrit, nous avons développé en *Java* un outil que nous présentons dans ce chapitre. Cet outil prend en entrée la machine B représentant la spécification du système ainsi qu'un fichier contenant la propriété à vérifier et l'ensemble des informations nécessaires à la vérification (la liste des chemins, les prédicats intermédiaires, etc.). Dans la suite, ce dernier fichier est désigné par *fichier de spécification*. Afin d'analyser ces fichiers et d'extraire les informations qu'ils contiennent, nous nous sommes inspirés des travaux présentés dans [1] et réalisés sous l'environnement `SableCC` [11]. Cet environnement a été conçu dans le cadre du développement de compilateurs et d'interpréteurs. Pour vérifier une propriété dynamique, notre outil procède en suivant les étapes illustrées par la figure 6.1 :

1. *Analyse syntaxique des fichiers d'entrée* : consiste à analyser la machine B et le fichier de spécification représentant la propriété à vérifier afin de construire les arbres syntaxiques correspondants.
2. *Extraction des informations* : à partir des arbres syntaxiques obtenus précédemment, cette étape a pour but de transformer ces arbres en objets Java qui sont par la suite utilisés pour la génération des obligations de preuve.
3. *Génération des obligations de preuve* : cette étape applique les différentes approches proposées au cours de ce travail de thèse pour la génération des obligations de preuve permettant d'établir la propriété à vérifier.
4. *Inclusion des obligations de preuve dans la machine B* : les obligations de preuve générées à l'étape précédente sont ajoutées à la clause **ASSERTIONS** de la machine B initiale.

Nous commençons par une présentation de l'environnement `SableCC`, puis décrivons l'architecture générale de l'outil développé à l'aide d'un diagramme de classes. Enfin, nous expliquons l'implémentation des différentes approches de vérification des propriétés dynamiques.

6.1 SableCC

`SableCC` [11] est un environnement orienté objet permettant le développement de compilateurs et d'interpréteurs dans le langage de programmation Java. Il prend en entrée un

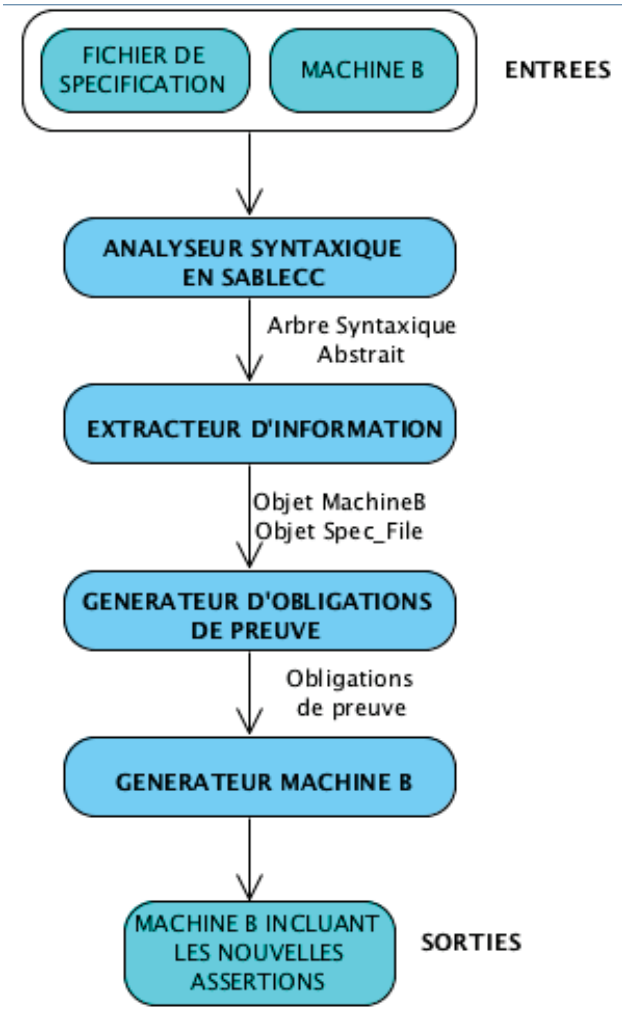

FIGURE 6.1 – Génération d'obligations de preuve pour les propriétés dynamiques

fichier contenant la description du langage pour lequel le compilateur doit être développé. Ce fichier peut contenir plusieurs clauses parmi lesquelles on distingue : la clause *Tokens* qui permet de définir les différents mots clés du langage, la clause *Production* qui définit la grammaire du langage, et enfin la clause *Abstract Syntax Tree* qui permet de créer l'arbre syntaxique. En sortie, `SableCC` génère plusieurs composants incluant, entre autres, un analyseur lexical, un analyseur syntaxique et un ensemble de classes. L'analyseur lexical permet de découper le texte à analyser en jetons conformément à ceux qui sont définis dans la clause *Tokens* ; l'analyseur syntaxique vérifie si ce texte respecte bien la structure fixée par la grammaire. Parmi les classes générées, nous retrouvons les classes *DepthFirstAdapter*

et *ReverseDepthFirstAdapter* qui permettent respectivement de parcourir en profondeur et en profondeur inverse l'arbre syntaxique. Pour effectuer des actions particulières sur cet arbre, il suffit d'étendre l'une de ces deux classes et de définir des méthodes dont les noms font référence aux nœuds sur lesquels on souhaite effectuer les actions.

Pour l'analyse des fichiers d'entrée de notre outil, nous avons sélectionné et étendu l'analyseur syntaxique du langage B développé à l'université de Dusseldorf [1] sous l'environnement `SableCC`. L'extension de cet analyseur a consisté en l'ajout de règles de grammaire pour analyser les différents éléments particuliers contenus dans les fichiers d'entrée de notre outil tout en réutilisant bien évidement la grammaire des éléments standards du langage B tels que les expressions, les substitutions et les prédicats.

6.2 Architecture structurelle générale de l'outil

L'architecture structurelle générale de notre outil de vérification de propriétés dynamiques est décrite par le diagramme de classes qui suit :

La figure 6.2 montre un ensemble de classes appelées *Extractors* qui héritent de la classe *DepthFirstAdapter*. Ces classes permettent d'extraire les informations contenues dans une machine B ou dans un fichier de spécification et de récupérer les objets stockant les informations extraites. Parmi ces classes, nous avons la classe *Extractor_MachineB* qui permet de récupérer un objet de type *MachineB* à travers la méthode *get_machine*() et les classes *Extractor_Spec_file_** qui permettent de récupérer des objets de type *Spec_File* à travers leurs méthodes *get_spec*(). Nous distinguons également la classe abstraite *Verif_Property* dont hérite toute classe qui implémente une approche de vérification d'une propriété dynamique. Cette classe compte, parmi ses attributs, un objet de type *MachineB* et un objet de type *Spec_File* qui sont instanciés au niveau du constructeur ; et déclare de manière abstraite une méthode *run*() qui doit être implémentée par chacune de ses sous-classes. Notons qu'une sous-classe abstraite peut déléguer à nouveau l'implémentation de cette méthode à ses descendants.

