Université de Franche-Comté **U.F.R. des Sciences et Techniques**

École doctorale Sciences Pour l'Ingénieur et Microtechniques

Vérification d'invariants de systèmes paramétrés par superposition

THÈSE

présentée et soutenue publiquement le 06 avril 2006

pour l'obtention du

Doctorat de l'Université de Franche-Comté

(Spécialité Informatique)

par

Jean-François Couchot

Composition du jury

Directeurs :	Françoise Bellegarde, Professeur à l'Université de Franche-Comté, LIFC Alain Giorgetti, Maître de conférences à l'Université de Franche-Comté, LIFC
Président :	Jacques Julliand, Professeur à l'Université de Franche-Comté, LIFC
Rapporteurs :	Pascal Gribomont, Professeur à l'Université de Liège Joseph Sifakis, Directeur de Recherche au CNRS, VERIMAG
Examinateur :	Silvio Ranise, Chargé de recherche INRIA, LORIA
Invité :	Susanne Graf, Chargé de recherche CNRS, VERIMAG

Laboratoire d'Informatique de l'Université de Franche-Comté — FRE 2661

Mis en page avec la classe thloria.

À ma famille.

Table des matières

Table des figures

Chapitre 1

Introduction

Le 25 avril 2005, l'airbus A380 fait son premier vol d'essai sur la base aérienne de Toulouse Blagnac. On peut lire le jour même dans "Le Monde" :

> *L'effet A380 [...] touche l'ensemble de l'Europe et met à contribution des laboratoires qui n'ont pas forcément de relations avec l'aéronautique. C'est le cas de l'Ecole normale supérieure de Paris, où Patrick Cousot, professeur d'informatique, anime une équipe de sept personnes travaillant sur le projet Astrée [...]. "L'ordinateur possède des limitations, explique M. Cousot. Lorsqu'une valeur est codée sur 5 chiffres, la machine ne peut accepter des nombres de taille supérieure, et elle les tronque." D'où des erreurs fatales, comme celle qui a conduit à la perte du contrôle de la fusée Ariane 5 le 4 juin 1996.*
> *Face à l'inflation galopante de la taille des logiciels embarqués sur les avions de ligne (20 000 lignes de code pour l'A320, 120 000 pour l'A340, près de 500 000 pour l'A380), la recherche de telles erreurs devient critique. "Nous sommes capables d'établir la preuve mathématique qu'un programme comme celui de la commande de vol de l'A380 ne comprend pas de bogues liés à la limite des capacités de l'ordinateur", rassure M. Cousot. [...] L'équipe de M. Cousot a d'ores et déjà établi un record : "L'A380 dispose du plus gros programme jamais "prouvé" dans le monde."*

Comme on peut le constater, vérifier qu'un logiciel est correct est une tâche qui mobilise conjointement chercheurs et industriels pour lequels la demande est récente, mais en forte croissance. Au delà du domaine de l'avionique, cette demande s'étend à tous les logiciels critiques comme ceux développés pour le ferroviaire (cf. la ligne parisienne de métro sans conducteur nommée Météor [Beh96, BBFM99]), les échanges de données bancaires, les protocoles de communication [ABB+05, Rus03a], les algorithmes de gestion de l'accès aux unités critiques d'un ordinateur ou d'un réseau [ACM03, Rus03b]...

Pragmatiques, les méthodes de test permettent d'identifier une grande partie des erreurs mais ne peuvent pas, par nature, couvrir l'ensemble des cas possibles : elles apportent une conviction que le programme est correct sans en fournir la garantie. Néanmoins, cette conviction peut se révéler suffisante pour le partenaire industriel [BLLP04].

Plus difficiles à mettre en œuvre car plus rigides et plus abstraites, les méthodes formelles de conception et de vérification sont une solution pour développer des logiciels sûrs, fiables,

corrects. La partie suivante présente les ressorts de ces méthodes formelles de conception et de vérification.

1.1 Conception et vérification formelle

Vérifier qu'un logiciel est correct est une tâche dont on commence à lever l'ambiguïté : un programme respectant les conventions standards d'indentation est correct vis à vis de celles-ci ; un programme respectant la syntaxe du langage dans lequel il est écrit est syntaxiquement correct. Le cadre de cette étude concerne la vérification d'une autre sorte de correction : on a vérifié qu'un programme est correct si, à l'aide d'une méthode formelle, on a pu *démontrer* que le programme et ses propriétés sont cohérents. En nommant *spécification* l'union d'un programme et des propriétés attendues pour celui-ci, il reste alors à définir la notion de cohérence d'une spécification par méthode formelle .

Une méthode formelle fournit

- un langage permettant de spécifier le programme et ses propriétés ; ce programme consiste à définir un ensemble de variables nommées *variables d'état* et une partie dynamique qui modifie ces variables d'état ; les propriétés, quant à elles, sont exprimées dans une logique particulière où l'ensemble des variables est celui des variables d'état ;
- une démarche qui prend en entrée cette spécification et dont le verdict est une certification du programme.

Pour chaque propriété, il y a trois verdicts possibles : (a) le programme satisfait la propriété ; la spécification est alors qualifiée de *modèle* ou de *cohérente* ; (b) le système ne satisfait pas la propriété ; il est alors incohérent ; (c) l'absence de conclusion. Dans l'absolu, la méthode n'acquière le statut de méthode formelle que lorsqu'elle a elle même été vérifiée comme correcte !

Néanmoins, vérifier formellement le programme ne garantit pas qu'il réponde aux besoins des utilisateurs décrits dans le cahier des charges : la méthode formelle qui se contente de vérifier si la spécification est un modèle n'assure pas que celui-ci est une formalisation fidèle à l'expresion des besoins : les modèles ne sont généralement que des vues partielles prises depuis différentes perspectives selon les propriétés que l'on souhaite vérifier. Répondre à la question "Construisons-nous une *bonne* spécification ?" relève du domaine de la *validation* et sort du cadre de cette étude.

Il existe deux principales approches de vérification formelle : le *model-checking* et la *preuve*, présentés dans les deux parties suivantes.

1.1.1 Vérification par model-checking

L'approche algorithmique par model-checking [CES86, CGP99] est basée sur l'idée simple suivante : une énumération exhaustive de toutes les configurations dans lesquelles le programme peut se trouver permet de savoir si les propriétés attendues pour celui-ci sont réellement assurées. Les propriétés sont vérifiées ainsi directement par l'outil, ceci sans qu'aucune interaction avec l'utilisateur ne soit nécessaire.

Bien qu'élégante, l'approche souffre d'un handicap de taille qui est la construction exhaustive de tous les états du programme, construction nommée *graphe d'atteignabilité*. En effet, la construction complète de ce graphe nécessite que celui-ci soit (i) fini car il doit être stocké dans une mémoire finie ; (ii) de taille raisonnable car la représentation de toutes les configurations

possibles conduit rapidement à une explosion dite *combinatoire* de la taille de ce graphe qui devient difficile, voire impossible, à stocker en mémoire.

Différentes stratégies permettent de traiter des systèmes de tailles croissantes. Parmi celles-ci, on trouve celles qui

- optimisent la représentation en mémoire en utilisant des structures de données compactes comme les arbres de décision binaires ordonnés (BDD), les tables de hachage...,
- réduisent directement la taille du graphe en abstrayant certains états ou certaines transitions,
- se contentent de vérifier les propriétés sur des composants [KL04] du système pour les étendre ensuite sur le système complet sans jamais construire tous ses états.

Néanmoins, aussi performantes que soient ces stratégies, aucune ne permet, ni ne permettra de traiter des systèmes infinis pourtant courants en informatique. En effet le développement d'un programme ne s'effectuant jamais en une seule étape, il est souvent pratique d'introduire des paramètres dans les premières phases de spécification, quitte à les préciser dans des étapes ultérieures. La correction du *système paramétré* doit être assurée pour toutes les valeurs potentielles de ces paramètres, ce qui engendre des graphes d'atteignabilité infinis. La vérification par preuve apporte une solution à ce problème.

1.1.2 Vérification par preuve

Comme son nom l'indique, une démarche de vérification par preuve consiste à prouver mathématiquement la validité d'un ensemble de formules –nommées obligations de preuve– engendrées à partir du programme et des propriétés d'une spécification formelle. Tout l'enjeu de cette démarche réside dans la capacité à faire réaliser – on emploie alors le terme *décharger*– ces preuves par des outils nommés *prouveurs*.

La première caractéristique d'un prouveur est la logique qu'il traite, qu'elle soit du premier ordre, du second ordre ou d'ordre supérieur. Pour simplifier, une formule d'une logique du premier ordre n'autorise pas les quantifications sur les variables de sorte fonction ou prédicat tandis qu'une logique du second ordre le permet. On trouve alors :

- Simplify [DNS05] et haRVey [DR02] qui traitent de la logique équationnelle (les formules élémentaires ne sont que des égalités) du premier ordre,
- MONA [HJJ+96] qui traite de la logique faible du second ordre avec un successeur (les variables du premier ordre sont interprétées par des entiers naturels tandis que les variables du second ordre sont interprétées comme des ensembles finis de naturels),
- l'Atelier B [Abr96], PVS [ORSSC98] et HOL [GM93] qui traitent de la logique du second ordre.

Une seconde caractéristique d'un prouveur est l'ensemble des *procédures de décision* qu'il implante, et qui lui permettent de décharger les obligations de preuve. On distingue, par exemple parmi ces procédures, celles qui s'appuient sur des techniques de réécriture [DJ90], celles qui s'appuient sur la construction d'un automate représentant la formule [Bö62], celles qui appliquent des techniques de tableaux [OS88]...

Une troisième caractéristique est l'interactivité (ou au contraire l'automaticité) de ces procédures : du plus interactif au plus automatique on trouve Coq [BP04], HOL, PVS, l'Atelier-B, Mona, haRVey, Simplify...

Si la logique retenue traite de structures de taille infinie (comme un ensemble contenant une infinité d'éléments par exemple) et si le prouveur permet de prouver les formules de cette

logique, l'approche par preuve se présente comme une réponse aux problèmes de taille infinie non traités par model-checking.

D'un point de vue application, même si la génération d'obligations de preuve nécessite que les spécifications soient exprimées dans un langage formel proche de celui des mathématiques et même si les prouveurs nécessitent souvent le guidage d'un expert pour aboutir à la preuve de la formule, cette démarche commence à séduire les industriels tels la RATP [Beh96, BBFM99] ou Airbus industrie [CCF+05, RSB+99] comme le montre l'exemple énoncé en introduction.

Le contexte et la problématique de ce travail, qui se focalise sur la vérification par preuve de systèmes paramétrés, sont présentés dans la partie suivante.

1.2 Contexte et problématique du travail

Cette partie indique dans quel contexte scientifique se situe ce travail et à quels problèmes il apporte une solution. Succinctement, ce travail a pour objectif d'appliquer une méthode de vérification de cohérence de spécification par construction d'invariants (partie 1.2.1) sur des spécifications exprimées en B (partie 1.2.2), par preuve automatique basée sur le raisonnement équationnel (partie 1.2.3) et montre son applicabilité sur une classe de systèmes paramétrés (partie 1.2.4).

1.2.1 Vérification symbolique d'invariant

Pour prouver qu'un programme ne mène pas dans un ensemble d'états d'erreur, il est nécessaire et suffisant de garantir que cet ensemble est disjoint de celui des états que peut prendre le programme depuis son initialisation. Cependant, sous une facilité trompeuse, la phrase précédente a occulté une réelle difficulté qu'est le calcul des états accessibles depuis l'initialisation, notamment lorsque ceux-ci sont en nombre infini.

Pour capturer l'infini, on peut exprimer les ensembles d'états à l'aide de formules, nommées *assertions*, puis itérer le calcul des successeurs de l'assertion initiale. Les règles de la logique sont ainsi utilisées comme des outils permettant de raisonner sur ces formules même si elles représentent des ensembles infinis.

On peut aussi déplacer le problème d'exprimer l'ensemble des états atteignables vers un calcul de sur-approximation de celui-ci. Si celle-ci est stable par toute opération du programme, on la qualifie dans ce cas d'*invariant*, et si elle est disjointe de l'ensemble d'états d'erreur, la propriété à vérifier est garantie.

Néanmoins, dans ces deux approches, on ne peut générer de telles assertions que si l'on dispose d'une spécification formelle du programme. La partie suivante propose d'utiliser le formalisme B comme langage de spécification.

1.2.2 Le formalisme B

Le Laboratoire d'Informatique de l'Université de Franche-Comté a une tradition forte de recherche autour de la méthode B : plusieurs directions ont ainsi été étudiées, dont, les systèmes d'événements [BJK02], leur raffinement [JB98, BJM99, BCJK01], la génération de tests [LP02, PLT00]... Deux outils issus de ces travaux sont utilisés dans la communauté B : les JBtools [TV01] qui sont une collection d'outils contenant en particulier un parseur B et un générateur de fichier XML correspondant à la spécification et l'environnement BZTT [BLP02]

qui consiste à produire des séquences de tests à partir d'analyses statiques et d'animation symbolique de la spécification B.

En exprimant des systèmes paramétrés dans le formalisme des machines B, en construisant et en vérifiant des propriétés d'invariance par preuve automatique, ce travail se présente comme une nouvelle direction de recherche tout en demeurant dans l'axe de recherche autour de la méthode B. Néanmoins l'environnement B classique fourni par l'Atelier-B possède de fortes contraintes qu'il est nécessaire de lever lorsqu'on s'attaque à une construction d'invariant : il est en effet judicieux (*i*) de posséder une démarche itérant un calcul symbolique de prédécesseurs ou de successeurs ; or cette itération, nommée calcul du *point fixe*, est absente dans la méthode B ; (*ii*) de disposer d'un prouveur automatique, pour décider de la poursuite ou non du calcul du point fixe ; le prouveur de l'Atelier-B est interactif par nature et les recherches précédentes de traduction d'obligations de preuve ont toujours été menées vers des prouveur interactifs [BF02]. La partie suivante présente une réponse possible au point (*ii*).

1.2.3 La logique équationnelle

Parmi les prouveurs employés dans des démarches de vérification de programmes, on trouve, sans prétendre être exhaustif, CAVEAT [GR95], basé sur la logique propositionnelle, MONA [HJJ+96] qui décide la logique monadique du second ordre faible, PVS [OR00], modulaire (il implante 150 théories) mais interactif pour certaines procédures, Uclid [LS04] qui décide les combinaisons des logiques avec compteurs arithmétiques, lambda expressions et fonctions non interprétées [Lah04] et CVCLite [BB04] qui décide les combinaisons des théories des fonctions non interprétées, des tableaux, de l'arithmétique linaire sur les réels. Néanmoins, hormis PVS, chacun de ces outils, conçu initialement pour une théorie, se révèle peu modulaire dans le sens où la prise en compte d'une autre théorie nécessite un nouveau développement complet de la procédure.

Sur le terrain des prouveurs automatiques et modulaires on trouve Otter [Kal01], Simplify [DNS05], Spass [Wei99], ... tous basés sur des techniques de réécriture et inférant un schéma de règles comme la paramodulation [NW01]. La démarche employée leur permet de traiter automatiquement les logiques équationnelles pour lesquelles il existe des théories finiment axiomatisables, la convergence de la procédure n'étant pas garantie dans tous les cas.

De nouveaux résultats de décidabilité par paramodulation concernant les listes, les tableaux et leur combinaison ont été établis par des membres du projet INRIA/LIFC CASSIS [ARR03]. Le prouveur haRVey [DR02], développé au sein du projet CASSIS, a validé ces résultats et se présente expérimentalement comme le plus rapide [DR03] grâce à une heuristique de découpage de la structure propositionnelle de la formule.

Néanmoins si les programmes que l'on souhaite vérifier par une démarche de construction d'invariants ne s'expriment pas dans ces théories décidables, ils sont cependant suffisamment proches pour que la question de l'adéquation entre la paramodulation telle qu'elle est implantée dans haRVey et la vérification de systèmes paramétrés exprimés en B soit étudiée. Cette adéquation se mesure expérimentalement, en remarquant tout d'abord la terminaison d'haRVey, puis en effectuant des comparaisons avec d'autres prouveurs, et théoriquement, en apportant une preuve que les théories décidables supportent des extensions.

1.2.4 Les systèmes uniformes distribués

La classe d'application du travail doit être la plus large possible, en ne privilégiant pas exclusivement une famille d'exemples. Telle est la consigne que tout chercheur tente en vain d'appliquer puisque toute considération simplificatrice le pousse vers une famille particulière.

Néanmoins, ne pas présenter une classe caractéristique d'application du travail est souvent néfaste en terme de communication scientifique : tout travail doit en effet montrer son applicabilité à une classe jugée comme importante en informatique pour permettre, entre autre, des comparaisons, le cas échéant.

Les systèmes uniformes distribués, n programmes identiques s'exécutant en parallèle, apparaissent à double titre comme une classe d'application intéressante : (*i*) ils servent régulièrement [GZ98, DP99, Del00, Mai01, PRZ01] d'exemples d'application des méthodes formelles de vérification de programmes ; (*ii*) ils font partie d'une thématique de recherche de l'équipe Systèmes Distribués du LIFC [TH01, HT01, GLT97].

Ce travail présente alors une démarche de vérification d'invariant sur une classe de spécifications logico-ensemblistes paramétrées exprimées en B, utilisant comme principe déductif la paramodulation outillée par haRVey. L'application est réalisée sur une famille de systèmes uniformes distribués.

1.3 Contributions

De plusieurs natures, les contributions de ce travail sont présentées dans l'ordre où elles apparaissent dans une chaîne de développement de méthode formelle : elles concernent l'impact du choix des structures de données sur la spécification (partie 1.3.1), la génération d'obligations de preuve (partie 1.3.2) ; de nouveaux résultats (partie 1.3.3) montrent que ces obligations de preuve sont décidables. L'implantation de la démarche est effectuée au travers de nouveaux outils (partie 1.3.4). La dernière partie (partie 1.3.5) présente les publications issues de ce travail.

1.3.1 Spécifications

C'est dans le cadre général des spécifications logico-ensemblistes B que ce situe ce travail. Néanmoins l'application aux systèmes uniformes distribués a suscité l'intérêt d'une extension dans la direction des spécifications avec fonctions totales.

En effet, pour un système uniforme distribué où chaque composant peut être dans un des n états $e_1, \ldots, e_n$, une configuration de l'état général du système consiste à décrire dans quel état e_i est chaque composant. Ceci s'entreprend soit en exhibant l'ensemble des composants dans chacun des e_i, soit en utilisant directement une fonction totale qui associe à chaque composant l'état dans lequel il est.

Le choix d'une des deux structures pour représenter l'état général du système a un impact direct sur la spécification et sur la logique considérée : on montre que si une spécification exprimée à l'aide de fonctions totales est plus concise que son homologue ensembliste, les obligations de preuve qui en découlent sont en outre plus directement traduisibles dans une logique pour laquelle nous avons réussi a établir des résultats de décidabilité.

1.3.2 Génération d'obligations de preuve

Indépendamment des structures de données, ce travail justifie l'emploi du calcul de plus faible précondition [Dij75] comme l'outil le plus adapté à la vérification symbolique de propriétés de sûreté vis à vis des principes de fonctionnement du prouveur cible haRVey. Sont prises en compte dans cette étude la taille des formules engendrées et la nature des quantificateurs introduits. Dans un souci d'efficacité, ce travail propose d'étendre la grammaire des prédicats B avec la structure propositionnelle *if then else*, présente nativement dans haRVey, en montrant théoriquement et pratiquement l'intérêt d'une telle extension sur la taille des obligations de preuve introduites et sur les temps de calcul résultant.

Ensuite, que ce soit pour des spécifications ensemblistes ou des spécifications construites à partir de structures de fonctions totales, cette thèse présente une traduction originale en logique équationnelle des obligations de preuve qui en résultent. Un premier pas en direction de l'abstraction est fait à ce niveau puisqu'une démarche de traduction abstrayant certains prédicats est présentée. Au niveau de la traduction, il en résulte des formules de taille plus petite dans une théorie plus simple.

En terme de complexité, la traduction est toujours linéaire en temps sur la taille de la formule. En terme de correction, et c'est une autre contribution, la traduction préserve la satisfaisabilité : la formule initiale est satisfaisable modulo la théorie dans laquelle elle est écrite si et seulement si la formule traduite est satisfaisable modulo la théorie équationnelle dans laquelle elle est traduite.

1.3.3 Procédures de décision

Présenter des démarches correctes de traduction de formules n'a d'intérêt que si l'on dispose également d'une démarche permettant de décider la satisfaisabilité des formules ainsi traduites. A ce niveau, nous présentons des classes de problèmes appartenant aux classes de systèmes uniformes distribués pour lesquels chaque obligation de preuve est décidable par une nouvelle procédure de décision. Celle-ci exploite un résultat récent de décidabilité par paramodulation [ARR03] et une instanciation originale des variables quantifiées qui tient compte du type des variables [FG03].

1.3.4 Outils

Les différentes démarches de génération d'obligations de preuve et de traduction ont été intégrées dans le prototype bam2rv[1]. Lorsque les obligations de preuve engendrées par cet outil contennaient des quantificateurs, elles ont été traitées par un second prototype rvqe[2] qui gérait ces quantificateurs pour les rendre traitables par le prouveur haRVey. Puis cette gestion a été intégrée dans la version suivante d'haRVey.

Différentes versions d'algorithmes de calcul de point fixe ont été introduits dans un prototype pour décider de l'atteignabilité d'une configuration et fournir, le cas échéant, un invariant inductif du système. De nombreux exemples[3] ont pu ainsi être traités automatiquement par la démarche.

[1] `http://lifc.univ-fcomte.fr/~giorgett/Rech/Software/bam2rv/index.html`
[2] `http://lifc.univ-fcomte.fr/~couchot/rvqe/`
[3] `http://lifc.univ-fcomte.fr/~couchot/specs/`

1.3.5 Publications

Ce travail a donné lieu à cinq publications en conférence et un article de journal, tous avec comité de lecture.

- Dans [CDD+03] nous avons présenté la démarche de génération d'obligations de preuve ensemblistes et leur traduction en logique équationnelle. Ce travail a été honoré du titre du meilleur article de la conférence. Le chapitre 8 reprend ce travail.
- Dans [CDGR03], version étendue de [CDD+03], nous avons prouvé la correction de la démarche de traduction en logique équationnelle et nous avons montré son efficacité sur quelques exemples, dont l'exemple industriel présenté dans ce mémoire. Les chapitres 7 et 8 reprennent ce travail.
- Dans [CDGR04] nous avons présenté l'interface barvey de gestion de preuve d'invariant B intégrant les outils bam2rv, rvqe et haRVey. La partie 11.1 reprend ce travail.
- Dans [CG04] nous avons présenté le langage des substitutions généralisées de tableaux en montrant qu'il s'applique à la classe des systèmes uniformes distribués se synchronisant par rendez-vous. La décidabilité des obligations de preuve y est posée en conjecture. La partie 10.2 reprend ce travail.
- Dans [Cou04], j'ai présenté une synthèse de la problématique de ce mémoire centré sur les spécifications de tableaux.
- Dans [CGK05], nous avons étendu la classe d'application de la méthode aux systèmes uniformes distribués se synchronisant par envoi multiple (broadcast). La décidabilité des obligations de preuve est établie pour une sous-classe de ces systèmes. Les chapitres 10 et 11 reprennent ce travail.

1.4 Plan du document

Le chapitre 2 de préliminaires introduit un cadre de logique multi-sortes : les langages, formules et quelques théorèmes principaux y sont définis comme autant d'outils permettant de raisonner simplement et efficacement dans les parties suivantes du mémoire. Il présente aussi la superposition, comme fondement théorique de l'outil haRVey. Le lecteur qui est familier avec les résultats classiques de logique peut se dispenser de sa lecture.

Le chapitre 3 présente les classes d'application de ce travail que sont les systèmes uniformes distribués et les spécifications industrielles logico-ensemblistes.

Le chapitre 4 replace la méthode de vérification d'invariant dans le contexte plus général de calculs de plus petit ou plus grand point fixe. Les semi-algorithmes déductifs qui implantent ces calculs y sont formalisés.

Le chapitre 5 est un rappel sur la méthode B, où l'on montre comment traduire les systèmes uniformes distribués dans un fragment de son langage de machine abstraite.

Le chapitre 6 est un état de l'art de la vérification de systèmes paramétrés par une méthode déductive.

Le chapitre 7 présente le prouveur de théorèmes haRVey, qui répond à la question de la validité en logique équationnelle d'une formule ϕ sans quantificateurs, modulo une théorie finiement axiomatisée. Il montre comment nous avons étendu sa classe d'application aux formules quantifiées.

Le chapitre 8 montre comment traduire une formule ensembliste en une formule équisatisfaisable exprimée en logique équationnelle. La correction de la démarche de traduction est établie.

Le chapitre 9 justifie, dans le cadre de la vérification de systèmes paramétrés, l'utilisation d'une classe de formules sur les structures de fonctions totales. La traduction présentée dans ce chapitre étend la traduction présentée au chapitre 8 et montre comment exprimer dans cette classe les systèmes uniformes distribués.

Le chapitre 10 introduit le langage des substitutions de tableaux, comme une abstraction fine du langage des substitutions généralisées sur les fonctions totales, permettant de décrire les systèmes uniformes distribués de manière concise. Les obligations de preuve engendrées sont traitables par raisonnement équationnel. De plus, ce chapitre montre quels résultats de décidabilité sont garantis selon la nature de la spécification.

En plus de présenter des considérations techniques déployées lors de l'implantation des différents prototypes, le chapitre 11 livre une collection de protocoles auxquels la démarche est appliquée.

Dans la conclusion, nous montrons les perspectives possibles de ce travail.

Chapitre 2

Préliminaire(s) logique(s)

Les liens étroits entre logique et informatique ne sont pas récents, avec pour exemple la citation suivante de plus de 40 ans : *"It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and mathematical elegance"* [McC64].

Ainsi, motivées ou non par des besoins informatiques, se sont développées les logiques temporelles [Pnu77, MP92] dont la clef est le temps, les logiques modales [Sti92] qui ont été exploitées pour la vérification de systèmes concurrents et non déterministes, la logique propositionnelle, la logique du premier ordre, la logique intuitionniste, ...

Dans un contexte de vérification de programme, lorsqu'il se pose, le choix d'une logique s'effectue selon deux critères, le premier étant la vue que l'on a du système à étudier et le second étant la disponibilité de méthodes (outillées) traitant cette logique : *"There is a tradition in logic, carried over into computer science, to think of pure first order logic as a universal language. In fact first order language is about as useful in verification as a Turing machine is in software engineering : cute to watch but not very useful. The sense in which first order logic is universal is that knowledge about domains can be embedded in it in the form of axioms, from which arguments may proceed. Unfortunately, axioms without methods tend not to be very useful in practice."* [Pra79]

Indécidable dans sa généralité, la logique du premier ordre contient néanmoins des fragments qui représentent de "bons compromis" entre expressivité et efficacité des traitements : ensembles et quantificateurs permettent d'exprimer simplement et de vérifier automatiquement des propriétés intéressantes de sûreté sur des systèmes simulant un nombre quelconque de processus au comportement uniforme et s'exécutant en parallèle [GS97, FG03, CGK05].

S'il est habituel de déclarer un type pour chaque variable utilisée dans un langage de programmation, il est également naturel d'utiliser les sortes en logique du premier ordre : quel intérêt aurait-on, par exemple, à comparer deux constantes de sortes différentes ? à utiliser un connecteur logique avec des opérandes mal sortés ? Supposons qu'un programme soit déclaré comme sûr si et seulement si une certaine formule mathématique est prouvée comme vraie. Que dire du programme si la formule n'est fausse que dans le cas où une variable prend une valeur en dehors de sa sorte ? Intégrer les sortes au niveau de la logique permet alors de raisonner à un niveau moins général que sans les sortes mais plus proche du problème de vérification de programmes.

Passer d'une logique multi-sortes à une logique mono-sorte s'effectue en introduisant un

prédicat de typage par sorte [End72, p. 279]. Tout résultat exprimé en logique du premier ordre mono-sorte qui supporte l'ajout d'un prédicat de typage engendre un résultat similaire en logique multi-sortes. Néanmoins la prise en compte des sortes directement dans la logique permet d'obtenir des résultats plus aisément que dans le cas mono-sorte (le chapitre 10 s'attache à justifier cette affirmation).

Pour permettre de raisonner simplement et efficacement dans les parties suivantes du mémoire, ce chapitre introduit un cadre de logique multi-sortes : les langages, formules et (quelques principaux) théorèmes sont définis comme autant d'outils rendant les preuves plus claires et plus concises. La première partie définit les langages du premier ordre multi-sortes. La seconde énonce comment est interprétée une formule de ce langage et quand elle est qualifiée de satisfaisable. Des formes particulières de formules sont données en troisième partie. La quatrième partie présente la méthode de Herbrand de vérification de satisfaisabilité de formule. La dernière partie présente une mise en œuvre pratique de cette méthode.

2.1 Langages du premier ordre multi-sortes égalitaires

<Cette partie, qui définit les notions classiques de langage du premier ordre, termes, atomes et formules, est inspirée par [DNR04, CL93, Mar03, Fon04].

Définition 1 (Langage du premier ordre multi-sortes) Un langage du premier ordre multi-sortes est un tuple $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, \alpha, \sigma \rangle$ tel que

- $\mathcal{S}$ est l'ensemble non vide et dénombrable des sortes (types) ;
- $\mathcal{V} = \bigcup_{\tau \in \mathcal{S}} \mathcal{V}_\tau$ est l'union des ensembles $\mathcal{V}_\tau$ de variables de sorte τ où les $\mathcal{V}_\tau$ sont disjoints et dénombrables ;
- $\mathcal{F}$ est l'ensemble fini ou dénombrable des symboles fonctionnels ;
- $\mathcal{P}$ est l'ensemble fini ou dénombrable des symboles prédicatifs ;
- $\alpha : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}$ retourne l'arité (i.e. le nombre d'arguments) de chaque symbole fonctionnel ou prédicatif ;
- $\sigma : \mathcal{F} \cup \mathcal{P} \to \mathcal{S}^*$ retourne pour un symbole fonctionnel $f \in \mathcal{F}$ le tuple de sortes dans $\mathcal{S}^{\alpha(f)+1}$ appelé signature de f où les $\alpha(f)$ premières sortes sont celles des arguments de f, la dernière étant celle de l'image de f. Pour un symbole prédicatif $p \in P$, σ retourne le tuple de $S^{\alpha(p)}$ des sortes des arguments de p.

Un langage du premier ordre multi-sortes est dit *égalitaire* lorsqu'il contient le prédicat particulier d'égalité $=_\tau$ pour chaque sorte τ.

Les règles de construction des formules du premier ordre multi-sortes pour un langage égalitaire $\langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, \alpha, \sigma \rangle$ sont données à la figure 2.1 où τ est une sorte dans $\mathcal{S}$. La première règle de cette grammaire, qui est en fait un schéma de règles, doit être instanciée pour chaque sorte $\tau \in \mathcal{S}$. Elle définit l'élément syntaxique τ-*terme* qui est un terme de sorte τ. Par la suite, un terme de sorte τ est appelé τ-*terme*. La seconde règle définit les formules atomiques (*atome*) tandis que la dernière définit la construction d'une formule du premier ordre multi-sortes (*fpoms*). On définit à présent la notion de littéral à la base du raisonnement équationnel.

Définition 2 (Littéral) Un littéral est une formule atomique ou la négation d'une formule atomique.

Certaines conventions sont posées pour améliorer la lisibilité ou par souci de concision :

$$
\begin{array}{rcl}
\tau\text{-}terme & ::= & x \mid f(\tau_1\text{-}term, \ldots, \tau_n\text{-}term) \\
 & & \quad \text{avec } x \in \mathcal{V}_\tau,\ f \in \mathcal{F} \text{ et } \sigma(f) = (\tau_1, \ldots, \tau_n, \tau) \\
atome & ::= & \tau\text{-}terme =_\tau \tau\text{-}terme \mid p(\tau_1\text{-}term, \ldots, \tau_n\text{-}term) \\
 & & \quad \text{avec } p \in \mathcal{P} \text{ et } \sigma(p) = (\tau_1, \ldots, \tau_n) \\
fpoms & ::= & atome \mid \neg fpoms \mid (Q_\tau x\,.\,fpoms) \mid (fpoms \diamond fpoms) \\
 & & \quad \text{avec } Q_\tau \in \{\forall_\tau, \exists_\tau\},\ x \in \mathcal{V}_\tau \text{ et } \diamond \in \{\wedge, \vee\}
\end{array}
$$

FIG. 2.1 – Grammaire des formules du premier ordre multi-sortes égalitaire

- On utilise les symboles dérivés $\Rightarrow$ et $\Leftrightarrow$; $\neg a \vee b$ s'écrit $a \Rightarrow b$ et $(\neg a \vee b) \wedge (\neg b \vee a)$ s'écrit $a \Leftrightarrow b$, où a et b sont deux formules du premier ordre multi-sortes.
- On utilise le prédicat ternaire $ite(\varphi_1, \varphi_2, \varphi_3)$ comme un raccourci pour $(\varphi_1 \Rightarrow \varphi_2) \wedge (\neg\varphi_1 \Rightarrow \varphi_3)$.
- La fonction σ est étendue aux termes en posant $\sigma(t) = \tau$ si t est un τ-*terme*.
- Un symbole fonctionnel d'arité nulle est appelé *constante*.
- Un atome toujours vrai (respectivement toujours faux) est noté $\top$ (resp. $\bot$) qui est une constante propositionnelle.
- On écrit $t \neq_\tau t'$ pour $\neg(t =_\tau t')$.
- On écrit $(Q_\tau x_1, \ldots, x_n\,.\,\varphi)$ pour $(Q_\tau x_1, (\ldots, (Q_\tau x_n\,.\,\varphi)))$.
- Dans une formule φ, on qualifie de *liée* une variable qui est sous la portée d'un quantificateur et de *libre* une variable qui ne l'est pas ; on note $\mathcal{V}_{lib}(\varphi)$ l'ensemble des variables libres de φ.
- Une formule est dite *close* si elle ne contient pas de variables libres et *ouverte* dans le cas contraire.

On définit alors formellement la démarche qui consiste à remplacer une variable par un terme dans une formule :

Définition 3 (Substitution) Une substitution θ est une application de $\mathcal{V}$ dans l'ensemble des termes de $\mathcal{L}$ telle que l'image de x par θ, notée $x\theta$, vaut x pour tout $x \in \mathcal{V}$ sauf pour un ensemble fini de variables noté $Dom(\theta)$. Chaque substitution est supposée correctement sortée : pour tout $x \in \mathcal{V}_\tau$, $x\theta$ est un terme de sorte τ.

Cette définition s'étend aux τ-termes construits à partir d'un symbole fonctionnel $f \in \mathcal{F}$ (et de manière similaire aux prédicats puis aux formules) en posant $f(t_1, \ldots t_n)\theta =_{def} f(t_1\theta, \ldots t_n\theta)$, puis s'étend naturellement aux formules. Par la suite, plutôt que de définir θ comme la substitution qui remplace toutes les occurrences de x par r et d'écrire $\varphi\theta$, on écrit directement $\varphi(r/x)$.

Lorsqu'on cherche à remplacer une partie d'une formule par une autre formule, on utilise la notion de "position" dans une formule, introduite par la définition suivante.

Définition 4 (Position) La position de ψ, sous-formule de φ, est un mot sur les entiers naturels non nuls qui est définie inductivement comme suit :

- si ψ est φ, la position est le mot vide ϵ ;
- si $\psi = \psi_1 \circ \psi_2$ a pour position π où $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$ alors la position de ψ_1 est $\pi.1$ et celle de ψ_2 est $\pi.2$;

- si $\psi = \neg\psi_1$, $\psi = (\forall\, x\,.\,\psi_1)$ ou $\psi = (\exists\, x\,.\,\psi_1)$ ont pour position π, alors la position de ψ_1 est $\pi.1$;
- si $\psi = \mathit{ite}(\psi_1, \psi_2, \psi_3)$ a pour position π, alors ψ_1 (resp. ψ_2, ψ_3) a pour position $\pi.1$ (resp $\pi.2$, $\pi.3$).

Pour une formule ϕ et une position π de l'ensemble des positions dans ϕ, on note

1. $\phi\,|_\pi$ la sous formule de ϕ à la position π. Si $\pi = \pi'.i$, alors $\phi\,|_{\pi'.i} = (\phi\,|_{\pi'})\,|_i$.
2. $\phi[\psi]_\pi$ la formule obtenue en remplaçant $\phi\,|_\pi$ dans ϕ par ψ.

De nombreux algorithmes de transformation de formules (cf chapitres 7 et 8) consistent à remplacer des sous-formules par des sous-formules plus simples. Une telle transformation est intéressante dès qu'elle garantit que la formule transformée se déduit de, implique, ou est équivalente à la formule initiale. Par exemple, remplacer dans une implication le membre droit par $\top$ (resp. par $\bot$) engendre une formule qui est une conséquence de la précédente (resp. qui implique la précédente). Ce qui importe, en fait, c'est la parité du nombre de négations qui existeraient entre la sous-formule et la racine de la formule si celle-ci était écrite sans implications ($\Rightarrow$) ni équivalence. Cette parité est dénommée *polarité* ([GGT00, lemme p. 368]). Lorsque la sous-formule n'est sous la portée ni d'une équivalence ni du premier membre d'un *ite*, si ce nombre est pair, on dit que la polarité de la sous-formule est $+1$, et s'il est impair, cette polarité est -1. Ceci est étendu aux formules avec $\Leftrightarrow$ et *ite(„)* en disant que la polarité est nulle sinon. La définition de la polarité est formalisée ci-dessous.

Définition 5 (Polarité) La *polarité* $\mathit{pol}(\phi\,|_\pi)$ d'une sous-formule $\phi\,|_\pi$ de ϕ est définie inductivement selon la structure de ϕ comme suit :

- $\mathit{pol}(\phi\,|_\epsilon) := +1$;
- $\mathit{pol}(\phi\,|_{\pi.i}) := \mathit{pol}(\phi\,|_\pi)$ si $\phi\,|_\pi$ est une conjonction, disjonction, formule dont le symbole de tête est un quantificateur, une implication avec $i = 2$, un *ite* avec $i = 2$ ou $i = 3$;
- $\mathit{pol}(\phi\,|_{\pi.i}) := -\mathit{pol}(\phi\,|_\pi)$ si $\phi\,|_\pi$ est une formule dont le symbole de tête est une négation ou une implication avec $i = 1$;
- $\mathit{pol}(\phi\,|_{\pi.i}) := 0$ si $\phi\,|_\pi$ est une formule dont le symbole de tête est une équivalence ou si c'est un *ite* avec $i = 1$.

On note qu'une sous-formule dont la polarité est nulle (sous la portée d'une équivalence ou comme condition d'un *ite*) posséderait deux occurrences, l'une avec une polarité positive et l'autre avec une polarité négative dans une version où les équivalences et les *ite* sont éliminés. La polarité nulle permet de traiter spécifiquement les opérateurs $\Leftrightarrow$ et *ite* sans avoir à les réduire.

Définition 6 (Clôture existentielle, clôture universelle) Pour une formule φ dont l'ensemble de variables libres est $\mathcal{V}_{lib}(\varphi) = \{x_1, \ldots, x_k\}$ avec $\sigma(x_i) = \tau_i$, on appelle

- clôture existentielle de φ et on note $\exists(\varphi)$ la formule $(\exists_{\tau_1} x_1 \,.\, \ldots\, (\exists_{\tau_k} x_k \,.\, \varphi))$
- clôture universelle de φ et on note $\forall(\varphi)$ la formule $(\forall_{\tau_1} x_1 \,.\, \ldots\, (\forall_{\tau_k} x_k \,.\, \varphi))$

La partie suivante décrit comment interpréter formellement une formule du premier ordre multi-sortes.

2.2 Satisfaisabilité

Définition 7 (Interprétation d'un langage) Soit un langage du premier ordre multi-sortes égalitaire $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, \alpha, \sigma \rangle$ et un domaine $D =_{def} \bigcup_{\tau \in \mathcal{S}} D_\tau$ de valeurs possibles pour les

constantes de ce langage, où les ensembles D_τ sont non vides et disjoints. Une *interprétation* d'un langage du premier ordre multi-sortes égalitaire est la paire $\mathcal{I} = (D, I)$ où

- I attribue un élément $d \in D_\tau$ à chaque variable $x \in \mathcal{V}_\tau$; on dit alors que I est une *valuation*.
- I attribue une fonction $I(f) : D_{\tau_1} \times \ldots \times D_{\tau_n} \to D_\tau$ à chaque symbole fonctionnel $f \in \mathcal{F}$ tel que $\sigma(f) = (\tau_1, \ldots, \tau_n, \tau)$; on pose alors $I(f(t_1, \ldots, t_n)) =_{def} I(f)(I(t_1), \ldots, I(t_n))$ pour chaque τ-*terme* $f(t_1, \ldots, t_n)$;
- I attribue une fonction $I(p) : D_{\tau_1} \times \ldots \times D_{\tau_n} \to \{\top, \bot\}$ à chaque symbole prédicatif $p \in \mathcal{P}$ avec $\sigma(p) = (\tau_1, \ldots, \tau_n)$;
- I attribue au symbole $=_\tau$ la relation d'identité sur τ définie par $I(=_\tau) : D_\tau \times D_\tau \to \{\top, \bot\}$ et $I(=_\tau)(t, t') = \top$ si $I(t) = I(t')$ et $\bot$ sinon.

Avec les notations précédentes, la valeur de vérité du prédicat atomique $p(t_1, \ldots, t_n)$ est $I(p)(I(t_1), \ldots, I(t_n))$. La valeur de vérité d'une formule logique φ dépend de la valeur de vérité de ses prédicats atomiques sur lesquels on applique les tables de vérités des connecteurs logiques et les règles suivantes pour les quantificateurs :

- $(\exists_\tau x \,.\, \varphi)$ est vraie par $\mathcal{I}$ s'il existe un élément $d \in D_\tau$ tel que $\varphi(d/x)$ est vraie ;
- $(\forall_\tau x \,.\, \varphi)$ est vraie par $\mathcal{I}$ si pour tout élément $d \in D_\tau$ la formule $\varphi(d/x)$ est vraie.

Une interprétation $\mathcal{I}$ associe ainsi un élément $I(\varphi) \in \{\top, \bot\}$ à chaque formule du premier ordre φ. Par abus de langage dans la suite, on utilise les termes "interprétation I de la formule φ dans D" pour "valeur de vérité de la formule φ par l'interprétation (D, I)".

Définition 8 (Modèle) Une interprétation $\mathcal{I} = (D, I)$ est un modèle d'une formule close φ (ce qu'on note $\mathcal{I} \Vdash \varphi$) si $I(\varphi) = \top$. Par extension, $\mathcal{I}$ est un modèle d'un ensemble de formules closes S (ce qu'on note $\mathcal{I} \Vdash S$) si c'est un modèle pour chaque formule de S.

Un ensemble de formules closes S' est une *conséquence logique* d'un ensemble de formules closes S (noté $S \vDash S'$) si tout modèle de S est un modèle de S'. Deux ensembles de formules closes sont *logiquement équivalents* si tout modèle de l'un est un modèle de l'autre.

Définition 9 (Satisfaisabilité, validité) Un ensemble S de formules closes est satisfaisable si S admet un modèle. Un ensemble S de formules closes est valide si toute interprétation est un modèle de S.

Une formule ouverte φ est satisfaisable[4] (resp. valide) si sa clôture existentielle $\exists(\varphi)$ (resp. sa clôture universelle $\forall(\varphi)$) est satisfaisable.

L'objectif de la partie suivante est de présenter quelques formes particulières (dites canoniques) de formules et d'établir une relation entre l'interprétation de cette forme canonique et celle de la forme initiale.

2.3 Formes canoniques

Certaines formes canoniques de formules sont détaillées ici pour permettre leur utilisation dans les preuves de théorèmes à venir. Pour chaque forme canonique obtenue, un théorème d'équivalence logique ou seulement d'équisatisfaisabilité établit le lien avec la forme initiale.

[4]Le terme de consistance sémantique est souvent employé à la place du terme de satisfaisabilité qui est un néologisme traduit de l'anglais *satisfiability*.

$$
\begin{array}{rcl}
dnf & ::= & dnf \vee dnf \,|\, nconj \\
cnf & ::= & cnf \wedge cnf \,|\, ndisj \\
nnf & ::= & nnf \diamond nnf \,|\, Q_\tau\, x\, .\, nnf \,|\, litt \\
nconj & ::= & litt \wedge litt \,|\, litt \\
ndisj & ::= & litt \vee litt \,|\, litt \\
litt & ::= & atome \,|\, \neg atome
\end{array}
$$

où *dnf*, *cnf*, *nnf* sont respectivement des formules en forme DNF, CNF et NNF, $Q_\tau \in \{\forall_\tau, \exists_\tau\}$, $x \in \mathcal{V}_\tau$, $\diamond \in \{\wedge, \vee\}$ et *atome* est l'élément syntaxique de la figure 2.1.

FIG. 2.2 – Grammaires des formules en forme DNF, CNF et NNF

Une formule sans quantificateurs est en *forme normale disjonctive* (DNF) si c'est une disjonction de conjonctions de littéraux, en *forme normale conjonctive* (CNF) si c'est une conjonction de disjonctions de littéraux. Une formule (contenant éventuellement des quantificateurs) est en *forme normale négative* (NNF) si elle ne contient ni symboles d'implication ($\Rightarrow$) ni symboles d'équivalence ($\Leftrightarrow$) et si toute négation n'est appliquée qu'aux atomes. Ces définitions sont formalisées par la grammaire donnée à la figure 2.2.

La figure 2.3 définit la fonction *neg* permettant d'obtenir une formule sous forme normale négative à partir d'une formule générale de type *fpoms* définie dans la partie 2.1. Cette fonction termine puisqu'elle diminue soit le nombre d'équivalences, soit le nombre d'implications sans augmenter celui d'équivalences, soit la taille de la formule à laquelle elle est appliquée. Nous allons établir que cette fonction transforme une formule en une formule équivalente.

Théorème 1 *Toute formule φ est logiquement équivalente à neg(φ).*

IDÉE DE PREUVE. La preuve se fait en plusieurs étapes. Par induction sur la structure de φ, on commence par montrer qu'elle est vraie pour toute formule sans quantificateurs ne contenant que des conjonctions et disjonctions ; ce résultat est étendu aux formules sans quantificateurs contenant en plus les opérateurs *ite*, $\Rightarrow$ et $\Leftrightarrow$ et $\neg$. L'extension aux formules quantifiées se fait alors naturellement. □

Pour caractériser les sous-formules quantifiées dont la quantification n'est pas modifiée par une mise en forme NNF, on introduit les termes *existentiellement$_{nf}$ quantifiée* et *universellement$_{nf}$ quantifiée* comme suit. Soit une formule φ et une position π telles que $\varphi\,|_\pi = Q\, x\, .\, \psi$. La variable x est dite existentiellement$_{nf}$ quantifiée dans la formule ψ quand Q est $\exists$ et $pol(\psi\,|_\pi) = +1$ ou quand Q est $\forall$ et $pol(\psi\,|_\pi) = -1$. Intuitivement, cette quantification a le sens d'une quantification existentielle. De manière similaire, x est dite universellement$_{nf}$ quantifiée dans la formule ψ quand Q est $\forall$ et $pol(\psi\,|_\pi) = +1$ ou quand Q est $\exists$ et $pol(\psi\,|_\pi) = -1$ Intuitivement, cette quantification a le sens d'une quantification universelle.

Théorème 2 *Toute formule sans quantificateur φ est équivalente à une formule ψ sous forme normale conjonctive (respectivement disjonctive).*

$$
\begin{array}{rcl}
neg(\varphi_1 \diamond \varphi_2) & = & neg(\varphi_1) \diamond neg(\varphi_2) \text{ où } \diamond \in \{\wedge, \vee\} \\
neg(a) & = & a \\
neg(\varphi_1 \Rightarrow \varphi_2) & = & neg(\neg\varphi_1 \vee \varphi_2) \\
neg(\varphi_1 \Leftrightarrow \varphi_2) & = & neg((\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)) \\
neg(ite(\varphi_1, \varphi_2, \varphi_3)) & = & neg((\varphi_1 \Rightarrow \varphi_2) \wedge (\neg\varphi_1 \Rightarrow \varphi_3)) \\
neg(\neg\neg\varphi) & = & neg(\varphi) \\
neg(\neg a) & = & \neg a \\
neg(\neg(\varphi_1 \Rightarrow \varphi_2)) & = & neg(\varphi_1 \wedge \neg\varphi_2) \\
neg(\neg(\varphi_1 \Leftrightarrow \varphi_2)) & = & neg(\neg(\varphi_1 \Rightarrow \varphi_2) \vee \neg(\varphi_2 \Rightarrow \varphi_1)) \\
neg(\neg ite(\varphi_1, \varphi_2, \varphi_3)) & = & neg(\neg(\varphi_1 \Rightarrow \varphi_2) \vee \neg(\neg\varphi_1 \Rightarrow \varphi_3)) \\
neg(\neg(\varphi_1 \wedge \varphi_2)) & = & neg(\neg\varphi_1 \vee \neg\varphi_2) \\
neg(\neg(\varphi_1 \vee \varphi_2)) & = & neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
neg(\neg(\forall_\tau x \,.\, \varphi)) & = & neg(\exists_\tau x \,.\, \neg\varphi) \\
neg(\neg(\exists_\tau x \,.\, \varphi)) & = & neg(\forall_\tau x \,.\, \neg\varphi) \\
neg(\, Q_\tau x \,.\, \varphi) & = & Q_\tau x \,.\, neg(\varphi) \text{ où } Q_\tau \in \{\forall_\tau, \exists_\tau\}
\end{array}
$$

où a est un mot engendré par l'élément syntaxique *atome* de la figure 2.1.

FIG. 2.3 – Mise en forme normale négative

PREUVE. En construisant la forme normale négative φ' de φ et en appliquant la règle de réécriture $P \wedge (Q \vee R) \rightarrow (P \wedge Q) \vee (P \wedge R)$ (resp. $P \vee (Q \wedge R) \rightarrow (P \vee Q) \wedge (P \vee R)$) qui préserve l'équivalence, on obtient une formule ψ en forme normale disjonctive (resp. conjonctive) équivalente à φ. □

Pour les formules avec quantificateurs, on introduit la forme prénexe et la forme de Skolem.

Définition 10 (Forme prénexe) Une formule est sous forme prénexe si elle s'écrit sous la forme $Q_{\tau_1} x_1 \ldots Q_{\tau_n} x_n \,.\, \varphi$ où φ est sans quantificateur, $Q_{\tau_i} \in \{\forall_{\tau_i}, \exists_{\tau_i}\}$ et $x_i \in \mathcal{V}_{\tau_i}$ pour $1 \leqslant i \leqslant n$.

Intuitivement, le théorème suivant montre que les quantificateurs d'une formule peuvent être "déplacés" en tête de celle-ci.

Théorème 3 *Toute formule φ est équivalente à une formule ψ sous forme prénexe.*

PREUVE. Comme dans la preuve du théorème 2, on réécrit φ sous la forme NNF. La preuve se fait alors par induction sur la taille de la formule qui ne contient plus alors que les connecteurs logiques $\wedge$, $\vee$, $\neg$ et les deux quantificateurs $\forall$ et $\exists$. □

Définition 11 (Forme de Skolem) Une formule est en forme de Skolem si elle s'écrit sous la forme $\forall_{\tau_1} x_1 \ldots \forall_{\tau_n} x_n \,.\, \xi$ où ξ est sans quantificateur et $x_i \in \mathcal{V}_{\tau_i}$ pour $1 \leqslant i \leqslant n$. Le processus de mise en forme de Skolem est appelé skolémisation.

La procédure suivante est une skolémisation. Soit $\varphi =_{def} Q_{\tau_1} x_1 \ldots Q_{\tau_n} x_n \,.\, \psi$ une formule prénexe où Q_{τ_i} pour $1 \leqslant i \leqslant n$ est un quantificateur typé et soit $i_1 < \ldots < i_m$ les indices i tels que Q_{τ_i} est un quantificateur existentiel. Pour chaque $1 \leqslant r \leqslant m$ on effectue successivement les étapes suivantes :

1. soit $j_1 < \ldots < j_k$ les indices j tels que Q_{τ_j} est un quantificateur universel et Q_{τ_j} est à gauche du quantificateur existentiel $Q_{\tau_{i_r}}$;
2. on ajoute le nouveau symbole fonctionnel f_r d'arité k tel que $\sigma(f_r) = (\sigma(x_{j_1}), \ldots, \sigma(x_{j_k}), \sigma(x_{i_r}))$ et qui est appelé fonction de Skolem ;
3. on définit le terme $t_r =_{def} f_r(x_{j_1}, \ldots, x_{j_k})$.

La formule φ_S, obtenue à partir de φ en supprimant, pour tout $1 \leqslant r \leqslant m$, toutes les quantifications existentielles $Q_{\tau_{i_r}} x_{i_r}$ et en remplaçant chaque occurrence de x_{i_r} par t_r, est la forme de Skolem de φ.

Exemple. Soit $\mathcal{L} =_{def} \langle \{\tau_1, \tau_2\}, \mathcal{V}, \{f\}, \{R\}, \alpha, \sigma \rangle$ un langage tel que $\alpha(f) = 1$, $\alpha(R) = 2$, $\sigma(f) = (\tau_1, \tau_2)$ et $\sigma(R) = (\tau_2, \tau_2)$, et soit φ la formule

$$\exists_{\tau_2} w \; \forall_{\tau_1} x \; \forall_{\tau_1} y \; \exists_{\tau_2} z \,.\, R(w, f(x)) \vee R(z, f(y)) \vee R(f(x), z).$$

Avec les notations de la procédure précédente, on a $m = 2$, $i_1 = 1$ et $i_2 = 4$. Pour r valant 1, i_1 vaut aussi 1 et comme il n'y a pas de quantificateur universel à gauche de Q_1 on a $t_1 = f_1$. Pour r valant 2, i_2 vaut 4, il y a deux quantificateurs universels à gauche de Q_4 et on a $t_2 = f_2(x, y)$ avec $\sigma(f_2) = (\tau_1, \tau_1, \tau_2)$. La formule $\varphi_S =_{def} \forall_{\tau_1} x \; \forall_{\tau_1} y \,.\, R(f_1, f(x)) \vee R(f_2(x, y), f(y)) \vee R(f(x), f_2(x, y))$ est alors la forme de Skolem de φ.

Théorème 4 *Une formule φ est satisfaisable si et seulement si une de ses formes de Skolem φ_S l'est aussi.*

PREUVE. On conserve les notations de la définition 11. Sans perte de généralité on considère une formule close en forme prénexe φ ; la preuve est en effet extensible aux formules ouvertes en prenant leur clôture existentielle. On montre que φ est satisfaisable si et seulement si la formule φ_s obtenue par la procédure de skolémisation détaillée ci-avant l'est aussi.

Soit $\mathcal{I} = (D, I)$ un modèle de φ. Pour chaque variable existentiellement quantifiée $x_{i_r} \in \mathcal{V}_{\tau_{i_r}}$ et pour toutes les substitutions θ_j d'une variable x_j universellement quantifiée et à gauche du quantificateur $\exists_{\tau_{i_r}}$ par un élément de D_{τ_j}, il existe un élément $e_{i_r} \in D_{\tau_{i_r}}$ tel que la formule φ, dans laquelle on a substitué chaque x_{i_r} par e_{i_r} et chaque x_j par $\theta_j(x_j)$, est interprétée à $\top$. Pour chaque symbole fonctionnel $f_r \in \mathcal{F}$ introduit par skolémisation de x_{i_r}, on construit la fonction $I(f_r) \in D_{\tau_{j_1}} \times \ldots \times D_{\tau_{j_k}} \to D_{\tau_{i_r}}$ qui associe e_{i_r} aux $(\theta_{j_1}(x_{j_1}), \ldots, \theta_{j_k}(x_{j_k}))$. L'interprétation $\mathcal{I}$' qui prolonge ainsi $\mathcal{I}$ aux symboles fonctionnels introduits par la skolémisation est un modèle pour φ_s.

Réciproquement, soit $\mathcal{I} = (D, I)$ un modèle de φ_s. Ceci signifie que pour chaque symbole fonctionnel f_r introduit par skolémisation, il existe une fonction $I(f_r) \in D_{\tau_{j_1}} \times \ldots \times D_{\tau_{j_k}} \to D_{\tau_{i_r}}$ telle que $I(\varphi)$ est $\top$. La formule obtenue en substituant dans φ chaque variable existentiellement quantifiée x_{i_r} par $I(f_r)(\theta_{j_1}(x_{j_1}), \ldots, \theta_{j_k}(x_{j_k})) \in D_{\tau_{i_r}}$ et en supprimant le quantificateur associé à x_{i_r}, est vraie dans $\mathcal{I}$; $\mathcal{I}$ est donc un modèle pour φ. □

La partie suivante présente alors la méthode de Herbrand qui permet de vérifier syntaxiquement la satisfaisabilité d'une formule écrite sous la forme d'un ensemble de clauses.

2.4 Méthode de Herbrand

La méthode de Herbrand est basée sur la construction d'un domaine (éventuellement infini) de termes, nommé *Univers de Herbrand* et sur une interprétation sur ce domaine. Comme dans [Fon04], cette partie présente une version multi-sortes de la méthode. Pour une version mono-sorte, on peut se référer à [Bus98, GG91].

Définition 12 (Clause) Une *clause* est une disjonction de littéraux dont les variables sont universellement quantifiées.

Définition 13 (Univers de Herbrand) L'univers de Herbrand $\mathbb{H}$ d'un langage multi-sortes $\langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, \alpha, \sigma \rangle$ est défini par $\mathbb{H} =_{def} \bigcup_{\tau \in \mathcal{S}} \mathbb{H}_\tau$, avec $\mathbb{H}_\tau =_{def} \bigcup_{n \in \mathbb{N}} \mathbb{H}_\tau^n$ où, pour chaque sorte $\tau \in \mathcal{S}$, la suite $(\mathbb{H}_\tau^n)_{n \in \mathbb{N}}$ est définie par :

- $\mathbb{H}_\tau^0$ qui est l'ensemble des constantes de $\mathcal{F}$ de sorte τ.
- $\mathbb{H}_\tau^{n+1}$ qui est l'ensemble des termes $f(t_1, \ldots, t_m)$ où $f \in \mathcal{F}$ avec $\sigma(f) = (\tau_1, \ldots, \tau_m, \tau)$ et $t_i \in \bigcup_{0 \leq k \leq n} \mathbb{H}_{\tau_i}^k$ pour $1 \leq i \leq m$.

Définition 14 (Interprétation de Herbrand) Pour un langage multi-sortes $\langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, \alpha, \sigma \rangle$, on appelle interprétation de Herbrand toute interprétation définie sur le domaine $\mathbb{H}$ qui est l'identité sur les constantes et qui fait correspondre à chaque symbole fonctionnel $f \in \mathcal{F}$ avec $\sigma(f) = (\tau_1, \ldots, \tau_m, \tau)$ la fonction $I(f)$ associant au terme $(t_1, \ldots, t_m) \in \mathbb{H}_{\tau_1} \times \ldots \times \mathbb{H}_{\tau_m}$ le terme $f(t_1, \ldots, t_m) \in \mathbb{H}_\tau$.

On appelle *base de Herbrand* (notée B_H) d'un langage $\mathcal{L}$ l'ensemble des littéraux clos de ce langage (c.à.d. l'ensemble des atomes clos et leur négation). Pour les littéraux de B_H, les interprétations de Herbrand ne diffèrent que dans la manière d'interpréter les symboles prédicatifs.

Lemme 1 (Lemme de Herbrand) *Un ensemble S de clauses est satisfaisable si et seulement s'il admet un modèle de Herbrand.*

PREUVE. La preuve du "si" est évidente. Réciproquement, soit $\mathcal{I} = (D, I)$ un modèle de S. On considère une fonction $f : \mathbb{H} \to D$ qui associe à tout terme $h \in \mathbb{H}_\tau$ un élément de D_τ. Soit l'interprétation de Herbrand $\mathcal{I}_H = (\mathbb{H}, I_H)$ telle que $I_H(p)(t_1, \ldots, t_n) = \top$ si $I(p)(f(t_1), \ldots, f(t_n)) = \top$ pour $t_1, \ldots, t_n$ des termes de $\mathbb{H}_{\tau_1}, \ldots, \mathbb{H}_{\tau_n}$.

Pour prouver que $\mathcal{I}_H$ est un modèle pour S, montrons que pour toute substitution $\theta_H : \mathcal{V} \to \mathbb{H}$ et pour toute clause $C \in S$, $I_H(C\theta_H) = \top$.

Supposons le contraire. Pour chaque littéral $l \in C$, on a

$$\begin{aligned} I_H(l\theta_H) &= \bot \text{ c.a.d.} \\ I_H(l(t_1\theta_H, \ldots, t_n\theta_H)) &= \bot \text{ ou encore} \\ I(l(f(t_1\theta_H), \ldots, f(t_n\theta_H))) &= \bot \end{aligned}$$

par définition de I_H. On a ainsi trouvé une substitution $\theta : \mathcal{V} \to D$ qui rend faux tout littéral de C, ce qui contredit l'hypothèse que $\mathcal{I}$ est un modèle de S. □

Définition 15 (Clause fondamentale) Une clause dans laquelle on a remplacé chaque variable par un symbole de même sorte puisé dans l'univers de Herbrand est appelée clause fondamentale.

Corollaire 1 *Un ensemble S de clauses est satisfaisable si et seulement si son ensemble de clauses fondamentales est satisfaisable.*

PREUVE. Un ensemble de clauses est satisfaisable si et seulement s'il admet un modèle de Herbrand, i.e. pour chaque clause $C =_{def} \forall_{\tau_1} x_1 \ldots \forall_{\tau_k} x_k \,.\, \varphi$ et chaque substitution $\theta : \mathcal{V} \to \mathbb{H}$, la formule sans quantificateur $\varphi\theta$ est satisfaisable, i.e. son ensemble de clauses fondamentales est satisfaisable. □

On peut alors en déduire le corollaire suivant

Corollaire 2 ([FG03, Cor. 1]) *Soit un langage multi-sortes $\mathcal{L} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, \alpha, \sigma \rangle$ et soit $\tau \in \mathcal{S}$ une sorte telle qu'il n'y ait pas de symbole fonctionnel f dans $\mathcal{F}$ de signature $\sigma(f) = (\tau_1 , \ldots , \tau_n , \tau)$ avec $n \geqslant 1$, et soit x une variable de sorte τ. Supposons que $\forall x \,.\, \Phi(x)$ est une formule close de Skolem de $\mathcal{L}$. Alors $\forall x \,.\, \Phi(x)$ est satisfaisable si et seulement si la conjonction finie $\bigwedge_{c \in H_\tau} \Phi(c)$ l'est.*

Détaillée dans [FG03, Cor. 1], la preuve de ce corollaire est fondée sur la finitude du domaine de Herbrand $\mathbb{H}_\tau$, permettant de transformer une quantification universelle en une conjonction finie.

Les atomes sans variable pouvant être vus comme des variables propositionnelles, ces deux corollaires permettent de passer d'un problème de satisfaisabilité exprimé en logique du premier ordre dans un problème de satisfaisabilité d'un ensemble de clauses propositionnelles. Le théorème de *compacité* [CL93, p. 62] de la logique propositionnelle (un ensemble T de clauses propositionnelles est satisfaisable si et seulement si tout sous-ensemble fini de T est satisfaisable) permet alors d'énoncer le théorème de Herbrand proprement dit :

$$\frac{C \vee A \quad D \vee \neg B}{(C \vee D)\sigma} \text{ où } \sigma =_{def} mgu(A, B)$$

(a) resolution

$$\frac{C \vee A \vee B}{(C \vee A)\sigma} \text{ où } \sigma =_{def} mgu(A, B)$$

(b) factoring

FIG. 2.4 – Règles du calcul de résolution [Rob65]

Théorème 5 (Théorème de Herbrand) *Un ensemble de clauses n'est pas satisfaisable si et seulement s'il existe un sous-ensemble fini de clauses fondamentales non satisfaisable.*

Cet ensemble fini se construit itérativement en ajoutant des instances de clauses fondamentales et en vérifiant "de temps en temps" si l'ensemble construit est satisfaisable.

Néanmoins, ce théorème ne donne pas de conclusion lorsque l'ensemble de clauses est satisfaisable. Pire, cette vérification itérative divergera dans ce cas. Ce résultat dissymétrique entre le cas où l'ensemble est satisfaisable et celui où il ne l'est pas est fréquent dans les procédure de détection de satisfaisabilité. On parle alors de *semi-décidabilité* lorsqu'elle convergent pour tout ensemble de clauses non satisfaisable et de décidabilité lorsqu'elles convergent dans les deux cas.

Des méthodes de vérification de satisfaisabilité de clauses fondamentales sont présentées dans la partie suivante.

2.5 De Herbrand à la superposition

Pour décider de la satisfaisabilité d'un ensemble fini S de clauses fondamentales, Robinson [Rob65] propose la méthode déductive de *résolution* : à partir de deux clauses fondamentales $C \vee l$ et $D \vee \neg l$ de S, on engendre la nouvelle clause $C \vee D$ (nommée résolvante). On détecte la non satisfaisabilité de S en comparant la résolvante au vide : dans le cas où elle est engendrée par deux clauses réduites aux deux littéraux complémentaires l et $\neg l$, S est insatisfaisable. Dans le cas contraire on réapplique la résolution sur d'autres clauses. Lorsque l'ensemble de clauses engendrées devient stable sans qu'il n'y ait de contradiction, on instancie "quelques" clauses fondamentales et on recommence la résolution.

2.5.1 Résolution

Présentée à la figure 2.4(a), la règle de résolution sur des clauses ouvertes permet d'intervertir instanciation et résolution : elle tire ainsi profit de la finitude du nombre de résolutions entre deux clauses ouvertes (ce nombre est infini lorsque le nombre de clauses fondamentales correspondantes est lui aussi infini). Cette démarche est basée sur la notion d'unificateur le plus général dont la définition est donnée ci-dessous.

Définition 16 (Unificateur, unificateur le plus général) La substitution θ est qualifiée
- d'unificateur entre deux termes t et t' de sorte τ, si $t\theta$ et $t'\theta$ sont syntaxiquement égaux, et
- d'unificateur le plus général (mgu) si, de plus, tout unificateur θ' de t et t' peut se réécrire comme la composée de θ et d'une autre substitution. On note $mgu(t, t')$ un tel unificateur entre les termes t et t'.

$$
\begin{array}{rl}
(\forall\, x\,.\, x = x) & \text{réflexivité} \\
(\forall\, x, y\,.\, x = y \Rightarrow y = x) & \text{symétrie} \\
(\forall\, x, y, z\,.\, (x = y \wedge y = z) \Rightarrow x = z) & \text{transitivité} \\
(\forall\, x_1, y_1 \ldots \forall\, x_n, y_n\,.\, (\bigwedge_{1 \leq i \leq n} x_i = y_i) \Rightarrow & \\
\quad f(x_1, \ldots x_n) = f(y_1, \ldots y_n)) & \text{monotonie}
\end{array}
$$

FIG. 2.5 – Axiomes de congruence $\mathcal{E}$

$$
\frac{C \vee s = t \quad D}{(C \vee D[t]_\pi)\sigma}
$$
où $\sigma =_{def} mgu(s, D\,|_\pi)$

FIG. 2.6 – Règle de paramodulation [RW69]

Couplée avec la règle de factoring (figure 2.4(b)) la règle de résolution engendre la clause vide à partir de tout ensemble insatisfaisable de clauses ; on parle alors de calcul *réfutationnellement complet*.

Dans le contexte général des logiques du premier ordre, l'égalité a un statut différent des autres prédicats : on ne l'interprète que par la diagonale du domaine. Cette spécificité peut être oubliée en définissant le symbole d'égalité comme le prédicat binaire vérifiant l'ensemble $\mathcal{E}$ des *axiomes de congruence* de la figure 2.5 et en ajoutant la forme clausale de $\mathcal{E}$ à l'ensemble de clauses initiales. Dans cette figure, chaque axiome est en fait un schéma d'axiomes : les trois premiers doivent être instanciés pour chaque sorte τ et l'axiome de monotonie doit être instancié pour chaque symbole fonctionnel de $\mathcal{F}$.

Cette interprétation est appelée interprétation modulo $\mathcal{E}$. Axiomatique, elle possède le défaut de générer un grand nombre de clauses inutiles. La paramodulation présentée ensuite permet de résoudre ce problème.

2.5.2 Paramodulation

Dans une interprétation modulo $\mathcal{E}$, pour limiter le nombre d'inférences entre la résolution et $\mathcal{E}$, Robinson et Wos [RW69] ont introduit la règle de *paramodulation* (figure 2.6) remplaçant plusieurs étapes de résolution par une instanciation suivie du remplacement d'un sous-terme. Cette règle permet de raisonner avec le symbole $=$ directement dans la logique, sans $\mathcal{E}$, et ne nécessite pas de paramoduler dans une variable. Cette dernière conséquence permet d'éviter de nombreuses inférences. Le calcul composé des règles de factoring, de résolution et de paramodulation est aussi réfutationnellement complet pour peu qu'on ajoute à l'ensemble de clauses des axiomes de réflexivité[NR01]. Ce calcul n'est pas détaillé davantage car la superposition, présentée ci-après, est encore plus performante.

2.5.3 Superposition

Ces améliorations successives ne servent qu'un seul but : produire des conséquences aussi petites que possible et en petit nombre pour converger au plus vite vers la clause vide. Pour limiter le nombre d'inférences il est judicieux de n'effectuer les substitutions que lorsqu'elles remplacent un terme par un autre de *plus petite taille*. De tels *systèmes de déduction* s'appuient donc sur l'existence d'une relation d'ordre entre les termes, appelée ordre de réduction.

Nom	Règle	Conditions
subsumption	$\frac{C \quad C'}{C}$	$\exists\theta\ .\ \theta(C) \subseteq C'$, $\forall\rho\ .\ \rho(C') \neq C$
simplification	$\frac{C[l'] \quad l=r}{C[\theta(r)] \quad l=r}$	$l' = \theta(l), \theta(l) \succ \theta(r)$, $C[\theta(l)] \succ (\theta(l) = \theta(r))$
deletion	$\frac{C \vee l=l}{}$	

où C et C' sont des clauses, θ et ρ des substitutions, l, l' et r des termes et $C[l]$ précise que la clause C contient un littéral dont un membre est le terme l.

FIG. 2.7 – Règles de simplification pour $\mathcal{SP}$.

A partir d'un ensemble fini S de clauses équationnelles, le calcul par superposition $\mathcal{SP}$ [ARR03] applique itérativement des règles de simplification (figure 2.7) et d'inférence (figure 2.8) en privilégiant systématiquement les simplifications. Ces règles sont fondées, d'une part, sur la construction préalable d'un ordre de réduction ($\succ$) sur les termes et son extension multi-ensemble [DJ90] aux littéraux puis aux clauses et, d'autre part, sur l'existence d'un unificateur le plus général (*mgu*) entre deux termes u et u'.

Dans le cas où $S = T \cup \varphi$ avec T un ensemble fini de clauses équationnelles et φ une disjonction de littéraux clos, il est possible de conclure, lorsque $\mathcal{SP}$ termine, que $T \cup \varphi$ est satisfaisable si et seulement si l'ensemble final de clauses ne contient pas la clause vide. Lorsque la clause vide est générée, $T \cup \varphi$ n'est pas satisfaisable.

Ainsi, pour décider par superposition de la satisfaisabilité d'une formule φ modulo une théorie $\mathcal{T}$ finiement axiomatisée par $Ax(\mathcal{T})$, il suffit de saturer l'ensemble des clauses issues de la mise en forme clausale de φ et des axiomes de $Ax(\mathcal{T})$.

Ce mémoire montre aux chapitres 8, 9 et 10 que les tableaux sont des structures permettant de modéliser et de vérifier automatiquement les systèmes uniformes distribués. La partie suivante définit alors la théorie des tableaux avec extensionnalité par un ensemble fini d'axiomes en logique équationnelle permettant ainsi l'utilisation de la superposition comme système déductif.

2.5.4 Théorie des tableaux avec extensionnalité

On nomme $\mathcal{A}_s^e$ la théorie des tableaux multi-sortes construite sur les sortes VALUE, INDEX et ARRAY, avec les symboles fonctionnels write (abrégé ensuite par wr) et read (abrégé ensuite par rd) de signature $\sigma(\mathsf{wr}) = (\text{ARRAY}, \text{INDEX}, \text{VALUE}, \text{ARRAY})$ et $\sigma(\mathsf{rd}) = (\text{ARRAY}, \text{INDEX}, \text{VALUE})$.

Intuitivement, pour un terme a de sorte ARRAY, un terme i de sorte INDEX et un terme v de sorte VALUE, les termes $\mathsf{rd}(a, i)$ et $\mathsf{wr}(a, i, v)$ définissent respectivement la valeur du tableau a à l'index i et le tableau obtenu à partir de a en fixant à v la valeur à l'index i.

Les propriétés de ces symboles fonctionnels sont définies par l'ensemble $Ax(\mathcal{A}_s^e)$ de la figure 2.9 qui est une présentation finie de $\mathcal{A}_s^e$. On note $\Sigma_{\mathcal{A}}$ la signature contenant les symboles fonctionnels rd, wr, un ensemble fini non vide de symboles constants de chaque sorte et $\mathcal{A}_s$ la théorie engendrée à partir de l'ensemble d'axiomes $Ax(\mathcal{A}_s) =_{def} \{(2.1), (2.2)\}$, qui est la

Nom	Règle	Conditions
paramodulation	$\frac{\Gamma \vee l[u'] \bowtie r \quad \Delta \vee u = v}{\sigma(\Gamma \vee \Delta \vee l[v] = r)}$	$\sigma(u) \not\preceq \sigma(v)$, $\sigma(u = v) \not\preceq \sigma(\Delta)$, $\sigma(l[u']) \not\preceq \sigma(r)$, $\sigma(l[u'] = r) \not\preceq \sigma(\Gamma \vee \Delta)$
reflection	$\frac{\Gamma \vee u' \neq u}{\sigma(\Gamma)}$	$\sigma(u' = u) \not\preceq \sigma(\Gamma)$
factoring	$\frac{\Gamma \vee u = v \vee u' = v'}{\sigma(\Gamma \vee v \neq v' \vee u = v')}$	$\sigma(u) \not\preceq \sigma(v)$, $\sigma(u) \not\preceq \sigma(\Gamma)$, $\sigma(u = v) \not\preceq \sigma(u' = v' \vee \Gamma)$

où Γ et Δ sont des disjonctions de littéraux, u, v, u', v', l et r sont des termes, σ est le *mgu* de u et u', $\bowtie$ est le symbole d'égalité ($=$) ou de distinction ($\neq$) et $l[u']$ précise que le terme l contient le terme u' à une certaine position.

FIG. 2.8 – Règles d'inférence pour $\mathcal{SP}$.

$$\forall\, A, X, E \,.\, \mathsf{rd}(\mathsf{wr}(A, X, E), X) = E \tag{2.1}$$

$$\forall\, A, X, J, E \,.\, X \neq J \Rightarrow \mathsf{rd}(\mathsf{wr}(A, X, E), J) = \mathsf{rd}(A, J) \tag{2.2}$$

$$\forall\, A, B \,.\, (\forall\, X \,.\, \mathsf{rd}(A, X) = \mathsf{rd}(B, X)) \Rightarrow A = B \tag{2.3}$$

avec A et B des variables de sorte ARRAY, X et J des variables de sorte INDEX, et E une variable de sorte VALUE.