Pour automatiser les deux méthodes de vérification de propriété d'atteignabilité basées sur les algorithmes de génération d'obligations de preuve (voir chapitre 3), nous avons implémenté la classe abstraite *Verif_Property_Reachability_Algo* qui définit les méthodes et attributs communs aux deux algorithmes proposés. Cette classe est étendue par les classes

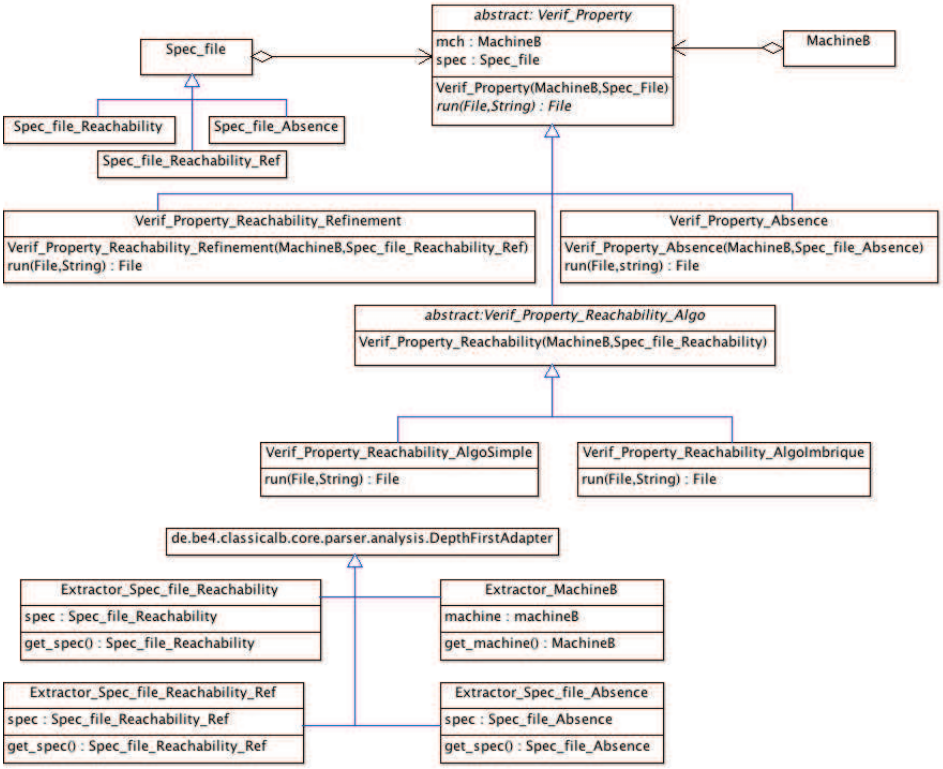

FIGURE 6.2 – Architecture structurelle de l'outil

Verif_Property_Reachability_AlgoSimple et *Verif_Property_Reachability_AlgoImbrique* qui implémentent respectivement la version simple et la version imbriquée de l'algorithme de génération d'obligations de preuve. Nous avons aussi défini la classe *Verif_Property_Reachability_Ref* qui implémente l'approche de vérification de propriété d'atteignabilité par raffinement présentée au chapitre 4.

Dans [3] et [10] ont été proposées deux approches de vérification de propriétés d'absence que nous avons également implémentées par les classes *Verif_Property_Absence_From* et *Verif_Property_Absence_After* qui héritent de la classe *Verif_Property_Absence*. Les sections suivantes détaillent l'implémentation de chacune de ces approches.

6.3 Implémentation de la vérification de la propriété d'atteignabilité

6.3.1 Approches basées sur les chemins

Cette section présente l'implémentation des deux approches de vérification de propriété d'atteignabilité basées sur des algorithmes de génération d'obligations de preuve. Rappelons que ces deux méthodes utilisent la même technique, mais se distinguent sur la forme finale des assertions générées. La deuxième version (imbriquée) a été définie dans le but de simplifier les assertions générées par la première en les imbriquant les unes dans les autres. Dans la section suivante, nous décrivons la grammaire BNF du fichier de spécification, celle-ci étant commune aux deux approches.

6.3.1.1 Grammaire du fichier de spécification

Nous définissons cette grammaire en réutilisant les termes "Predicate" et "Expression" déjà définis dans l'analyseur syntaxique que nous étendons. Ces termes permettent de prendre en compte tous les prédicats et expressions définis dans le langage B. Le fichier de spécification que doit fournir l'utilisateur est décrit selon la grammaire BNF suivante :

$FICHIER\_SPEC ::=$ **SPECIFICATION FILE**
FROM *Predicate*
TO *Predicate*
PATHS
$(PATH$ **END** $)^+$
$PATH ::= Predicate \sim> (ACTION)^{+";"}$
$ACTION ::= Operation(param_1, .., param_n) \mid$
$\mid_{id \in Expression} Operation(id, ...) \mid$
LOOP $(Predicate, \mid_{id \in Expression} Operation(id, ...), Expression) \mid$
IF *Predicate* **THEN** $ACTION$

La première ligne indique qu'il s'agit d'un fichier de spécification. Ensuite, nous avons deux clauses **FROM** et **TO** qui expriment respectivement les deux prédicats ψ et ϕ d'une

propriété d'atteignabilité $\mathsf{AG}(\psi \Rightarrow \mathsf{EF}\phi)$. Enfin une liste de chemins possibles pour une telle propriétés est précisée dans la clause **PATHS**. Comme le montre la grammaire BNF ci-dessus, chaque chemin peut être gardé par un prédicat et consiste en une séquence d'actions. Le fichier de spécification suivant correspond à la propriété d'atteignabilité (voir section 3.2.2) du chapitre 3.

SPECIFICATION FILE
FROM $ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt$
TO $ca \mapsto cl \in Emprunt$
PATHS
$Reservation(ca) = [] \rightsquigarrow$
IF $ca \in dom(Emprunt)$ **THEN** $Rendre(ca)$;
IF $card(Emprunt^{-1}[\{cl\}]) = MaxNbLoan$ **THEN**
$| ca_1 \in Emprunt^{-1}[\{cl\}]$ $Rendre(ca_1)$;
$Emprunter(cl, ca)$

END $Reservation(ca) \neq [] \rightsquigarrow$
IF $cl \notin ran(Reservation(ca))$ **THEN** $Reserver(cl, ca)$;
LOOP $(reservation(ca) \neq [cl]$,
$| oo \in ran(Reservation(ca)) - \{cl\}$ $Annuler\_Reservation(oo, ca)$,
$size(ran(Reservation(ca))))$;
IF $ca \in dom(Emprunt)$ **THEN** $Rendre(ca)$;
IF $card(Emprunt^{-1}[\{cl\}]) = 2$ **THEN** $| ca_1 \in Emprunt^{-1}[\{cl\}]$ $Rendre(ca_1)$;
$Empunter(cl, ca)$
END

6.3.1.2 Implémentation

Une fois l'analyse syntaxique et l'extraction des informations effectuées sur le fichier de spécification et la machine B, nous pouvons maintenant appliquer nos approches pour la génération des obligations de preuve. Comme les deux algorithmes proposés ne se distinguent que sur la forme des assertions générées, nous avons défini une classe abstraite *Verif_ Property_ Reachability_ Algo* qui hérite de la classe *Verif_ Property* et qui regroupe l'ensemble des attributs et méthodes communs à ces deux algorithmes. Les classes *Verif_ Property_ Reachability_ AlgoSimple* et *Verif_ Property_ Reachability_ AlgoImbrique* qui implémentent respectivement la version simple et la version imbriquée de l'algorithme héritent ainsi de la classe *Verif_ Property_ Reachability_ Algo*. Elles implémentent égale-

ment la méthode *run*() définie dans la classe *Verif_Property*. Ces classes sont décrites dans les paragraphes suivants.

Classe *Verif_Property_Reachability_Algo* Nous avons défini dans cette classe deux méthodes $getOP\_SUBS()$ et $getOP\_PRE()$ qui permettent de renvoyer la substitution et la précondition de l'opération associées à une action donnée. Aussi, ces opérations substituent les paramètres formels de l'opération par les paramètres effectifs en utilisant la méthode *get_param_appel*() qui permet de récupérer les paramètres formels d'une opération. Dans cette classe est aussi définie la méthode *Pre*() qui retourne la précondition d'une action. Les principales opérations utilisées pour l'implémentation de ces algorithmes sont localisées dans les classes *Operations* et *Substitutions*.

1. **Classe *Operations*** : les deux fonctions *Update* et *Cons_Po* utilisées dans l'algorithme de génération d'obligations de preuve pour mettre à jour respectivement les hypothèses et l'ensemble des formules à prouver sont implémentées par les méthodes *Update*() et *Cons_Po*() de la classe *Operations*. Cette classe inclut *GammaAct* qui permet d'effectuer la substitution $\gamma_j(act) = [||\ x_i := x_i^{j-1}]act$ en prenant en compte les différents types d'action. Elle définit aussi la méthode *Post*() qui permet de calculer la postcondition d'une action.
2. **Classe *Substitutions*** : dans cette classe sont définies toutes les opérations permettant d'effectuer des substitutions. Comme exemple, nous citons la méthode *substitute*() qui remplace, dans une substitution, les paramètres formels d'une opération par les paramètres effectifs en utilisant les méthodes *subsExpression*() et *subsPredicate*(). Les méthodes *subsExpression*() et *subsPredicate*() permettent de substituer, dans une expression (resp. un prédicat), les paramètres formels d'une opération par les paramètres effectifs. Ces deux méthodes ont également été utilisées par l'opération *GammaAct* de la classe *Operations* afin d'effectuer la substitution des variables x_i par les variables x_i^{j-1}.

Classe *Verif_Property_Reachability_AlgoSimple* La méthode *run*() de la classe *Verif_Pro- perty_Reachability_AlgoSimple* correspond à la traduction Java de la version simple ou sans imbrication de l'algorithme de génération d'obligations de preuve présenté dans le chapitre 3. Cette méthode utilise les opérations définies dans la classe

Verif_Property_Reachability_- Algo dont elle hérite, ainsi que celles des classes *Operations* et *Substitutions* introduites précédemment.

Classe *Verif_Property_Reachability_AlgoImbrique* La version imbriquée de l'algorithme de génération d'obligations de preuve introduite dans le chapitre 3 est implémentée par la méthode *run*() de la classe *Verif_Property_Reachability_AlgoImbrique*. Cette méthode fait appel à l'opération $Operations\_vers\_imbriquee.Update()$ qui, contrairement à $Operations.Update()$, permet d'imbriquer les assertions générées. Exceptée la méthode $Update()$, dont l'implémentation constitue la principale différence entre les deux versions de l'algorithme, la méthode $Verif\_Property\_Reachability\_AlgoImbrique.run()$ réutilise les mêmes opérations que $Verif\_Property\_Reachability\_AlgoSimple.run()$.

6.3.2 Approche basée sur le raffinement

Cette section décrit l'implémentation de la méthode de vérification de propriété d'atteignabilité par raffinement introduite dans le chapitre 4. Elle est organisée en trois parties. D'abord, nous allons définir la grammaire BNF du fichier de spécification. Ensuite, nous allons identifier les vérifications à effectuer sur ce fichier afin de garantir que les informations qu'il contient sont correctes. Pour finir, nous allons expliquer l'implémentation de l'approche de vérification.

6.3.2.1 Grammaire du fichier de spécification

Le fichier de spécification fourni par l'utilisateur doit décrire la propriété d'atteignabilité à vérifier ainsi que l'arbre de raffinement permettant à l'outil de générer les obligations de preuve nécessaires à la vérification de cette propriété. La grammaire de ce fichier utilise la notion de *Substitution* dont la grammaire BNF est définie comme suit :

Substitution ::=

Operation$(param_1, .., param_n)$ |

ANY x **WHERE** $x \in$ *Expression* **THEN** *Substitution* **END** |

IF *Predicate* **THEN** *Substitution*[**ELSE** *Substitution*] **END** |

CHOICE *Substitution* **OR** *Substitution* **END** |

SELECT *Predicate* **THEN** *Substitution*

(**WHEN** *Predicate* **THEN** *Substitution*)*

[**ELSE** *Substitution*]

END |

WHILE *Predicate* **DO** *Substitution*

INVARIANT *Predicate*

VARIANT *Expression*

END |

Substitution **;***Substitution* |

Spec$(Predicate, Predicate)$ |

IncreasedBy$(Predicate, Predicate)$

avec la fonction *IncreasedBy* définie par :

$$IncreasedBy(P_1, P_2) == Spec(P_1, P_1 \wedge P_2)$$

cette fonction a été définie dans le but de simplifier l'écriture des spécifications en évitant de faire des répétitions. Nous réduisons ainsi la taille du fichier de spécification fourni par l'utilisateur. On définit aussi une substitution *terminale* comme toute substitution qui n'est ni de type *Spec* ni de type *IncreasedBy*. De plus, elle ne doit inclure aucune substitution de type *spec* ou *IncreasedBy*. La grammaire BNF décrivant le fichier de spécification est le suivant :

$Fichier\_Spec ::=$ **SPECIFICATION FILE**
FROM $Predicate$
TO $Predicate$
DEFINITIONS
(**LET** $Name = Predicate$)*
REFINEMENT TREE
((**Spec**$(Predicate, Predicate)$ |
IncreasedBy$(Predicate, Predicate)) \sqsubseteq Substitution)^+$

La première ligne indique que ce fichier est un fichier de spécification. Les deux prédicats *FROM* et *TO* permettent de décrire la propriété d'atteignabilité. La suite du fichier est organisée en deux clauses : *DEFINITIONS* et *REFINEMENT TREE*. La clause *DEFINITIONS* permet d'associer un nom à un prédicat afin de faciliter la description de l'arbre de raffinement dans la clause *REFINEMENT TREE*. Cette deuxième clause est constituée d'une suite de relations de raffinement ; chacune est composée d'une partie gauche correspondant à une spécification et d'une partie droite représentant une substitution. Cependant, une analyse syntaxique du fichier de spécification ne garantit pas la bonne définition et la correction de l'arbre de raffinement. Par conséquent, des vérifications supplémentaires, identifiées dans la section suivante, doivent être réalisées. Le fichier de spécification suivant correspond à la propriété d'atteignabilité (voir 3.2.2) du chapitre 3.

SPECIFICATION FILE

FROM $ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt$

TO $ca \mapsto cl \in Emprunt$

DEFINITIONS

LET $P_1 \triangleq ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt \wedge card(Emprunt^{-1}[\{cl\}]) < 2$
$\wedge ca \notin dom(Emprunt) \wedge Reservation(ca) = []$

LET $P_2 \triangleq ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt \wedge card(Emprunt^{-1}[\{cl\}]) < 2$

LET $P_3 \triangleq ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt \wedge card(Emprunt^{-1}[\{cl\}]) < 2$
$\wedge ca \notin dom(Emprunt)$

REFINEMENT TREE

$Spec(ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt, ca \mapsto cl \in Emprunt)$

REF

$Spec(ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt, P_1);$

$Spec(P_1, ca \mapsto cl \in Emprunt)$

$Spec(P_1, ca \mapsto cl \in Emprunt)$

REF

$Emprunter(cl, ca)$

$Spec(ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt, P_1)$

REF

$IncreasedBy(ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt,$
$card(Emprunt^{-1}[\{cl\}]) < 2);$

$Spec(P_2, P_1)$

$IncreasedBy(ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt,$
$card(Emprunt^{-1}[\{cl\}]) < 2)$

REF

IF $not(card(Emprunt^{-1}[\{cl\}]) < 2)$ **THEN**

ANY ca_1 **WHERE** $ca_1 \in Emprunt^{-1}[\{cl\}]$ **THEN** $Rendre(ca_1)$ **END**

END

$Spec(P_2, P_1)$

REF

$IncreasedBy(P_2, ca \notin dom(Emprunt)); Spec(P_3, P_1)$

$IncreasedBy(P_2, ca \notin dom(Emprunt))$

REF

IF $not(ca \notin dom(Emprunt))$ **THEN** $Rendre(ca)$ **END**

$Spec(P_3, P_1)$
REF
WHILE $\neg(Reservation(ca) = [])$**DO**
ANY cl_1 **WHERE** $cl_1 \in ran(Reservation(ca))$
THEN $Annuler\_Reservation(cl_1, ca)$ **END**
INVARIANT $ca \in Cassettes \wedge cl \in Clients \wedge ca \mapsto cl \notin Emprunt \wedge$
$card(Emprunt^{-1}[\{cl\}]) < 2 \wedge ca \notin dom(Emprunt)$
VARIANT $size(Reservation(ca))$
END

6.3.2.2 Vérification du fichier de spécification

La conformité du fichier de spécification par rapport à sa grammaire BNF n'est pas une condition suffisante pour garantir la bonne définition et la correction de ce dernier. Nous considérons qu'un tel arbre est bien défini et correct si et seulement si les conditions suivantes sont vérifiées :

1. la racine de l'arbre correspond à la spécification $Spec(\psi, \phi)$ avec ψ et ϕ désignant les prédicats apparaissant dans les clauses *FROM* et *TO*,
2. la spécification correspondante à la racine de l'arbre est bien raffinée, c.à.d., il doit exister, dans la clause *REFINEMENT TREE*, une et une seule relation de raffinement dont la partie gauche correspond à cette spécification que nous appelons spécification principale.
3. toutes les feuilles de l'arbre sont des substitutions terminales, c.à".d., pour toute spécification définie, il existe une relation de raffinement dans ce fichier telle que cette spécification soit sa composante gauche. Ainsi, nous garantissons que toute branche de l'arbre se terminer par une substitution terminale.