FIG. 2.9 – Axiomatisation $Ax(\mathcal{A}_s^e)$

$$\frac{s_i \succeq t}{f(\ldots, s_i, \ldots) \succ t}$$

$$\frac{\mathsf{wr}(s_1, s_2, s_3) \succ t_1 \quad \mathsf{wr}(s_1, s_2, s_3) \succ t_2}{\mathsf{wr}(s_1, s_2, s_3) \succ \mathsf{rd}(t_1, t_2)}$$

$$\frac{(s_1, \ldots, s_m) =^{\text{lex}} (t_1, \ldots, t_n)}{f(s_1, \ldots, s_m) \approx f(t_1, \ldots, t_n)}$$

$$\frac{(s_1, \ldots, s_n) \succ^{\text{lex}} (t_1, \ldots, t_n) \qquad f(s_1, \ldots, s_n) \succ t_i,\ i \in 1, \ldots, n}{f(s_1, \ldots, s_n) \succ f(t_1, \ldots, t_n)}$$

où s_i, t_i et t sont des termes, f est le symbole fonctionnel rd ou wr et $\succ^{\text{lex}}$ est l'extension lexicographique de $\succ$ qui est définie récursivement par $(t_1, \ldots, t_m) \succ^{\text{lex}} (s_1, \ldots, s_n)$ ssi $t_1 \succ s_1$ ou $t_1 = s_1$ et $(t_2, \ldots, t_m) \succ^{\text{lex}} (s_2, \ldots, s_n)$

FIG. 2.10 – Ordre LPO appliqué aux symboles de $\Sigma_{\mathcal{A}}$

théorie des tableaux privée de l'axiome (2.3) dit d'extensionnalité.

Voici un exemple de saturation d'un ensemble de clauses par superposition modulo la théorie des tableaux.

Exemple. Soit la formule

$$\varphi_{sat} =_{\text{def}} \mathsf{rd}(\mathsf{a}, \mathsf{k}) = \mathsf{r} \wedge \mathsf{rd}(\mathsf{wr}(\mathsf{a}, \mathsf{k}, \mathsf{c}), \mathsf{i}) = \mathsf{c} \ \wedge \\ \mathsf{rd}(\mathsf{wr}(\mathsf{a}, \mathsf{k}, \mathsf{c}), \mathsf{j}) = \mathsf{c} \wedge \mathsf{i} \neq \mathsf{j}$$

On souhaite décider de la satisfaisabilité de φ_{sat} modulo $\mathcal{A}_s^e$. Comme φ_{sat} ne contient pas de distinctions entre termes de sorte ARRAY, il suffit [ARR03, Lemme 7.1] de vérifier la satisfaisabilité de φ_{sat} modulo $\mathcal{A}_s$. Ceci revient à saturer l'ensemble composé de la clause

$$\mathsf{rd}(\mathsf{wr}(A, I, E), J) = \mathsf{rd}(A, J) \vee I = J \tag{2.4}$$

(qui est l'axiome (2.2) sous forme clausale), de l'axiome (2.1) et des clauses suivantes :

$$\mathsf{rd}(\mathsf{a}, \mathsf{k}) = \mathsf{r}, \tag{2.5}$$

$$\mathsf{rd}(\mathsf{wr}(\mathsf{a}, \mathsf{k}, \mathsf{c}), \mathsf{i}) = \mathsf{c}, \tag{2.6}$$

$$\mathsf{rd}(\mathsf{wr}(\mathsf{a}, \mathsf{k}, \mathsf{c}), \mathsf{j}) = \mathsf{c}, \tag{2.7}$$

$$\mathsf{i} \neq \mathsf{j}. \tag{2.8}$$

Il reste à définir un ordre de réduction $\succ$, paramètre de $\mathcal{SP}$. Plutôt que d'expliciter une relation d'ordre entre les termes du langage et de vérifier ensuite qu'elle est bien un ordre de réduction (stabilité, monotonie et bonne fondation), on engendre cet ordre à l'aide d'un constructeur : l'ordre de réduction $\succ$ retenu est induit par la relation de précédence $>$, considérée comme totale sur les constantes et telle que $\mathsf{wr} > \mathsf{rd}$ et il est construit comme l'ordre de chemin lexicographique LPO [DJ90] selon les règles de la figure 2.10.

Sous cet ensemble de règles, la clause

$$\mathsf{rd}(\mathsf{a}, \mathsf{i}) = \mathsf{c} \vee \mathsf{k} = \mathsf{i} \tag{2.9}$$

est engendrée par *paramodulation* entre (2.4) et (2.6). Dans ce cas en effet, on pose d'une part $\Gamma = \emptyset$, $l[u'] = u' = \mathsf{rd}(\mathsf{wr}(\mathsf{a},\mathsf{k},\mathsf{c}),\mathsf{i})$, $u = \mathsf{rd}(\mathsf{wr}(A,I,E),J)$, $v = \mathsf{rd}(A,J)$, $r = \mathsf{c}$, $\Delta = (I = J)$ et $\sigma = [A := \mathsf{a}, I := \mathsf{k}, E := \mathsf{c}, J := \mathsf{i}]$. D'autre part, en appliquant la première règle de la figure 2.10, on obtient $\mathsf{wr}(\mathsf{a},\mathsf{k},\mathsf{c}) \succ \mathsf{a}$ qui induit $(\mathsf{wr}(\mathsf{a},\mathsf{k},\mathsf{c}),\mathsf{i}) \succ^{\text{lex}} (\mathsf{a},\mathsf{i})$ par extension lexicographique. En appliquant alors la dernière règle de LPO, on en déduit que $\mathsf{rd}(\mathsf{wr}(\mathsf{a},\mathsf{k},\mathsf{c}),\mathsf{i}) \succ \mathsf{rd}(\mathsf{a},\mathsf{i})$ et donc que $\sigma(u) \not\preceq \sigma(v)$. Vérifier les autres contraintes d'ordre s'effectue de manière similaire.

De même, on obtient la formule

$$\mathsf{rd}(\mathsf{a},\mathsf{j}) = \mathsf{c} \vee \mathsf{k} = \mathsf{j} \tag{2.10}$$

par *paramodulation* entre (2.4) et (2.7). Enfin les axiomes (2.1) et (2.4) génèrent par *paramodulation* la clause

$$\mathsf{rd}(A,I) = E \vee I = I \tag{2.11}$$

(éliminée par *deletion*) et aucune règle ne peut plus s'appliquer. La formule φ_{sat} est donc $\mathcal{A}_s$-satisfaisable.

Prouvé comme réfutationnellement complet pour la logique équationnelle du premier ordre [NR01], le calcul $\mathcal{SP}$ peut diverger face à un ensemble de clauses satisfaisables. La partie suivante présente un résultat de décidabilité par ce calcul pour la théorie $\mathcal{A}_s^e$.

2.5.5 Résultat de décidabilité par $\mathcal{SP}$

Dans [ARR03], Armando, Ranise et Rusinowitch présentent un ensemble de résultats de décidabilité utilisant le calcul par superposition $\mathcal{SP}$. Parmi ceux-ci, on reprend celui qui va servir de base théorique pour ce mémoire.

Théorème 6 ([ARR03, Th. 7.2]) *Le calcul par superposition $\mathcal{SP}$ décide de la satisfaisabilité de toute formule de $\mathcal{A}_s^e$ sans quantificateurs.*

2.6 Résumé

Ce chapitre a présenté une collection d'outils (définitions et théorèmes) en logique et en preuve par système déductif permettant une meilleure compréhension des preuves théoriques de ce mémoire. Parmi ceux-ci, on retiendra essentiellement le lemme 1 de Herbrand permettant de se ramener à un raisonnement syntaxique et le théorème 6 de décidabilité du calcul $\mathcal{SP}$ pour la théorie des tableaux.

Pour approfondir le sujet on peut se référer à [BG01] pour une étude plus précise sur la résolution, [NR01, GN01] qui détaille davantage la paramodulation, [Wei01] qui présente une combinaison originale de superposition et d'utilisation des sortes, [DJ90] qui précise les systèmes de réécriture tandis que [Rus89] s'intéresse à la complétude des systèmes d'inférence de paramodulation et de superposition.

Le chapitre suivant présente les classes d'application de ce travail.

Chapitre 3

Classes d'application

Les motivations relatives au traitement d'une classe particulière d'exemples peuvent être (*i*) soit à l'origine d'une étude : il s'agit dans ce cas de trouver une démarche apportant une solution à cette classe ; (*ii*) soit un cadre d'application d'une étude : dans ce cas, c'est la démarche qui importe et l'enjeu consiste à la rendre suffisamment générique pour traiter autant de familles d'exemples que possible, dont la classe particulière. Néanmoins, quelles que soient les motivations, cette classe particulière se caractérise à l'aide d'une description formelle des comportements qu'elle autorise et des propriétés qu'elle traite.

Dans ce chapitre, le formalisme des systèmes de transitions étiquetés est rappelé et utilisé pour présenter les exemples classés ici en deux familles. La première famille est à rattacher aux systèmes distribués (ou répartis) qui correspondent au besoin de faire travailler indépendamment des processeurs, des machines parallèles avec, pourquoi pas, des morceaux de mémoire partagée. La seconde famille s'intéresse aux exemples industriels. Ces exemples de grande taille servent, au chapitre 11, à montrer (sans démontrer) l'efficacité des démarches présentées dans cette thèse en justifiant qu'elles supportent un passage à une grande échelle.

Au delà de la présentation de classes d'exemples, ce chapitre montre quelles structures de données sont nécessaires pour exprimer et vérifier ces exemples. Dans un cadre de résolution de conflits entre processus, par exemple, la citation suivante laisse à penser a priori que les structures d'éléments seront suffisantes : "*The problem of resolving conflicts between processes in distributed systems is of practical importance. A conflict between a set of processes must be resolved in favor of some (usually one) process and against the others : a favored process must have some property that distinguishes it from others*"[CM84]. Après réflexion, la structure d'ensemble de processus apparaît comme judicieuse : on peut alors construire un premier ensemble des éléments qui ont cette propriété de distinction et un second ensemble de ceux qui ne l'ont pas, puis prouver que le premier ensemble est réduit à un singleton. Ce chapitre montre qu'au delà de la structure "naturelle" d'ensemble, d'autres structures alternatives sont utilisables dans ce contexte.

Ce chapitre s'organise de la façon suivante : la partie 3.1 définit les systèmes uniformes distribués comme des systèmes issus, notamment, de synchronisations de systèmes de transitions étiquetés ; la partie 3.2 traite de propriétés que doivent garantir de tels systèmes, et montre alors l'intérêt des structures ensemblistes ; la partie 3.3 présente l'exemple jouet du MESI qui sert de fil rouge à toute la suite du travail, tandis que la partie 3.4 décrit à quelle famille de spécifications industrielles ont aussi été appliquées les théories et techniques de ce travail ; la partie 3.5 présente les enjeux de la vérification de systèmes abstraits paramétrés et quels défis

sont à relever. Un résumé conclut alors le chapitre.

3.1 Systèmes uniformes distribués

Les systèmes distribués ont connu un essor important ces deux dernières décennies avec l'arrivée de machines multiprocesseurs, de grilles de calcul, d'imprimantes en réseau, ... Ces applications nécessitent de savoir partager efficacement des ressources communes, de travailler avec des copies locales d'une mémoire globale, de synchroniser des tâches...

S'inspirant des ouvrages [MP92, Ray92, Bau03, CDK01, SBB+99], cette partie introduit les systèmes de transitions étiquetés, définit ensuite leur composition parallèle avec ou sans synchronisation et se termine par leur composition au moyen de variables partagées.

Dans ce qui suit, pour ne pas alourdir la présentation, on emploie avec le même sens les termes de composant(e)s, entités, sachant que cela s'applique essentiellement à des processus ou des processeurs.

3.1.1 Système de transitions étiqueté

Pour caractériser formellement les propriétés des systèmes, il est nécessaire de choisir un langage formel de spécification dont il existe une représentation sous la forme de pseudo-code informatique suffisamment expressif et qui permet en outre de définir chaque module du système individuellement pour les composer entre eux ultérieurement.

A la manière de [MP95], on commence par caractériser individuellement chaque composant à l'aide d'un système de transitions étiqueté dont la définition est donnée ci-après.

Définition 17 (Système de transitions étiqueté) Pour un langage $\mathcal{L}$ du premier ordre multi-sortes, un système de transitions étiqueté TS est un quintuplet $\langle X, D, I, act, \rightarrow \rangle$ où

- l'ensemble fini X de *variables d'état* comprend notamment les *variables de données* qui sont explicitement déclarées et modifiées par des instructions du programme et les *variables de contrôle* qui mémorisent la progression de l'exécution du programme ;
- D est l'ensemble des domaines des variables de X ;
- chaque état est une valuation ρ qui affecte à chaque élément $x \in X$ une valeur de son domaine dans D ; un état ρ satisfait alors une formule $\varphi \in \mathcal{L}$ s'il existe une interprétation $\mathcal{I}$ de $\mathcal{L}$ telle que $\mathcal{I} \Vdash \varphi(\rho(x))$;
- la formule I de $\mathcal{L}$ caractérise les états dans lesquels le programme débute son exécution ; tout état ρ satisfaisant I est appelé *état initial* ;
- act est l'ensemble des étiquettes des transitions ;
- un triplet (ρ, l, ρ') avec ρ, ρ' deux états et $l \in act$ appartient à l'ensemble des transitions $\rightarrow$ signifie que ρ' est un successeur de ρ par la transition étiquetée l.

Plus que la description syntaxique du composant par un système de transitions, c'est l'ensemble $[[TS]]$ des *exécutions* de ce système de transitions qui nous intéresse. Chaque exécution est une séquence $\rho_0, \rho_1, \ldots$ telle que ρ_0 est un état initial et pour tout $i \geq 0$, il existe un $l \in act$ tel que (ρ_i, l, ρ_{i+1}) est une transition de $\rightarrow$.

3.1.2 Composition parallèle

L'*exécution parallèle* de deux systèmes TS_1 et TS_2 de transitions, puis par extension de n systèmes, est définie comme l'exécution du système $TS_1 \otimes TS_2$ construit comme un produit

asynchrone de ces systèmes de transitions dont on donne la définition ci-après.

Définition 18 (Produit asynchrone) Pour $i = 1, 2$, soit $TS_i =_{def} \langle X, D, I_i, act_i, \rightarrow_i \rangle$ deux systèmes de transitions. Le produit asynchrone entre TS_1 et TS_2, noté $TS_1 \otimes TS_2$, est défini par

$$\langle X, D, I, act_1 \cup act_2 \cup act_1 \times act_2, \rightarrow \rangle$$

où
- l'état (ρ_1, ρ_2) vérifie I si et seulement si ρ_1 vérifie I_1 et ρ_2 vérifie I_2,
- une transition $((\rho_1, \rho_2), l, (\rho'_1, \rho'_2))$ appartient à l'ensemble des transitions $\rightarrow$ si
  - $l \in act_1$, $(\rho_1, l, \rho'_1) \in \rightarrow_1$ et $\rho_2 = \rho'_2$, ou
  - $l \in act_2$, $(\rho_2, l, \rho'_2) \in \rightarrow_2$ et $\rho_1 = \rho'_1$, ou
  - $l = (l_1, l_2)$, $l_1 \in act_1$, $l_2 \in act_2$, $(\rho_1, l_1, \rho'_1) \in \rightarrow_1$ et $(\rho_2, l_2, \rho'_2) \in \rightarrow_2$.

3.1.3 Synchronisation

Le produit asynchrone permet de définir l'ensemble des états pour un système dont les composantes n'interagissent pas entre elles. Dans le cas contraire, on parle de *synchronisation*, notion détaillée dans la définition de produit synchrone qui suit.

Définition 19 (Produit synchrone) Pour $1 \leqslant i \leqslant n$, soit $TS_i =_{def} \langle X, D, I_i, act_i, \rightarrow_i \rangle$ des systèmes de transitions étiquetés. Le produit synchronisé $\oplus$ entre TS_1, TS_2, ..., TS_n est défini comme la restriction du produit asynchrone aux transitions autorisées, définies par un ensemble *Sync* d'étiquettes d'actions.

La synchronisation par envoi/réception de message est un cas particulier de produit synchronisé. Elle concerne les systèmes ayant trois sortes de transitions : les transitions *internes*, les *rendez-vous* et celles de *diffusion*. Elles sont décrites dans une sémantique d'entrelacement, c.a.d. un contexte où deux transitions autorisées de *Sync* ne peuvent avoir lieu simultanément.

Une transition interne d'étiquette l est une action locale à un seul processus qui passe d'un état dans un autre état. Cette transition, qui ne se synchronise avec aucune autre, correspond à l'inclusion de $\{l\}$ dans l'ensemble *Sync*.

Un rendez-vous est une transition qui synchronise une transition d'un processus p avec une transition d'un processus p', par exemple par l'envoi et la réception d'un message m bloquant, étiquetés respectivement $m!$ et $m?$. Le processus effectuant la transition $m!$ est l'émetteur, celui effectuant la transition $m?$ le récepteur. Dans le produit synchronisé, seules sont autorisées les transitions où une émission $m!$ est accompagnée d'une réception $m?$, ce qui se traduit par l'inclusion de $\{(m!, m?)\}$ dans l'ensemble *Sync*.

Enfin une transition de diffusion (*broadcast*) change l'état de chaque processus. Un processus envoie un message de diffusion à tous les autres en franchissant la transition $(\rho_1, l!!, \rho'_1)$ de $\rightarrow$. Tous les autres se déplacent d'un état ρ_2 vers un état ρ'_2 lorsque $(\rho_2, l??, \rho'_2)$ est dans $\rightarrow$. Il se déplace alors depuis l'état ρ_1 vers l'état ρ'_1. Par souci de clarté, ce mémoire se restreint à l'étude de réception déterministe de broadcast, c'est à dire telle que, pour chaque état ρ et chaque étiquette de broadcast $l??$, il existe exactement un état ρ' tel que $(\rho, l??, \rho')$ est dans $\rightarrow$. Ainsi les transitions $(\rho, l??, \rho)$ ne sont pas dessinées sur les figures.

La figure 3.3 est un exemple de système de transitions avec synchronisation par broadcast de message. Les systèmes avec transitions internes, rendez-vous ou broadcast composent la classe des *systèmes de broadcast*.

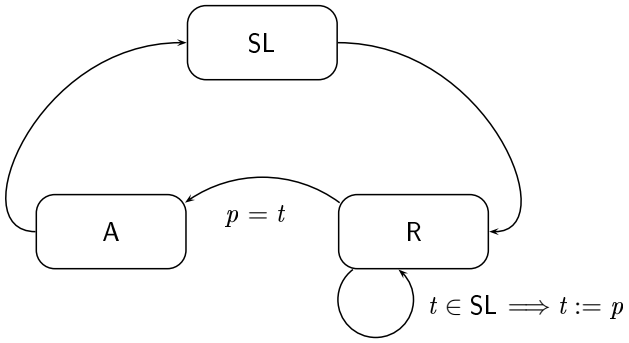


FIG. 3.1 – Système de transitions du MutEx.

3.1.4 Partage de variables

Lorsque les composantes d'un système partagent des variables, au lieu de définir une synchronisation à l'aide d'une restriction sur les transitions du produit asynchrone, il est plus judicieux de contraindre l'exécution d'une transition à la validité d'un prédicat portant sur ces variables. On appelle alors *garde* le prédicat qui bloque ou autorise une transition suivant sa valeur de validité.

En guise d'exemple de synchronisation de systèmes de transitions par variables partagées, on présente l'algorithme MutEx qui est une version simplifiée (issue de [BLS01]) de l'algorithme d'exclusion mutuelle de Dijkstra [Dij65]. Chaque processus p contrôlé par cet algorithme suit le système de transitions donné à la figure 3.1, où SL, R et A définissent respectivement les ensembles des processus endormis (*sleeping*), prêts (*ready*) et actifs. La variable t partagée entre tous les processus indique lequel a le droit d'être actif.

La transition de SL vers R exprime qu'un processus peut devenir prêt sans aucune condition. Celle qui effectue une boucle autour de l'état R garantit au processus p le droit d'obtenir la section critique lorsque le précédent détenteur de ce droit, t, est endormi. La transition de R vers A n'est franchie que par le processus autorisé –au cours des autres transitions– à utiliser la section critique. Celle de A vers SL exprime qu'un processus actif peut s'endormir sans aucune condition.

MutEx est un exemple d'algorithme d'exclusion mutuelle basé sur l'utilisation d'une variable sémaphore.

La partie suivante présente plus généralement les propriétés des systèmes distribués auxquelles s'intéresse ce travail.

3.2 Propriétés

Cette partie se focalise sur les propriétés d'exclusion mutuelle et de cohérence de caches auxquelles s'intéresse ce travail. Elle se conclut en exhibant les besoins, en terme de structures de données, que soulèvent ces propriétés.

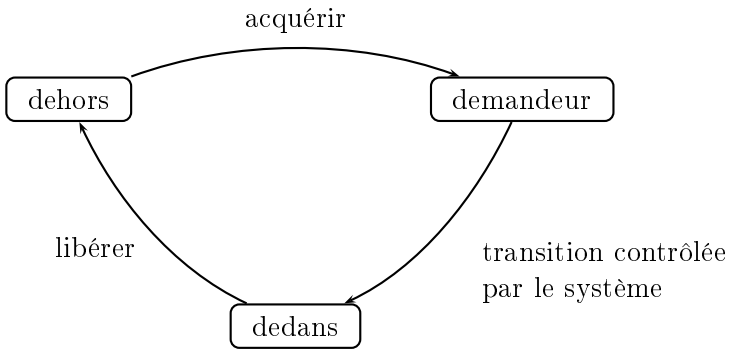


FIG. 3.2 – Exclusion mutuelle : transition d'un processus.

3.2.1 Exclusion mutuelle

Lorsqu'une ressource physique ou logique (comme une séquence d'instructions) partagée par plusieurs entités ne doit être accessible que par une seule de ces entités à la fois, cette ressource est appelée *section critique*. Tout programme qui garantit l'atomicité des accès à une section critique est dit posséder la *propriété d'exclusion mutuelle*.

Toute entité, dont le comportement est contrôlé par un algorithme d'exclusion mutuelle, aussi complexe soit-il, suit le graphe de transitions de la figure 3.2 [Ray92]. Dans ce graphe, chaque entité initialement *dehors* cherche à *acquérir* le droit d'entrer dans la section critique et devient alors un *demandeur*. L'attribution par le système du droit permettant d'accéder à cette section critique constitue la spécificité de l'algorithme. L'entité en ayant terminé avec la section critique invoque une transition permettant de la *libérer*. Parmi les protocoles d'exclusion mutuelle, on trouve Szymanski [Szy88], bakery [Lam74], MuxSem [MP92], Dijkstra [Dij65], ticket, Peterson, ...

3.2.2 Cohérence de caches

La seconde propriété étudiée est celle de *cohérence de caches* [PP84, Han93, PD97] : lorsqu'une partie de la mémoire centrale est physiquement dupliquée en plusieurs exemplaires nommés *caches*, il est nécessaire de garantir que la mémoire centrale et l'ensemble des caches sont cohérents. Lorsqu'un cache est modifié, les autres doivent soit être mis à jour pour garantir la cohérence des informations et on parle d'écriture-généralisée (write-update en anglais), soit être déclarés comme invalides pour en interdire la lecture et on parle d'écriture invalidante (write-invalidate en anglais). Un algorithme de cohérence de caches définit un ensemble de règles qui coordonnent les contrôleurs de cache, les contrôleurs de mémoire centrale et les processeurs. On se limite par la suite aux protocoles d'écriture-invalidante. Parmi les protocoles de cohérence de caches, on trouve Mesi [PP84], Illinois [PP84], Berkeley [KEW+85], German [Ger00], DEC Firefly [TS87].

3.2.3 Patron de propriétés

Cette partie montre que les deux principales propriétés étudiées dans ce mémoire s'écrivent sous une forme normale.

Pour une propriété d'exclusion mutuelle, on nomme $\mathit{Crit}(a)$ le prédicat qui vaut $\top$ si et seulement si l'entité a est dans la section critique. Un programme garantit une propriété d'exclusion mutuelle si la formule

$$\forall\, a, b\, .a \neq b \wedge \mathit{Crit}(a) \Rightarrow \neg \mathit{Crit}(b) \tag{3.1}$$

est vraie dans tous ses états.

Pour une propriété de cohérence de caches, en se plaçant dans le cadre de la partie 3.2.2 on nomme $\mathit{Modified}(a)$ (resp. $\mathit{Shares}(a)$) le prédicat qui vaut $\top$ si et seulement si l'entité a a modifié son cache (resp. partage une copie de la mémoire centrale). Il s'agit alors de vérifier si la formule

$$\forall\, a, b\, .a \neq b \wedge \mathit{Modified}(a) \Rightarrow \neg \mathit{Shares}(b) \tag{3.2}$$

est vraie dans tous les états du programme. Classiquement, on ajoute à cet invariant la propriété d'exclusion mutuelle

$$\forall\, a, b\, .a \neq b \wedge \mathit{Modified}(a) \Rightarrow \neg \mathit{Modified}(b) \tag{3.3}$$

qui stipule que deux processeurs ne peuvent modifier simultanément leur propre cache.

Sans difficulté, on remarque que (3.1), (3.2) et (3.3) sont toutes des instances du patron de propriétés

$$\forall\, a, b\, .a \neq b \wedge P(a) \Rightarrow \neg Q(b) \tag{3.4}$$

où P et Q sont des symboles prédicatifs éventuellement égaux.

3.2.4 Structures de spécification adéquates

La partie précédente a montré que les systèmes au centre de ce travail doivent garantir des propriétés d'exclusion mutuelle. Ces algorithmes se partagent en deux classes : les algorithmes fondés sur l'obtention de permissions et ceux fondés sur l'utilisation d'un jeton. Dans le premier cas un processus désireux d'accéder à la section critique doit obtenir une permission des autres ou d'un contrôleur pour accéder à l'*ensemble* des états autorisés. Cette permission n'est accordée que lorsque le système est dans une configuration particulière qui dépend des états d'un ensemble de processus. Dans le second cas l'unicité du jeton garantit l'exclusivité de l'accès à la section critique. Dans ce cas la structure représentant le jeton peut être la même que celle qui représente un processus.

Pour établir une spécification, il est donc nécessaire de posséder au moins deux sortes et donc deux structures distinctes : une pour mémoriser une entité de type processus et une autre pour représenter un ensemble de processus. Pour mémoriser une entité, on utilise par la suite une sorte d'*éléments* ou d'*indices* ; pour représenter un ensemble d'entités on utilise les sortes d'*ensemble d'éléments*, de *fonction totale* et de *tableau* pour lesquelles il faut au moins prévoir des opérateurs classiques tels que

- union, intersection, ... dans le cas des ensembles,
- image, surcharge, ... dans le cas des fonctions et
- lecture/écriture d'une valeur à un indice précis dans le cas des tableaux.

Le chapitre 5 montrera comment spécifier ces systèmes à l'aide d'ensembles ; pour les mêmes systèmes, le chapitre 9 utilisera des fonctions et le chapitre 10 des tableaux.

3.3 Exemple fil rouge

Cette partie présente l'algorithme MESI [PP84] qui sert d'illustration aux démarches de vérification présentées dans ce mémoire. Cet algorithme modélise le comportement de caches de processeurs distribués ayant un comportement uniforme. Le système de transitions d'un cache est représenté à la figure 3.3 et possède quatre états M, E, S, I, donnant leur nom à l'algorithme.

Un cache dans l'état I est invalide : les données qu'il contient ne doivent pas être lues car elles ne sont pas identiques à celles de la mémoire centrale. Toute transition qui sort de cet état met à jour le cache avec le contenu de la mémoire centrale. Dans l'état partagé S (pour "Shared"), le contenu du cache est une copie. Un cache dans l'état E est aussi une copie de la mémoire centrale, mais il dispose, de plus, de l'exclusivité pour se modifier. Dans l'état M (pour "modifié"), un cache n'est plus cohérent par rapport à la mémoire centrale. Toute transition qui fait sortir un cache de cet état met à jour la mémoire centrale avec le contenu de ce cache.

On détaille ci-après le comportement global du système qui doit garantir que les deux propriétés de cohérence de cache sont respectées, à savoir :

- il n'y a qu'un cache dans l'état M et
- lorsqu'un cache est en S aucun cache n'est en M.

Initialement, chaque processeur a un cache invalidé, c'est à dire dans l'état I.

Chaque cache peut être lu par l'action locale *read* s'il est dans un état quelconque sauf dans I. Un cache c invalidé (c.a.d. dans l'état I) souhaitant lire dans la mémoire envoie le message de broadcast *read*, auquel réagissent tous les autres caches dans les états M et E en se déplaçant dans l'état S. Le retour dans l'état S du cache qui était dans l'état M s'accompagne d'une mise à jour de la mémoire. Le cache c parvient alors à l'état S après une mise à jour de son contenu par celui de la mémoire centrale.

Pour modifier son contenu, un cache dans l'état S demande l'invalidation des autres caches en leur envoyant le message de broadcast *writeInv*. Il accède alors à l'état E à partir duquel il peut modifier son contenu, ce qui est représenté par une action locale *write* qui le fait passer dans l'état M.

L'exemple MESI possède les caractéristiques requises pour illustrer notre approche : le système de transitions de chaque entité est suffisamment petit (une transition interne et deux transitions de broadcast) pour être présenté sous plusieurs perspectives et c'est une propriété de cohérence de cache qui doit être vérifiée. La partie suivante justifie l'étude de spécifications industrielles même si elles sortent du cadre strict posé dans ce chapitre.

3.4 Spécifications industrielles

Les exemples dits "industriels" composent la deuxième famille sur laquelle la démarche est appliquée. Leur grande taille induit plusieurs conséquences. Elle permet expérimentalement de :

- s'assurer que la démarche proposée supporte un passage à une grande échelle,
- comparer les outils implantant les démarches de vérification en exhibant un écart majeur de temps de calcul qui cette fois est significatif,
- convaincre les partenaires académiques et industriels du bien fondé de l'approche.

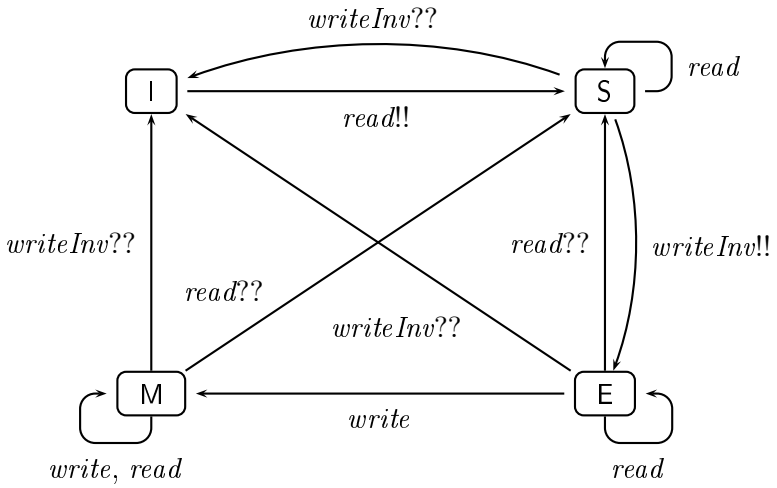


FIG. 3.3 – Système de transition d'un cache MESI.

De plus les besoins industriels n'étant pas en bijection avec les connaissances théoriques et pratiques du moment, ces spécifications proposent des pistes d'extension des méthodes à privilégier, du moins ponctuellement, et évitent de s'engouffrer immédiatement dans certaines branches théoriques qui ne seraient pas utilisées en pratique.

Ces exemples concernent principalement la validation de protocoles de communication autour de la carte à puce, de commandes automobiles et de porte monnaie électroniques. Il sont classiquement représentés par un seul système de transitions avec un grand nombre d'états (fini), contrairement aux compositions parallèles des exemples précédents ; ils sont ainsi plus proches de l'implantation que d'un modèle abstrait du problème.

Ces exemples sont exprimés dans le langage des machines abstraites B (cf chapitre 5) pour lesquelles des invariants doivent être établis (cf. chapitre 4).

3.5 Systèmes abstraits paramétrés

Au lieu de s'attaquer péniblement à des systèmes complexes et trop détaillés, le spécifieur élabore souvent ses premiers modèles sur des structures de données abstraites pour mettre en valeur le cœur des algorithmes et repousser l'introduction de détails à des étapes ultérieures de raffinement. Ceci conduit souvent à remplacer des domaines bornés par des domaines infinis, et des structures ordonnées (listes, files, ...) par des ensembles sans structure. Cette méthodologie nommée *développement incrémental* permet d'étudier des systèmes plus petits mais aussi plus abstraits.

Lorsqu'elles sont établies sur le modèle abstrait, les propriétés de sûreté, comme celles présentées à la partie précédente, sont conservées par tout raffinement car celui-ci ne fait qu'enlever des comportements. Par conséquent, le spécifieur a tout à gagner à vérifier ses propriétés au plus tôt.

Si l'on prend pour exemple un système uniforme distribué où le nombre de composants est abstrait par un entier n positif, vérifier en une seule étape les propriétés pour tout n est un véritable challenge : en cas de réussite, plus aucune autre vérification n'est nécessaire quelque

que soit ledit nombre n.

Cependant, derrière cet idéal séduisant se cachent des difficultés : les techniques classiques de model-checking ne s'appliquant plus en raison de l'infinitude du graphe d'exécution, il est nécessaire de mettre en place des méthodes symboliques capturant la notion d'infini comme celles présentées aux chapitres suivants.

3.6 Résumé

Ce chapitre a présenté les classes d'exemples et de propriétés que la démarche présentée dans ce mémoire vise à traiter. Ces exemples sont définis à l'aide d'une composition synchronisée de systèmes de transitions étiquetés.

La partie suivante s'intéresse aux langages de spécification qui préservent la sémantique des systèmes de transitions (sans en imposer la lourdeur syntaxique) et aux techniques de vérification associées. Un tel langage doit posséder des structures permettant de représenter un élément et un ensemble d'éléments.

Chapitre 4

Construction d'invariants

Le chapitre précédent a introduit les classes de systèmes pour lesquelles ce travail apporte une solution de vérification en insistant sur le challenge que représente la vérification de systèmes abstraits paramétrés. Dans de tels systèmes, l'espace des états étant infini, seule une méthode capturant cette infinitude est à même d'apporter un verdict.

Traitant de formules quantifiées, les logiques du premier ordre multi-sortes répondent à cette attente mais nécessitent, pour être employées, que les questions de vérification soient traduites dans leur langage de prédicats. Pour mettre en œuvre ceci il est suffisant de traduire les ensembles d'états en formules du premier ordre nommées *assertions*, les programmes en transformateurs d'assertions et de disposer d'une procédure de décision adaptée.

La partie 4.1 introduit des éléments de terminologie et de méthodologie de vérification de propriétés de sûreté par construction d'invariants. Cette méthode est définie au niveau des systèmes de transitions. La partie 4.2 définit deux langages : celui des assertions et celui des substitutions généralisées comme langage de spécification. A l'aide d'opérateurs classiques [Sif82], la partie 4.3 montre que les programmes peuvent être vus comme des transformateurs d'assertions, dont la pertinence est évaluée dans un contexte de vérification par preuve automatique. Elle établit ainsi que la méthode déductive de vérification d'invariant correspond à la méthode de la partie 4.1 et traite en plus le cas infini. Différents semi-algorithmes qui implantent cette méthode déductive de construction d'invariants sont donnés en partie 4.4.

4.1 Méthode

Les programmes étudiés étant paramétrés, chacun d'entre eux peut potentiellement débuter dans différents états. Cet ensemble d'états, appelé ensemble d'états initiaux et noté I, est inclus dans l'espace Σ des états du programme. Dans ce contexte, une propriété de *sûreté* exprime qu'un programme ne doit pas évoluer vers certains états critiques ou d'erreur depuis un état initial choisi dans I. Notons J le complémentaire de ces états critiques c'est à dire l'ensemble des états de Σ non critiques. Par la suite, on confond la propriété de sûreté avec cet ensemble J d'états non critiques.

En nommant K l'ensemble des *états atteignables* c.a.d. tous les états qui sont accessibles à partir des états initiaux du programme, la condition nécessaire et suffisante pour que la propriété de sûreté J soit établie est que l'ensemble K soit inclus dans J.

Cependant K n'est pas nécessairement donné, tandis que J l'est. Pour vérifier la propriété J, il suffit, par exemple, de s'assurer que chaque état initial vérifie la propriété (c.a.d. que $I \subseteq$

J) et que toute opération du programme depuis un état de J mène dans un état de J. Si c'est le cas, l'ensemble J et la propriété sont qualifiés d'*invariants* du programme dont l'ensemble d'états initiaux est I. Cependant, cette condition sur J n'est que suffisante : une propriété de sûreté peut être établie sur K sans être invariante. Lorsqu'il est non vide, l'ensemble $J \setminus K$ peut en effet contenir des états qui mènent à des états critiques, rendant ainsi J non invariant. L'exemple suivant illustre un tel cas.

Exemple. Considérons le programme qui est défini pour une variable $x \in \mathbb{Z}$ et tel que
- si $x > 0$, le programme incrémente x de 1 et
- si $x \leqslant 0$, le programme décrémente x de 1.

La propriété de sûreté $J =_{def} \{x \mid x \in \mathbb{N}\}$ est établie pour le programme précédent lorsqu'il débute dans l'ensemble d'états initiaux particulier $I =_{def} \{x \mid x = 1\}$: à chaque itération, la valeur de x augmente de 1 et est ainsi strictement positive. L'ensemble des états atteignables est donc $K =_{def} \{x \mid x \in \mathbb{N} \setminus \{0\}\}$. Par contre la propriété J n'est pas invariante : lorsque x vaut 0, le programme est dans un état qui vérifie la propriété mais il retourne la valeur -1 qui ne vérifie plus celle-ci.