6.3.2.3 Implémentation

Pour automatiser l'approche de vérification de propriété d'atteignabilité par raffinement, nous avons implémenté la classe *Verif_ Property_ Reachability_ Ref* qui est une sous-classe de la classe *Verif_ Property*. La classe *Verif_ Property_ Reachability_ Ref* hérite les attributs de sa classe mère dont un de type *MachineB* et un autre de type *Spec_ file* qu'elle

instancie au niveau de son constructeur. L'attribut de type *MachineB* comprend les informations de la machine B ; celui de type *Spec_file* inclut les informations du fichier de spécification. La méthode principale de cette classe est la méthode *run*() définie comme suit :

```
1  public File run(File machine, String name) {
2  /*Remplacement des prédicats déclarés dans la partie DEFINITIONS */
3      for(int i=0; i< ref.getList_refinements().size(); i++){
4     rel= (ARefinementRelation)ref.getList_refinements().get(i);
5     ref.getList_refinements().set(i, replace_variable(rel));
6      }
7  /*Vérification de l'arbre de raffinement */
8      Verification_tree_refinement verif= new Verification_tree_refinement();
9      try{
10    verif.run(ref);
11     }
12     catch(Spec_Not_Refined_Exception e)
13    {System.out.println(e.getMessage()); }
14
15 /*Génération de l'obligation de preuve correspondant  */
16 /*à  chaque relation de raffinement */
17     for(int i=0; i< ref.getList_refinements().size(); i++){
18    rel= (ARefinementRelation)ref.getList_refinements().get(i);
19    ArrayList<PPredicate> results= generated_PO(rel);
20    if(!(results.isEmpty())){
21       POs.addAll(results);
22    }
23    }
24   return machine_with_assertions(POs, machine, name);
25 }
```

L'objet *ref* de la méthode *run*() est un attribut de la classe *Verif_Property_Reachability_Ref* ; il est de type *Spec_file_Reachability_Ref*. Cette méthode comporte les quatre parties suivantes :

– **Remplacement des prédicats** (lignes de 2 à 6) : cette phase consiste à remplacer, au niveau des relations de raffinement, tous les prédicats déclarés dans la clause

DEFINITIONS par leurs valeurs respectives. Si l'identifiant d'un prédicat mentionné dans une relation de raffinement n'est pas défini dans la clause *DEFINITIONS*, alors une exception notifiant cette erreur est levée.

- **Vérification du fichier de spécification** (lignes 7 à 13) : dans cette partie, nous effectuons, sur le fichier de spécification, les vérifications identifiées à la section 6.3.2.2. Pour ce faire, on fait appel à la méthode $Verification\_tree\_refinement.run()$ en lui transmettant un objet de type *Spec_file_Reachability_Ref*.
- **Génération des obligations de preuve** (lignes 17 à 23) : cette troisième partie permet de générer les obligations de preuve relatives à chaque relation de raffinement en faisant appel à la méthode *generated_PO*. Les obligations de preuve ainsi générées sont rajoutées dans la clause *ASSERTIONS* de la machine B pour être prouvées à partir des invariants.
- **Génération de la nouvelle machine B** (ligne 24) : cette dernière partie consiste à générer un fichier qui correspond à la machine B initiale donnée en paramètre et à laquelle sont rajoutées les obligations de preuves générées comme des assertions.

6.4 Vérification de la propriété d'absence

6.4.1 Introduction

Dans cette section, nous présentons l'implémentation de la vérification de deux propriétés d'absence. La première approche de vérification, introduite dans [3], permet de prouver une propriété d'absence exprimée par $\mathsf{Abs}(P_2,\ \mathsf{From}\ P_1\ \mathsf{Until}\ P_3)$. Cette propriété spécifie qu'à partir d'un état satisfaisant le prédicat P_1, il est impossible d'atteindre un état où P_2 serait vrai tant qu'on a pas eu un état où le prédicat P_3 est satisfait.

La deuxième approche, proposée dans [10], permet quant à elle de vérifier une propriété d'absence de la forme $\mathsf{Abs}(P_2,\ \mathsf{After}\ P_1\ \mathsf{Until}\ P_3)$ qui exprime : après un état satisfaisant P_1, on ne peut avoir un état où P_2 est vrai tant qu'on a pas atteint un état qui satisfait P_3. Le pattern **From-Until** spécifie qu'à l'état où P_1 est vrai, soit P_3 est vrai soit P_2 n'est pas satisfait contrairement au pattern **After-Until** qui ne prend pas en compte cet état mais l'état suivant. Les obligations de preuve nécessaires et suffisantes à la vérification de ces deux patterns sont données ci-dessous.

6.4.1.1 Vérification du pattern From-Until

Les travaux réalisés dans [3] montrent que pour prouver une propriété d'absence de la forme $\mathsf{Abs}(P_2,\ \mathsf{From}\ P_1\ \mathsf{Until}\ P_3)$, il faut et suffit d'exhiber un prédicat P' tel que pour chaque opération *op* du système :

- **OP1** : $\forall \vec{v}.(P_1 \Rightarrow (\neg(P_2 \vee P') \vee P_3))$
- **OP2** : $\forall \vec{v}.(\neg(P_2 \vee P') \wedge Pre(op) \Rightarrow [S_{op}](\neg(P_2 \vee P') \vee P_3))$

avec v incluant les paramètres formels de l'opération *op* et les variables libres des prédicats P_1, P_2 et P_3. L'obligation de preuve **OP1** vérifie que si le prédicat P_1 est vrai, alors soit P_3 est vrai soit P_2 n'est pas établi. L'obligation de preuve **OP2** permet de prouver qu'après l'exécution de chaque opération, le système se trouve dans un état où P_3 est vrai ou P_2 reste non satisfait. Le prédicat $\neg(P_2 \vee P')$ permet de caractériser les états intermédiaires entre ceux vérifiant P_1 et P_3.

6.4.1.2 Vérification du pattern pattern After-Until

Afin de prouver une propriété d'absence de la forme $\mathsf{Abs}(P_2,\ \mathsf{After}\ P_1\ \mathsf{Until}\ P_3)$, les travaux réalisés dans [10] montrent qu'il faut et suffit d'exhiber un prédicat P' tel que pour chaque opération *op* du système :

- **OP3** : $\forall \vec{v}.(P_1 \wedge Pre(op) \Rightarrow [S_{op}](\neg(P_2 \vee P') \vee P_3))$
- **OP4** : $\forall \vec{v}.(\neg(P_2 \vee P') \wedge pre(op) \Rightarrow [S_{op}](\neg(P_2 \vee P') \vee P_3))$

avec v incluant les paramètres formels de l'opération *op* et les variables libres des prédicats P_1, P_2 et P_3. L'obligation de preuve **OP3** permet d'établir que si P_1 est vrai, alors après exécution d'une opération *op*, on se retrouve dans un état où soit P_3 devient vrai soit P_2 reste non satisfait. L'obligation de preuve **OP4** est similaire à **OP2** définie précédemment.

Nous avons implémenté ces approches afin de permettre une génération automatique de ces obligations de preuve. Pour cela, l'utilisateur doit fournir à l'outil une machine B et un fichier de spécification dont la grammaire est définie dans la section suivante.

6.4.2 Grammaire du fichier de spécification

La grammaire suivante décrit le fichier de spécification, point d'entrée de l'approche de vérification d'une propriété d'absence :

$FICHIER\_SPEC ::=$
LOCAL VARIABLES $(TYPAGE)^{*","}$
ABSENCE_OF $Predicate$
FROM | **AFTER** $Predicate$
TO $Predicate$
INVARIANT $Predicate$

$TYPAGE ::= id : Expression$

Les prédicats P_2, P_3 et P' définis précédemment sont exprimés respectivement dans les clauses **ABSENCE_OF**, **TO** et **INVARIANT**. En fonction de la propriété d'absence à vérifier, le prédicat P_1 est exprimé dans la clause **FROM** ou **AFTER**. Les types des variables utilisées pour l'expression des prédicats sont définis dans la clause **LOCAL VARIABLES**. Remarquons que l'utilisateur n'a pas besoin de spécifier les types des variables dans les fichiers de spécification précédents. En effet dans ces cas, les variables sont utilisées comme paramètres formels d'opérations et on peut ainsi déduire leur type à partir de leur précondition. Le fichier de spécification suivant correspond à la propriété d'absence :

$$\begin{array}{l}\mathsf{Abs}(ca \mapsto cl_2 \in Emprunt,\\ \quad \mathsf{After}(cl_1 \in ran(Reservation(ca)) \wedge cl_2 \notin ran(Reservation(ca)) \wedge\\ \qquad ca \mapsto cl_2 \notin Emprunt)\\ \quad \mathsf{Until}(ca \mapsto cl_1 \in Emprunt))\end{array}$$

SPECIFICATION FILE
LOCAL VARIABLES $cl_1 \in Clients,\ cl_2 \in Clients,\ ca \in Cassettes$
ABSENCE_OF $ca \mapsto cl_2 \in Emprunt$
AFTER $cl_1 \in ran(Reservation(ca) \wedge cl_2 \notin ran(Reservation(ca)) \wedge ca \mapsto cl_2 \notin Emprunt$
TO $ca \mapsto cl_1 \in Emprunt$
INVARIANT
$(cl_1 \notin ran(Reservation(ca)))\vee$
$(cl_2 \in ran(Reservation(ca)) \wedge Indice(ca, cl_2) < Indice(ca, cl_1))$

6.4.3 Implémentation

Les deux approches présentées précédemment ont été implémentées en définissant les classes *Verif_Property_Absence_From* et *Verif_Property_Absence_After* qui héritent de la classe abstraite *Verif_Property_Absence*. Parmi les attributs de la classe *Verif_Property_Absence*, nous distinguons un objet de type *MachineB* qui contient les éléments de la machine B et un objet de type *Spec_file_Absence* qui correspond au fichier de spécification. Les méthodes $Verif\_Property\_Absence\_From.run()$ et *Verif_Property_Absence_After.run()* génèrent respectivement les obligations de preuve vérifiant les propriétés de la forme :

$$\mathsf{Abs}(P_2, \mathsf{From}\ P_1\ \mathsf{Until}\ P_3)$$
$$\mathsf{Abs}(P_2,\ \mathsf{After}\ P_1\ \mathsf{Until} P_3).$$

Présentons tout d'abord la méthode $Verif\_Property\_Absence\_From.run()$ qui peut être décomposée en quatre parties :

- **Récupération des prédicats** : cette première partie permet de récupérer les informations contenues dans le fichier de spécification à travers l'objet *Spec_file_Absence*. à ce stade, sont aussi extraites les variables typées dans la clause **LOCAL VARIABLES**. Ces variables sont quantifiées universellement lors de la génération des obligations de preuve.
- **Génération de OP1** : cette partie permet de générer l'obligation de preuve **OP1**.
- **Génération de OP2** : dans cette troisième phase, nous générons l'obligation de preuve **OP2** pour chaque opération de la machine B. Afin d'appliquer la substitution de l'opération sur le prédicat $(\neg(P_2 \vee P') \vee P_3)$, nous faisons appel à la méthode $Substitution.subs\_in\_pred()$ qui calcule l'effet de l'exécution d'une substitution sur un prédicat.
- **Génération de la nouvelle machine B** : cette dernière phase consiste à générer un fichier qui correspond à la machine B initiale à laquelle sont rajoutées les obligations de preuve générées dans la clause **ASSERTIONS**.

La méthode $Verif\_Property\_Absence\_After.run()$ peut être décomposée en les trois étape suivantes :

- **Récupération des prédicats** : cette étape est similaire à la première étape de la méthode $Spec\_file\_Absence\_From.run()$.

– **Génération de OP3 et OP4** : cette étape consiste à générer les obligations de preuve **OP3** et **OP4** pour chaque opération de la machine B.
– **Génération de la nouvelle machine B** : cette étape est similaire à la quatrième étape de la méthode $Spec\_file\_Absence\_From.run()$.

6.5 Conclusion

Dans ce chapitre, nous avons présenté l'architecture structurelle et fonctionnelle de l'outil que nous avons développé pour la vérification de diverses propriétés dynamiques en générant des obligations de preuve intégrées comme assertion à la machine B modélisant le système sur laquelle portent les propriétés à vérifier. Programmé en langage Java (environ 4400 lignes de code), cet outil peut être étendu facilement pour la prise en compte d'autres types de propriétés dynamiques. D'un point de vue performance et temps d'exécution, l'outil génère les obligations de preuve de la propriété d'atteignabilité associées au ch_1 (Voir page 59 du chapitre 3) en moins de deux secondes et ce pour les différentes approches de vérification présentées dans ce manuscrit. On retrouve les mêmes performances pour la propriété d'absence pour une machine B constituée de cinq operations.

Chapitre 7

Conclusion et perspectives

Sommaire

Les méthodes formelles sont de plus en plus utilisées pour le développement de systèmes non réellement critiques comme les systèmes d'information pour risque financier par exemple. En effet, une mauvaise conception des systèmes d'information pourrait entrainer des dysfonctionnements susceptibles de causer des dommages importants. Néanmoins, la plupart des travaux développés jusqu'à présent s'intéresse plus à des propriétés statiques, dites d'invariance, qui ne prennent en compte qu'un seul état du système à la fois. Cependant, la qualité d'un système ne se restreint pas à la vérification de propriétés statiques mais doit aussi considérer d'autres types de propriétés, dynamiques par exemple. Une propriété dynamique porte sur plusieurs états du système pris à des instants différents. Le travail présenté dans cette thèse répond à ce besoin de vérifier des propriétés dynamiques sur les systèmes d'information en proposant des approches formelles fondées sur la preuve en utilisant la méthode B. Nous avons opté pour l'utilisation de la preuve au détriment du model checking car ce dernier souffre du problème d'explosion combinatoire de l'espace des états et limite donc la taille des systèmes qu'on peut vérifier.

7.1 Contributions

Les contributions de ce travail de thèse sont les suivantes :

1. *vérification de propriété d'atteignabilité* : une propriété d'atteignabilité permet de s'assurer qu'à partir d'un état donné s, le système a la possibilité d'évoluer pour atteindre un état cible s'. Pour prouver une telle propriété, nous avons proposé trois approches principales que nous avons évidemment prouvées :
   - la première approche se base sur la notion de chemin (séquence d'actions) et définit un algorithme de génération d'assertions B suffisantes qui vérifient que le chemin permet bien de mener de l'état s à l'état s'. L'idée clé de ces assertions consiste à s'assurer qu'il est possible d'exécuter la première action à partir de l'état initial s, que les valeurs intermédiaires des variables permettent d'exécuter l'action suivante, et puis enfin l'exécution de la dernière action permet d'atteindre l'état s'.
   - la seconde utilise le calcul de la plus faible pré-condition et produit également des assertions à établir. Cette approche adopte un raisonnement inversé car, dans ce cas, il s'agit d'établir que l'état s appartient au plus large ensemble des états

permettant d'atteindre s' en exécutant un chemin donné.