La proposition suivante établit une condition nécessaire et suffisante à l'établissement d'une propriété de sûreté. Celle-ci est à la base de toutes les méthodes de vérification de propriétés de sûreté de ce travail.

Proposition 1 Une propriété de sûreté J est établie par un programme si et seulement s'il existe un ensemble J' qui est un invariant du programme et qui est inclus dans J. Dans ce cas, on dit que J' est un *renforcement* de J.

Preuve. La preuve du si ($\Leftarrow$) s'effectue en remarquant que J' étant un invariant, l'ensemble I des états initiaux du programme est inclus dans J' et J' est stable par ce programme. L'ensemble K des états atteignables du programme est donc inclus dans J', donc dans J par hypothèse.

La preuve du seulement si ($\Rightarrow$) s'effectue en choisissant pour J' l'ensemble K des états atteignables depuis l'ensemble des états initiaux du programme : cet ensemble contient I et toute opération du programme depuis un état atteignable mène dans un état atteignable, donc cet ensemble est un invariant. □

On s'intéresse dans la suite à des moyens algorithmiques de construire J'. On rappelle ci-après deux méthodes, nommées *propagation avant* et *propagation arrière* qui, à partir des ensembles I et J respectivement, calculent ce J'.

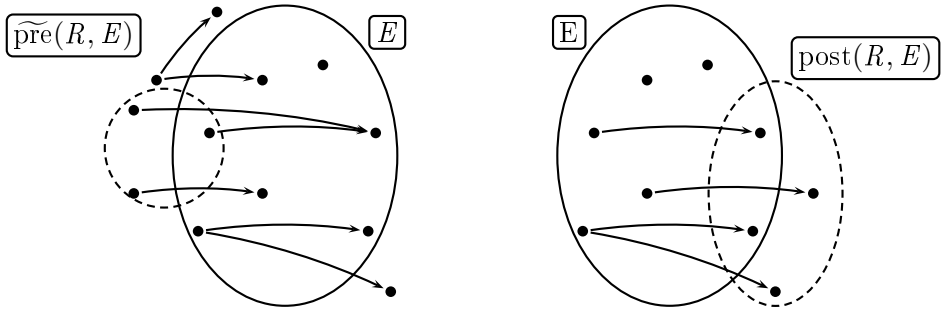
La propagation avant calcule la limite J_m d'une suite croissante, au sens de l'inclusion, d'ensembles des états accessibles depuis I, I compris. De manière duale, étant donné un ensemble d'états J, la propagation arrière calcule la limite J_M d'une suite décroissante d'ensembles des états dont tous les successeurs appartiennent à J, J_M compris. Pour construire ces suites, on a besoin de la définition suivante.

Définition 20 (post et $\widetilde{\text{pre}}$) Soit (Σ, R) un système de transitions et E une partie de Σ. Les fonctions post et $\widetilde{\text{pre}}$ définies par

$$\text{post}(R, E) = \{\rho \mid \rho \in \Sigma \wedge (\exists\, \rho' \,.\, (\rho', \rho) \in R \wedge \rho' \in E)\} \quad (4.1)$$

$$\widetilde{\text{pre}}(R, E) = \{\rho \mid \rho \in \Sigma \wedge (\forall\, \rho' \,.\, (\rho, \rho') \in R \Rightarrow \rho' \in E)\} \quad (4.2)$$

construisent respectivement l'ensemble des successeurs de E selon une application de R et l'ensemble des prédécesseurs de E selon une application de R qui mènent uniquement dans E.



(a) Prédécesseurs ne menant que vers l'ensemble E

(b) Successeurs de l'ensemble E

FIG. 4.1 – Transformateurs d'ensembles

Dans la figure 4.1 qui représente ces deux ensembles, on constate que l'indéterminisme est un point essentiel dans leur construction : certains prédécesseurs d'éléments de E n'appartiennent pas à $\widetilde{\text{pre}}(R, E)$ car ils sont à l'origine de transitions dont l'image n'appartient pas à E. On donne ci-après la définition d'invariant basée sur ces transformateurs d'ensemble.

Définition 21 (Invariant de programme) Soit (Σ, R) un système de transitions et I une partie de Σ, définissant l'ensemble des états initiaux d'un programme dont l'ensemble des états est donné par Σ et dont les transitions sont définies par la relation R. U est un invariant du programme défini par (Σ, R, I)

1. si et seulement si $I \subseteq U$ et $\text{post}(R, U) \subseteq U$ ou
2. si et seulement si $I \subseteq U$ et $U \subseteq \widetilde{\text{pre}}(R, U)$.

L'équivalence entre 1 et 2 s'obtient en remarquant que

$$
\begin{array}{lll}
 & U \subseteq \widetilde{\text{pre}}(R, U) & \\
\equiv & \forall\, \rho.\, \rho \in U \Rightarrow \rho \in \widetilde{\text{pre}}(R, U) & \text{(def. inclusion)} \\
\equiv & \forall\, \rho.\, \rho \in U \Rightarrow (\forall\, \rho'\,.\, (\rho, \rho') \in R \Rightarrow \rho' \in U) & \text{(def. } \widetilde{\text{pre}}\text{)} \\
\equiv & \forall\, \rho, \rho'.\, (\rho \in U \wedge (\rho, \rho') \in R) \Rightarrow \rho' \in U & \text{(mise en forme prénexe)} \\
\equiv & \forall\, \rho', \rho.\, (\rho' \in U \wedge (\rho', \rho) \in R) \Rightarrow \rho \in U & \text{(renommage de variables)} \\
\equiv & \forall\, \rho.\, (\exists\, \rho'\,.\, \rho' \in U \wedge (\rho', \rho) \in R) \Rightarrow \rho \in U & \text{(réduction de portée de la quantification sur } \rho'\text{)} \\
\equiv & \forall\, \rho.\, \rho \in \text{post}(R, U) \Rightarrow \rho \in U & \text{(def. post)} \\
\equiv & \text{post}(R, U) \subseteq U & \text{(def. inclusion)}
\end{array}
$$

On note que post est distributif vis à vis de l'union et que $\widetilde{\text{pre}}$ est distributif vis à vis de l'intersection. La preuve est immédiate : considérons deux ensembles E et F de l'espace des états ; dans $\text{post}(R, E \cup F)$ (resp. dans $\widetilde{\text{pre}}(R, E \cap F)$) on réécrit $\rho' \in E \cup F$ (respectivement $\rho' \in E \cap F$) en $\rho' \in E \vee \rho' \in \cup F$ (resp. $\rho' \in E \wedge \rho' \in \cup F$) et on exploite la distributivité du $\wedge$ sur le $\vee$.

On construit avec ces notations les suites $(I_n)_{n\in\mathbb{N}}$ et $(J_n)_{n\in\mathbb{N}}$ telles que $I_0 = I$, $I_n = I_{n-1} \cup \text{post}(R, I_{n-1})$, $J_0 = J$ et $J_n = J_{n-1} \cap \widetilde{\text{pre}}(R, J_{n-1})$. Par récurrence sur n et en utilisant la distributivité de post (resp. de $\widetilde{\text{pre}}$) par rapport à l'union (resp. à l'intersection), on a, pour $n \geqslant$

1, $I_n = I_{n-1} \cup \text{post}^n(R, I)$ et $J_n = J_{n-1} \cap \widetilde{\text{pre}}^n(R, J)$, où $\text{post}^n(R, I) = \text{post}(R, \text{post}^{n-1}(R, I))$, avec $\text{post}^1(R, I) = \text{post}(R, I)$ et de même pour $\widetilde{\text{pre}}$.

La propagation avant (respectivement arrière) est une itération du calcul de I_n (resp. de J_n). Les limites de $(I_n)_{n\in\mathbb{N}}$ et de de $(J_n)_{n\in\mathbb{N}}$ sont respectivement notées J_m et J_M lorsqu'elles existent.

Détaillons à présent en quoi consiste la méthode de vérification. Supposons dans un premier temps la terminaison de la méthode de propagation avant, ce qui revient à supposer que l'on dispose de J_m. D'après la proposition 1, la propriété de sûreté J est vérifiée si et seulement si l'inclusion $J_m \subseteq J$ est établie. De manière similaire, si l'on suppose la terminaison de la méthode de propagation arrière et donc l'existence de J_M, il est nécessaire et suffisant que l'inclusion $I \subseteq J_M$ soit vraie pour que la propriété de sûreté soit établie.

Pour répondre au problème de terminaison du calcul de J_m (resp. de J_M), il suffit de savoir décider l'inclusion de I_{n+1} dans I_n (resp. de J_n dans J_{n+1}). Cependant, même lorsque les réponses aux deux questions précédentes sont constamment négatives c.a.d. que les termes des suites $(I_n)_{n\in\mathbb{N}}$ et $(J_n)_{n\in\mathbb{N}}$ divergent, on peut vérifier "de temps en temps" l'inclusion d'un élément I_n dans J (resp. celle de I dans J_n), selon le sens de la propagation choisi. Si l'inclusion n'est pas établie, il est inutile de poursuivre le calcul car le raisonnement énoncé au paragraphe précédent s'applique : la propriété de sûreté est violée.

La proposition suivante établit quelques propriétés sur J_m et J_M lorsqu'ils existent.

Proposition 2 (Plus petit invariant, plus grand invariant) Pour un système de transitions (Σ, R) et deux ensembles I et J tels que $I \subseteq J \subseteq \Sigma$.

1. Si l'ensemble J_m existe, alors c'est le plus petit des invariants de (Σ, R, I).
2. Si l'ensemble J_M existe et contient I, alors c'est le plus grand des invariants de (Σ, R, I) inclus dans J.

PREUVE. (adaptée de [GGT00, p. 348]) On ne détaille que la preuve de l'item 1, l'autre étant similaire. On rappelle tout d'abord que J_m est l'union des états accessibles depuis I, I compris ; étant eux mêmes accessibles, les ensembles de successeurs de J_m sont donc inclus dans J_m. Par conséquent, J_m est un invariant.

Montrons à présent que J_m est inclus dans tout invariant J'. Soit un invariant J' ; par définition, (i) $I \subseteq J'$ et (ii) $\text{post}(R, J') \subseteq J'$. De (ii), on a que l'ensemble J' contient l'ensemble des états accessibles depuis J' ; (i) ajoute que cet ensemble contient I, donc contient aussi l'ensemble J_m des états accessibles depuis I. Par conséquent, J_m est inclus dans J'. □

En décrivant syntaxiquement les ensembles d'états et les programmes, la partie suivante transfère au niveau symbolique ce qui a été fait, dans cette partie, au niveau des systèmes de transitions.

4.2 Langage d'assertions et de programmes

Vérifier, par construction d'invariant, des propriétés de sûreté sur des systèmes spécifiés à l'aide de systèmes de transitions est réaliste lorsque l'espace total des états est fini et de taille raisonnable.

Dans le cas contraire, il est nécessaire de proposer une démarche prenant en compte cette infinitude ou cette grande taille. L'utilisation d'une logique traitant des formules quantifiées est

une réponse, mais celle-ci nécessite de savoir transformer les questions d'inclusion d'ensembles en des obligations de preuve de cette logique.

Pour cela, les ensembles d'états sont représentés par des formules (partie 4.2.1) et le programme qui modifie ces ensembles d'états est décrit à l'aide de substitutions généralisées. Le choix d'un tel langage est motivé à la partie 4.2.2, sa syntaxe est détaillée à la partie 4.2.3 et sa sémantique opérationnelle est donnée à la partie 4.2.4.

4.2.1 Des ensembles d'états aux assertions

Pouvoir exprimer les ensembles d'états du programme à l'aide d'une formule logique est un pré-requis à l'utilisation de la logique comme démarche de vérification.

Soit un programme portant sur un vecteur $X = (x_1, \ldots, x_n)$ de variables d'état de domaine $D = D_1 \times \ldots \times D_n$. Un état de ce programme étant une valuation ρ de chacune de ces variables par un élément de son domaine, on définit un ensemble E d'états comme une formule logique φ dont les seules variables libres appartiennent à X et telle que ρ est un état de E si et seulement si $\varphi\rho$ est interprétée à vrai. On dit que la formule φ est une *assertion* qui définit E. On note $\varphi(X)$ pour montrer que les seules variables libres de φ appartiennent à X. De manière usuelle [GGT00, MP95, BLS01], on confond parfois dans la suite une assertion avec l'ensemble d'états qu'elle représente.

4.2.2 Motivations quant au formalisme du programme

Les langages développés pour la vérification de systèmes uniformes distribués sont fondés soit sur des prédicats, soit sur des affectations.

Dans les langages à base de prédicats [DP99], chaque opération est décrite à l'aide d'un prédicat P portant sur les variables d'état du programme $x_1, \ldots, x_n$ et sur leur version primée $x'_1, \ldots, x'_n$. Ce prédicat, nommé *prédicat avant-après* dans[Abr96], décrit la relation entre les états du programme avant et après son exécution.

La seconde famille de langages ([Sif82], SPL [MP92], ALV [YKBB05], [RV03b], ASM [BS03]), utilise un formalisme d'affectations gardées. En constatant qu'une seule affectation, $x_1 := E$, est suffisante pour représenter une conjonction d'égalités de la forme $x'_1 = E \wedge x'_2 = x_2 \wedge \ldots \wedge x'_n = x_n$, on comprend que le premier intérêt d'un langage à base d'affectations est sa compacité. On rejoint en ce sens la remarque de [BS03, p. 293] comparant les prédicats avant-après avec la notion d'affectations gardées portant sur un ensemble de variables (dénommée méthode de "Parnas-table-based") : "*The frame problem enters Parnas-table-based methods through the declarative x/x'-notation for values of variables x in a given state and their value x' in the next state, yielding the overwhelming so-called NC-clauses (No-Change).*"

On remarque ensuite que les langages de prédicats sont plus expressifs que les programmes utiles à modéliser. Par exemple, le prédicat $x'^2 = x^3+1$ est certes une représentation concise de l'opération qui affecte de manière indéterministe à la variable x soit $\sqrt{x^3+1}$ soit $-\sqrt{x^3+1}$; cependant lors d'une propagation avant ou arrière, la résolution d'une telle égalité va se poser. Tandis que l'affectation indéterministe fournit directement les solutions, la forme prédicative nécessite une procédure spécifique.

On note enfin que définir des opérations à l'aide d'affectations gardées permet d'obtenir, pour une des deux propagations, des assertions dont le nombre de quantificateurs, qui ont été introduits par cette propagation, est aussi petit que possible (cf partie 4.3.2).

$$
\begin{array}{rcl}
\mathit{subst} & ::= & \mathsf{skip} \mid \mathit{affec} \mid \mathit{choix} \mid \mathit{pred} \Longrightarrow \mathit{subst} \mid \\
 & & \mathsf{if}\ \mathit{pred}\ \mathsf{then}\ \mathit{subst}\ \mathsf{else}\ \mathit{subst} \\
\mathit{affec} & ::= & x := \mathit{expr} \mid x := \mathit{expr} \parallel \mathit{affec} \\
\mathit{choix} & ::= & \mathit{subst} [] \mathit{subst} \mid (@x \ .\ \mathit{subst})
\end{array}
$$

FIG. 4.2 – Syntaxe des substitutions généralisées.

La partie suivante présente la syntaxe du langage des substitutions généralisées, qui est une extension des langages d'affectations gardées.

4.2.3 Syntaxe des substitutions généralisées

Cette partie définit un langage de substitutions généralisées qui est une restriction du langage des machines abstraites B. Chaque substitution est engendrée par l'élément syntaxique *subst* de la grammaire de la figure 4.2, paramétrée par les éléments syntaxiques *expr* et *pred*. La syntaxe des expressions (*expr*) dépend des structures de données utilisées. Les prédicats *pred* sont exprimés, quant à eux, dans une logique du premier ordre multi-sortes dépendant des structures de données des expressions. Les chapitres 8, 9 et 10 instancient ces éléments syntaxiques.

Détaillons le sens de chacune de ces substitutions. Les substitutions de base sont skip, qui est sans effet, et l'affectation simple (:=), qui modifie la valeur d'une variable x. Toutes les affectations sont supposées bien sortées, c'est à dire que les membres gauche et droit d'une affectation sont de la même sorte. Deux substitutions modifiant des variables différentes peuvent être effectuées simultanément en utilisant l'opérateur associatif et commutatif (||) de composition[5]. Néanmoins, les règles de la figure 4.3 [BP03] appliquées de gauche à droite permettent de réduire cet opérateur de composition de substitutions généralisées à son action sur des affectations simples (un simple renommage permet de garantir la condition de la dernière règle de cette figure). Cet opérateur réduit est donc dit "d'affectation multiple". Toutes les variables d'une affectation multiple doivent être différentes.

L'indéterminisme est modélisé par le choix borné ([]) entre deux substitutions et par un choix non borné (@) permettant de choisir une variable fraîche x servant ensuite de paramètre à la substitution devant être effectuée. En restreignant l'utilisation de cette variable aux éléments syntaxiques de type *expr* qui sont sortés, celle-ci acquiert une sorte implicite liée à un domaine d'interprétation.

Les substitutions peuvent être gardées ($\Longrightarrow$) par un prédicat (élément syntaxique *pred*).

Pour un prédicat P et deux substitutions généralisées S_1 et S_2, la substitution conditionnelle if P else S_1 else S_2 relève du sucre syntaxique : elle peut en effet se réécrire sous la forme $P \Longrightarrow S_1 [] \neg P \Longrightarrow S_2$. Néanmoins le principal inconvénient d'un tel développement est qu'il est exponentiel sur le nombre de if then else imbriqués. On préfère donc conserver cette construction verbeuse, d'autant plus qu'elle apparaît fréquemment dans les spécifications industrielles et qu'elle est gérée nativement dans le prouveur cible haRVey présenté au chapitre 7.

[5]Cette composition parallèle qui s'applique à des substitutions est différente de la composition parallèle de systèmes de transitions qui est présentée au chapitre 3.

$$
\begin{array}{rcl}
S \mid\mid (S' \mid\mid S'') & = & (S \mid\mid S') \mid\mid S'' \\
S \mid\mid S' & = & S' \mid\mid S \\
\mathsf{skip} \mid\mid S & = & S \\
(P \Longrightarrow S) \mid\mid S' & = & P \Longrightarrow (S \mid\mid S') \\
(S \,[]\, S') \mid\mid S'' & = & (S \mid\mid S'') \,[]\, (S' \mid\mid S'') \\
(\mathsf{if}\ P\ \mathsf{then}\ S\ \mathsf{else}\ S') \mid\mid S'' & = & \mathsf{if}\ P\ \mathsf{then}\ (S \mid\mid S'')\ \mathsf{else}\ (S' \mid\mid S'') \\
(@x\ .\ S) \mid\mid S' & = & (@x\ .\ (S \mid\mid S')) \text{ si } x \text{ n'est pas libre dans } S'
\end{array}
$$

où S, S' ,S'' sont des substitutions généralisées, P est un prédicat et x est une variable.

FIG. 4.3 – Réduction de l'opérateur $\mid\mid$.

$$
\begin{array}{rcl}
rel_{x_i := E} & = & \{\big((a_1, \ldots, a_i, \ldots, a_n), (a_1, \ldots, E, \ldots, a_n)\big) \in \Sigma \times \Sigma\} \\
rel_{x_i := E_i \mid\mid \cdots \mid\mid x_k := E_k} & = & \{\big((a_1, \ldots, a_i, \ldots, a_k, \ldots, a_n), (a_1, \ldots, E_i, \ldots, E_k, \ldots, a_n)\big) \\
 & & \quad \in \Sigma \times \Sigma\} \\
rel_{\mathsf{skip}} & = & \text{ est la relation d'identité sur } \Sigma \\
rel_{P \Longrightarrow S} & = & \big(\Sigma_P \times \Sigma\big) \cap rel_S \\
rel_{S \,[]\, T} & = & rel_S \cup rel_T \\
rel_{@x\ .\ S} & = & \bigcup_x rel_S \text{ où } x \text{ apparaît dans les expressions de } rel_S \\
rel_{\mathsf{if}\ P\ \mathsf{then}\ S\ \mathsf{else}\ T} & = & rel_{P \Longrightarrow S \,[]\, \neg P \Longrightarrow T}
\end{array}
$$

FIG. 4.4 – Sémantique opérationnelle des substitutions généralisées.

4.2.4 Sémantique opérationnelle des substitutions généralisées

On définit la sémantique opérationnelle d'une substitution généralisée S en construisant un système de transitions (Σ, rel_S) qui porte sur un vecteur $X = (x_1, \ldots, x_n)$ de variables de domaine $D = D_1 \times \ldots \times D_n$. L'espace Σ des états du programme est donc l'ensemble des valuations possibles de X dans $D_1 \times \ldots \times D_n$.

La relation binaire rel_S est définie pour chaque substitution S dans la figure 4.4. Dans celle-ci, à chaque assertion φ correspond une partie Σ_φ de Σ définie par

$$\Sigma_\varphi = \{\rho \in \Sigma \mid \varphi\rho \text{ est interprétée à vrai}\}$$

La partie suivante présente les transformateurs d'assertions exprimés à partir des substitutions généralisées, qui permettent, entre autre, de générer syntaxiquement les plus faibles et plus forts invariants.

4.3 Transformateurs d'assertions

On présente dans cette partie les transformateurs d'assertions qui permettent de définir les suites $(I_n)_{n\in\mathbb{N}}$ et $(J_n)_{n\in\mathbb{N}}$, cette fois au niveau logique.

La partie 4.3.1 définit les deux transformateurs d'assertions *post* et $\widetilde{pre}$ qui simulent post et $\widetilde{\text{pre}}$ au niveau des assertions et des substitutions. En montrant en 4.3.2 que ces deux opérateurs n'induisent pas la même classe d'obligations de preuve, l'une étant plus facilement traitable que l'autre, on en déduit que l'une des deux propagations est à privilégier par rapport à l'autre dans un contexte de vérification automatique. La partie 4.3.3 présente l'opérateur [] équivalent à $\widetilde{pre}$ mais qui possède des avantages sur celui-ci en terme de quantificateurs.

4.3.1 Transformateurs d'assertions

On commence par définir les opérateurs $\widetilde{pre}$ et *post* dont les comportements respectifs simulent ceux de post et $\widetilde{\text{pre}}$ au niveau des substitutions généralisées.

Définition 22 ([Sif82] Transformateurs d'assertions) Soit X un vecteur de variables d'états de domaine $\Sigma = D_1 \times \ldots \times D_n$, S une substitution généralisée portant sur X et φ une assertion définissant une partie de Σ.

$$post(S,\varphi) = \exists\,\rho' \,.\, (\rho',\rho) \in rel_S \wedge \varphi(\rho') \tag{4.3}$$

$$\widetilde{pre}(S,\varphi) = \forall\,\rho' \,.\, (\rho,\rho') \in rel_S \Rightarrow \varphi(\rho') \tag{4.4}$$

L'assertion $post(S,\varphi)$ caractérise l'ensemble $\text{post}(rel_S, \Sigma_\varphi)$ et, de même, l'assertion $\widetilde{pre}(S,\varphi)$ définit $\widetilde{\text{pre}}(rel_S, \Sigma_\varphi)$. Les transformateurs d'assertion *post* et $\widetilde{pre}$ sont couramment qualifiés respectivement de plus forte postcondition (strongest postcondition) et de plus faible précondition (weakest precondition) [Dij75] dans la mesure où le langage de substitutions généralisées retenu n'introduit que des transitions dont l'exécution se termine. La proposition suivante établit la définition d'invariant à l'aide de ces transformateurs.

Proposition 3 Soit une substitution généralisée S et un état initial I. J est un invariant de S

- si et seulement si

$$I \Rightarrow J \text{ et } post(S,J) \Rightarrow J \tag{4.5}$$

- si et seulement si

$$I \Rightarrow J \text{ et } J \Rightarrow \widetilde{pre}(S,J) \tag{4.6}$$

On donne ci-après une définition de J_m et J_M que l'on exprime cette fois à l'aide des transformateurs d'assertions introduits dans la partie précédente.

Définition 23 (J_M et J_m) Soit X un vecteur de variables, S une substitution généralisée ne modifiant que des variables de X et I, J deux assertions dont les variables libres appartiennent à X et telles que $I \Rightarrow J$.

- J_m est la plus forte assertion, au sens de l'implication, telle que $I \Rightarrow J_m$ et $post(S, J_m) \Rightarrow J_m$.
- J_M est la plus faible assertion, au sens de l'implication, telle que $I \Rightarrow J_M$ et $J_M \Rightarrow \widetilde{pre}(S, J_M)$

Pour construire J_m et J_M, on considère ainsi les suites d'assertions $(I_n)_{n\in\mathbb{N}}$ et $(J_n)_{n\in\mathbb{N}}$ telles que $I_0 = I, I_n = I_{n-1} \vee post(S, I_{n-1})$, $J_0 = J$ et $J_n = J_{n-1} \wedge \widetilde{pre}(S, J_{n-1})$

Pour des raisons similaires à la partie 4.1, pour $n \geqslant 1$, on a

$$I_n \equiv I_{n-1} \vee post^n(S, I) \text{ et } J_n \equiv J_{n-1} \wedge \widetilde{pre}^n(S, J),$$

avec $post^n(S, I) = post(S, post^{n-1}(S, I))$, $\widetilde{pre}^n(S, I) = \widetilde{pre}(S, \widetilde{pre}^{n-1}(S, I))$, $post^1(S, I) = post(S, I)$ et $\widetilde{pre}^1(S, I) = \widetilde{pre}(S, I)$. De même, les limites de $(I_n)_{n\in\mathbb{N}}$ et $(J_n)_{n\in\mathbb{N}}$ sont respectivement notées J_m et J_M lorsqu'elles existent.

Cependant, il n'est pas aisé d'automatiser la construction de ces limites. La proposition suivante corrige ce défaut en donnant une méthode constructive de J_m et de J_M. Elle sera utilisée dans les parties qui suivent.

Proposition 4 (Construction inductive de J_m et J_M) Soit S une substitution sur Σ, I une assertion définissant un ensemble d'états initiaux, et J une assertion définissant une propriété de sûreté. En considérant $(I_n)_{n\in\mathbb{N}}$ et $(J_n)_{n\in\mathbb{N}}$ les suites d'assertions définies comme ci-dessus,

1. J_m est le premier I_i tel que $i \in \mathbb{N}$, $I \Rightarrow I_i$ et $post(S, I_i) \Rightarrow I_i$.
2. J_M est le premier J_i tel que $i \in \mathbb{N}$, $I \Rightarrow J_i$ et $J_i \Rightarrow \widetilde{pre}(S, J_i)$

Pour appliquer la proposition précédente, il est nécessaire de savoir décider l'implication aussi bien dans l'approche par *post* que dans l'approche par $\widetilde{pre}$. La partie suivante compare ces deux approches dans la perspective de décharger toutes les obligations de preuve successives dans un prouveur.

4.3.2 Choisir entre $\widetilde{pre}$ et *post* dans une approche par preuve automatique

Cette partie montre qu'il n'est pas indifférent, selon le critère d'automaticité des preuves, d'utiliser la propagation avant basée sur l'opérateur *post* ou la propagation arrière basée sur $\widetilde{pre}$.

Dans le cas où la substitution S de (4.4) est une affectation gardée, la quantification la plus externe de cette formule s'élimine. Pour illustrer cette partie, considérons $X = \{x, y\}$ l'ensemble des variables d'état d'un programme et l'affectation gardée

$$P(x, y) \Longrightarrow x := E(x, y).$$

L'instance correspondante de (4.4) est

$$\forall\, x', y' \,.\, \big(((x, y), (x', y')) \in rel_{P(x,y)\Longrightarrow x:=E(x,y)}\big) \Rightarrow \varphi(x', y')$$

soit, d'après la définition de *rel*,

$$\forall\, x', y' \,.\, \big(((x, y), (x', y')) \in \{((x, y), (x', y')) \mid P(x, y) \wedge x' = E(x, y) \wedge y' = y\}\big) \Rightarrow \varphi(x', y')$$

soit,

$$\forall\, x', y' \,.\, \big(P(x, y) \wedge x' = E(x, y) \wedge y' = y\big) \Rightarrow \varphi(x', y')$$

qui se simplifie successivement en

$$\begin{aligned} & P(x, y) \Rightarrow (\varphi(x', y')(E(x, y)/x')(y/y')) \text{ (suppression de } \forall) \\ \equiv\ & P(x, y) \Rightarrow \varphi(E(x, y), y) \end{aligned} \tag{4.7}$$

$$
\begin{array}{rcl}
[P \Longrightarrow S]O & = & P \Rightarrow [S]O \\
[\mathsf{skip}]O & = & O \\
[x := Exp]O & = & O(Exp/x) \\
[affec]\ O & = & O(z_2/y_2)\ldots(z_n/y_n)(E_1/y_1) \\
 & & \quad (E_2/z_2)\ldots(E_n/z_n) \\
[S\ []\ S']O & = & [S]O \wedge [S']O \\
[\mathsf{if}\ P\ \mathsf{then}\ S\ \mathsf{else}\ S']O & = & ite(P, [S]O, [S']O) \\
[(@x\ .\ S)]O & = & (\forall\, p\ .\ [S]O(p/x)) \text{ où } p \notin \mathcal{V}_{lib}(O)
\end{array}
$$

où S, S' sont des substitutions généralisées (définies à la figure 4.2), P et O sont des prédicats, x est une variable, *affec* dénote l'affectation parallèle $y_1 := E_1 \mid\mid \cdots \mid\mid y_n := E_n$ où les variables $y_1, \ldots, y_n$ sont des variables distinctes deux à deux et $z_1, \ldots, z_n$ sont des variables distinctes fraîches.

FIG. 4.5 – Opérateur [].

qui ne contient plus de quantificateur.

En instanciant la formule (4.3) avec la même substitution, on obtient une formule qui, de manière similaire, se simplifie en

$$\exists\, x' \,.\, P(x', y) \wedge x = E(x', y) \wedge \varphi(x', y)$$

et pour laquelle la quantification ne se supprime pas. Ainsi, lorsque les variables d'état sont ensemblistes, même si φ est une formule logique du premier ordre, l'opérateur *post* engendre des formules du second ordre dans le cas d'affectations gardées.

Cependant, en terme de gestion des quantificateurs, l'utilisation du transformateur *post* convient dans certains cas. Pour une substitution S et un ensemble d'états initiaux I, lorsqu'il s'agit uniquement de vérifier qu'une propriété J fournie est invariante ou non (formule (4.5) de la proposition 3) l'obligation de preuve contenant l'opérateur *post* est en effet $post(S, J) \Rightarrow J$ et le quantificateur existentiel introduit par *post* se supprime par skolémisation. Il en est de même lors de la première vérification de l'item 1 de la proposition 4 où l'obligation de preuve est dans ce cas $post(S, I) \Rightarrow I$. Cependant, lorsque ces formules ne sont pas valides et que la méthode proposée ici suggère de calculer I_1, cette technique de skolémisation ne suffit plus pour supprimer les quantificateurs qui se situent aussi à droite du symbole d'implication.

4.3.3 De $\widetilde{pre}$ à l'opérateur []

La partie précédente a montré que, dans le cas d'une affectation gardée, l'équation (4.4) est équivalente à une formule dont la quantification la plus externe est remplacée par une substitution (cf. équation (4.7)). Cette démarche de suppression de quantificateur se généralise à toutes les substitutions généralisées de la figure 4.2. La preuve se fait par induction sur la structure de la substitution.

En nommant [] le calcul qui compose $\widetilde{pre}$ avec cette élimination de quantificateur externe,

on obtient les règles de calcul présentées à la figure 4.5. On remarque que ce calcul exploite l'opérateur logique *ite* qui rend plus compact le cas des substitutions conditionnelles.

Proposition 5 (Construction inductive de J_M avec []) Soit S une substitution, I une assertion définissant un ensemble d'états initiaux et J une assertion définissant une propriété de sûreté. Soit $(J_n)_{n\in\mathbb{N}}$ la suite d'assertions dont le terme général est défini par $J_0 = J$ et $J_{n+1} = J_n \wedge [S]^n J$. Alors J_M est le premier J_i tel que $i \in \mathbb{N}$, $I \Rightarrow J_i$ et $J_i \Rightarrow [S]J_i$

Remarques

1. Comme $\widetilde{pre}$, l'opérateur [] est distributif vis à vis des conjonctions. On a formellement $[S](\varphi_1 \wedge \varphi_2) = ([S]\varphi_1) \wedge ([S]\varphi_2)$ pour toute substitution S et toutes les assertions φ_1 et φ_2.
2. La sémantique rel_S d'une substitution généralisée S s'exprime aussi à l'aide du prédicat $\neg[S]\neg(X = X')$ qui définit la relation entre l'ensemble X des variables d'état avant l'opération et l'ensemble X' des variables d'état après l'opération.
3. Dans une substitution de choix non borné ($@x \,.\, S$), le calcul de $[S]$ introduit un quantificateur universel. Cependant, dans le cas de systèmes uniformes distribués, cette substitution est classiquement utilisée pour paramétrer la substitution par l'élément qui initie (ou intervient dans) l'action (cf chapitre suivant). La quantification qui en résulte porte donc sur un élément et non sur un ensemble. On demeure dans la logique du premier ordre.

La partie suivante détaille des semi-algorithmes de construction de plus faible invariant J_M exploitant la proposition 5.

4.4 Semi-algorithmes de construction du plus faible invariant

Cette partie montre comment itérer le calcul de plus faible invariant J_M en exploitant la proposition 5. A des fins pédagogiques, un premier semi-algorithme est présenté en partie 4.4.1. La partie 4.4.2 en présente une version affaiblie qui est suffisante pour vérifier que des propriétés de sûreté sont violées. La dernière partie en présente enfin des optimisations successives permettant d'envisager raisonnablement une implantation efficace.

4.4.1 Construction naïve d'invariant

Le semi-algorithme visant à construire l'invariant J_M de manière itérative est donné à la figure 4.6, une *itération* étant un cycle qui commence et se termine dans l'état Updated et qui passe une seule fois par les transitions compute et update.

A chaque itération, les deux conditions de la proposition 5 sont confiées à une procédure de décision externe représentée par un hexagone et J_i est mis à jour comme détaillé dans cette même proposition. Ce premier semi-algorithme s'arrête dans deux cas :

- si l'état initial n'est plus inclus dans J_i ; dans ce cas J n'est pas une propriété de sûreté.
- si I est inclus dans J_i et J_i est inclus dans ses prédécesseurs qui ne mènent que dans lui ; la propriété de sûreté J est alors vérifiée et J_i est un invariant du programme. On reconnaît là une détection de point fixe.

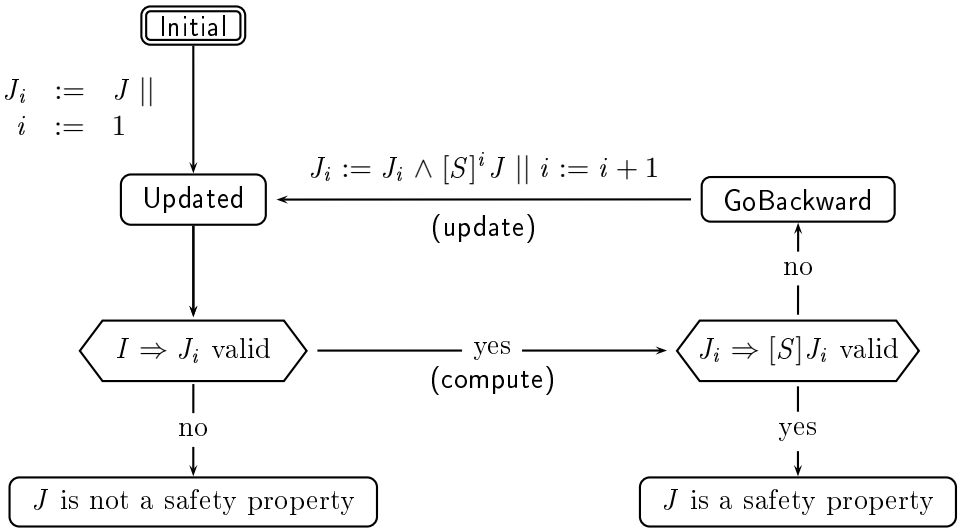


FIG. 4.6 – Construction naïve du plus grand invariant.

On note que cette construction itérative n'est qu'un semi-algorithme pour deux raisons. La première est la *divergence* potentielle du calcul du prédicat J_i construit au cours des itérations de la transition update, qui correspond à l'indécidabilité de la vérification d'invariance pour les systèmes paramétrés généraux [AK86]. La seconde est la divergence de la procédure externe devant vérifier la validité des deux conditions d'évolution.

La partie suivante s'intéresse à ce second cas et montre qu'en affaiblissant ce semi-algorithme on obtient une démarche permettant de détecter qu'une propriété de sûreté est violée.

4.4.2 Montrer qu'une propriété de sûreté est violée

Lorsqu'on pressent qu'une propriété de sûreté J est violée, il apparaît judicieux de simplifier le semi-algorithme précédent en supprimant la vérification de la condition $J_i \Rightarrow [S]J_i$. Le semi-algorithme résultant, présenté à la figure 4.7, s'arrête dès que $I \Rightarrow J_i$ n'est plus valide, ce qui signifie qu'au moins un élément de I n'est pas dans J_i c'est à dire qu'au moins un successeur d'un élément de I, ou un élément de I n'appartient pas à J.

Ce semi-algorithme possède l'avantage de ne pas diverger au cours d'une vérification (inutile) de $J_i \Rightarrow [S]J_i$. De plus, il ne génère qu'une seule famille de formules à décharger dans un prouveur, ce qui, en terme de procédures de décision, a des conséquences intéressantes qui seront détaillées au chapitre 10.