– la dernière se base sur le principe de raffinement de substitutions de B et consiste à générer par raffinement successif un programme (chemin) dont l'exécution de l'état s termine dans l'état s'. La correction d'une telle approche est assurée par la preuve des différentes étapes de raffinement.

Pour chacune de ces approches, nous avons dégagé ses avantages et ses limites.

2. *vérification de propriété de précédence* : une propriété de précédence permet d'exprimer qu'un état donné du système précède toujours un autre état. Une telle propriété permet d'exprimer un ordre sur certains états du système. Pour vérifier une telle propriété, nous avons défini des conditions nécessaires et suffisantes dont nous avons aussi prouvé la correction. Pour une propriété de précédence "P_2 précède P_1", de telles conditions permettent de vérifier que sur tout chemin, issu de l'état initial, s'il existe un état qui vérifie un prédicat P_2 alors ce dernier est précédé par un état vérifiant un prédicat P_1. Nous avons bien évidemment prouvé le caractère suffisant et nécessaire des conditions définies.

3. développement d'un outil support : dans le but de rendre les approches proposées opérationnelles et décharger l'utilisateur de la tâche longue et fastidieuse de génération d'obligations de preuve, nous avons réalisé un outil qui implémente les différentes approches. Développé en JAVA, cet outil génère les différentes obligations de preuve associées à une propriété dynamique et ce à partir de la spécification B du système et de la description textuelle de la propriété. La phase de génération des obligations de preuve est précédée par une étape de vérification syntaxique pour s'assurer, en particulier, de la bonne expression des propriétés et des éléments nécessaires à la génération des obligations de preuve.

7.2 Perspectives

Le travail de recherche présenté dans ce manuscrit ouvre plusieurs perspectives possibles. Notre travail pourrait être étendu afin de prendre en compte d'autres types de propriétés qui pourraient être pertinents pour les systèmes d'information, comme les réponses en chaîne pour exprimer par exemple que l'annulation d'un voyage par un client

doit être suivie par l'annulation de sa réservation d'hôtel, du remboursement éventuel des payements, etc. Concernant l'implémentation, le développement d'une interface graphique permettrait de rendre l'utilisation de notre outil plus intuitive et plus aisée. Une telle interface comportera, par exemple, des champs texte pour la saisie des propriétés à vérifier, mais également un menu permettant de parcourir l'arborescence de la machine pour la sélection des fichiers représentant la spécification du système ou de la propriété.

Annexe A. Compléments sur le langage B

A1. Principaux opérateurs relationnels du langage B

Nom	Notation B	Définition	Condition
Produit cartésien	$s \times t$	$\{x \mapsto y \mid x \in s \wedge y \in t\}$	
Ensemble des parties de S	$\mathbb{P}\ S$	$\{x \mid x \subseteq S\}$	
Relation	$s \leftrightarrow t$	$\mathbb{P}(s \times t)$	
Inverse	r^{-1}	$\{x \mapsto y \mid y \mapsto x \in r\}$	$r \in s \leftrightarrow t$
Domaine	$dom(r)$	$\{x \mid x \in s \wedge \exists y.(y \in t \wedge x \mapsto y \in r\}$	$r \in s \leftrightarrow t$
Codomaine	$ran(r)$	$\{y \mid y \in t \wedge \exists x.(x \in s \wedge x \mapsto y \in r\}$	$r \in s \leftrightarrow t$
Image	$r[X]$	$\{y \mid y \in t \wedge \exists x.(x \in X \wedge x \mapsto y \in r)\}$	$r \in s \leftrightarrow t$ $X \subseteq s$
Surcharge	$f \lhd\!\!+ g$	$\{x \mapsto y \mid x \mapsto y \in f \wedge x \notin dom(g)\} \cup g$	$f \in s \leftrightarrow t$ $g \in s \leftrightarrow t$
Anti-Restriction	$u \lhd\!\!\!- r$	$\{x \mapsto y \mid x \mapsto y \in r \wedge x \notin u\}$	$r \in s \leftrightarrow t$ $u \subseteq s$
Fonction partielle	$s \nrightarrow t$	$\{f \mid f \in s \leftrightarrow t \wedge$ $\forall x.(x \in s \Rightarrow card(f[\{x\}]) \leq 1)\}$	
Fonction totale	$s \rightarrow t$	$\{f \mid f \in s \nrightarrow t \wedge dom(f) = s\}$	
Injection partielle	$s \rightarrowtail t$	$\{f \mid f \in s \nrightarrow t \wedge f^{-1} \in t \nrightarrow s\}$	
Injection totale	$s \rightarrowtail t$	$s \rightarrowtail t \cap s \rightarrow t$	

Ensemble des séquences injectives sur T	$\mathbf{iseq}(T)$	$\bigcup(1..n \rightarrowtail T)$ avec $n \in \mathbb{N}$	
Insertion en queue	$s \leftarrow x$	$s \cup \{size(s)+1 \mapsto x\}$	$s \in seq(T)$ $x \in T$
Restriction en tête	$s \uparrow n$	$1..n \lhd s$	$s \in seq(T)$ $0 \leq n$ $n \leq size(s)$
Restriction en queue	$s \downarrow n$	$\lambda i.(i \in 1..size(s)-n \mid s(n+i))$	$s \in seq(T)$ $0 \leq n$ $n \leq size(s)$
Concaténation de séquences	$s_1 \frown s_2$	$s_1 \cup$ $\lambda i.(i \in size(s_1)+1..size(s_1)+size(s_2) \mid$ $s_2(i-size(s_1)))$	$s_1 \in seq(T)$ $s_2 \in seq(T)$

A2. Principales substitutions du langage B

Nom	Notation B	Signification	Condition de définition
Skip	*skip*	Ne rien faire	
Substitution élémentaire	$x := E$	Affecter E à la variable x	x : variable E : expression
Précondition	**PRE** P **THEN** S **END** (notée $(P\|S)$)	S'assurer de P et exécuter S	P : prédicat S : substitution
IF	**IF** P **THEN** S **ELSE** T **END**	Si P est vrai, exécuter S sinon exécuter T	P : prédicat S et T : substitutions
Garde	**SELECT** P **THEN** S **END**	Exécuter S si P est vrai	P : prédicat S : substitution
ANY	**ANY** X **WHERE** P **THEN** S **END**	Sélectionner une valeur de X qui vérifie P et exécuter S	X : liste de variables P : prédicat S : substitution
Simultanée	$S\|\|T$	Exécuter S et T en même temps	S et T : substitution
Séquence	$S\,;T$	Exécuter S puis T	S et T : substitution
Devient tel que	$X : (P)$	Sélectionner des valeurs des variables X telles que P est vrai	X : liste de variables P : prédicat
Appel d'opération	$[u \leftarrow]Op[(v)]$	appeler l'opération Op avec les paramètres v et affecter son résultat à u	

Annexe B. Preuve de l'algorithme de génération d'obligations de preuve pour la propriété d'atteignabilité

Dans cette annexe est donnée la preuve de correction de l'algorithme de génération d'obligations de preuve pour l'établissement de la propriété d'atteignabilité $\mathsf{AG}(\psi \Rightarrow \mathsf{EF}\phi)$. Cet algorithme est décrit au chapitre 3, à la page 57 de cette thèse. Prouver l'exactitude de cet algorithme revient à établir que les obligations de preuve qu'il génère sont équivalentes à la formule (3.1). Pour ce faire, considérons :

- $(cond \rightsquigarrow (a_1; \ldots; a_k))$ un chemin quelconque,
- $\{{x_1}^i, \ldots, {x_n}^i\}$ les valeurs des variables $\{x_1, \ldots, x_n\}$ après exécution de l'action a_i et avant exécution de l'action a_{i+1}.

Pour prouver la correction de l'algorithme, nous devons établir que :

(a) La formule $([a_1; \ldots; a_k]\phi)$ est équivalente à :

$$Pre(a_1) \wedge \forall \overrightarrow{x_i}^1 .(Post_{x_i, {x_i}^1}(S_{a_1}) \Rightarrow \gamma_2(\phi)) \qquad (\mathrm{H}_1)$$

si $k = 1$, et à :

$$
\begin{aligned}
& Pre(a_1) \wedge \\
& \forall \vec{x_i}^1 .(Post_{x_i,{x_i}^1}(a_1) \Rightarrow Pre(\gamma_2(a_2)) \\
& \wedge \ldots \wedge \\
& \forall (\vec{x_i}^1, \ldots, \vec{x_i}^{k-1}). \\
& (\bigwedge_{j=1}^{k-1} Post_{{x_i}^{j-1},{x_i}^j}(\gamma_j(S_{a_j})) \Rightarrow Pre(\gamma_k(a_k))) \wedge \qquad (\mathrm{H}_2) \\
\forall (\vec{x_i}^1, \ldots, \vec{x_i}^k). & (\bigwedge_{j=1}^{k} Post_{{x_i}^{j-1},{x_i}^j}(\gamma_j(S_{a_j})) \Rightarrow \gamma_{k+1}(\phi))
\end{aligned}
$$

si $k \geq 2$.

(b) Les obligations de preuve que l'algorithme produit sont équivalentes aux formules (H_1) et (H_2).

Par soucis de concision, nous détaillons cette preuve uniquement pour les actions de la forme $op(\ldots)$. Les autres cas se traitent de manière similaire. Afin de prouver (a) et (b), adoptons un raisonnement par induction sur la longueur du chemin :

– Cas de base ($longueur = 1$) : le chemin est composé d'une seule action a_1. Par définition, la formule $([a_1]\phi)$ est équivalente à (H_1). donc, $(cond \wedge \psi) \Rightarrow ([a_1]\phi)$ est équivalent à :

$$(cond \wedge \psi) \Rightarrow (Pre(a_1) \wedge \forall \vec{x_i}^1 .(Post_{x_i,{x_i}^1}(S_{a_1}) \Rightarrow \gamma_2(\phi)))$$

En appliquant l'algorithme, nous obtenons :

– les lignes 3–6 donnent : $For = \{Pre(a_1)\}$.

– la ligne 12 donne :

$$Hyp = \forall \vec{x_i}^1 .Post_{x_i,{x_i}^1}(S_{a_1}).$$

– la ligne 23 donne :

$$For = For \cup \{\forall \vec{x_i}^1 .(Post_{x_i,{x_i}^1}(S_{a_1}) \Rightarrow \gamma_2(\phi)).$$

– la ligne 26 donne :

$$(cond \wedge \psi) \Rightarrow (Pre(a_1) \wedge \forall \vec{x_i}^1 .(Post_{x_i,{x_i}^1}(S_{a_1}) \Rightarrow \gamma_2(\phi)))$$

L'algorithme est donc correct pour tout chemin de longueur 1.

– l'hypothèse d'induction ($longueur = k$) : supposons que la formule (3.1) est équivalente à la formule (H_2) et que l'algorithme est correct pour tout chemin de longueur inférieure ou égale à $k (k \geq 2)$.

– l'étape d'induction ($longueur = k + 1$) : prouvons que la formule (3.1) est équivalente à la formule (H_2) pour tout chemin de longueur $(k + 1)$. Par définition, nous avons :

$$([a_1; \ldots; a_k; a_{k+1}]\phi) \Leftrightarrow ([a_1; \ldots; a_k]([a_{k+1}]\phi)) \tag{H_3}$$

Aussi, le terme $([a_{k+1}]\phi)$ est équivalent à :

$$Pre(a_{k+1}) \wedge \forall \overrightarrow{x_i}^{k+1}.(Post_{{x_i}^k, {x_i}^{k+1}}(S_{a_{k+1}}) \Rightarrow \gamma_{k+2}(\phi)) \tag{H_4}$$

En substituons (H_4) dans (H_3), nous obtenons :

$$[a_1; \ldots; a_k](Pre(a_{k+1}) \wedge \forall \overrightarrow{x_i}^{k+1}.(Post_{{x_i}^k, {x_i}^{k+1}}(S_{a_{k+1}}) \Rightarrow \gamma_{k+2}(\phi)))$$

En utilisant l'hypothèse d'induction, nous obtenons :

$$\begin{array}{l} \quad Pre(a_1) \wedge \\ \forall \overrightarrow{x_i}^1.(Post_{x_i, {x_i}^1}(S_{a_1}) \Rightarrow Pre(\gamma_2(a_2))) \\ \qquad\qquad \wedge \ldots \wedge \\ \quad \forall(\overrightarrow{x_i}^1, \ldots, \overrightarrow{x_i}^{k-1}). \\ \quad (\bigwedge_{j=1}^{k-1} Post_{{x_i}^{j-1}, {x_i}^j}(\gamma_j(S_{a_j})) \Rightarrow Pre(\gamma_k(a_k))) \wedge \\ \quad \forall(\overrightarrow{x_i}^1, \ldots, \overrightarrow{x_i}^k). \\ (\bigwedge_{j=1}^{k} Post_{{x_i}^{j-1}, {x_i}^j}(\gamma_j(S_{a_j})) \Rightarrow \\ \qquad Pre(\gamma_{k+1}(a_{k+1}) \wedge \\ \qquad \forall \overrightarrow{x_i}^{k+1}.(Post_{{x_i}^k, {x_i}^{k+1}}(S_{a_{k+1}}) \Rightarrow \gamma_{k+2}(\phi)))) \end{array}$$

En utilisant l'équivalence suivante :

$$\forall X.(A \Rightarrow (B \wedge \forall Y.(C \Rightarrow D))) \Leftrightarrow \forall X.(A \Rightarrow B) \wedge \forall (X,Y).(A \wedge C \Rightarrow D)$$

nous obtenons :

$$\begin{aligned}
& Pre(a_1) \wedge \\
& \forall \vec{x_i}^1.(Post_{x_i,x_i{}^1}(a_1) \Rightarrow Pre(\gamma_2(a_2))) \\
& \wedge \ldots \wedge \\
& \forall (\vec{x_i}^1, \ldots, \vec{x_i}^{k-1}).(\bigwedge_{j=1}^{k-1} Post_{x_i{}^{j-1},x_i{}^j}(\gamma_j(S_{a_j})) \Rightarrow \\
& (Pre(\gamma_k(a_k))) \wedge \\
& \forall (\vec{x_i}^1, \ldots, \vec{x_i}^{k}).(\bigwedge_{j=1}^{k} Post_{x_i{}^{j-1},x_i{}^j}(\gamma_j(S_{a_j})) \\
& \Rightarrow Pre(\gamma_{k+1}(a_{k+1}))) \wedge \\
& \forall (\vec{x_i}^1, \ldots, \vec{x_i}^{k+1}). \\
& (\bigwedge_{j=1}^{k} Post_{x_i{}^{j-1},x_i{}^j}(\gamma_j(S_{a_j})) \wedge \\
& Post_{x_i{}^k,x_i{}^{k+1}}(\gamma_{k+1}(S_{a_{k+1}})) \Rightarrow \\
& \gamma_{k+2}(\phi))))
\end{aligned} \tag{H$_5$}$$

qui est égale à (H$_2$). Maintenant, nous devons prouver que l'algorithme produit les mêmes formules. À l'étape $(k+1)$ de la boucle (ligne 13 de l'algorithme), nous avons :

– $Hyp = \forall (\vec{x_i}^1, \ldots, \vec{x_i}^k).(\bigwedge_{j=1}^{k} Post_{x_i{}^{j-1},x_i{}^j}(\gamma_j(S_{a_j}))$

–

$$\begin{aligned}
& For = \{Pre(a_1), \\
& \forall \vec{x_i}^1.(Post_{x_i,x_i{}^1}(S_{a_1}) \Rightarrow Pre(\gamma_2(a_2))) \\
& \qquad\qquad ,\ldots, \\
& \forall (\vec{x_i}^1, \ldots, \vec{x_i}^{k-1}). \\
& (\bigwedge_{j=1}^{k-1} Post_{x_i{}^{j-1},x_i{}^j}(\gamma_j(S_{a_j})) \Rightarrow Pre(\gamma_k(a_k)))\}
\end{aligned}$$