La partie suivante présente des raffinements successifs du semi-algorithme de la figure 4.6, dont l'objectif principal est de simplifier les obligations de preuve à décharger dans le prouveur.

4.4.3 Quelques optimisations

Conçu naïvement, le semi-algorithme de la figure 4.6 est perfectible en ce qui concerne les obligations de preuve fournies au prouveur, qui peuvent être aisément simplifiées. Cette partie montre comment réduire les formules fournies au prouveur, tout en donnant un sens à ces réductions.

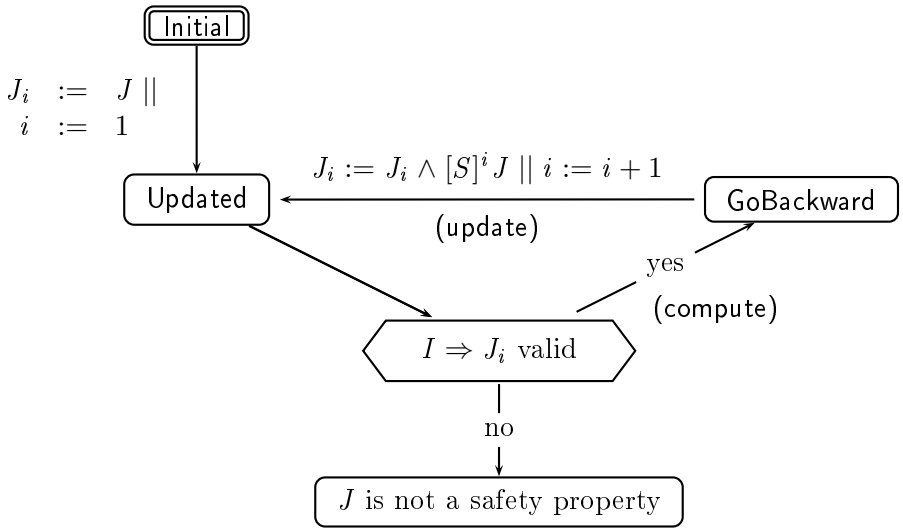


FIG. 4.7 – Vérification de violation de propriété de sûreté.

On rappelle tout d'abord que, par définition

$$J_i = J_{i-1} \wedge [S]^{i-1} J = J \wedge [S]J \wedge \ldots [S]^{i-1} J.$$

Ainsi, pour vérifier que J_i est un invariant à la $i^{ème}$ itération, en plus de vérifier la condition sur l'état initial, il suffit de vérifier que J_i est inclus dans ses nouveaux prédécesseurs qui ne mènent qu'à lui et qui sont décrits par $[S]^i J$. Cette intuition se retrouve au niveau logique en constatant que la formule $J_i \Rightarrow [S]J_i$ se réécrit comme

$$(J \wedge [S]J \wedge \ldots \wedge [S]^{i-1} J) \Rightarrow [S](J \wedge [S]J \wedge \ldots \wedge [S]^{i-1} J),$$

en développant J_i. Cette équation est équivalente à $J_i \Rightarrow [S]^i J$ d'après la propriété de distributivité de $[\,]$ par rapport aux conjonctions et par simplification logique.

Supposons maintenant que la substitution S soit décrite comme un choix borné entre n substitutions S_1, ..., S_n qui modélisent un programme défini par n opérations. Lors de la transition update, dans l'assertion

$$[S]^i J = [S_1][S]^{i-1} J \wedge \ldots \wedge [S_n][S]^{i-1} J = \bigwedge_{1 \leqslant \alpha_1, \ldots, \alpha_i \leqslant n} [S_{\alpha_1}] \ldots [S_{\alpha_i}] J \tag{4.8}$$

qui est ajoutée à J_i et qui est écrite sous la forme d'une conjonction d'assertions, seules sont intéressantes celles qui ne se déduisent pas de J_i. En effet, celles qui s'en déduisent ne le renforcent pas. Pour mettre en place cette optimisation, il suffit de savoir détecter cette implication, ce qui est réalisable lors de la détection $J_i \Rightarrow [S]^i J$ du point fixe, pour peu qu'on la réécrive en

$$J_i \Rightarrow \bigwedge_{1 \leqslant \alpha_1, \ldots, \alpha_i \leqslant n} [S_{\alpha_1}] \ldots [S_{\alpha_i}] J. \tag{4.9}$$

Ainsi lors de la transition update, il suffit de n'ajouter à J_i –sous la forme d'une conjonction– que les assertions $[S_{\alpha_1}] \ldots [S_{\alpha_i}] J$ qui ne vérifient pas l'équation (4.9). Dans la figure 4.8 qui

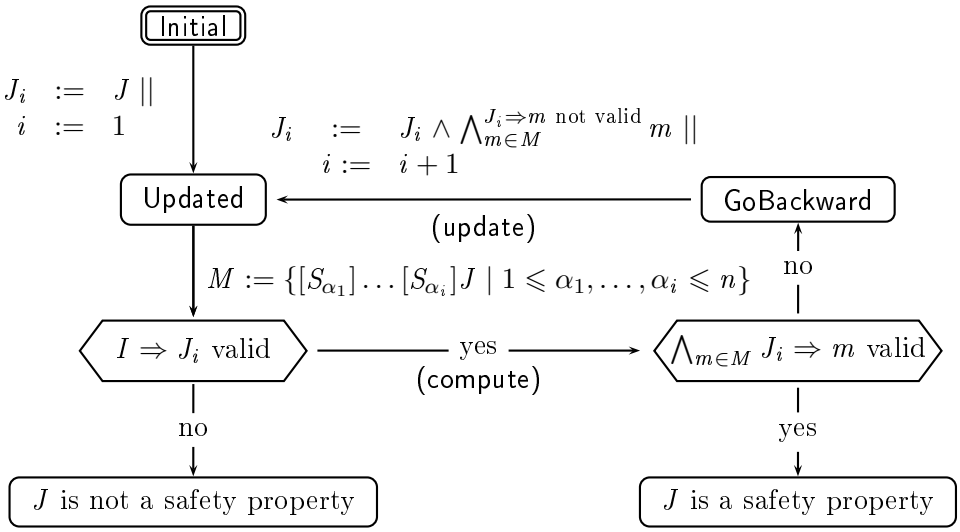


FIG. 4.8 – Construction d'invariant par réduction de J_i, de la condition d'inclusion et par introduction du choix borné.

synthétise ces premières optimisations, les ensembles des prédeceseurs de J de la $i^{ème}$ itération sont calculés (par la transition compute) en écrivant la substitution comme un choix borné et sont mémorisés dans l'ensemble M (pour prédécésseurs qui ne mènent que dans J). La détection du point fixe est une réécriture directe de l'équation (4.9) en tenant compte de M. La mise à jour de J_i s'en déduit immédiatement.

Ensuite, le calcul de l'ensemble $[S]^i J$ des $i^{ème}$ prédécesseurs ne menant que dans J, calcul effectué lors de la transition compute de la figure 4.8, peut être optimisé en exploitant le calcul de l'itération précédente comme présenté de manière inductive dans l'équation (4.8). A nouveau, parmi les assertions $[S_{\alpha_1}] \ldots [S_{\alpha_i}] J$, seules nous intéressent celles qui ne se déduisent pas de J_i. Cette optimisation est mise en œuvre en mémorisant, à chaque itération dans un ensemble N (pour nouveaux prédécesseurs), les prédécesseurs d'un élément de M qui ne se déduisent pas de J_i. On remarque, de plus, que cette optimisation s'adapte bien au fait que la propriété de sûreté exprime généralement une combinaison de contraintes, qui se traduit ainsi en une conjonction d'assertions. Dans ce cas, en initialisant N comme l'ensemble de ces assertions, le calcul des prédécesseurs, qui se fait par rapport à chacune d'elle, engendre des formules de taille plus réduite et donc plus facilement traitables automatiquement.

La figure 4.9 présente ce semi-algorithme, dénommé "calcul local des prédécesseurs", par opposition au calcul global des prédécesseurs effectués dans les semi-algorithmes précédents. Dans celui-ci, à partir de J, exprimée sous la forme d'une conjonction d'assertions, la fonction *ass* engendre l'ensemble de ces assertions.

Enfin, dans le semi-algorithme de la figure 4.9, la détection de point fixe est réalisée en vérifiant la validité de $J_i \Rightarrow m$ pour chaque prédécesseur $m \in M$. En nommant $\mathcal{J}$ l'ensemble des assertions composant J_i, la vérification de cette formule est équivalente à la vérification de la validité de $\bigvee_{j \in \mathcal{J}} (j \Rightarrow m)$, qui peut être déchargée en autant d'appels au prouveur que d'éléments dans $\mathcal{J}$. Implantant cette variante, le semi-algorithme de la figure 4.10 mémorise l'invariant J_i sous la forme d'un ensemble $\mathcal{J}$ d'assertions, plutôt que sous la forme d'une

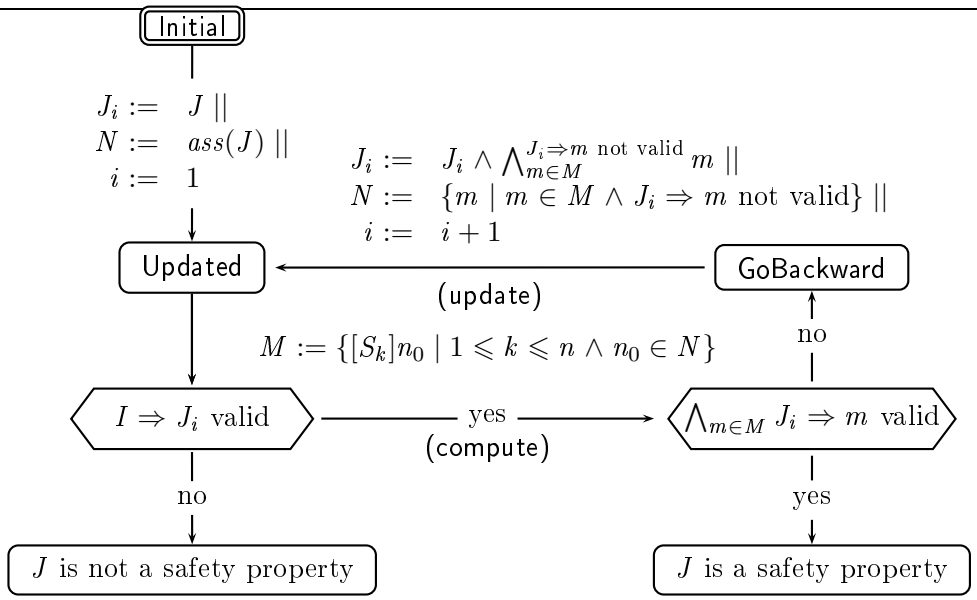


FIG. 4.9 – Construction d'invariant par calcul local des prédécesseurs.

conjonction. Comme N, $\mathcal{J}$ est initialisé à l'aide de la fonction *ass* vue dans l'optimisation précédente.

Ce découpage en vérification locale de d'implication est inspiré de [Sri93] qui montre, dans le cadre de contraintes arithmétiques linéaires, que la vérification de l'implication classique est co-NP tandis que la vérification locale est polynomiale.

4.5 Résumé

Ce chapitre a rappelé comment vérifier des propriétés de sûreté en construisant des invariants par propagation avant ou arrière. Cette construction se fait de manière itérative en appliquant un transformateur d'assertions. Nous avons alors montré que le langage des substitutions généralisées muni de l'opérateur [] allie une syntaxe compacte à un transformateur d'assertions intégrant nativement une technique d'élimination de quantificateurs.

Du plus naïf au plus raffiné, différents semi-algorithmes itérant ce calcul d'invariant et déchargeant les obligations de preuve dans un prouveur adapté ont été présentés. Successivement, chacun des algorithmes fournit des obligations de preuve plus réduites que la version qu'il raffine. La divergence de ces semi-algorithmes sera étudiée au chapitre 10 dans un cadre particulier de langage de prédicats et d'expressions.

La partie suivante présente la méthode B comme une démarche outillée permettant de vérifier des invariants fournis par le spécifieur pour des programmes exprimés à l'aide de substitutions généralisées.

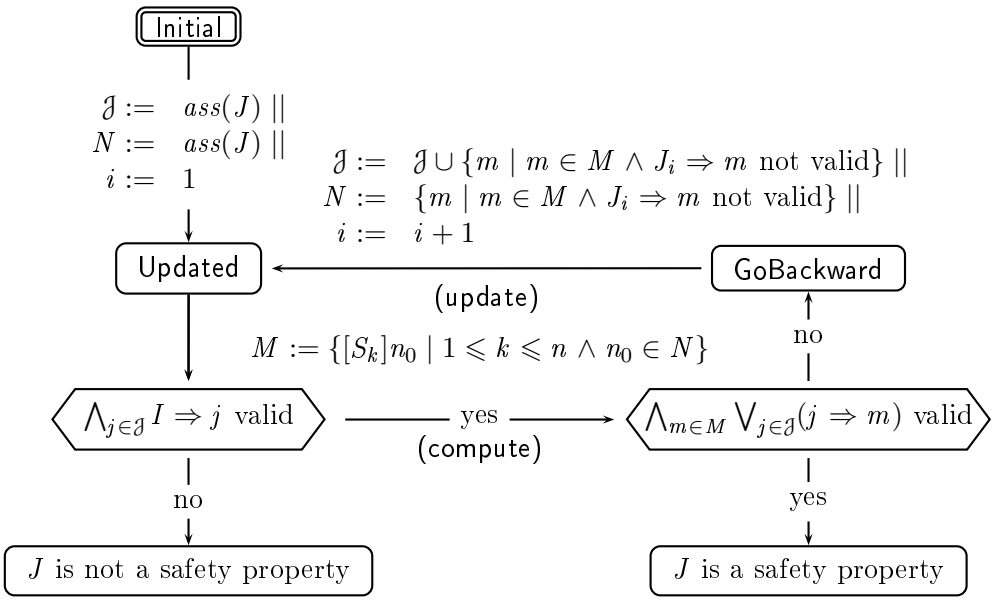


FIG. 4.10 – Optimisation par vérification locale de l'implication.

Chapitre 5

La méthode B

Le chapitre précédent a présenté l'intérêt du langage des substitutions généralisées dans le cadre de vérification de propriétés de sûreté par construction d'invariants. Ce langage est en fait extrait du langage des machines abstraites B, s'intégrant dans le contexte de la méthode B et mobilisant une grande communauté de chercheurs.

Ce chapitre vise à présenter cette méthode et à montrer l'applicabilité du langage des machines abstraites B à la vérification de systèmes uniformes distribués. Plus précisément, la partie 5.1 décrit la forme générale d'une machine abstraite B dont la syntaxe est donnée en partie 5.2. La partie 5.3 présente en détail les obligations de preuve de cohérence de machine B et établit le lien avec les OPs d'invariance vues au chapitre précédent. La partie 5.4 montre comment traduire un système uniforme distribué en une machine abstraite B prête à être vérifiée par un outil, dont les principaux sont présentés en partie 5.5.

5.1 Ressorts de la méthode

Conçue par J.-R. Abrial, la méthode formelle B [Abr96, Lan96] couvre les étapes du développement d'un logiciel, de la spécification abstraite jusqu'à la génération de code dans un langage de programmation classique, par étapes successives dites de "raffinement". Les principaux ressorts de la méthode B sont :

- La *machine abstraite (MA)*. C'est la spécification à la base de toute la démarche B.
- Les *substitutions généralisées*. Elles permettent de décrire toute transition de la machine à l'aide d'un langage dont un fragment a déjà été présenté au chapitre précédent. Par rapport à ce dernier, quelques différences syntaxiques existent : le langage B accepte notamment la notation $x_1, x_2 := E_1, E_2$ avec la sémantique de l'affectation multiple ($||$) et `ANY` x `WHERE` P `THEN` S `END` est une notation verbeuse pour $(@x \,.\, P \Rightarrow S)$.
- Les *composants*. Réaliser un programme complexe sans faute en un seule étape est une tâche difficile. La notation des machines abstraites B prévoit des clauses permettant de développer le programme en assemblant des composants décrits aussi à l'aide de machines. Ils ne seront pas étudiés plus en détail par la suite.
- Le *raffinement*. Le raffinement consiste à enlever de l'indéterminisme dans la machine sans que l'utilisateur de celle-ci ne s'en rende compte : vue de l'extérieur, la machine a les mêmes préconditions et les mêmes postconditions. Seules les données et les opérations, devenues plus concrètes, plus précises, changent. Les étapes de raffinement permettent d'introduire progressivement les détails de conception qui n'étaient pas pris en compte

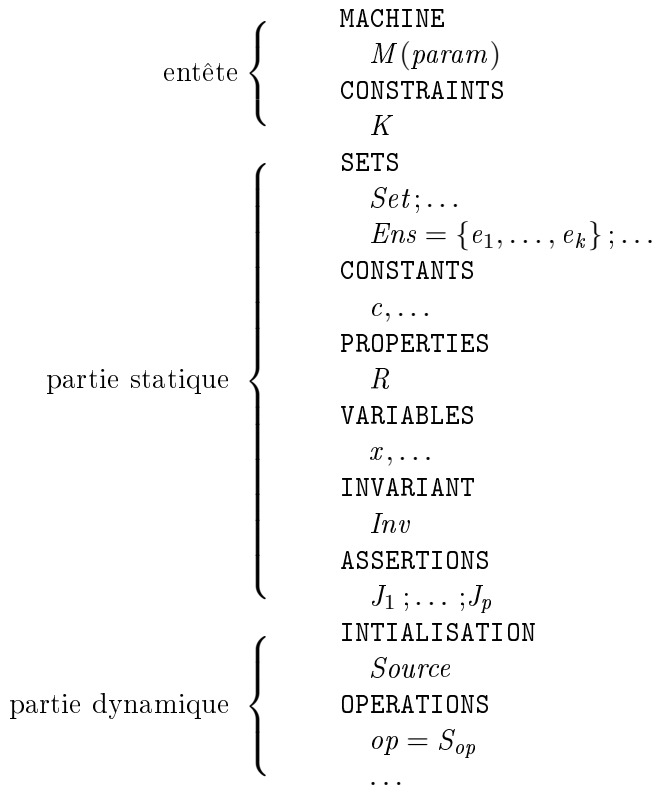


FIG. 5.1 – Forme générale d'une machine abstraite B.

auparavant.

- Les *obligations de preuve (OP)*. Construites à chaque étape du développement B, les OPs correspondent aux conditions à vérifier pour que le système possède les propriétés attendues. On distingue deux familles d'OPs : celles qui vérifient que l'invariant proposé en est bien un, auquel cas la machine est dite cohérente, et celles qui montrent la correction du raffinement. Dans les deux cas, les OPs sont construites automatiquement par un générateur d'OPs.
- Les *prouveurs*. Leur rôle est de décharger les OPs générées précédemment. Interactifs dans la majorité des cas, ils nécessitent l'intervention d'un expert pour être aiguillés vers une solution.

5.2 Syntaxe d'une machine abstraite B

La figure 5.1 décrit la forme générale d'une machine abstraite (MA) B. Elle peut se partager en trois parties appelées *entête*, partie *statique* et partie *dynamique*.

L'entête contient nécessairement la clause `MACHINE` et éventuellement la clause `CONSTRAINTS`.

La clause `MACHINE` fixe le nom de la machine. Ce nom est éventuellement suivi, entre parenthèses, de la liste des paramètres de la machine, qui sont soit des scalaires (écrits en minuscules), soit des ensembles finis, non vides et deux à deux disjoints (écrits en majuscules).

La clause `CONSTRAINTS`, le cas échéant, exprime sous la forme d'un prédicat K les contraintes sur ces paramètres.

La partie statique contient (éventuellement) les clauses `SETS`, `CONSTANTS`, `PROPERTIES`, `VARIABLES`, `INVARIANT` et `ASSERTIONS`.

La clause `SETS` définit les ensembles utiles pour le typage des éléments de la spécification. Ces ensembles sont caractérisés soit par un identifiant ; l'ensemble est alors fini, non vide et nom énuméré ; il est qualifié d'*abstrait* ; soit par une égalité entre un identifiant et un ensemble énuméré de constantes. Les éléments d'un ensemble énuméré sont toujours considérés comme distincts. On traduit cette condition par le prédicat

$$B =_{def} \bigwedge_{Ens} \bigwedge_{e_i, e_j \in Ens}^{1 \leq i < j \leq k} e_i \neq e_j$$

qui précise que, pour chaque ensemble *Ens* défini dans cette clause par $Ens = \{e_1, \ldots e_k\}$, les éléments constitutifs de *Ens* sont deux à deux distincts.

La clause `CONSTANTS` définit une collection d'identifiants représentant des constantes, c'est à dire des données dont la valeur ne peut être modifiée.

La clause `PROPERTIES` est composée d'une conjonction R de prédicats définissant les constantes. Cette conjonction est incluse dans toutes les OPs.

La clause `VARIABLES` contient la collection de variables d'état de la machine : ce sont les variables qui pourront être modifiées par les opérations.

La clause `INVARIANT` contient un prédicat *Inv* qui définit les propriétés sur ces variables qui doivent être établies initialement puis préservées par l'exécution de chaque opération.

La clause `ASSERTIONS` est composée d'une séquence de prédicats $J_1 ; \ldots ; J_p$. Chaque prédicat J_i, $1 \leqslant i \leqslant p$, est un lemme permettant de décharger plus facilement les preuves de cohérence. Il est donc ajouté dans la partie des hypothèses des obligations de preuve de cohérence. Chaque prédicat J_i, $1 \leqslant i \leqslant p$, doit être prouvé comme une conséquence de tous les prédicats nommés ci-avant, à savoir : les distinctions entre éléments des ensembles énumérés (prédicat B), la contrainte K de la clause `CONSTRAINTS`, les propriétés R de la clause `PROPERTIES`, le prédicat invariant *Inv* et les assertions $J_1, \ldots, J_{i-1}$ qui le précèdent.

La partie dynamique contient les clauses `INITIALISATION` et `OPERATIONS`. La première est une opération, définie à l'aide d'une substitution généralisée *Source*, qui attribue à chacune des variables d'état une valeur initiale. Celle-ci est déterminée par une expression dont les variables libres n'appartiennent pas à la clause `VARIABLES`. La clause `OPERATIONS` contient n opérations, $n \in \mathbb{N}$, définies à l'aide de substitutions généralisées qui modifient les valeurs des variables d'état. Chaque opération de cette clause porte un nom *op* ; on désigne par S_{op} la substitution généralisée la définissant.

5.3 Vérification de cohérence

Exprimé en logique du premier ordre multi-sortes, le prédicat *Inv* de la clause `INVARIANT` doit être inductif : les propriétés sur les variables d'état doivent (*i*) être établies par la substitution généralisée *Source* de la clause `INITIALISATION` et (*ii*) être préservées pour chaque substitution généralisée S de la clause `OPERATIONS`.

Formellement, le spécifieur doit vérifier la validité des formules

$$B \wedge K \wedge R \Rightarrow [Source]Inv \tag{5.1}$$

$$B \wedge K \wedge R \wedge J_1 \wedge \ldots \wedge J_n \wedge Inv \Rightarrow [S]Inv \tag{5.2}$$

qui correspondent respectivement aux cas (i) et (ii). Les obligations de preuve relatives à la correction des assertions $J_1, \ldots, J_n$ étant de la forme

$$B \wedge K \wedge R \wedge Inv \Rightarrow J_1 \text{ et}$$
$$B \wedge K \wedge R \wedge Inv \wedge J_1 \wedge ... \wedge J_{i-1} \Rightarrow J_i, \text{ pour } 2 \leqslant i \leqslant n,$$

le prédicat $J_1 \wedge \ldots \wedge J_n$ est bien une conséquence de $B \wedge K \wedge R \wedge Inv$ et peut ainsi être supprimé de l'obligation de preuve (5.3) sans en changer la valeur de vérité. Par souci de concision, cette simplification sera retenue dans la suite de ce mémoire pour donner

$$B \wedge K \wedge R \wedge Inv \Rightarrow [S]Inv. \tag{5.3}$$

Comparons maintenant les obligations de preuve (5.1) et (5.3) avec celles de l'équation (4.6) du chapitre précédent. D'une manière générale, on note que la démarche B n'étant pas dédiée à une structure particulière de données ni à une classe particulière de programmes paramétrés, les obligations de preuve intègrent explicitement ces particularités au moyen des prédicats B, K et R dans leurs hypothèses. Ces prédicats peuvent aussi être vus comme une théorie propre à chaque programme et donc être intégrés directement dans l'outil de preuve, plutôt qu'apparaître explicitement dans les OPs.

Avec cette considération, la seconde obligation de preuve de (4.6) est équivalente à l'obligation de preuve (5.3).

Intuitivement, on constate que la première OP de (4.6) et (5.1) ont le même sens : dans le premier cas il s'agit de vérifier que l'ensemble I des états initiaux est inclus dans l'invariant Inv ; dans le second, il est nécessaire de s'assurer que la substitution d'initialisation *Source* mène systématiquement dans l'invariant Inv depuis tout état.

5.4 Spécification B de systèmes uniformes distribués

Cette partie montre comment spécifier, au moyen de machines abstraites B ensemblistes, les systèmes uniformes distribués présentés au chapitre 3 et définis comme un produit synchronisé de systèmes de transitions spécifiques.

Avec la sémantique d'entrelacement retenue, la mise à jour contradictoire d'une variable partagée par différents composants ne peut survenir. Dans de telles conditions, plutôt que de modéliser chaque composant individuellement, c'est le système global qui est modélisé par une machine. Celle-ci est paramétrée par l'ensemble abstrait $PROC$ des identifiants des processus, qui exécutent tous le même système de transitions. A chaque état e du système de transitions est associée une variable ensembliste E qui mémorise quels processus sont dans cet état. Pour un système de transitions à n états $e_1, \ldots, e_n$, on a n variables ensemblistes $E_1, \ldots, E_n$, deux à deux disjointes et telles que $E_1 \cup \ldots \cup E_n = PROC$.

On détaille maintenant la syntaxe des substitutions généralisées qui simulent au niveau du système global les différentes sortes d'actions des systèmes de broadcast.

Une action locale à un seul processus p qui passe d'un état e_l dans un état e_m sous hypothèse éventuelle d'une garde P se traduit par la substitution

$$\texttt{ANY } p \texttt{ WHERE } p \in PROC \wedge p \in E_l \wedge P \texttt{ THEN } E_l, E_m := E_l \setminus \{p\}, E_m \cup \{p\} \texttt{ END}$$

Au niveau du système global, lorsque P est établie et qu'il existe un processus (quelconque) p qui appartient à E_l, celui-ci est enlevé de E_l pour être inséré dans E_m.

Une action de rendez-vous synchronise une transition d'un processus p de l'état e_l dans e_m, sous hypothèse éventuelle d'une garde P, avec une transition d'un processus p' de l'état e_o dans e_p, sous hypothèse éventuelle d'une garde P'. Lorsque e_l, e_m, e_o et e_p sont deux à deux distincts, l'action se traduit par la substitution

```
ANY p WHERE p ∈ PROC ∧ p ∈ E_l ∧ P THEN
    ANY p' WHERE p' ∈ PROC ∧ p' ∈ E_m ∧ P' THEN
        E_l, E_m := E_l \ {p}, E_m ∪ {p} ||
        E_o, E_p := E_o \ {p'}, E_p ∪ {p'}
    END
END.
```

Les cas où certains des états ne sont pas distincts s'en déduisent naturellement. Par exemple, le cas où deux processus échangent leurs états e_l et e_m au moyen d'un rendez-vous se traduit en

```
ANY p WHERE p ∈ PROC ∧ p ∈ E_l THEN
    ANY p' WHERE p' ∈ PROC ∧ p' ∈ E_m THEN
        E_l, E_m := E_l \ {p} ∪ {p'}, E_m ∪ {p} \ {'p'}
    END
END.
```

L'action de broadcast effectuée par envoi et réception multiple d'un message l est provoquée par un processus qui, sous hypothèse éventuelle d'une garde P et en passant de e_l à e_m impose à tous les autres un déplacement d'un état e_i vers un état $e_i{}'$ puisque, par hypothèse de déterminisme, pour tout état e_i, il existe une unique transition $(e_i, l??, e_i')$. On considère que le système de transitions possède j états, e_1, ..., e_j. On nomme

- $e_{1_1}, \ldots, e_{1_.}$ les états des processus , qui, à réception du message l, vont dans e_1 (c.a.d. tels que la transition $(e_{1_.}, l??, e_1)$ existe),
- ...
- $e_{j_1}, \ldots, e_{j_.}$ les états des processus , qui, à réception du message l, vont en e_j,

où les $e_{1_.}$ (resp. $e_{2_.}$, ... $e_{j_.}$) sont distincts deux à deux. L'action se traduit par la substitution

```
ANY p WHERE p ∈ PROC ∧ p ∈ E_l ∧ P THEN
    E_1 := E_{1_1} ∪ ... ∪ E_{1_.} ||
    ...
    E_l := E_{l_1} ∪ ... ∪ E_{l_.} \ {p} ||
    ...
    E_m := E_{m_1} ∪ ... ∪ E_{m_.} ∪ {p} ||
    ...
    E_j := E_{j_1} ∪ ... ∪ E_{j_.}
END
```

On note que l'ensemble vide est affecté à E lorsque toutes les transitions de broadcast sortent de l'état correspondant à E.

Au niveau du système global, lorsqu'il existe un processus p dans E_l et que P est établie, ce processus est enlevé de E_l pour être ajouté à E_m et, simultanément, les autres ensembles E_i sont formés des ensembles $E_{i_.}$, qui sont les ensembles de processus dans l'état $e_{i_.}$ à partir duquel la réception de l conduit à e_i.

```
MACHINE
  mesiSet(c)
VARIABLES
  m, e, s, i
INITIALISATION
  m, e, s, i := ∅, ∅, ∅, c
INVARIANT
  m ∪ e ∪ s ∪ i = c ∧
  m ∩ e = ∅ ∧ m ∩ s = ∅ ∧ m ∩ i = ∅ ∧
  e ∩ s = ∅ ∧ e ∩ i = ∅ ∧ s ∩ i = ∅ ∧
  card(m) ⩽ 1 ∧ (s ≠ ∅ ⇒ m = ∅)
OPERATIONS
  sendWriteInvalidate =
    ANY p
    WHERE p ∈ c ∧ p ∈ s
    THEN m, e, s, i :=
        ∅, {p}, ∅, i ∪ m ∪ e ∪ (s \ {p})
    END;
  sendRead =
    ANY p
    WHERE p ∈ c ∧ p ∈ i
    THEN m, e, s, i :=
        ∅, ∅, m ∪ e ∪ s ∪ {p}, i \ {p}
    END;
  write =
    ANY p
    WHERE p ∈ c ∧ p ∈ e
    THEN e, m := e \ {p}, m ∪ {p}
    END
END
```

FIG. 5.2 – Spécification B ensembliste du MESI.

Retour à l'exemple MESI. La machine abstraite donnée à la figure 5.2 spécifie le système global du MESI vu comme le produit synchronisé de systèmes de transitions identiques à celui représenté à la figure 3.3.

D'un point de vue syntaxique, le langage B exige que le nom de chaque variable et de chaque paramètre soit composé, au minimum, de deux caractères et que les ensembles abstraits et les paramètres soient en majuscules. Pour des raisons de lisibilité, cette contrainte n'est pas respectée.

Pour prouver l'algorithme quelque soit le nombre de caches pris en compte, ce nombre est abstrait par le paramètre c qui représente un ensemble de taille finie de caches dont la machine spécifie les comportements.

Chaque cache peut être dans un des quatre états M, E, S et I. On définit l'ensemble m (respectivement e, s, i) des caches dans l'état M (respectivement E, S, I), comme dans la partie précédente.

Initialement chaque cache est dans l'état I, ce qui revient à initialiser les ensembles m, e, s à l'ensemble vide et i à c dans la clause `INITIALISATION`.

La clause `INVARIANT` exprime les contraintes suivantes :

- chaque cache est dans un des états M, E, S, I ($m \cup e \cup s \cup i = c$) mais ne peut être simultanément dans deux états différents ; les ensembles qui les représentent sont donc deux à deux disjoints ($m \cap e = \emptyset$, ...) ;
- il y a au plus un cache dans l'état M ($card(m) \leqslant 1$) ;
- il ne peut y avoir simultanément un cache dans l'état S tandis qu'un autre est dans l'état M ($s \neq \emptyset \Rightarrow m = \emptyset$).

Ces deux dernières formules sont naturellement des instances du patron de formules correspondant aux propriétés de cohérence de cache (cf. équation (3.2) et (3.3)).

En exécutant l'opération sendWriteInvalidate gardée par l'existence d'un cache p dans l'état S, tous les caches se retrouvent dans l'état I sauf p qui passe dans l'état E. L'opération sendRead gardée par l'existence d'un cache p dans l'état I fait passer dans l'état S le cache p et les autres caches dans les états E ou M. La dernière opération précise qu'un cache dans l'état E peut passer sans contrainte dans l'état M.

Soit

$$\begin{array}{lll} Inv(m,e,s,i,c) & =_{def} & m \cup e \cup s \cup i = c \wedge \\ & & m \cap e = \emptyset \wedge m \cap s = \emptyset \wedge m \cap i = \emptyset \wedge \\ & & e \cap s = \emptyset \wedge e \cap i = \emptyset \wedge s \cap i = \emptyset \wedge \\ & & card(m) \leqslant 1 \wedge (s \neq \emptyset \Rightarrow m = \emptyset) \end{array}$$

le prédicat de la clause `INVARIANT`.

A titre d'exemple, l'obligation de preuve relative à l'opération write est

$$\begin{array}{ll} & Inv(m,e,s,i,c) \Rightarrow [S_{\mathsf{write}}]Inv(m,e,s,i,c) \\ = & Inv(m,e,s,i,c) \Rightarrow (\forall\, p \,.\, p \in c \wedge p \in e \Rightarrow Inv(m \cup \{p\}, e \setminus \{p\}, s, i, c)). \end{array} \tag{5.4}$$

Pour montrer qu'elle n'est pas valide, on construit l'interprétation $\mathcal{I}$ classique sur les symboles prédicatifs et fonctionnels ensemblistes telle que les ensembles sont interprétés sur $\mathbb{P}(\{a,b\})$ et les éléments sur $\{a,b\}$ où a et b sont deux éléments distincts.

On calcule la valeur de vérité de la partie gauche et de la partie droite de l'implication de (5.4) pour $m = \emptyset$, $e = \{a\}$, $s = \{b\}$, $i = \emptyset$ et $c = \{a,b\}$. Il est aisé de vérifier *manuellement* que la partie gauche

$$\begin{array}{ll} & Inv(m,e,s,i,c)(\emptyset/m)(\{a\}/e)(\{b\}/s)(\emptyset/i)(\{a,b\}/c) \\ = & Inv(\emptyset,\{a\},\{b\},\emptyset,\{a,b\}) \\ = & \emptyset \cup \{a\} \cup \{b\} \cup \emptyset = \{a,b\} \wedge \\ & \emptyset \cap \{a\} = \emptyset \wedge \emptyset \cap \{b\} = \emptyset \wedge \emptyset \cap \emptyset = \emptyset \wedge \\ & \{a\} \cap \{b\} = \emptyset \wedge \{a\} \cap \emptyset = \emptyset \wedge \{b\} \cap \emptyset = \emptyset \wedge \\ & card(\emptyset) \leqslant 1 \wedge (\{b\} \neq \emptyset \Rightarrow \emptyset = \emptyset) \end{array}$$

est interprétée à vrai ($\top$) tandis que la partie droite

$$\begin{array}{ll} & \big(\forall\, p \,.\, p \in c \wedge p \in e \Rightarrow Inv(m \cup \{p\}, e \setminus \{p\}, s, i, c)\big)(\emptyset/m)(\{a\}/e)(\{b\}/s)(\emptyset/i)(\{a,b\}/c) \\ = & \forall\, p \,.\, p \in \{a,b\}, \wedge\, p \in \{a\} \Rightarrow \\ & \quad \emptyset \cup \{p\} \cup \{a\} \cup \{b\} \cup \emptyset = \{a,b\} \wedge \ldots \wedge \\ & \quad (card(\emptyset \cup \{p\}) \leqslant 1 \wedge (\{b\} \neq \emptyset \Rightarrow \emptyset \cup \{p\} = \emptyset)) \end{array}$$

est interprétée à faux : pour $p = a$ la partie gauche de cette seconde implication est vraie tandis la partie droite ne l'est pas.

La formule (5.4) n'étant pas valide la méthode B suggère à l'utilisateur de modifier sa spécification tandis que la méthode de propagation arrière va renforcer la formule *Inv* pour tenter de la rendre invariante.

5.5 Outils de vérification pour la méthode B

Cette partie présente les différents outils permettant de vérifier la cohérence de machines B.

L'Atelier B [Cle04], de la société ClearSY se compose d'un noyau intégrant un prouveur interactif, basé sur la méthode de déduction des séquents et est couplé avec une interface utilisateur permettant de gérer des projets B, d'éditer les spécifications... Une version académique nommée B4free [Cle] composée d'un noyau bridant le nombre d'obligations de preuve

peut être couplée avec l'interface Click'n Prove [AC03] d'assistance à la preuve interactive. Les obligations de preuve générées correspondent aux formules (5.1) et (5.3).

Le B-toolKit [Wor96] de la société B-core est similaire à l'Atelier B dans le sens où il propose un prouveur dédié et une interface de développement de projet. Il s'appuie sur le prouveur interactif B-platform qui est basé sur des règles de déduction naturelle.

L'outil Pro-B [LB03] se restreint aux MAs dont tous les ensembles sont énumérés et en vérifie la cohérence par un parcours exhaustif de l'ensemble des états du système.

5.6 Résumé

Ce chapitre a présenté les ressorts de la méthode B, la syntaxe de machines abstraites et leur vérification de cohérence.

La syntaxe des machines B, basée sur des langages de prédicats et de substitutions généralisées, permet de spécifier les systèmes uniformes distribués de la classe des protocoles de broadcast en simulant le système dans sa globalité.

En terme d'obligations de preuve, nous avons montré que les OPs de cohérence d'une machine abstraite B sont équivalentes aux OPs de vérification d'invariance vues au chapitre précédent.

Par rapport au chapitre précédent, nous avons montré que la méthode se limite à une seule itération dans la construction d'invariant. Le chapitre 11 présentera comment, pratiquement, nous y avons intégré ces semi-algorithmes de construction.

D'autres démarches déductives sont présentées au chapitre suivant.

Chapitre 6

Cas particuliers d'approches déductives

Les deux chapitres précédents se sont intéresses aux principes généraux des démarches de vérification d'invariant et aux classes de systèmes paramétrés auxquels on souhaite les appliquer. Dans ce cadre général, la vérification automatique est un problème indécidable [AK86, Suz88]. Néanmoins, de nombreuses recherches ont lieu dans ce domaine avec des objectifs plus restreints. Certaines s'attachent à exhiber *une classe* décidable de systèmes. Cette classe est construite en exploitant la structure du réseau (anneau, bus...), la nature des synchronisations et, le cas échéant, des messages envoyés (rendez-vous, broadcast...), ou la forme des propriétés vérifiées (sûreté, vivacité ...). D'autres présentent *une méthode* qui ne traite pas le problème dans toute sa généralité mais qui, en pratique, permet de vérifier (efficacement) un (grand) nombre de cas issus des classes présentées ci-avant.

État de l'art des méthodes de vérification de systèmes uniformes distribués, ce chapitre montre dans la première partie (partie 6.1) comment certaines recherches ont exploité le caractère numérique des propriétés à vérifier pour se ramener à la vérification de systèmes exprimés à l'aide de contraintes numériques. Il indique dans la partie 6.2 les classes connues pour lesquelles la vérification automatique est décidable et décrit (partie 6.3) un état de l'art des méthodes, à base de déduction, qui vérifient une classe plus large de systèmes uniformes distribués sans pour autant être des procédures de décision. La partie 6.4 met alors le doigt sur les limitations concernant ces méthodes de vérification.

6.1 Abstraction de comptage

Vérifier des propriétés d'exclusion mutuelle et de cohérence de cache revient à s'assurer que le nombre de processus dans un certain état n'est pas supérieur à un cardinal déterminé. Partant de ce principe, de nombreux travaux [EN95, EN96, GZ98, DP99, EFM99, Del00, DP01, PRZ01] ont étudié les systèmes où seul est retenu le nombre de processus dans chaque état. Ainsi, pour chaque état ρ du système, cette *abstraction de comptage*, (de l'anglais "counting abstraction") mémorise dans une variable entière n_ρ le nombre de processus en ρ.

Gribomont et Zenner [GZ98] sont (à notre connaissance) les premiers à avoir montré l'intérêt de l'abstraction de comptage pour une preuve automatique d'un algorithme non trivial d'exclusion mutuelle, en l'occurrence celui de Szymanski [Szy88]. En construisant manuellement l'invariant inductif abstrait et en déchargeant le système dans l'outil CAVEAT [GR95],

la preuve est déchargée automatiquement.

Delzanno [DP99] combine cette approche avec les outils de programmation logique avec contraintes (CLP) (et leurs calculs de point fixe sous-jacents) pour vérifier automatiquement avec l'outil DMC [DP01] des propriétés de sûreté. Le système y est spécifié à l'aide de clauses PROLOG définissant l'état courant en fonction de ses successeurs. La localité du calcul des prédécesseurs (cf figure 4.9) et de la vérification de l'inclusion (cf figure 4.10) est déjà présente dans la démarche de vérification. Par extensions successives, la démarche est appliquée à la vérification de gestionnaires de processus communiquant par envoi et réception de messages [ADMP01], par broadcast [DEP99, DP01], à la famille des protocoles de cohérence de cache [Del00, DP01] et à celle des protocoles de type client-server [BD01].

Orthogonale à la représentation des contraintes arithmétiques sous la forme d'un *polyèdre* comme en CLP (les formules sont représentées en DNF dont chaque conjonction est un polyèdre convexe) la représentation sous forme d'un automate qui accepte les solutions du système a été envisagée [WB95, BB03]. En optimisant le calcul des prédécesseurs, Bultan [BB03] montre que la complexité de la construction de cet automate peut être réduite de l'exponentiel au linéaire et exploite cette optimisation dans l'outil Action Language Verifier [Bul00, BYK01, YKBB05] basé sur la Composite Symbolic Library [YKTB01].

Les bases de Hilbert sont exploitées [RV02b] dans une troisième démarche pour résoudre chaque contrainte arithmétique C écrite comme un système d'inéquations de la forme $L\mathcal{V}+l \leqslant 0$ où L est une matrice à coefficients entiers, $\mathcal{V}$ le vecteur des variables d'état et l est un vecteur d'entiers. Un algorithme calcule alors la base (N, H) d'ensembles de vecteurs, tels que toutes les solutions de C se représentent comme la somme d'un vecteur de N et d'une combinaison linéaire de vecteurs dans H. Basé sur une analyse d'atteignabilité arrière et exploitant un algorithme incrémental – au sens que la base de $C_1 \wedge C_2$ est calculée à partir de C_2 et de la base de C_1 – de calcul de base de Hilbert, l'outil Brain [RV02a] semble être plus rapide [RV02b, RV03b, RV03a] que les outils DMC et Action Language Verifier.

Les cas de décidabilité des démarches basées sur les contraintes arithmétiques, ou plus généralement, les cas où une démarche permet de décider de l'atteignabilité ou non d'un ensemble d'états sont détaillés à la partie suivante.

6.2 Classes décidables

Disposer des deux conditions suivantes rend décidable une démarche de vérification d'invariant : les conditions d'arrêt du calcul du point fixe sont décidables et la convergence du calcul du point fixe est garantie. Dans ce qui suit, chacune des deux conditions est en effet assurée pour chaque méthode.

6.2.1 Token-Ring

Indécidable en général, la vérification de propriétés temporelles quelconques sur des systèmes paramétrés le demeure même lorsque les processus, à états finis, sont organisés en token ring unidirectionnel [Suz88].

Cependant, Emerson et Namjoshi ont établi [EN95] la décidabilité des propriétés exprimées en logique temporelle arborescente [CGB86] sans l'opérateur de successeurs immédiats – et donc la décidabilité des propriétés d'invariance – pour des processus organisés en anneau et qui communiquent en transmettant un jeton, puis plusieurs jetons booléen dans les deux

directions [EK04]. La méthode consiste dans les deux cas à réduire le système paramétré en un système fini pour lequel la terminaison est garantie par énumération.

6.2.2 Broadcast

Esparza, Finkel et Mayr [EFM99] ont montré que les systèmes exprimés à l'aide d'abstraction de comptage de la famille des protocoles de broadcast (c.f. partie 3.1.3). se ramènent par construction d'automates à des systèmes de transitions bien structurés [ACJT96, FS01] dont le calcul des prédécesseurs converge [FS01] et dont les conditions d'évolution sont décidables aussi par automates.

6.2.3 Méthode unificatrice

Maidl [Mai01] unifie les travaux des deux parties précédentes en proposant une méthode qui accepte la synchronisation restreinte des token-rings et la synchronisation des protocoles de broadcast. En utilisant de plus des structures de tableaux indexés par des termes de l'arithmétique de Presburger, Maidl définit un cadre où la satisfaisabilité de chaque condition d'arrêt est décidable et où le calcul du point fixe termine. Aucune implantation n'est néanmoins proposée.

6.2.4 Cas de la programmation logique avec contraintes (CLP)

En exhibant des familles de contraintes compatibles avec les critères de décidabilité de vérification des protocoles de broadcast (cf partie 6.2.2) et en exploitant la possibilité de vérifier localement (et donc avec une complexité moindre) les conditions d'arrêt du calcul du point fixe, Delzanno, Esparza et Podelski [DEP99] montrent que la CLP est une procédure de décision effective et efficace pour la vérification de protocoles de broadcast. Une de ces familles de contraintes est réutilisée par Delzanno et Bultan [BD01] pour montrer que les propriétés de cohérence de cache de l'algorithme de S. German [Ger00], dont certaines synchronisations se réalisent à l'aide de variables partagées, sortant ainsi du cadre strict des protocoles de broadcast, sont décidables en CLP.

6.3 Méthodes sans garantie de convergence

Présentées à la partie précédente, les procédures décidant les propriétés de sûreté des systèmes uniformes distribués ne permettent pas de traiter tous les cas : la figure 3.1 donne en effet un exemple de système qui possède une variable partagée et qui n'entre pas dans ces critères de décidabilité. Cette partie s'intéresse aux démarches de vérification déductive sans garantie théorique de convergence mais qui, en pratique, décident des exemples intéressants.

6.3.1 Instanciation des variables quantifiées

Dans une démarche déductive, savoir décider certaines classes de formules quantifiées est un prérequis dès lors que l'assertion qui décrit l'ensemble des états à ne pas atteindre en contient. L'exclusion mutuelle se traduit par exemple en la non atteignabilité de l'assertion $\exists\, p, q\,.\, p \neq q \wedge p \in \mathit{Crit} \wedge q \in \mathit{Crit}$ quantifiée existentiellement, où p et q sont des processus et *Crit* est la section critique.

Le corollaire 2 (chapitre 2) a montré comment réduire certains problèmes de satisfaisabilité de formules quantifiées en un problème décidable de satisfaisabilité de formule sans quantificateurs. En exprimant les hypothèses de cette proposition au niveau du système global, c.a.d au niveau des gardes des transitions et de l'invariant, [FG03] présente un cadre où la vérification de l'inductivité d'une formule est décidable. Rien n'est néanmoins établi sur la terminaison du calcul du point fixe.

L'instanciation des variables quantifiées est sous-jacente à la vérification par *invariant invisible* [APR+01, PRZ01, ZP04] qui s'intéresse aux propriétés d'exclusion mutuelle s'exprimant sous la forme d'une formule p. Une méthode basée sur l'exploration par model checking d'une instance finie du système permet de générer une formule φ, invisible à l'utilisateur, d'une classe de formules C, telle que $\varphi \Rightarrow p$ par construction. φ correspond à un renforcement de p. La classe C étant bien choisie, il est alors possible d'instancier les variables quantifiées de φ pour décider son inductivité. Les exemples qui ont pu être vérifiés grâce à cette approche sont les protocoles d'exclusion mutuelle MuxSem [PRZ01], Szymanski [Szy88], Bakery [RB86], Peterson [SP89] et de cohérence de cache de S. German [Ger00] et de l'université de l'Illinois [PP84].

Totalement automatique, la démarche dépend néanmoins de la stratégie utilisée pour engendrer φ. Lorsque cette formule est déclarée non inductive par la procédure de décision, se pose alors la question de la validité de p : est-ce p qui n'est pas invariant ou est-ce la méthode qui n'est pas complète ? Cette absence de certitude en cas de réponse négative de la méthode se retrouve aussi dans les démarches à base d'abstraction qui, comme celle-ci, construisent un modèle intermédiaire qu'elles décident ensuite.

6.3.2 Abstraction

Dans une méthode basée sur l'*abstraction* [GS97, BGL03], la preuve d'une propriété du système concret est établie sur un système abstrait obtenu en ne retenant qu'une partie des informations du système concret (l'abstraction de comptage en est un cas particulier). La principale difficulté d'une telle méthode consiste à trouver la relation d'abstraction. On note que même une méthode qui construit automatiquement un système abstrait fini et donc traitable par model checking n'en est pas pour autant une procédure de décision : le système abstrait – qui peut contenir plus de comportements – qui atteint un état non sûr n'induit pas que le système concret n'est pas sûr ; la seule conclusion est qu'il est nécessaire, dans ce cas, de raffiner la relation d'abstraction. Cette partie détaille la démarche d'abstraction de prédicats qui permet de construire le système abstrait par une approche déductive.

L'abstraction de prédicats [GS97, BLO98a, RS99, LB04] est un cas particulier de l'interprétation abstraite [CC77]. La méthode considère un langage $\mathcal{L}$ d'assertions définissant les états concrets, le vecteur $\overline{\varphi} =_{def} (\varphi_1, \ldots, \varphi_l)$ composé de l prédicats de ce langage tels que $\bigvee_{1 \leq i \leq l} \varphi_i$ est valide et le vecteur $\overline{B} =_{def} (B_1, \ldots, B_l)$ composé de l variables booléennes.

Les états du système *abstrait* considéré correspondent à un sous ensemble de l'ensemble des prédicats constructibles à partir des variables $B_1, \ldots, B_l$ et contient ainsi au maximum 2^{2^l} états. Chaque état noté $exp^A(B_1, \ldots, B_l)$ peut s'exprimer sous la forme CNF où chaque disjonction ne contient qu'une seule occurrence de B_i. On pose γ la fonction qui, à tout $exp^A(B_1, \ldots, B_l)$, associe $exp^A(B_1, \ldots, B_l)(\overline{\varphi}/\overline{B}) \in \mathcal{L}$, c'est à dire un ensemble d'éléments concrets. Réciproquement, on pose α qui, à toute formule φ, associe $\bigcap_l \{exp^A(B_1, \ldots, B_l) \mid \varphi \Rightarrow exp^A(B_1, \ldots, B_l)(\overline{\varphi}/\overline{B})\}$, un ensemble d'états abstraits. A partir des fonctions α et γ respectivement d'abstraction et de concrétisation, pour chaque substitution S au niveau abstrait,

la relation de transition $\rightarrow^A$ au niveau abstrait est définie exactement par

$$(exp^A(B_1, \ldots, B_l), \rightarrow^A, \alpha(\gamma(exp^A(B_1, \ldots, B_l))[S]^o))$$

qui nécessite un (grand) nombre fini d'appels à un prouveur.

Les prototypes Invariant Checker [Sai97] puis InVeSt [BLO98b] implantent cette approche en déchargeant leur obligation de preuve dans le prouveur PVS [ORSSC98] et en parcourant le système de transitions abstrait fini à l'aide respectivement des model-checkers ALDEBARAN [FGK+96] et SMV [McM92].

Baukus a montré [BBLS00, BLS01, Bau03] que la logique WS1S dispose d'une expressivité suffisante pour exprimer les assertions des systèmes uniformes distribués considérés dans ce mémoire et construire le système abstrait en déchargeant ses obligations de preuve dans le prouveur MONA [HJJ+96]. L'outil PAX qui implante cette abstraction appelle ensuite les model-checkers SMV et SPIN [Hol97]. Les algorithmes de Szymanski [Szy88], MutEx [BLS01], S. German [Ger00] ont ainsi été vérifiés [Bau03, BLS02] avec cette méthode.

6.3.3 Langages réguliers

Connue sous le nom de *regular model-checking*, cette approche [KMM+97, BJNT00] garantit l'invariant de systèmes uniformes distribués en réduisant le problème en la détection d'appartenance de chaque état atteignable du système à une expression régulière.

Plus formellement, soit un système où chaque processus parcourt un ensemble fini d'états Σ qui est cette fois représenté par un alphabet. Une configuration peut être vue comme un mot de Σ^* qui est l'ensemble des mots constructibles à partir des lettres de Σ et, plus généralement, une configuration *symbolique* est un ensemble de configurations décrit par un langage régulier sur Σ. Pour une propriété de sûreté φ, la démarche construit un langage régulier L_φ définissant la configuration symbolique où $\neg\varphi$ est vraie, c.a.d. où la propriété est violée. L'ensemble des configurations atteignables est l'automate défini par compositions iteratives des automates (nommés transducteurs) reconnaissant les relations de transition du système avec l'automate caractérisant la configuration initiale. A chaque itération, l'appartenance à L_φ de l'expression régulière correspondant à cet automate est vérifiée. Si elle est établie, il existe une configuration qui viole φ. Dans le cas contraire et si l'ensemble des configurations atteignables devient stable, φ est un invariant. Le troisième cas où le calcul des configurations atteignables diverge peut être évité par des techniques d'accélération [AJMd02] ou d'élargissement [BJNT00].

6.4 Discussion

Même si la vérification par abstraction de comptage est extrêmement efficace [DP01, RV03b], sa mise en œuvre dans un processus constructif de programme correct possède deux défauts.

Le premier concerne le verdict de la vérification : comment renseigner le spécifieur en terme de contraintes numériques alors que son programme initial ne contenait que des ensembles d'états ?

Le second concerne la modification de sémantique que l'abstraction de comptage engendre parfois. Prenons, en effet, l'exemple de l'algorithme du MutEx présenté à la partie 3.1.4. La figure 6.1 donne la machine B de l'abstraction de comptage qui lui correspond. Dans cette machine, les variables `sl`, `re`, `ac` mémorisent le nombre de processus dans les états SL, RE

```
MACHINE
  MutExCounting(n)
CONSTRAINTS
  n ∈ ℕ
SETS
  turnStatus = {slS, reS, acS}
VARIABLES
  sl, re, ac, turn
INVARIANT
  sl ∈ ℕ ∧
  re ∈ ℕ ∧
  ac ∈ ℕ ∧
  turn ∈ turnStatus ∧
  re + sl + ac = n ∧
  ac ⩽ 1
INITIALISATION
  sl, re, ac := n, 0, 0 ||
  turn := slS
OPERATIONS
  ask =  SELECT sl > 0 THEN
    sl, re := sl − 1, re + 1 ||
     SELECT turn = slS THEN
       CHOICE turn := reS OR skip END
     END
   END ;
  mturn =  SELECT re > 0 ∧ turn = slS THEN
    turn := reS
   END ;
  activate =  SELECT re > 0 ∧ turn = reS THEN
    re, ac := re − 1, ac + 1 ||
    turn := acS
   END ;
  sleep =  SELECT ac > 0 THEN
    ac, sl := ac − 1, sl + 1 ||
    turn := slS
   END
END
```

FIG. 6.1 – Spécification B par abstraction de comptage du MutEx

et AC respectivement et `turn` est la variable qui mémorise dans quel état est le détenteur du droit : `turn` = `acS` signifie par exemple qu'il est dans l'état AC. Dans la spécification initiale, un processus p devient celui qui a le droit d'utiliser la section critique soit lors de la transition de SL vers R, si c'était le détenteur de ce droit au tour précédent, soit lors de la boucle autour de R. Seul ce processus peut alors utiliser la section critique. Dans la spécification par abstraction de comptage, l'opération `mturn` qui correspond à cette boucle n'a plus la même sémantique : elle permet d'acquérir un droit d'utilisation de la section critique pour un processus quelconque de R : celui qui accède ensuite à la section critique n'est alors pas nécessairement celui qui a franchi `mturn`.

L'approche par abstraction de prédicats souffre d'un autre inconvénient qu'est la construction du système abstrait. Celle-ci nécessite en effet un grand nombre de preuves : construire, par exemple le système abstrait de l'algorithme de Dijkstra (cf partie 11.3.4) requiert déjà plus de 10 minutes (sur un Sun Sparc 9 à 750 MHz) et plus de 350 Mo de mémoire [Bau03]. Le passage à des algorithmes plus conséquents en taille laisse alors craindre le pire.

6.5 Résumé

Ce chapitre a présenté un état de l'art des méthodes déductives de vérification d'invariant appliquées aux systèmes paramétrés, qui se classent en deux parties : celles qui décident une classe déterminée, par convergence garantie du calcul du point fixe et celles pour lesquelles la convergence n'est pas garantie mais est obtenue pratiquement. Ce chapitre a aussi montré que ces méthodes n'étaient pas applicables en l'état soit en raison de leur classe d'application qui ne recouvre pas celles que l'on souhaite traiter, soit en raison de leur complexité trop importante ne leur permettant pas de supporter un passage à l'échelle.

L'alternative retenue consiste à construire la spécification sans abstraction (susceptible de modifier la sémantique), puis à réaliser une vérification d'invariant par calcul de plus faible point fixe où les obligations de preuve sont optimisées pour être déchargées avec une convergence aussi rapide que possible dans un prouveur par superposition, en l'occurrence haRVey, pour mesurer l'efficacité de cette approche. Le chapitre suivant présente ce prouveur.

Chapitre 7

Le prouveur de théorèmes haRVey

Harvey existait bien avant le commencement de ce travail de thèse, non pas comme prouveur de théorèmes, mais comme héros de la bande dessinée Batman : Harvey est l'homme aux deux visages, mi ange mi démon. David Déharbe et Silvio Ranise sont à l'origine [DR02] du prouveur qui, comme le héros, est construit sur l'ambivalence de faire cohabiter un calcul par superposition et une simplification propositionnelle.

Dans [ARR03], Armando, Ranise et Rusinowitch présentent des procédures permettant de décider de la satisfaisabilité de formules logiques modulo des théories équationnelles finiment axiomatisables, comme la théorie des tableaux (chapitre 2), des listes ou la combinaison des deux. Ces procédures sont basées sur des techniques de réécriture et sur un calcul par superposition (chapitre 2) raisonnant à partir d'un ensemble de clauses. Le calcul par superposition est cette première face.

Mais traduire une formule quelconque en forme clausale engendre rapidement une explosion combinatoire sur la taille de la formule. Ceci est fréquent lorsque la formule est engendrée à partir de if then else imbriqués dans la spécification et qui ont été ensuite soigneusement dépliés lors de la génération des obligations de preuve. Ceci peut être évité en introduisant le symbole prédicatif ite (chapitre 2) dont la traduction en DNF est immédiate. Cette mise en forme DNF, implantée à l'aide de BDD, constitue la deuxième face du prouveur.

Ce chapitre, qui traite du prouveur haRVey, est organisé comme suit : La partie 7.1 présente haRVey en terme de flux de données, en s'attachant particulièrement à la structure propositionnelle de la formule fournie en entrée. La partie 7.2 montre comment sont gérés les quantificateurs dans haRVey.

7.1 Principes généraux

Une première partie présente tout d'abord les principes du prouveur de théorèmes haRVey, au travers des flux de données dans l'outil. La seconde partie, se focalise sur le symbole prédicatif ite de son langage d'entrée.

7.1.1 Flux dans l'outil

Pour être efficace dans la gestion de la preuve, l'outil haRVey[DR02, DR03] ajoute une couche propositionnelle (gérée par BDD) au calcul par superposition exploitant les outils SPASS [Wei99] et E-prover [Sch04]. Son fonctionnement est sommairement rappelé ci-dessous.

A partir d'une paire $(\phi, Ax(\mathcal{T}))$ fourni en entrée où ϕ est une formule sans quantificateur et $Ax(\mathcal{T})$ est un ensemble d'axiomes, le tout en logique équationnelle du premier ordre,

1. l'outil SPASS construit l'ensemble S de clauses qui est le résultat de la mise en forme CNF de l'ensemble des axiomes de $Ax(\mathcal{T})$;
2. la négation $\neg\phi$ est nommée $\overline{\phi}$;
3. une bijection f est crée ; elle associe à chaque atome de $\overline{\phi}$ un symbole propositionnel p ; sa réciproque est notée f^{-1} ; soit alors $\overline{\phi}_p$ la formule obtenue en remplaçant dans $\overline{\phi}$ chaque atome par son image par f ;
4. $\overline{\phi}_p$ est mise en DNF à l'aide d'une structure de BDD ; on nomme $bdd(\overline{\phi}_p)$ le résultat ;
5. pour chaque branche du BDD menant à la feuille vraie (c.a.d. chaque conjonction de $bdd(\overline{\phi}_p)$), chaque symbole propositionnel p est remplacé par l'atome correspondant selon f^{-1} ; la formule obtenue, nommée ξ, est alors couplée avec S pour être transmise au prouveur par superposition E-prover ;
6. si la clause vide n'appartient pas à l'ensemble C de clauses engendrées, le prouveur s'arrête en concluant que ξ est un contre exemple à la validité de ϕ modulo $Ax(\mathcal{T})$; sinon il reprend à l'étape 5 en simplifiant les autres branches puisque chaque littéral de ξ qui appartient aussi à une clause unitaire de C est alors faux.

La partie suivante montre comment prétraiter une formule que l'on souhaite décharger dans haRVey et qui contient des quantificateurs.

7.1.2 Structure ite

Pour des prédicats p, q et r d'une logique du premier ordre, on rappelle que le prédicat $ite(p, q, r)$ est une abréviation de $(p \Rightarrow q) \wedge (\neg p \Rightarrow r)$. Le chapitre 5 a montré que cette structure est engendrée par application de l'opérateur [] sur un prédicat et la substitution généralisée if P then S else S' où P est un prédicat, S et S' sont des substitutions généralisées.

L'intérêt majeur de cet opérateur réside dans le fait qu il lui correspond un BDD à partir duquel le BDD de la formule complète se construit directement. Cette heuristique permet de traiter des formules de grande taille, qui mettent en échec d'autres méthodes de décomposition (voir chapitre 11).

Par exemple, la figure 7.1 montre comment construire une forme DNF de la négation de la formule $ite(p, q, ite(p', q', r'))$. Celle-ci est obtenue à partir du BDD de la négation de la sous-formule $ite(p', q', r')$ (encadré par un rectange dans la figure) qui est considéré comme un nœud –relié aux feuilles $\top$ et $\bot$ – du BDD de $ite(p, q, ite(p', q', r'))$.

7.2 Gestion des quantificateurs

Notre approche de vérification nécessite de savoir traiter des obligations de preuve quantifiées.

Comme réponse à ce besoin, cette partie présente une démarche en amont de l'outil haRVey. qui pemet de fournir une paire $(\phi, Ax(\mathcal{T}))$ possédant les caractéristiques d'entrée de haRVey à partir d'une paire $(\phi', Ax(\mathcal{T})')$ où ϕ' contient des variables quantifiées, ϕ' et chaque axiome de $Ax(\mathcal{T})'$ sont en logique équationnelle du premier ordre, de telle sorte que $Ax(\mathcal{T})' \vDash \phi'$ si et seulement si $Ax(\mathcal{T}) \vDash \phi$.

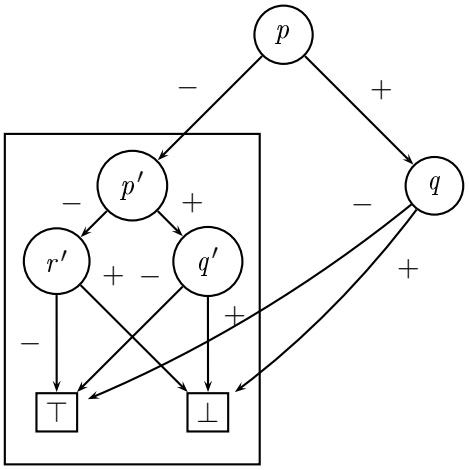


FIG. 7.1 – BDD de la négation de $ite(p, q, ite(p', q', r'))$

Cette démarche nécessite trois procédures : la première (partie 7.2.1) réduit la portée des quantificateurs, la seconde (partie 7.2.2) est une skolémisation restreinte et la troisième (partie 7.2.3) renomme les formules quantifiées résiduelles à l'aide d'un prédicat.

Chaque procédure est présentée à l'aide de règles de réécriture et ses propriétés sont exprimées en terme de satisfaisabilité.

7.2.1 Réduction de portée des quantificateurs

La fonction

$$(\mathsf{gq} : \text{formula} \rightarrow \text{formula}) = (\lambda\, \phi \;.\; \mathsf{dropExistential}(\mathsf{mini}(\phi))) \tag{7.1}$$

gère une partie des quantificateurs présents dans une formule ϕ.

Dans un premier temps, la portée des quantificateurs de ϕ est réduite autant que possible par application d'un algorithme de *miniscoping* [NW01]. Par exemple $(\forall\, x \;.\; \xi \wedge \psi)$ est remplacé par $\xi \wedge (\forall\, x \;.\; \psi)$ si x n'apparaît pas dans ξ. Cette transformation préserve l'équivalence logique et termine car elle se réduit à un seul parcours de la structure de la formule.

Les règles d'automatisation de cette procédure sont données à la figure 7.2. La fonction est l'identité sur les formules atomiques et elle invoque la fonction auxiliaire m_q lorsqu'une sous formule quantifiée est rencontrée. Le premier et le second argument de m_q sont respectivement le quantificateur et la variable quantifiée dont la portée est en cours de réduction. Le troisième argument est la sous-formule déjà traitée.

Intuitivement, dans une formule ϕ, la fonction m_q supprime le quantificateur et sa variable lorsque cette dernière n'est pas libre dans ϕ (règle (7.7)), traverse une négation en modifiant le quantificateur (règle (7.8)) – $\exists$ devient $\forall$ et réciproquement–, enlève de la portée du quantificateur toute sous-formule φ qui ne contient pas d'occurrence libre de la variable quantifiée lorsque la formule en cours est une conjonction ou disjonction de φ (règles (7.9) et (7.10)), exploite l'équivalence entre $(\forall\, x \,.\, \varphi \wedge \psi)$ et $(\forall\, x \,.\, \varphi) \wedge (\forall\, x \,.\, \psi)$ et son dual avec $\exists$ (règles (7.11) et (7.12)) et applique cette série de quatre règles au cas particulier de l'implication (de (7.13) à (7.15)) qui est une disjonction particulière et à l'opérateur *ite* qui est une conjonction de deux implications particulières (règles (7.16) et (7.17)).

$$
\begin{array}{rcll}
\mathsf{mini}(\phi \circ \psi) & := & \mathsf{mini}(\phi) \circ \mathsf{mini}(\psi) & (7.2)\\
\mathsf{mini}(\neg\phi) & := & \neg\mathsf{mini}(\phi) & (7.3)\\
\mathsf{mini}(\mathit{ite}(\phi,\psi,\xi)) & := & \mathit{ite}(\mathsf{mini}(\phi),\mathsf{mini}(\psi),\mathsf{mini}(\xi)) & (7.4)\\
\mathsf{mini}(\phi) & := & \phi \text{ si } \phi \text{ est atomique} & (7.5)\\
\mathsf{mini}(Q\,x_1,\cdots,x_n.\phi) & := & \mathsf{m}_q(Q,x_1,\ldots\mathsf{m}_q(Q,x_n,\mathsf{mini}(\phi))) & (7.6)\\
\mathsf{m}_q(Q,x,\phi) & := & \phi \text{ si } x \notin \mathcal{V}_{lib}(\phi) & (7.7)\\
\mathsf{m}_q(Q,x,\neg\phi) & := & \neg\mathsf{m}_q(-Q,x,\phi) & (7.8)\\
\mathsf{m}_q(Q,x,\phi \diamond \psi) & := & \mathsf{m}_q(Q,x,\phi) \diamond \psi \text{ si } x \notin \mathcal{V}_{lib}(\psi) & (7.9)\\
\mathsf{m}_q(Q,x,\phi \diamond \psi) & := & \phi \diamond \mathsf{m}_q(Q,x,\psi) \text{ si } x \notin \mathcal{V}_{lib}(\phi) & (7.10)\\
\mathsf{m}_q(\forall,x,\phi \wedge \psi) & := & \mathsf{m}_q(\forall,x,\phi) \wedge \mathsf{m}_q(\forall,x',\psi(x'/x)) \text{ sinon} & (7.11)\\
\mathsf{m}_q(\exists,x,\phi \vee \psi) & := & \mathsf{m}_q(\exists,x,\phi) \vee \mathsf{m}_q(\exists,x',\psi(x'/x)) \text{ sinon} & (7.12)\\
\mathsf{m}_q(Q,x,\phi \Rightarrow \psi) & := & \phi \Rightarrow \mathsf{m}_q(Q,x,\psi) \text{ si } x \notin \mathcal{V}_{lib}(\phi) & (7.13)\\
\mathsf{m}_q(Q,x,\phi \Rightarrow \psi) & := & \mathsf{m}_q(-Q,x,\phi) \Rightarrow \psi \text{ si } x \notin \mathcal{V}_{lib}(\psi) & (7.14)\\
\mathsf{m}_q(\exists,x,\phi \Rightarrow \psi) & := & \mathsf{m}_q(\forall,x,\phi) \Rightarrow \mathsf{m}_q(\exists,x',\psi(x'/x)) & (7.15)\\
\mathsf{m}_q(Q,x,\mathit{ite}(\phi,\psi,\xi)) & := & \mathit{ite}(\phi,\psi,\mathsf{m}_q(Q,x,\xi)) \text{ si } x \notin \mathcal{V}_{lib}(\phi) \cup \mathcal{V}_{lib}(\psi) & (7.16)\\
\mathsf{m}_q(Q,x,\mathit{ite}(\phi,\psi,\xi) & := & \mathit{ite}(\phi,\mathsf{m}_q(Q,x,\psi),\xi) \text{ si } x \notin \mathcal{V}_{lib}(\phi) \cup \mathcal{V}_{lib}(\xi) & (7.17)\\
\mathsf{m}_q(Q,x,\phi) & := & Q\,x.\phi \text{ sinon} & (7.18)
\end{array}
$$

où $\circ \in \{\wedge,\vee,\Rightarrow,\Leftrightarrow\}$, $\diamond \in \{\wedge,\vee\}$, $Q \in \{\forall,\exists\}$ et $-Q$ vaut $\forall$ (resp. $\exists$) quand Q est $\exists$ (resp. $\forall$).

FIG. 7.2 – Définition de mini et m_q.

$$\begin{aligned}
\mathsf{dropExistential}(\phi) &:= \mathsf{de}(\phi, +1) && (7.19)\\
\mathsf{de}(\phi, p) &:= \phi \text{ si } \phi \text{ est atomique,} && (7.20)\\
\mathsf{de}(\neg\phi, p) &:= \neg\mathsf{de}(\phi, -p) && (7.21)\\
\mathsf{de}(\phi \wedge \psi, p) &:= \mathsf{de}(\phi, p) \wedge \mathsf{de}(\psi, p) && (7.22)\\
\mathsf{de}(\phi \vee \psi, p) &:= \mathsf{de}(\phi, p) \vee \mathsf{de}(\psi, p) && (7.23)\\
\mathsf{de}(\mathit{ite}(\phi, \psi, \xi), p) &:= \mathit{ite}(\phi, \mathsf{de}(\psi, p), \mathsf{de}(\xi, p)) && (7.24)\\
\mathsf{de}(\phi \Rightarrow \psi, p) &:= \mathsf{de}(\phi, -p) \Rightarrow \mathsf{de}(\psi, p) && (7.25)\\
\mathsf{de}(\phi \Leftrightarrow \psi, p) &:= \phi \Leftrightarrow \psi && (7.26)\\
\mathsf{de}(\forall x.\phi, +1) &:= \forall x.\phi && (7.27)\\
\mathsf{de}(\exists x.\phi, -1) &:= \exists x.\phi && (7.28)\\
\mathsf{de}(\forall x.\phi, -1) &:= \mathsf{de}(\phi(c/x), -1) && (7.29)\\
\mathsf{de}(\exists x.\phi, +1) &:= \mathsf{de}(\phi(c/x), +1) && (7.30)
\end{aligned}$$

où c est une constante fraîche.

FIG. 7.3 – Définition de dropExistential et de.

7.2.2 Skolémisation restreinte

Dans un second temps, les variables quantifiées existentiellement$_{nf}$ (cf chapitre 2) et *extérieures* (pas sous la portée d'un quantificateur universel) sont skolémisées.

Ceci est automatisé en invoquant la fonction dropExistential définie à la figure 7.3 qui appelle la fonction auxiliaire de dont le second paramètre est la polarité $p \in \{+1, -1\}$ de la sous formule considérée en cours de traitement.

Cette fonction traverse la structure prédicative en mettant, le cas échéant, à jour la polarité : celle-ci est inversée pour une formule sous la portée d'une négation ou dans le membre gauche d'une implication (règles (7.21) et (7.25)). La fonction est récursive et s'arrête dès qu'un quantificateur ne peut pas être enlevé ; il est alors universel$_{nf}$ (règles (7.27) et (7.28)). Les variables existentiellement$_{nf}$ quantifiées sont remplacées par des constantes fraîches, et leur quantificateur est ôté (règles (7.29) et (7.30).

La transformation préserve la satisfaisabilité de la formule (cf théorème 4) et termine car elle traite au plus une fois chaque quantificateur. Ainsi pour toute théorie $\mathcal{T}$et toute formule ϕ, les formules ϕ et $\mathsf{gq}(\phi)$ sont équisatisfaisables modulo $\mathcal{T}$.