L'application de l'algorithme donne :

– les lignes 14–15 donnent :

$$
\begin{array}{l}
For = \{Pre(a_1), \\
\forall \overrightarrow{x_i}^1.(Post_{x_i,{x_i}^1}(S_{a_1}) \Rightarrow Pre(\gamma_2(a_2))) \\
\qquad\qquad\qquad\qquad\qquad\qquad , \ldots , \\
\forall (\overrightarrow{x_i}^1, \ldots , \overrightarrow{x_i}^{k-1}). \\
(\bigwedge_{j=1}^{k-1} Post_{{x_i}^{j-1},{x_i}^j}(\gamma_j(S_{a_j})) \Rightarrow Pre(\gamma_k(a_k))), \\
\\
\forall (\overrightarrow{x_i}^1, \ldots , \overrightarrow{x_i}^{k}).(\\
\qquad \bigwedge_{j=1}^{k} Post_{{x_i}^{j-1},{x_i}^j}(\gamma_j(S_{a_j})) \Rightarrow Pre(\gamma_{k+1}(a_{k+1}))) \\
\}
\end{array}
$$

– la ligne 21 donne :
$Hyp = \forall (\overrightarrow{x_i}^1, \ldots , \overrightarrow{x_i}^{k+1}).(\bigwedge_{j=1}^{k} Post_{{x_i}^{j-1},{x_i}^j}(\gamma_j(S_{a_j})) \wedge$
$Post_{{x_i}^k,{x_i}^{k+1}}(\gamma_{k+1}(S_{a_{k+1}})))$

qui est équivalente à :

$Hyp = \forall (\overrightarrow{x_i}^1, \ldots , \overrightarrow{x_i}^{k+1}).(\bigwedge_{j=1}^{k+1} Post_{{x_i}^{j-1},{x_i}^j}(\gamma_j(S_{a_j})))$

– la ligne 23 donne :

$$
\begin{array}{l}
For = \{Pre(a_1), \\
\forall \overrightarrow{x_i}^1.(Post_{x_i,{x_i}^1}(S_{a_1}) \Rightarrow Pre(\gamma_2(a_2))) \\
\qquad\qquad\qquad\qquad\qquad\qquad , \ldots , \\
\forall (\overrightarrow{x_i}^1, \ldots , \overrightarrow{x_i}^{k-1}). \\
(\bigwedge_{j=1}^{k-1} Post_{{x_i}^{j-1},{x_i}^j}(\gamma_j(S_{a_j})) \Rightarrow Pre(\gamma_k(a_k))), \\
\forall (\overrightarrow{x_i}^1, \ldots , \overrightarrow{x_i}^{k}). \\
(\bigwedge_{j=1}^{k} Post_{{x_i}^{j-1},{x_i}^j}(\gamma_j S_{a_j}) \Rightarrow Pre(\gamma_{k+1}(a_{k+1}))) \\
\\
\forall (\overrightarrow{x_i}^1, \ldots , \overrightarrow{x_i}^{k+1}). \\
\qquad (\bigwedge_{j=1}^{k+1} Post_{{x_i}^{j-1},{x_i}^j}(\gamma_j(S_{a_j})) \Rightarrow (\gamma_{k+2}(\phi)))\}
\end{array}
$$

Finalement, les lignes 25–26 donnent une formule égale à la formule (3.1), ce qui prouve la correction de notre algorithme.

Bibliographie

[1] https ://github.com/bendisposto/probparsers/tree/develop/bparser.

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

[3] A.Mammar, M. Frappier, and R. Chane-Yack-Fa. Proving the Absence Property Pattern Using the B Method. In *$14^t h$ International IEEE Symposium on High-Assurance Systems Engineering, HASE 2012*, pages 167–170. IEEE Computer Society, 2012.

[4] H.-R. Barradas and D. Bert. Specification and Proof of Liveness Properties under Fairness Assumptions in B Event Systems. In M.-J. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15-18, 2002, Proceedings*, volume 2335 of *Lecture Notes in Computer Science*, pages 360–379. Springer, 2002.

[5] E.-M. Clarke and E-A Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 196–215. Springer, 2008.

[6] ClearSy. *Manuel de Reference du Langage B*. ClearSy, France, 2007.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[8] M.-B. Dwyer, G.-S. Avrunin, and J.-C. Corbett. Patterns in Property Specifications for Finite-State Verification. In B.-W. Boehm, D. Garlan, and J. Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*, pages 411–420. ACM, 1999.

[9] M. Frappier, F. Diagne, and A. Mammar. Proving Reachability in B using Substitution Refinement. *Electr. Notes Theor. Comput. Sci.*, 280 :47–56, 2011.

[10] M. Frappier and A. Mammar. An Assertions-Based Approach to Verifying the Absence Property Pattern. In *23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*, pages 361–370. IEEE, 2012.

[11] E.-M. Gagnon and L.-J. Hendren. SableCC, an Object-Oriented Compiler Framework. In *TOOLS 1998 : 26th International Conference on Technology of Object-Oriented Languages and Systems, 3-7 August 1998, Santa Barbara, CA, USA*, pages 140–154. IEEE Computer Society, 1998.

[12] P.-C. Héam. *Automates Finis pour la Fiabilité Logicielle et l'Analyse d'Accessibilité*. PhD thesis, Université de Franche-Comté, 2009.

[13] T-Son Hoang and J-R Abrial. Reasoning about Liveness Properties in Event-B. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods*, volume 6991 of *Lecture Notes in Computer Science*, pages 456–471. Springer, 2011.

[14] J.-P. Katoen. *Concepts, Algorithms, and Tools for Model Checking Joost-Pieter*. Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung. Inst. für Mathematische Maschinen und Datenverarbeitung, 1999.

[15] M. Leuschel and M.-J. Butler. ProB : A Model Checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003 : Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer, 2003.

[16] M. Mach-Król. Perspectives of Using Temporal Logics for Knowledge Management. In M. Ganzha, L.-A. Maciaszek, and M. Paprzycki, editors, *Federated Conference on Computer Science and Information Systems - FedCSIS 2012, Wroclaw, Poland, 9-12 September 2012, Proceedings*, pages 935–938, 2012.

[17] A. Mammar, M. Frappier, and F. Diagne. A Proof-Based Approach to Verifying Reachability Properties. In W.-C. Chu, W. Eric Wong, M.-J. Palakal, and C.-C. Hung, editors, *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, pages 1651–1657. ACM, 2011.

[18] C.-K. Mani. *Parallel Program Design : A Foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

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

[20] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems : Safety*. Springer-Verlag New York, Inc., 1995.

[21] C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, 1994.

[22] A. Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[23] A. Pnueli and Y. Kesten. A Deductive Proof System for CTL*. In L. Brim, P. Jancar, M. Kretínský, and A. Kucera, editors, *CONCUR 2002 - Concurrency Theory, 13th International Conference, Brno, Czech Republic, August 20-23, 2002, Proceedings*, volume 2421 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2002.

[24] A. Pnueli and Y. Sa'ar. All You Need is Compassion. In F. Logozzo, D.-Peled, and L.-D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings*, volume 4905 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2008.

[25] R. Reix. *Système d'Information et Management des Organisations*. Vuibert, Paris, 2002.

[26] M. Saad and L. Jemni Ben Ayed. Introducing Dynamic Properties with Past Temporal Operators in the B Refinement. In D. Peled and Y.-K. Tsay, editors, *Automated Technology for Verification and Analysis, Third International Symposium, ATVA 2005, Taipei, Taiwan, October 4-7, 2005, Proceedings*, volume 3707 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2005.

[27] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, and A. Petit. *Vérification de Logiciels : Techniques et Outils du Model-Checking*. Vuibert, April 1999.

[28] W. Su, J.-R. Abrial, R. Huang, and H. Zhu. From Requirements to Development : Methodology and Example. In *Formal Methods and Software Engineering - 13th*

International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings, volume 6991 of *Lecture Notes in Computer Science*, pages 437–455. Springer, 2011.