7.2.3 Renommage propositionnel

La fonction

$$(\mathsf{gqe} : \text{formula} \rightarrow \text{formula} \times \text{theory}) = (\lambda\,\phi\ .\ \mathsf{renameFormula}(\mathsf{gq}(\phi))) \qquad (7.41)$$

est une extension de la fonction gq qui gère cette fois tous les quantificateurs d'une formule ϕ. Cette fonction utilise gq puis la fonction renameFormula qui gère les quantificateurs résiduels par renommage propositionnel.

$$\begin{aligned}
\mathsf{renameFormula}(\phi) &:= \mathsf{rf}(\phi, +1) && (7.31)\\
\mathsf{rf}(\phi, p) &:= (\phi, \emptyset) \text{ où } \phi \text{ est atomique} && (7.32)\\
\mathsf{rf}(\neg\phi, p) &:= (\neg\psi, \mathcal{A}) \text{ où } (\psi, \mathcal{A}) = \mathsf{rf}(\phi, -p) && (7.33)\\
\mathsf{rf}(\phi_1 \diamond \phi_2, p) &:= (\psi_1 \diamond \psi_2, \mathcal{A}_1 \cup \mathcal{A}_2) \text{ où } (\psi_i, \mathcal{A}_i) = \mathsf{rf}(\phi_i, p) && (7.34)\\
\mathsf{rf}(\mathit{ite}(\phi_1, \phi_2, \phi_3), p) &:= (\mathit{ite}(\psi_1, \psi_2, \psi_3), \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3)\\
&\quad \text{où } (\psi_1, \mathcal{A}_1) = \mathsf{rf}(\phi_1, 0),\ (\psi_2, \mathcal{A}_2) = \mathsf{rf}(\phi_2, p)\\
&\quad \text{et } (\psi_3, \mathcal{A}_3) = \mathsf{rf}(\phi_3, p) && (7.35)\\
\mathsf{rf}(\phi_1 \Rightarrow \phi_2, p) &:= (\psi_1 \Rightarrow \psi_2, \mathcal{A}_1 \cup \mathcal{A}_2)\\
&\quad \text{où } (\psi_1, \mathcal{A}_1) = \mathsf{rf}(\phi_1, -p) \text{ et } (\psi_2, \mathcal{A}_2) = \mathsf{rf}(\phi_2, p) && (7.36)\\
\mathsf{rf}(\phi_1 \Leftrightarrow \phi_2, p) &:= (\psi_1 \Leftrightarrow \psi_2, \mathcal{A}_1 \cup \mathcal{A}_2) \text{ où } (\psi_i, \mathcal{A}_i) = \mathsf{rf}(\phi_i, 0) && (7.37)\\
\mathsf{rf}(Qx\,.\,\phi, 0) &:= (\mathbf{q}, \{\mathbf{q} \Leftrightarrow Qx\,.\,\phi\}) && (7.38)\\
\mathsf{rf}(\forall x\,.\,\phi, +1) &:= (\mathbf{q}, \{\mathbf{q} \Rightarrow (\forall x\,.\,\phi)\}) && (7.39)\\
\mathsf{rf}(\exists x\,.\,\phi, -1) &:= (\mathbf{q}, \{(\exists x\,.\,\phi) \Rightarrow \mathbf{q}\}) && (7.40)
\end{aligned}$$

où $\diamond \in \{\wedge, \vee\}$, $Q \in \{\forall, \exists\}$ et $\mathbf{q}$ est une constante propositionnelle fraîche.

FIG. 7.4 – Définition de renameFormula and rf.

D'un point de vue général, cette fonction parcourt la formule $\mathsf{gq}(\phi)$ de haut en bas à partir de sa racine et la transforme en une formule ϕ_g en remplaçant chaque sous-formule quantifiée ψ rencontrée par une constante propositionnelle $\mathbf{q}$ fraîche. Elle mémorise sa définition $\mathbf{q} \Leftrightarrow \psi$ dans un ensemble Δ d'axiomes nommé *définitions des constantes propositionnelles* et retourne la paire (ϕ_g, Δ) où ϕ_g est sans quantificateur. Cette fonction transforme alors le problème de vérification de la satisfaisabilité d'une formule quantifiée ϕ quelconque modulo une théorie $\mathcal{T}$ en la vérification de la satisfaisabilité de ϕ_g sans quantificateur modulo $\mathcal{T} \cup \Delta$.

D'un point de vue logique, l'équivalence $\mathbf{q} \Leftrightarrow \psi$ entre la proposition introduite et sa définition n'est pas nécessaire pour préserver la satisfaisabilité : la formule ψ étant quantifiée universellement$_{nf}$, seule l'implication $\mathbf{q} \Rightarrow \psi$ est ajoutée à Δ [NW01].

L'automatisation de cette dernière phase est donnée à la figure 7.4. Le premier argument de la fonction auxiliaire rf est la formule en cours de traitement tandis que le second est sa polarité. Elle ne modifie pas la structure propositionnelle de la formule auquelle on l'applique puisqu'elle ne fait que la traverser en fixant la polarité de la formule traitée (règles (7.33) à (7.37)). Elle effectue le remplacement par une constante fraiche $\mathbf{q}$ de chaque formule quantifiée suivant la polarité celle-ci (règles (7.38) à (7.40)). Le calcul de renameFormula termine puisqu'il traite chaque quantificateur au plus une fois.

La preuve de l'équisatisfaisabilité entre ϕ et $\phi' \bigwedge_{d \in \Delta} d$ où $(\phi', \Delta) = \mathsf{renameFormula}(\phi)$ se fait par induction sur la structure de la formula ϕ. En guise d'exemple soit la formule $\phi =_{def} \xi \vee \forall x.\psi$ telle que $\mathsf{renameFormula}(\phi) = (\xi \vee q, \{q \Rightarrow \forall x.\psi\})$, où q est une constante propositionnelle fraîche. Chaque modèle de $(\xi \vee q) \wedge (q \Rightarrow \forall x.\psi)$ est un modèle de $\xi \vee \exists x.\psi$ puisque cette dernière est une conséquence logique de $\xi \vee q$ et $q \Rightarrow \forall x.\psi$. Réciproquement, un modèle $\mathcal{M}'$ pour $(\xi \vee q) \wedge (q \Rightarrow \forall x.\psi)$ est obtenu depuis un modèle $\mathcal{M}$ pour $\xi \vee \forall x.\psi$ en définissant que l'interprétation de q est vraie dans $\mathcal{M}'$ si $\forall x.\psi$ est interprétée à vraie dans $\mathcal{M}$.

Théorème 7 *La formule ϕ est satisfaisable modulo une théorie $\mathcal{T}$ si et seulement si la formule (sans quantificateur) ϕ_g est satisfaisable modulo $\mathcal{T} \cup \Delta$ où* $\mathsf{gqe}(\phi) = (\phi_g, \Delta)$.

La preuve de ce théorème est une conséquence immédiate des remarques énoncées ci dessus.

7.3 Résumé

Ce chapitre a présenté le prouveur de théorèmes haRVey. Pour une formule fournie en entrée, cet outil simplifie au mieux sa structure propositionnelle par BDD en tirant profit de l'opérateur conditionnel ite. L'outil invoque ensuite un prouveur par superposition et en analyse la trace.

Notre approche nécessitant de savoir traiter des formules quantifiées, nous avons présenté dans ce chapitre une méthode permettant d'étendre la syntaxe d'entrée de l'outil aux formules contenant des quantificateurs. L'implantation correspondante et son bilan sont présentés au chapitre 11.

Le chapitre suivant décrit une démarche permettant de traduire une formule d'une théorie ensembliste en une formule appartenant à une logique équationnelle. La validité de cette formule engendrée est vérifiable par haRVey après la suppression des quantificateurs par gqe.

Chapitre 8

Des ensembles aux tableaux de booléens

Dans le contexte fixé de vérification de propriétés de sûreté par construction d'invariants, nous souhaitons décider de la validité d'obligations de preuve ensemblistes avec le prouveur de théorèmes haRVey présenté au chapitre précédent.

Celui-ci prenant en entrée des formules et une théorie (finiement axiomatisée) exprimées en logique équationnelle du premier ordre, il nous faut prévoir un traducteur des formules ensemblistes dans un tel formalisme. On rejoint en ce sens Kleene, qui écrivait [Kle67] (extrait de [GG91]) : *Il est caractéristique des logiciens contemporains de travailler avec souplesse, en utilisant plusieurs formulations et en passant de l'une à l'autre selon les exigences de leur propos.*

Se pose alors la question de choisir la théorie cible de la traduction. Disposant de résultats récents de décidabilité par superposition concernant la théorie des tableaux (cf théorème 6 chapitre 2), on peut penser à traduire chaque ensemble par sa fonction caractéristique, représentable par un tableau de booléens. Cette intuition est renforcée par le fait qu'en programmation, la structure d'ensemble est classiquement représentée par une structure de tableau.

La question qui suit est d'ordre logique : cette représentation pratique d'un ensemble par un tableau est-elle correcte en logique ? En d'autres termes, étant donnée une axiomatisation (restreinte) de la théorie des ensembles et une axiomatisation de la théorie des tableaux, une formule satisfaisable ou valide dans la première théorie le demeure-t-elle dans la seconde ?

Ce chapitre présente comment traduire une formule exprimée dans une théorie ensembliste en une formule dans une extension de $\mathcal{A}_s^e$, la théorie des tableaux avec extensionnalité vue au chapitre 2. La théorie ensembliste retenue $SSET$ est présentée en premier lieu, ainsi qu'un prétraitement pour l'opérateur de cardinalité. La théorie cible retenue $\mathcal{BA}_s^e$, extension de $\mathcal{A}_s^e$, est présentée en partie 8.2. La traduction est formellement définie en partie 8.3. La preuve de sa correction est établie à la partie 8.4. L'application sur l'exemple du MESI clôt ce chapitre.

8.1 $SSET$, une théorie ensembliste simple

Cette partie présente un fragment restreint de la théorie des ensembles nommé $SSET$, en adéquation avec les obligations de preuve ensemblistes que l'on souhaite décharger.

Cette partie s'organise comme suit : la partie 8.1.1 présente une axiomatisation finie de $SSET$; la partie 8.1.2 présente un pré-traitement permettant de réduire un problème de

$$\forall E \,.\, E \notin \emptyset \tag{8.1}$$
$$\forall E \,.\, E \in \{E\} \tag{8.2}$$
$$\forall E, E' \,.\, E \neq E' \Rightarrow E \notin \{E'\} \tag{8.3}$$
$$\forall E, S_1, S_2 \,.\, E \in S_1 \cup S_2 \Leftrightarrow (E \in S_1 \vee E \in S_2) \tag{8.4}$$
$$\forall E, S_1, S_2 \,.\, E \in S_1 \cap S_2 \Leftrightarrow (E \in S_1 \wedge E \in S_2) \tag{8.5}$$
$$\forall E, S_1, S_2 \,.\, E \in S_1 \setminus S_2 \Leftrightarrow (E \in S_1 \wedge E \notin S_2) \tag{8.6}$$
$$\forall S_1, S_2 \,.\, S_1 = S_2 \Leftrightarrow (\forall E \,.\, E \in S_1 \Leftrightarrow E \in S_2) \tag{8.7}$$
$$\forall S_1, S_2 \,.\, S_1 \subseteq S_2 \Leftrightarrow (\forall E \,.\, E \in S_1 \Rightarrow E \in S_2) \tag{8.8}$$

avec E, E' deux variables de sorte ELEM et S_1, S_2 des variables de sorte SET.

FIG. 8.1 – Axiomatisation $Ax(SSET)$

satisfaisabilité d'une formule modulo une théorie ensembliste avec l'opérateur de cardinalité en un problème de satisfaisabilité modulo $SSET$.

8.1.1 Axiomatisation finie $Ax(SSET)$

Soit $SSET$ un langage du premier ordre multi-sortes ne contenant que deux sortes distinctes ELEM, SET et tel que $\mathcal{P} = \{=_{\text{ELEM}}, =_{\text{SET}}, \subseteq, \in\}$ et $\mathcal{F}$ contient $\{\emptyset, \cup, \cap, \setminus, \{.\}\}$ et un ensemble de constantes de chaque sorte. Pour améliorer la lisibilité du document, on écrira $=$ pour $=_{\text{ELEM}}$ et $=_{\text{SET}}$ lorsque cela n'introduit pas d'ambiguïté.

Le terme constant $\emptyset$ est de sorte SET et pour un terme e de sorte ELEM le terme $\{e\}$ est de sorte SET. De même si s_1 et s_2 sont deux termes de sorte SET, alors $s_1 \bowtie s_2$, est un terme de sorte SET, pour $\bowtie \in \{\cap, \cup, \setminus\}$. On écrit alors $\{e_1, e_2, \ldots, e_n\}$ comme une abréviation de $((\{e_1\} \cup \{e_2\}) \ldots \cup \{e_n\})$.

L'ensemble des atomes de $SSET$ est constitué des expressions de la forme $e_1 = e_2$, $e \in s$, $s_1 \subseteq s_2$, $s_1 = s_2$, où e, e_1, e_2 sont des termes de sorte ELEM et s, s_1, s_2 sont des termes de sorte SET. Les littéraux, combinaisons booléennes et formules quantifiées de ce langage sont définies comme à la partie 2.1, et $e \notin s$ est introduit comme une abréviation de $\neg(e \in s)$ où e (resp. s) est de sorte ELEM (resp. SET). Les définitions de ces symboles fonctionnels sont formalisées par l'ensemble fini $Ax(SSET)$ d'axiomes donné dans la figure 8.1.

La partie suivante présente une extension de $SSET$.

8.1.2 Étendre $SSET$ avec l'opérateur de cardinalité

Fréquemment utilisé dans les spécifications ensemblistes, le symbole fonctionnel *card* d'arité 1 et tel que $\sigma(card) = (\text{SET}, \mathbb{N})$ donne le nombre d'éléments de l'ensemble fourni en paramètre. L'axiome

$$card(S) = N \Leftrightarrow (\exists_{\text{ELEM}}\, e_1, \ldots, e_N \,.\, \{e_1, \ldots, e_N\} = S \wedge \bigwedge_{1 \leqslant i < j \leqslant N} e_i \neq e_j). \tag{8.9}$$

Polarité	Littéral		Réécriture
positive	$card(s) = n$	$\rightarrow$	$\exists_{\text{ELEM}}\, e_1, \ldots, e_n \,.\, \bigwedge_{1 \leq i < j \leq n} e_i \neq e_j \wedge \{e_1, \ldots, e_n\} = s$
	$card(s) > n$	$\rightarrow$	$\exists_{\text{ELEM}}\, e_1, \ldots, e_n \,.\, \bigwedge_{1 \leq i < j \leq n+1} e_i \neq e_j \wedge \{e_1, \ldots, e_{n+1}\} \subseteq s$
négative	$card(s) = n$	$\rightarrow$	$(\forall_{\text{ELEM}}\, e_1, \ldots, e_n \,.\, \{e_1, \ldots, e_{n+1}\} \subseteq s \Rightarrow \bigvee_{1 \leq i < j \leq n+1} e_i = e_j) \wedge$ $\forall_{\text{ELEM}}\, f_1, \ldots, f_n \,.\, \neg(s \subseteq \{f_1, \ldots, f_{n-1}\})$
	$card(s) > n$	$\rightarrow$	$\forall_{\text{ELEM}}\, e_1, \ldots, e_n \,.\, \neg(s \subseteq \{e_1, \ldots, e_n\})$

où s est un terme de sorte SET, n est un entier naturel, e_1,...,e_n et f_1,...,f_n sont des variables fraîches de sorte ELEM.

FIG. 8.2 – Suppression de cardinalité dans des formules sans quantificateurs

universellement quantifié en S de sorte SET, et en N de sorte $\mathbb{N}$ en définit la sémantique. Cet axiome pourrait être ajouté à $SSET$. Ceci nécessiterait alors de détailler le domaine d'interprétation $D_{\mathbb{N}}$, puis d'étendre la démarche de traduction à ce symbole.

Au lieu de cela, on remplace ce littéral par le prétraitement présenté à la figure 8.2. Celle-ci montre comment réécrire un littéral comparant une cardinalité avec un entier naturel en une formule quantifiée de $SSET$, selon la polarité de cet opérateur. Ces réécritures sont optimisées pour éviter, autant que possible, l'introduction de quantificateurs.

En effet un ensemble est de cardinalité strictement supérieure à n si et seulement s'il existe $n+1$ éléments distincts qui lui appartiennent ou bien, de manière duale, si et seulement s'il n'est pas inclus dans quelque ensemble composé de n éléments ou moins. C'est cette double formulation qui est exploitée, pour n'introduire, par réécriture, que des variables quantifiées qui sont ensuite skolémisées par des constantes. On remarque, en effet, qu'une quantification existentielle est introduite lorsque la polarité est positive, et qu'elle est universelle dans le cas contraire. Il est donc ensuite possible de supprimer cette quantification par skolémisation.

Naïve et coûteuse, cette réécriture permet de traiter de petites cardinalités que l'on rencontre, notamment, dans les algorithmes d'exclusion mutuelle. On note aussi qu'elle se limite à la comparaison d'une cardinalité avec une constante n entière connue, excluant ainsi les comparaisons de la forme $card(s_1) \leqslant card(s_2)$, par exemple.

8.2 $\mathcal{BA}_s^e$, une théorie des tableaux de booléens

L'intuition sous-jacente à la démarche de traduction d'une formule de $SSET$ dans une (extension de) $\mathcal{A}_s^e$ est basée sur la représentation d'un ensemble par sa fonction caractéristique. Une telle fonction peut être encodée par un tableau de booléens dont les indices sont les éléments de l'ensemble. Par exemple, l'ensemble $s = \{1, 2\}$ peut être représenté par $s[1] = s[2] = \mathit{true}$ pour peu que $s[x] = \mathit{false}$, pour tout x distinct de 1 et 2.

Tout d'abord, soit $\mathcal{BA}_s^e \supset \mathcal{A}_s^e$ une spécialisation de $\mathcal{A}_s^e$ où l'on se limite aux tableaux qui ne stockent que des valeurs booléennes (c'est le '$\mathcal{B}$' devant le '$\mathcal{A}_s^e$' dans '$\mathcal{BA}_s^e$'). Cette théorie contient les deux constantes tt et ff de sorte VALUE et le symbole constant mty de sorte ARRAY qui définit le tableau valant partout ff. Intuitivement, mty est la représentation de l'ensemble vide dans $\mathcal{A}_s^e$. L'axiomatisation $Ax(\mathcal{BA}_s^e)$ est donnée à la figure 8.3.

$$\forall\, A, J, E\,.\, \mathsf{rd}(\mathsf{wr}(A, J, E), J) = E \tag{8.10}$$

$$\forall\, A, J, J', E\,.\, J \neq J' \Rightarrow \mathsf{rd}(\mathsf{wr}(A, J, E), J') = \mathsf{rd}(A, J') \tag{8.11}$$

$$\forall\, A, A'\,.\, (\forall\, J\,.\, \mathsf{rd}(A, J) = \mathsf{rd}(A', J)) \Rightarrow A = A' \tag{8.12}$$

$$\mathsf{tt} \neq \mathsf{ff} \tag{8.13}$$

$$\forall\, J\,.\, \mathsf{rd}(\mathsf{mty}, J) = \mathsf{ff} \tag{8.14}$$

avec A et A' des variables de sorte ARRAY, J et J' des variables de sorte INDEX, et E une variable de sorte VALUE.

FIG. 8.3 – Axiomatisation $Ax(\mathcal{BA}_s^e)$

8.3 De $SSET$ à $\mathcal{BA}_s^e$

Les parties précédentes ont présenté dans l'ordre $SSET$, la théorie ensembliste définissant la logique des formules en entrée de notre démarche de traduction et $\mathcal{BA}_s^e$, la théorie des tableaux de booléens définissant la logique des formules cibles. Cette partie présente les fonctions réalisant cette traduction.

On définit trois fonctions S, T et F telles que :

- S associe à toute sorte de $SSET$ une sorte dans $\mathcal{BA}_s^e$,
- T associe à tout terme de $SSET$ une paire dont le premier membre est un terme de $\mathcal{BA}_s^e$ et le second est un ensemble de formules de $\mathcal{BA}_s^e$,
- F associe à toute formule de $SSET$ une paire dont le premier membre est une formule de $\mathcal{BA}_s^e$ et le second est un ensemble de formules de $\mathcal{BA}_s^e$.

La suite donne tout d'abord la sémantique de ces fonctions, puis leur définition et décrit ensuite formellement comment d'autres symboles et axiomes sont ajoutés à $Ax(\mathcal{BA}_s^e)$ lors de la traduction d'une formule de $SSET$ dans une formule de la logique du premier ordre avec égalité lors de l'appel des fonctions T et F.

On pose $\mathsf{S}(\text{ELEM}) :=$ INDEX et $\mathsf{S}(\text{SET}) :=$ ARRAY. Intuitivement, la fonction T associe à chaque terme de $SSET$ de sorte ELEM (resp. de sorte SET) une paire dont le premier membre est la correspondance dans INDEX (resp. dans ARRAY) dudit terme et le second membre est un ensemble de formules de $\mathcal{BA}_s^e$ définissant cette correspondance. Les règles formelles de traduction de cette fonction sont données à la figure 8.4. On note que pour chaque u engendré, ajouter l'axiome δ_u permet de définir $\{\mathsf{tt}, \mathsf{ff}\}$ comme le co-domaine de la fonction caractéristique représentée par le symbole u.

On détaille, par exemple, la traduction d'une union $s_1 \cup s_2$ où s_1 et s_2 sont deux termes de sorte SET. Ce terme est traduit en un tableau u et un ensemble de définitions

$$\{\alpha_\cup\} \cup \alpha_1 \cup \alpha_2 \cup \{\delta_u\}$$

où

- $\alpha_\cup$ est la définition du tableau u qui représente l'ensemble $s_1 \cup s_2$. C'est formellement la formule

  $$\forall\, J\,.\, (\mathsf{rd}(u_1, J) = \mathsf{tt} \vee \mathsf{rd}(u_2, J) = \mathsf{tt}) \Leftrightarrow \mathsf{rd}(u, J) = \mathsf{tt}$$

  qui se lit "pour tout indice J, le tableau u vaut vrai en J si et seulement si le tableau u_1, qui correspond à l'ensemble s_1, ou le tableau u_2, qui correspond à l'ensemble s_2, vaut

$$
\begin{array}{rcll}
\mathsf{T}(\emptyset) & := & (\mathsf{mty}, \emptyset) & (8.15)\\
\mathsf{T}(k) & := & (i, \emptyset) \text{ si } k \text{ est une constante ou une variable de sorte ELEM} & (8.16)\\
\mathsf{T}(k) & := & (u, \{\delta_u\}) \text{ si } k \text{ est une constante ou une variable de sorte SET} & (8.17)\\
\mathsf{T}(\{k\}) & := & (\mathsf{wr}(\mathsf{mty}, i, \mathsf{tt}), \emptyset) \text{ si } k \text{ est une constante ou une variable} & \\
 & & \quad \text{de sorte ELEM telle que } T(k) = (i, \emptyset) & (8.18)\\
\mathsf{T}(s_1 \cup s_2) & := & (u, \{\alpha_\cup\} \cup \alpha_1 \cup \alpha_2 \cup \{\delta_u\}) \text{ où } \alpha_\cup \text{ est} & \\
 & & \quad \forall J \,.\, (\mathsf{rd}(u_1, J) = \mathsf{tt} \vee \mathsf{rd}(u_2, J) = \mathsf{tt}) \Leftrightarrow \mathsf{rd}(u, J) = \mathsf{tt} & (8.19)\\
\mathsf{T}(s_1 \cap s_2) & := & (u, \{\alpha_\cap\} \cup \alpha_1 \cup \alpha_2 \cup \{\delta_u\}) \text{ où } \alpha_\cap \text{ est} & \\
 & & \quad \forall J \,.\, (\mathsf{rd}(u_1, J) = \mathsf{tt} \wedge \mathsf{rd}(u_2, J) = \mathsf{tt}) \Leftrightarrow \mathsf{rd}(u, J) = \mathsf{tt} & (8.20)\\
\mathsf{T}(s_1 \setminus s_2) & := & (u, \{\alpha_\setminus\} \cup \alpha_1 \cup \alpha_2 \cup \{\delta_u\}) \text{ où } \alpha_\setminus \text{ est} & \\
 & & \quad \forall J \,.\, (\mathsf{rd}(u_1, J) = \mathsf{tt} \wedge \mathsf{rd}(u_2, J) = \mathsf{ff}) \Leftrightarrow \mathsf{rd}(u, J) = \mathsf{tt} & (8.21)
\end{array}
$$

où i est une constante (resp. variable) fraîche de sorte INDEX si k est une constante (resp. variable) de sorte ELEM, u est une constante (resp. variable) fraîche de sorte ARRAY si k est une constante (resp. variable) de sorte SET, s_1, s_2 sont des termes de sorte ARRAY tels que $\mathsf{T}(s_i) = (u_i, \alpha_i)$ pour $i = 1, 2$, J est de sorte INDEX et δ_u est définie pour chaque u par $\forall J \,.\, \mathsf{rd}(u, J) = \mathsf{tt} \vee \mathsf{rd}(u, J) = \mathsf{ff}$.

FIG. 8.4 – Fonction T de traduction des termes de $SSET$ dans $\mathcal{BA}_s^e$

aussi vrai en J".

- α_1 est l'ensemble des formules qui définissent le tableau u_1 comme étant la traduction de l'ensemble s_1.
- α_2 est l'ensemble des formules qui définissent le tableau u_2 comme étant la traduction de l'ensemble s_2.
- δ_u est la formule qui fixe le codomaine du tableau u.

Selon le même principe, les formules de $SSET$ sont traduites dans $\mathcal{BA}_s^e$ à l'aide de la fonction F définie par un ensemble de règles données à la figure 8.5.

Pour expliciter le comportement de F, détaillons, par exemple, la traduction de $\exists\, p \,.\, p \in c \wedge p \in e$ pour p une variable de sorte ELEM et c et e deux variables de sorte SET. En posant

$$
\begin{array}{rcl}
\mathsf{T}(p) & = & (i_p, \emptyset),\\
\mathsf{T}(c) & = & (u_c, \{\delta_{u_c}\}) \text{ et}\\
\mathsf{T}(e) & = & (u_e, \{\delta_{u_e}\}), \text{ on a}\\
\mathsf{F}(p \in c \wedge p \in e) & = & (\mathsf{rd}(u_c, i_p) = \mathsf{tt} \wedge \mathsf{rd}(u_e, i_p) = \mathsf{tt}, \{\delta_{u_c}, \delta_{u_e}\})
\end{array}
$$

et $\mathsf{F}(\exists\, p \,.\, p \in c \wedge p \in e)$ vaut $(\exists\, i_p \,.\, \delta_{u_c} \wedge \delta_{u_e} \wedge \mathsf{rd}(u_c, i_p) = \mathsf{tt}, \emptyset)$.

Retour à l'exemple MESI. Pour décider de la satisfaisabilité de la négation de l'obligation de preuve (5.4), on considère sa clôture existentielle

$$\exists\, m, e, s, i, c \,.\, Inv(m, e, s, i, c) \wedge (\exists\, p \,.\, p \in c \wedge p \in e \wedge \neg Inv(m \cup \{p\}, e \setminus \{p\}, s, i, c))$$

$$
\begin{array}{rcll}
\mathsf{F}(e \in s_1) & := & (\mathsf{rd}(u_1, i) = \mathsf{tt}, \alpha_1) & (8.22)\\
\mathsf{F}(s_1 = s_2) & := & (u_1 = u_2, \alpha_1 \cup \alpha_2) & (8.23)\\
\mathsf{F}(s_1 \subseteq s_2) & := & \big((\forall\, J\,.\, \mathsf{rd}(u_1, J) = \mathsf{tt} \Rightarrow \mathsf{rd}(u_2, J) = \mathsf{tt}), \alpha_1 \cup \alpha_2\big) & (8.24)\\
\mathsf{F}(\neg\varphi) & := & (\neg\, \widehat{\varphi}, \alpha) & (8.25)\\
\mathsf{F}(\varphi_1 \circ \varphi_2) & := & (\widehat{\varphi_1} \circ\ \widehat{\varphi_2}, \alpha_1 \cup \alpha_2) \text{ où } \circ \in \{\vee, \wedge, \Rightarrow \Leftrightarrow\} & (8.26)\\
\mathsf{F}(\mathit{ite}(\varphi, \varphi_1, \varphi_2)) & := & (\mathit{ite}(\widehat{\varphi}, \widehat{\varphi}_1, \widehat{\varphi}_2), \alpha \cup \alpha_1 \cup \alpha_2) & (8.27)\\
\mathsf{F}(\forall\, y\,.\, \varphi) & := & (\forall\, \widehat{y}\,.\, \alpha' \Rightarrow \widehat{\varphi}, \emptyset) & (8.28)\\
\mathsf{F}(\exists\, y\,.\, \varphi) & := & (\exists\, \widehat{y}\,.\, \alpha' \wedge \widehat{\varphi}, \emptyset) & (8.29)
\end{array}
$$

où e est une constante ou une variable de sorte ELEM telle que $T(e) = (i, \emptyset)$, s_1, s_2 sont des termes de sorte ARRAY tels que $\mathsf{T}(s_i) = (u_i, \alpha_i)$ pour $i = 1, 2$, J est une variable de sorte INDEX, φ, φ_1, φ_2 sont des formules de $SSET$ telles que $\mathsf{F}(\varphi) = (\widehat{\varphi}, \alpha)$ et $\mathsf{F}(\varphi_i) = (\widehat{\varphi}_i, \alpha_i)$ pour $i = 1, 2$, y est une variable de sorte SET ou ELEM telle que $\mathsf{T}(y) = (\widehat{y}, \alpha_y)$ pour $i = 1, 2$, et $\alpha' = \bigwedge_{d \in \alpha} d \wedge \bigwedge_{d \in \alpha_y} d$.

FIG. 8.5 – Fonction F de traduction des formules de $SSET$ dans $\mathcal{BA}_s^e$.

qui est satisfaisable si et seulement si la forme de Skolem

$$\mathit{Inv}(\widetilde{m}, \widetilde{e}, \widetilde{s}, \widetilde{\imath}, \widetilde{c}) \wedge \widetilde{p} \in \widetilde{c} \wedge \widetilde{p} \in \widetilde{e} \wedge \neg \mathit{Inv}(\widetilde{m} \cup \{\widetilde{p}\}, \widetilde{e} \setminus \{\widetilde{p}\}, \widetilde{s}, \widetilde{\imath}, \widetilde{c}) \tag{8.30}$$

l'est, où $\widetilde{m}$, $\widetilde{e}$, $\widetilde{s}$, $\widetilde{\imath}$, $\widetilde{c}$ sont des constantes fraîches de sorte SET et $\widetilde{p}$ est une constante fraîche de sorte ELEM.

On explique la démarche de traduction vers $\mathcal{BA}_s^e$ de la sous-formule $\mathit{card}(\widetilde{m} \cup \{\widetilde{p}\}) > 1$ apparaissant dans $\mathit{Inv}(\widetilde{m} \cup \{\widetilde{p}\}, \widetilde{e} \setminus \{\widetilde{p}\}, \widetilde{s}, \widetilde{\imath}, \widetilde{c})$ de (8.30). On commence par supprimer l'opérateur de cardinalité dans (8.30) comme présenté à la partie 8.1.2. Apparaissant à une polarité positive dans (8.30), $\mathit{card}(\widetilde{m} \cup \{\widetilde{p}\}) > 1$ se réécrit en $\neg(\widetilde{m} \cup \{\widetilde{p}\} \subseteq \{\widetilde{e}_1\})$ où $\widetilde{e}_1$ est une constante fraîche.

Pour traduire $\neg(\widetilde{m} \cup \{\widetilde{p}\} \subseteq \{\widetilde{e}_1\})$, on pose $\mathsf{T}(\widetilde{e}_1) = (i_{e_1}, \emptyset)$, $\mathsf{T}(\widetilde{p}) = (i_p, \emptyset)$, $\mathsf{T}(\widetilde{m}) = (u_m, \{\delta_{u_m}\})$. On a alors $\mathsf{T}(\widetilde{m} \cup \{\widetilde{p}\}) = (u_{m \cup \{p\}}, \{(8.31), \delta_{u_m}, \delta_{u_{m \cup \{p\}}}\})$ où (8.31) est l'axiome

$$\forall\, X\,.\, (\mathsf{rd}(u_m, X) = \mathsf{tt} \vee \mathsf{rd}(\mathsf{wr}(\mathsf{mty}, i_p, \mathsf{tt}), X) = \mathsf{tt}) \Leftrightarrow \mathsf{rd}(u_{m \cup \{p\}}, X) = \mathsf{tt} \tag{8.31}$$

qui définit l'union $\widetilde{m} \cup \{\widetilde{p}\}$.

Ainsi

$$
\begin{array}{rcl}
\mathsf{F}(\widetilde{m} \cup \{\widetilde{p}\} \subseteq \{\widetilde{e}_1\}) & = & (\forall\, J\,.\, \mathsf{rd}(u_{m \cup \{p\}}, J) = \mathsf{tt} \Rightarrow \mathsf{rd}(\mathsf{wr}(\mathsf{mty}, i_{e_1}, \mathsf{tt}), J) = \mathsf{tt},\\
& & \{(8.31), \delta_{u_m}, \delta_{u_{m \cup \{p\}}}\}),
\end{array}
$$

et donc

$$
\begin{array}{rcl}
\mathsf{F}(\neg(\widetilde{m} \cup \{\widetilde{p}\} \subseteq \{\widetilde{e}_1\})) & = & (\neg(\forall\, J\,.\, \mathsf{rd}(u_{m \cup \{p\}}, J) = \mathsf{tt} \Rightarrow \mathsf{rd}(\mathsf{wr}(\mathsf{mty}, i_{e_1}, \mathsf{tt}), J) = \mathsf{tt}),\\
& & \{(8.31), \delta_{u_m}, \delta_{u_{m \cup \{p\}}}\}).
\end{array}
$$

Il reste alors à établir la correction de la traduction ce que réalise la partie suivante.

$$
\begin{aligned}
I'(\mathsf{rd}) &\in D'_{\text{ARRAY}} \times D'_{\text{INDEX}} \rightarrow D'_{\text{VALUE}} \\
I'(\mathsf{rd})(s,e) &= \mathsf{tt} \text{ si } I(\in)(e,s) = \top && (8.32)\\
I'(\mathsf{rd})(s,e) &= \mathsf{ff} \text{ sinon} && (8.33)
\end{aligned}
$$

$$
\begin{aligned}
I'(\mathsf{wr}) &\in D'_{\text{ARRAY}} \times D'_{\text{INDEX}} \times D'_{\text{VALUE}} \rightarrow D'_{\text{ARRAY}} \\
I'(\mathsf{wr})(s,e,\mathsf{tt}) &= s \cup \{e\} && (8.34)\\
I'(\mathsf{wr})(s,e,\mathsf{ff}) &= s \setminus \{e\} && (8.35)
\end{aligned}
$$

$$
\begin{aligned}
I'(\mathsf{mty}) &= \emptyset && (8.36)\\
I'(u) &= s \text{ si } \mathsf{T}(s) = (u, \_) && (8.37)\\
I'(i) &= e \text{ si } \mathsf{T}(e) = (i, \emptyset) && (8.38)\\
I'(\mathsf{tt}) &= \mathsf{tt} && (8.39)\\
I'(\mathsf{ff}) &= \mathsf{ff} && (8.40)
\end{aligned}
$$

où $s \in D'_{\text{ARRAY}} = D_{\text{SET}}$, $e \in D'_{\text{INDEX}} = D_{\text{ELEM}}$, u est une constante de sorte ARRAY, i est une constante de sorte INDEX, $s \cup \{e\}$ et $s \setminus \{e\}$ sont des termes de $SSET$.

FIG. 8.6 – Interprétation des symboles fonctionnels de $\mathcal{BA}^e_s$

8.4 Correction de la traduction

La partie précédente a présenté une démarche de traduction d'une formule dans la théorie $SSET$ en une formule dans une extension de la théorie $\mathcal{A}^e_s$. Intuitivement, cette partie montre que cette traduction est correcte, c'est à dire que la valeur de vérité de la formule initiale dans $SSET$ est la même que la valeur de vérité de la formule traduite dans l'extension de $\mathcal{A}^e_s$. Le théorème suivant formalise cette intuition.

Théorème 8 *Soit φ une formule close sans quantificateur de $SSET$ et soit $(\psi, \alpha) = F(\varphi)$. φ est satisfaisable modulo $SSET$ si et seulement si ψ est satisfaisable modulo $\mathcal{BA}^e_s \cup \alpha$.*

Preuve du seulement si

Supposons qu'il existe un modèle pour $SSET$ et φ. D'après le lemme 1 (chapitre 2), il existe un modèle de Herbrand $\mathcal{I} = (D, I)$ pour $SSET$ et φ. Dans cette interprétation, les domaines d'interprétation des constantes de sorte SET et ELEM sont les ensembles $D_{\text{SET}} = \mathbb{H}_{\text{SET}}$ et $D_{\text{ELEM}} = \mathbb{H}_{\text{ELEM}}$ construits comme à la définition 13 (chapitre 2), à partir de la forme clausale de $Ax(SSET)$ et de φ.

On construit une interprétation particulière $\mathcal{I}' = (D', I')$ sur $\mathcal{BA}^e_s$ dont on démontre qu'elle est un modèle pour $\mathcal{BA}^e_s \cup \alpha$ et ψ. On pose $D'_{\text{INDEX}} = D_{\text{ELEM}}$, $D'_{\text{VALUE}} = \{\mathsf{tt}, \mathsf{ff}\}$ et $D'_{\text{ARRAY}} = D_{\text{SET}}$. La figure 8.6 donne une interprétation spécifique des symboles fonctionnels de $\mathcal{BA}^e_s$. Le seul symbole prédicatif de la théorie $\mathcal{BA}^e_s$ est l'égalité. I' l'interprète par la diagonale du domaine auquel elle s'applique.

Fait 1 Si l'interprétation $\mathcal{I}$ définie ci-dessus est un modèle de $SSET$, alors l'interprétation $\mathcal{I}'$ définie ci-dessus est un modèle pour $\mathcal{BA}_s^e$.

PREUVE. Il suffit de calculer la valeur de vérité de chaque axiome de $Ax(\mathcal{BA}_s^e)$ par $\mathcal{I}'$. Considérons, par exemple, l'axiome (8.10).

D'une part, chaque axiome de $Ax(SSET)$ étant interprété à $\top$, il en est ainsi pour (8.2) et pour (8.4). Ainsi, pour toute constante e de D_{ELEM} et tout terme s_1, s_2 de D_{SET} on a

$$I(\in)(e, \{e\}) \quad = \quad \top \tag{8.41}$$

$$I(\in)(e, s_1 \cup s_2) = \top \quad \text{ssi} \quad I(\in)(e, s_1) = \top \text{ ou } I(\in)(e, s_2) = \top. \tag{8.42}$$

D'autre part, étudier la valeur de vérité de l'axiome (8.10), qui est

$$\forall\, A, J, E \,.\, \mathsf{rd}(\mathsf{wr}(A, J, E), J) = E,$$

revient à vérifier que quelque soit la substitution de A par un terme s de D'_{ARRAY}, de J par un terme e de D'_{INDEX} et de E par un terme b de D'_{VALUE}, la formule est interprétée à $\top$. Ainsi, pour b valant tt, on a successivement

	$I'(\mathsf{rd}(\mathsf{wr}(s, e, \mathsf{tt}), e) = \mathsf{tt}) = \top$	
ssi	$I'(=)(\mathsf{rd}(\mathsf{wr}(s, e, \mathsf{tt}), e), \mathsf{tt}) = \top$	(définition de l'interprétation)
ssi	$I'(\mathsf{rd})(I'(\mathsf{wr})(s, e, \mathsf{tt}), e) = \mathsf{tt}$	(interprétation syntaxique de l'égalité)
ssi	$I'(\mathsf{rd})(s \cup \{e\}, e) = \mathsf{tt}$	(règle (8.34))
ssi	$I(\in)(e, s \cup \{e\}) = \top$	(règle (8.32))
ssi	$I(\in)(e, s) = \top \vee I(\in)(e, \{e\}) = \top$	(règle (8.42) en constatant que $\{e\} \in D_{\text{SET}}$)

qui est vrai d'après (8.41). La conclusion étant la même lorsqu'on substitue b par ff et la preuve étant indépendante de s et de e, on en déduit que (8.10) est interprété à $\top$. La preuve étant similaire pour les autres axiomes de $\mathcal{BA}_s^e$, on peut conclure. □

Fait 2 Soit φ une formule close sans quantificateur satisfaisable dans $SSET$. L'interprétation $\mathcal{I}'$ définie ci-dessus est un modèle pour α avec $\mathsf{F}(\varphi) = (\psi, \alpha)$.

PREUVE. Toutes les formules de α sont issues de la traduction par T des termes de φ. Autrement dit, F n'ajoute dans α que les formules issues d'une traduction par T.

Soit alors $t(\varphi)$ l'ensemble des termes de φ. On montre par induction sur la taille de $t(\varphi)$ que pour tout terme $t \in t(\varphi)$ tel que $\mathsf{T}(t) = (\_, \alpha_t)$, on a $I'(\alpha_t) = \top$.

La constante $\emptyset$, les termes de sorte ELEM et ceux de la forme $\{e\}$, où e est une constante de sorte ELEM, n'engendrent pas de formules par T. La preuve est donc triviale pour ces cas de base. On s'intéresse maintenant aux constantes s de sorte SET dans $t(\varphi)$. On a $\mathsf{T}(s) = (u_s, (8.43))$ avec

$$\forall\, J \,.\, \mathsf{rd}(u_s, J) = \mathsf{tt} \vee \mathsf{rd}(u_s, J) = \mathsf{ff}. \tag{8.43}$$

Pour évaluer la valeur de vérité de (8.43), on remplace J par n'importe quel terme e de $D'_{\text{INDEX}} = D_{\text{ELEM}}$. On a ainsi

	$I'(\mathsf{rd}(u_s, e) = \mathsf{tt} \vee \mathsf{rd}(u_s, e) = \mathsf{ff}) = \top$	
ssi	$I'(\mathsf{rd}(u_s, e) = \mathsf{tt}) = \top$ ou $I'(\mathsf{rd}(u_s, e) = \mathsf{ff}) = \top$	(définition de l'interprétation)
ssi	$I'(\mathsf{rd})(I'(u_s), e) = \mathsf{tt}$ ou $I'(\mathsf{rd})(I'(u_s), e) = \mathsf{ff}$	(interprétation de l'égalité)
ssi	$I(\in)(e, s) = \top$ ou $I(\in)(e, s) = \bot$	(règles (8.32) et (8.33))

qui est vraie.

Supposons que $s_1 \cap s_2$ appartienne à $t(\varphi)$. Posons $\mathsf{T}(s_1) = (u_1, \alpha_1)$, $\mathsf{T}(s_2) = (u_2, \alpha_2)$ et $T(s_1 \cap s_2) = (u_{s_1 \cap s_2}, \alpha_{s_1 \cap s_2})$ avec $\alpha_{s_1 \cap s_2} = \alpha_1 \cup \alpha_2 \cup \{(8.44), \delta_{u_{s_1 \cap s_2}}\}$ où (8.44) est l'axiome

$$\forall\, J\,.\,(\mathsf{rd}(u_1, J) = \mathsf{tt} \wedge \mathsf{rd}(u_2, J) = \mathsf{tt}) \Leftrightarrow \mathsf{rd}(u_{s_1 \cap s_2}, J) = \mathsf{tt} \tag{8.44}$$

qui définit l'union $s_1 \cap s_2$.

Comme $\alpha_1 \cup \alpha_2$ est interprété à vrai par hypothèse d'induction, il reste à montrer que la formule (8.44) est interprétée à vrai, la preuve pour $\delta_{u_{s_1 \cap s_2}}$ étant analogue. Pour étudier la valeur de vérité de (8.44), on substitue J par n'importe quel élément e de $D'_{\text{INDEX}} = D_{\text{ELEM}}$. On a alors successivement

$$\begin{array}{ll}
& I'(8.44) = \top \\
\text{ssi} & I'\big((\mathsf{rd}(u_1, e) = \mathsf{tt} \wedge \mathsf{rd}(u_2, e) = \mathsf{tt}) \Leftrightarrow \mathsf{rd}(u_{s_1 \cap s_2}, e) = \mathsf{tt}\big) = \top \\
\text{ssi} & \big(I'(\mathsf{rd}(u_1, e) = \mathsf{tt}) = \top \text{ et } I'(\mathsf{rd}(u_2, e) = \mathsf{tt}) = \top\big) \text{ ssi } I'(\mathsf{rd}(u_{s_1 \cap s_2}, e) = \mathsf{tt}) = \top \\
\text{ssi} & I'(\mathsf{rd})(I'(u_{s_1 \cap s_2}), e) = \mathsf{tt} \text{ ssi } \big(I'(\mathsf{rd})(I'(u_1), e) = \mathsf{tt} \text{ et } I'(\mathsf{rd})(I'(u_2), e) = \mathsf{tt}\big) \\
\text{ssi} & I'(\mathsf{rd})(s_1 \cap s_2, e) = \mathsf{tt} \text{ ssi } \big(I'(\mathsf{rd})(s_1, e) = \mathsf{tt} \text{ et } I'(\mathsf{rd})(s_2, e) = \mathsf{tt}\big) \qquad \text{(règle(8.36))} \\
\text{ssi} & I(\in)(e, s_1 \cap s_2) = \top \text{ ssi } \big(I(\in)(e, s_1) = \top \text{ et } I(\in)(e, s_2) = \top\big) \qquad \text{(règle (8.32))}
\end{array}$$

qui est vraie puisque par hypothèse, I est un modèle de $SSET$ et donc, en particulier, $I((8.5)) = \top$.

La preuve étant similaire pour les autres termes $s_1 \cup s_2$ et $s_1 \setminus s_2$, on peut conclure. □

Fait 3 Soit φ une formule close sans quantificateur et une formule ψ telle que $\mathsf{F}(\varphi) = (\psi, \alpha)$. $\mathcal{I}$ est un modèle pour φ si et seulement si $\mathcal{I}'$ est un modèle pour ψ.

PREUVE. La preuve se fait par induction sur la structure de φ. On montre tout d'abord que le fait est vrai pour les cas de prédicats élémentaires $e_1 \in s_1$, $s_1 \subseteq s_2$, $e_1 = e_2$ et enfin $s_1 = s_2$ avec e_1, e_2 de sorte ELEM, s_1 et s_2 de sorte SET. On traitera ensuite le cas de $\neg\varphi$ et enfin le cas de $\varphi_1 \diamond \varphi_2$ où $\diamond \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow, ite\}$.

Le raisonnement étant similaire pour tous les cas élémentaires, on ne détaille que celui où φ est est de la forme $s_1 \subseteq s_2$ avec s_1 et s_2 deux termes de sorte SET.

En posant $\mathsf{T}(s_1) = (u_1, \alpha_1)$ et $\mathsf{T}(s_2) = (u_2, \alpha_2)$, on a $F(s_1 \subseteq s_2) = (\psi, \alpha_1 \cup \alpha_2)$ où ψ est $\forall\, J\,.\,\mathsf{rd}(u_1, J) = \mathsf{tt} \Rightarrow \mathsf{rd}(u_2, J) = \mathsf{tt}$.

On obtient la valeur de vérité de ψ en y remplaçant J par n'importe quel élément e de $D'_{\text{INDEX}} = D_{\text{ELEM}}$. Ainsi

$$\begin{array}{ll}
& I'(\psi') = \top \\
\text{ssi} & I'\big(\mathsf{rd}(u_1, e) = \mathsf{tt} \Rightarrow \mathsf{rd}(u_2, e) = \mathsf{tt}\big) = \top \\
\text{ssi} & I'(\mathsf{rd}(u_1, e) = \mathsf{tt}) = \top \text{ implique } I'(\mathsf{rd}(u_2, e)) = \mathsf{tt}) = \top \\
\text{ssi} & I'(\mathsf{rd})(I'(u_1), e) = \mathsf{tt} \text{ implique } I'(\mathsf{rd})(I'(u_2), e) = \mathsf{tt} \\
\text{ssi} & I'(\mathsf{rd})(s_1, e) = \mathsf{tt} \text{ implique } I'(\mathsf{rd})(s_2, e) = \mathsf{tt} \qquad \text{(règle (8.36))} \\
\text{ssi} & I(\in)(e, s_1) = \top \text{ implique } I(\in)(e, s_2) = \top \qquad \text{(définition de } I'(\mathsf{rd})).
\end{array}$$

Or $\mathcal{I}$ est un modèle de $SSET$, donc satisfait les axiomes (8.1) à (8.8) et en particulier (8.8) , donc

$$I\,(\forall\, S_1, S_2\,.\,S_1 \subseteq S_2 \Leftrightarrow (\forall\, E\,.\,E \in S_1 \Rightarrow E \in S_2)) = \top.$$

Or s_1 et s_2 appartiennent à D_{SET}, donc

$$I(s_1 \subseteq s_2) = \top \text{ si et seulement si } I(\forall\, E\,.\,E \in s_1 \Rightarrow E \in s_2) = \top.$$

Avec l'hypothèse que $I(s_1 \subseteq s_2) = \top$ on a $I(\forall\, E\,.\, E \in s_1 \Rightarrow E \in s_2) = \top$. Or $e \in D_{\text{ELEM}} = D'_{\text{INDEX}}$ donc on a

$$I(e \in s_1 \Rightarrow e \in s_2) = \top.$$

et on en déduit

$$I(\in)(e, s_1) = \top \text{ implique } I(\in)(e, s_2) = \top,$$

et donc que $I'(\psi) = \top$.

Avec l'hypothèse que $I(s_1 \subseteq s_2) = \bot$, on a trouvé un $\widehat{e}$ tel que

$$I(\in)(\widehat{e}, s_1) = \top \text{ et } I(\in)(\widehat{e}, s_2) = \bot,$$

et donc tel que $I'(\psi) = \bot$. Ainsi le fait est vrai pour $\varphi = s_1 \subseteq s_2$.

Ensuite, on remarque que la formule φ n'étant pas quantifiée, sa traduction ψ a la même structure propositionnelle.

Supposons maintenant que $I(\varphi) = \top \Leftrightarrow I'(\psi) = \top$. On montre que le fait est inductif vis à vis de la négation. Le raisonnement suivant montre que ce fait est inductif

$$\begin{array}{ll} & I(\neg\varphi) = \top \\ \Leftrightarrow & I(\varphi) = \bot \\ \Leftrightarrow & I'(\psi) = \bot \qquad \text{(par hypothèse d'induction)} \\ \Leftrightarrow & I(\neg\psi) = \top. \end{array}$$

Supposons maintenant φ_i, ψ_i et α_i, $1 \leqslant i \leqslant 2$ tels que $\mathsf{F}(\varphi_i) = (\psi_i, \alpha_i)$. Les interprétations $\mathcal{I}$ et $\mathcal{I}'$ appliquant les tables de vérité pour les opérateurs logiques, on déduit de $I(\varphi_i) = \top \Leftrightarrow I'(\psi_i) = \top$ que $I(\varphi_1 \wedge \varphi_2) = \top \Leftrightarrow I'(\psi_1 \wedge \psi_2) = \top$, puis le même résultat pour $\vee$,$\Rightarrow$ et $\equiv$, *ite* . On conclut alors la preuve du fait. □

La preuve du "seulement si" est la conjonction des trois faits précédents.

Preuve du si

Soit une formule φ de $SSET$, soit $(\psi, \alpha) = \mathsf{F}(\varphi)$ et soit $\mathcal{I}' = (D', I')$ un modèle de $Ax(\mathcal{BA}^e_s) \cup \alpha$ et ψ. Montrons qu'il existe un modèle pour φ et $SSET$.

On construit pour cela l'interprétation $\mathcal{I} = (D, I)$ de Herbrand avec $D_{\text{SET}} = \mathbb{H}_{\text{SET}}$, $D_{\text{ELEM}} = \mathbb{H}_{\text{ELEM}}$ et pour laquelle la figure 8.7 définit l'interprétation I des symboles prédicatifs. L'égalité sur les termes de sorte ELEM est définie comme la diagonale du domaine de D_{ELEM}.

L'interprétation I' étant un modèle de ψ et $Ax(\mathcal{BA}^e_s) \cup \alpha$, on en déduit que

- tous les α sont interprétés à $\top$ par I' ; d'après la définition de I, la valeur de vérité de chaque prédicat ensembliste P par I est donc la même que celle de sa traduction Ψ_P par I' ;
- ψ est interprété à $\top$ par I'.

Comme φ et sa traduction ont la même structure propositionnelle (φ n'est pas quantifiée), on en déduit que φ est interprétée à $\top$ par I.

Il reste à montrer que l'interprétation $\mathcal{I}$ définie ci-dessus est un modèle pour $SSET$, ceci en calculant la valeur de vérité de chaque axiome de $Ax(SSET)$. Établissons, par exemple, la valeur de vérité de l'axiome (8.5). Pour toute substitution de E par un élément e de D_{ELEM}, de S_1 par un élément s_1 de D_{SET}, de S_2 par un élément s_2 de D_{SET}, on note i, u_1, α_1, u_2, α_2,

$$I(P)(t_1, t_2) = \top \text{ si } I'(\psi_P) = \top \text{ et } I'(\alpha_P) = \top \quad (8.45)$$
$$= \bot \text{ sinon} \quad (8.46)$$

où $P \in \{\in, \subseteq, =_{SSET}\}$, $F(P(t_1, t_2)) = (\psi_P, \alpha_P)$, t_1 et t_2 sont des termes de D_{ELEM} ou D_{SET}, en accord avec la signature de P.

FIG. 8.7 – Interprétation des symboles prédicatifs par $\mathcal{I}$

$u_{s_1 \cap s_2}$, $\alpha_{s_1 \cap s_2}$ les termes et ensembles de formules définis par $T(e) = (i, \emptyset)$, $T(s_1) = (u_1, \alpha_1)$, $T(s_2) = (u_2, \alpha_2)$ et $T(s_1 \cap s_2) = (u_{s_1 \cap s_2}, \alpha_{s_1 \cap s_2})$. Ainsi, $\mathsf{F}(e \in s_1 \cap s_2)$ est $(\mathsf{rd}(\alpha, i) = \mathsf{tt}, \alpha_{s_1 \cap s_2})$. Alors,

$$\begin{array}{lll}
& I(e \in s_1 \cap s_2) = \top & \\
\text{ssi} & I(\in)(e, s_1 \cap s_2) = \top & \\
\text{ssi} & I'(\alpha_{s_1 \cap s_2}) = \top \text{ et } I'(\mathsf{rd}(u_{s_1 \cap s_2}, i) = \mathsf{tt}) = \top & \text{(définition (8.45))} \\
\text{ssi} & I'(\alpha_1) = \top \text{ et } I'(\alpha_2) = \top \text{ et } I'(\mathsf{rd}(u_{s_1 \cap s_2}, i) = \mathsf{tt}) = \top \text{ et} & \\
& I'\big(\forall J \,.\, (\mathsf{rd}(u_1, J) = \mathsf{tt} \wedge \mathsf{rd}(u_2, J) = \mathsf{tt}) \Leftrightarrow \mathsf{rd}(u_{s_1 \cap s_2}, J) = \mathsf{tt}\big) = \top & \text{(décomposition de } s_1 \cap s_2) \\
\text{ssi} & I'(\alpha_1) = \top \text{ et } I'(\mathsf{rd}(u_1, i) = \mathsf{tt}) = \top \text{ et} & \\
& I'(\alpha_2) = \top \text{ et } I'(\mathsf{rd}(u_2, i) = \mathsf{tt}) = \top & \text{(simplification)} \\
\text{ssi} & I(\in)(e, s_1) = \top \text{ et } I(\in)(e, s_2) = \top &
\end{array}$$

L'interprétation $\mathcal{I}$ est donc un modèle pour l'axiome (8.5). Similaire, la preuve pour les autres axiomes est omise ; on peut donc conclure la démonstration.

8.5 Résumé

Ce chapitre a montré comment traduire une formule de la théorie ensembliste restreinte *SSET* en une formule équisatisfaisable dans $\mathcal{BA}^e_s$, une extension de la théorie des tableaux avec extensionnalité. Ce chapitre a aussi montré comment étendre cette traduction à d'autres opérateurs (notamment celui de cardinalité). La correction de la traduction a été établie à l'aide d'un théorème prouvé.

Comme le présente plus en détail le chapitre 11, cette traduction se révèle efficace en comparaison avec d'autres outils spécialisés dans la preuve de formules ensemblistes.

Se pose alors la question de savoir si la structure d'ensemble est la plus à même pour spécifier le comportement global des systèmes d'intérêt : certes les ensembles fournissent un confort d'utilisation, ils nécessitent néanmoins un traitement coûteux pour être réduits en des tableaux. Le chapitre suivant montre comment les systèmes uniformes distribués peuvent être spécifiés à l'aide de fonctions totales, plus directement traduisibles dans la théorie des tableaux.

Chapitre 9

Des fonctions totales aux tableaux de valeurs

Dans le cadre de la vérification de systèmes uniformes distribués, les ensembles fournissent un confort certain dans l'expression des opérations correspondant aux actions locales, aux synchronisations par rendez-vous ou par broadcast. En effet, un opérateur d'union exprime qu'un ensemble de processus entre dans un nouvel état et un opérateur de différence ensembliste exprime qu'un tel ensemble quitte un état.

Néanmoins, les ensembles n'en demeurent pas pour autant la structure idéale pour toute spécification. En effet, lorsque des ensembles $E_1, \ldots, E_n$ mémorisent respectivement les éléments dans les états $e_1, \ldots, e_n$, le spécifieur doit préciser que ces ensembles sont deux à deux disjoints, ce qui se représente par $\frac{n(n-1)}{2}$ intersections de la forme $E_i \cap E_j = \emptyset$ pour $1 \leqslant i < j \leqslant n$. Toute ces intersections vides sont traduites en autant de formules universellement quantifiées dans la théorie des tableaux, occasionnant pour le prouveur des ralentissements engendrés par des possibilités supplémentaires de superposition ou des éventuels problèmes de mémoire. Puisque chaque processus est dans un des états du système, pourquoi ne pas mémoriser l'état global du système dans une fonction totale qui associerait à chaque processus l'état dans lequel il se trouve ? Ces fonctions totales étant directement traduisibles dans la théorie des tableaux, on peut espérer d'importants gains de temps de preuve. Certes le co-domaine de chaque fonction totale étant un ensemble énuméré de la forme $\{e_1, \ldots, e_n\}$, chaque obligation de preuve doit intégrer en hypothèses $\frac{n(n-1)}{2}$ distinctions entre constantes de la forme $e_i \neq e_j$ pour $1 \leq i < j \leq n$, mais ce nombre est à rapprocher des $\frac{n(n-1)}{2}$ formules universellement quantifiées exprimées plus haut.

Ce chapitre suit un plan similaire à celui du chapitre précédent : la partie 9.1 présente *SFUNC*, une théorie simple de fonctions totales, la partie 9.2 montre que *SFUNC* est suffisante pour spécifier des systèmes uniformes distribués et la partie 9.3 montre comment traduire une forme de *SFUNC* en une formule de $\mathcal{BA}_s^e$.

9.1 *SFUNC*, une théorie simple des fonctions totales

L'objectif de cette partie est d'étendre *SSET* aux fonctions totales, pour lesquelles on définit un ensemble d'opérateurs minimal permettant de spécifier facilement les systèmes uniformes distribués. En plus des sortes SET et ELEM vues au chapitre précédent, on considère

Symbole	Signature	Sens	Axiomes
$\_ \lhd \{\_ \mapsto \_\}$	FUNC × ELEM × STATEVALUE × FUNC	surcharge d'une fonction avec une paire	(9.2) et (9.3)
$\_ \times \{\_\}$	SET × STATEVALUE × FUNC	produit cartésien entre ensemble et singleton	(9.4)
im	FUNC × STATEVALUE × STATEVALUE × ... ×STATEVALUE × STATEVALUE × FUNC	modification par bloc de la fonction	(9.5)

FIG. 9.1 – Signature des symboles de *SFUNC*

la sorte FUNC des fonctions totales dont le domaine est un ensemble de sorte SET et dont le co-domaine est un ensemble énuméré constitué de constantes de sorte STATEVALUE.

Chaque fonction totale $f \in s \to \{v_1, \ldots, v_n\}$ avec s de sorte SET et v_i de sorte STATEVALUE pour $1 \leqslant i \leqslant n$ est définie comme un ensemble de paires ordonnées inclus dans $s \times \{v_1, \ldots, v_n\}$ où chaque élément a du domaine s a au moins une image dans $\{v_1, \ldots, v_n\}$ (la relation est totale) et cette image est unique (la relation est une fonction). Formellement, cela revient à poser l'axiome

$$\begin{array}{l} f \in s \to \{v_1, \ldots, v_n\} \Leftrightarrow \\ \quad \forall\, a \,.\, a \in s \Rightarrow \left(\left(\bigvee_{1 \leqslant i \leqslant n} (a, v_i) \in f \right) \wedge (\forall\, v, w \,.\, ((a, v) \in f \wedge (a, w) \in f) \Rightarrow v = w) \right). \end{array} \tag{9.1}$$

Par la suite, pour $f \in$ FUNC et e (resp. v) appartenant au domaine (resp. au codomaine) de f, on note $f(e) = v$ à la place de $(e, v) \in f$.

Pour ces fonctions totales, se pose le choix d'un ensemble d'opérateurs suffisant pour spécifier les systèmes uniformes distribués définis au chapitre 3. Pour représenter la transition d'un processus d'un état vers un autre, un opérateur de *surcharge de fonction* suffit ; pour représenter le fait que les systèmes sont couramment initialisés en fixant tous les processus dans un même état e, un opérateur de *produit cartésien* entre l'ensemble des processus et le singleton $\{e\}$ suffit ; pour représenter les transitions issues d'un broadcast qui vont déplacer chaque processus (excepté celui qui émet le message) d'un état e_1 vers un état e'_1, un état e_2 vers un état e'_2 ..., un opérateur dit de *modification par bloc* qui modifie chaque image de la fonction uniquement selon sa valeur avant cette modification est suffisant.

On définit les trois symboles fonctionnels $\lhd$, $\times$ et im correspondant respectivement à la surcharge, au produit cartésien et à la modification par bloc. La figure 9.1 donne leur signature et leur sens. Leur sémantique est, quant à elle, formalisée sous forme d'axiomes (figure 9.2).

Intuitivement, pour un terme f de sorte FUNC, un terme s de sorte SET, un terme e de sorte ELEM, des termes v, v_1, ..., v_n, v'_1, ..., v'_n, de sorte STATEVALUE,

- le terme $f \lhd \{e \mapsto v\}$ représente la fonction totale identique à f sauf en e où elle vaut v. Pour lire sa valeur en un élément e', il suffit de distinguer le cas où $e = e'$ (axiome 9.2) et le cas contraire (9.3).
- le terme $s \times \{v\}$ représente le produit cartésien classique, c.a.d. la fonction totale valant v en tout élément de son domaine s.
- le terme $\mathsf{im}(f, v_1, v'_1, \ldots, v_n, v'_n)$, pour $f \in s \to \{v_1, \ldots v_n\}$, désigne la fonction totale de domaine s qui vaut v'_i en tout élément x où $f(x)$ vaut v_i, pour $1 \leqslant i \leqslant n$.

L'axiome (9.5) est en fait un schéma d'axiome qui doit être instancié pour chaque valeur

$$(F \lhd\!\!\!\!- \{(E \mapsto V)\})(E) = V \tag{9.2}$$
$$E \neq E' \Rightarrow (F \lhd\!\!\!\!- \{(E \mapsto V)\})(E') = F(E') \tag{9.3}$$
$$E \in S \Rightarrow (S \times \{V\})(E) = V \tag{9.4}$$
$$F \in S \rightarrow \{V_1, \ldots, V_n\} \wedge F(E) = V_i \Rightarrow \mathsf{im}(F, V_1, V_1', \ldots, V_n, V_n')(E) = V_i' \tag{9.5}$$

où toutes les variables sont universellement quantifiées, F est de sorte FUNC, E, E' sont de sorte ELEM, S est de sorte SET, V, V_1, …, V_n, V_1', …, V_n' sont de sorte STATEVALUE et $1 \leqslant i \leqslant n$.

FIG. 9.2 – Axiomatisation $Ax(SFUNC)$ de $SFUNC$

de i entre 1 et n. On note que l'opérateur im aurait pu être défini par le schéma d'axiomes

$$\mathsf{im}(F, V_1', \ldots, V_n') = \mathrm{dom}(F \rhd \{V_1\}) \times \{V_1'\} \cup \ldots \cup \mathrm{dom}(F \rhd \{V_n\}) \times \{V_n'\}. \tag{9.6}$$

pour toute fonction $F \in s \rightarrow \{V_1, \ldots, V_n\}$. Si (9.5) et (9.6) sont équivalents, (9.5) est préféré puisqu'il ne nécessite de définir ni le symbole de domaine (dom), ni celui de restriction de codomaine ($\rhd$), ni celui d'union ($\cup$), qui est problématique pour les fonctions totales dans le sens où l'union de deux fonctions totales n'en est pas nécessairement une.

Le seul symbole prédicatif supplémentaire est l'appartenance ($\in$) de signature FUNC $\times$ SET $\times \mathbb{P}$(STATEVALUE) et dont l'axiomatisation est donnée en (9.1).

La partie suivante montre comment spécifier des systèmes uniformes distribués à l'aide des opérateurs sur les fonctions totales définis dans cette partie.

9.2 Spécifications à l'aide de fonctions totales

Le chapitre 5 a montré que les systèmes uniformes distribués du chapitre 3 sont modélisables dans le langage des machines abstraites B, basé sur les expressions ensemblistes. A nouveau, ici, c'est le comportement global du système qui est modélisé, mais cette fois, l'état du système est mémorisé à l'aide d'une variable gs de type fonction totale au lieu de plusieurs ensembles comme précédemment. Le domaine de cette fonction est l'ensemble abstrait $PROC$ des identifiants des processus ($CACHE$ pour l'exemple du MESI) et le codomaine est un ensemble énuméré $STATUS$ des états de ces processus ($vAL = \{m, e, s, i\}$ pour le MESI). En d'autres termes, la fonction gs associe à chaque processus l'état dans lequel il est. Avec cette notation, nous présentons les substitutions généralisées qui expriment les actions locales, de rendez-vous et de broadcast puis l'assertion qui correspond au patron de propriétés vu au chapitre 3.

Une action locale à un seul processus qui passe d'un état $v_l \in STATUS$ dans un état $v_m \in STATUS$ sous hypothèse éventuelle d'une garde P se traduit par la substitution

$$\texttt{ANY } p \texttt{ WHERE } p \in PROC \wedge gs(p) = v_l \wedge P \texttt{ THEN } gs := gs \lhd\!\!\!\!- \{p \mapsto v_m\} \texttt{ END}$$

Le rendez-vous qui synchronise une transition d'un processus p de v_l dans v_m, sous hypothèse éventuelle d'une garde P, avec une transition d'un processus p' de v_o dans v_p, sous

MACHINE
  $mesi(CACHE)$
SETS
  $vAL = \{m, e, s, i\}$
VARIABLES
  gs
DEFINITIONS
  $im(f, s1, s1p, s2, s2p, s3, s3p, s4, s4p) ==$
    $\mathrm{dom}(f \rhd \{s1\}) \times \{s1p\} \cup$
    $\mathrm{dom}(f \rhd \{s2\}) \times \{s2p\} \cup$
    $\mathrm{dom}(f \rhd \{s3\}) \times \{s3p\} \cup$
    $\mathrm{dom}(f \rhd \{s4\}) \times \{s4p\}$
INITIALISATION
  $gs := CACHE \times \{i\}$
INVARIANT
  $gs \in CACHE \rightarrow vAL \wedge$
  $(\forall (p_1, p_2) . ((p_1 \in CACHE \wedge p_2 \in CACHE \wedge$
    $gs(p_1) = s) \Rightarrow gs(p_2) \neq m)) \wedge$
  $(\forall (p_1, p_2) . ((p_1 \in CACHE \wedge p_2 \in CACHE \wedge$
    $gs(p_1) = m \wedge gs(p_2) = m) \Rightarrow p_1 \neq p_2))$
OPERATIONS
  sendWriteInvalidate =
    ANY p
    WHERE $p \in CACHE \wedge gs(p) = s$
    THEN $gs := (CACHE \times \{i\}) \lhd\!\!\lhd \{p \mapsto e\}$
    END ;
  write =
    ANY p
    WHERE $p \in CACHE \wedge gs(p) = e$
    THEN $gs := gs \lhd\!\!\lhd \{p \mapsto m\}$
    END ;
  sendRead =
    ANY p
    WHERE $p \in CACHE \wedge gs(p) = i$
    THEN
      $gs := im(gs, m, s, e, s, s, s, i, i) \lhd\!\!\lhd \{p \mapsto s\}$
    END
 END

FIG. 9.3 – Spécification du MESI exprimée à l'aide de fonctions totales

hypothèse éventuelle d'une garde P', se traduit par la substitution

$$\begin{array}{l} \texttt{ANY } p \texttt{ WHERE } p \in PROC \wedge gs(p) = v_l \wedge P \texttt{ THEN} \\ \quad \texttt{ANY } p' \texttt{ WHERE } p' \in PROC \wedge gs(p') = v_o \wedge P' \texttt{ THEN} \\ \quad\quad gs := gs \lhd\!\!\lhd \{p \mapsto v_m\} \lhd\!\!\lhd \{p' \mapsto v_p\} \texttt{ END END} \end{array}$$

où les deux processus sont choisis de manière non déterministe.

L'action de broadcast provoquée par un processus qui, sous hypothèse éventuelle d'une garde P et en passant de v_l à v_m impose à chaque autre processus de passer de v_1 dans l'état v_1', avec $v_1, v_1' \in STATUS$, ..., de v_n de passer dans l'état v_n', avec $v_n, v_n' \in STATUS$ se traduit par la substitution

$$\begin{array}{l} \texttt{ANY } p \texttt{ WHERE } p \in PROC \wedge gs(p) = v_l \wedge P \texttt{ THEN} \\ \quad gs := \mathsf{im}(gs, v_1, v_1', \ldots, v_n, v_n') \lhd\!\!\lhd \{p \mapsto v_m\} \texttt{ END} \end{array}$$

On note que ces deux derniers patrons d'opérations de rendez-vous et de broadcast ne distinguent pas les cas particuliers où certains états v_i sont égaux comme dans les spécifications purement ensemblistes. La construction des spécifications s'en trouve simplifiée.

Dans ce langage, le patron de propriétés présenté à l'équation (3.4) s'exprime sous la forme

$$\forall\, p_a, p_b\, .\, (p_a \in PROC \wedge p_b \in PROC \wedge p_a \neq p_b \wedge gs(p_a) = v_p) \Rightarrow gs(p_b) \neq v_q \qquad (9.7)$$

où p_a, p_b sont des variables de sorte ELEM et v_p et v_q sont des constantes de sorte STATEVALUE, éventuellement égales.

Retour à l'exemple MESI. La figure 9.3 donne la spécification du MESI exprimée à l'aide de la fonction totale gs qui associe à chaque cache de $CACHE$ son statut dans vAL. On remarque que l'opérateur im étant absent du langage B, pour rester correct vis à vis de celui-ci, nous avons ajouté la définition de cet opérateur (donnée par l'égalité (9.6)) dans la clause `DEFINITIONS` de la machine B à vérifier.

La partie suivante montre comment une obligation de preuve de $SFUNC$ est traduite dans la spécialisation $\mathcal{BA}_s^e$ de la théorie des tableaux.

9.3 de *SFUNC* à $\mathcal{BA}_s^e$

La démarche de traduction s'appuie sur l'analogie entre les paires d'axiomes ((2.1),(2.2)) et ((9.2),(9.3)). De la même manière qu'un ensemble était représenté par un tableau de booléens, une fonction va être représentée par un tableau de valeurs. On étend ainsi les domaines d'application des fonctions S, T, et F vues au chapitre précédent.

La fonction S est enrichie par $\mathsf{S}(\text{FUNC}) = \text{ARRAY}$, $\mathsf{S}(\text{STATEVALUE}) = \text{VALUE}$. Les extensions de T aux symboles fonctionnels $\lhd$, $\times$ et im et de F au seul symbole prédicatif introduit, $\in$, sont données à la figure 9.4. Dans cette figure, null est une constante de type VALUE distincte des v_i, $1 \leqslant i \leqslant n$, qui, lorsqu'elle est employée dans l'égalité $\mathsf{rd}(u, X) = \mathsf{null}$, précise que le tableau u n'est pas défini en X. On explique la traduction de im (règle 9.13) et du prédicat $\in$ (règle 9.14) comme suit :

- pour chaque terme v_i de sorte VALUE et paramètre de $\mathsf{im}(f, v_1, v_1', \ldots, v_n, v_n')$, le tableau u codant $\mathsf{im}(f, v_1, v_1', \ldots, v_n, v_n')$ vaut $\widehat{v_i'}$ (représenté par $\mathsf{rd}(u, X) = \widehat{v_i'}$) partout où le tableau $\widehat{f}$ codant f vaut $\widehat{v_i}$ (représenté par $\mathsf{rd}(\widehat{f}, X) = \widehat{v_i'}$) et null ailleurs ;
- le tableau $\widehat{f}$ code la fonction totale f de s dans $\{v_1, \ldots, v_n\}$ si pour tout indice X, si ce X représente un élément de s, $\widehat{f}$ vaut un certain $\widehat{v_i}$ en X et, sinon, $\widehat{f}$ n'est pas défini en ce X.

Par la suite, pour simplifier la présentation, on note v_l à la place de $\widehat{v_l}$ lorsque cela n'introduit pas d'ambiguïté.

Retour à l'exemple MESI. On détaille la traduction par T du terme $CACHE \times \{i\}$ qui apparaît à la fois dans l'obligation de preuve relative à l'initialisation et dans l'obligation de preuve de l'opération sendWriteInvalidate. En posant $\mathsf{T}(CACHE) = (u_{CACHE}, \{\delta_{CACHE}\})$ (règle (8.17)) et $\mathsf{T}(i) = (\widehat{i}, \emptyset)$ (règle (9.8)), on a

$$\begin{aligned}\mathsf{T}(CACHE \times \{i\}) \quad = \quad & (u, \{\delta_{CACHE}\} \cup \{(\forall\, X \,.\, \mathsf{rd}(u_{CACHE}, X) = \mathsf{tt} \Rightarrow \mathsf{rd}(u, X) = \widehat{i}),\\ & \qquad (\forall\, X \,.\, \mathsf{rd}(u_{CACHE}, X) = \mathsf{ff} \Rightarrow \mathsf{rd}(u, X) = \mathsf{null})\}).\end{aligned}$$

Dans cette traduction, u est le tableau qui représente le produit cartésien $CACHE \times \{i\}$.

Conjecture 1 *Une formule φ est satisfaisable modulo $Ax(SFUNC)$ ssi $\widehat{\varphi}$ est satisfaisable modulo $\mathcal{BA}_s^e \cup \alpha$ où $\mathsf{F}(\varphi) = (\widehat{\varphi}, \alpha)$.*

IDÉE DE PREUVE. Les traducteurs des symboles fonctionnels et prédicatifs étant le reflet des axiomes qui les définissent, la démarche de preuve, technique et fastidieuse, est similaire à celle du théorème 8 du chapitre précédent. □

Elle n'a pas été entièrement développée dans ce travail pour son faible intérêt pratique. En effet, on peut aussi voir les notations des fonctions totales comme une abréviation des notations sur les tableaux et considérer ainsi que les fonctions totales sont définies directement à l'aide des axiomes définissant les tableaux.

9.4 Résumé

Ce chapitre a présenté *SFUNC*, une théorie simple de fonctions totales définissant un nombre minimal d'opérateurs pour spécifier les systèmes uniformes distribués de cette étude où chaque système de transitions a un nombre fini d'états. Celle-ci possède plusieurs avantages :

$$
\begin{aligned}
\mathsf{T}(w) &:= (\widehat{w}, \emptyset) \text{ si } w \text{ est une constante ou une variable} \\
&\quad \text{de sorte } \textsc{stateValue} && (9.8)\\
\mathsf{T}(g) &:= (\widehat{g}, \emptyset) \text{ si } g \text{ est une constante ou une variable} \\
&\quad \text{de sorte } \textsc{func} && (9.9)\\
\mathsf{T}(f \lhd \{e \mapsto v\}) &:= (\mathsf{wr}(\widehat{f}, i, \widehat{v}), \alpha_f \cup \alpha_v) && (9.10)\\
\mathsf{T}(f(e)) &:= (\mathsf{rd}(\widehat{f}, i), \alpha_f) && (9.11)\\
\mathsf{T}(s \times \{v\}) &:= (u, \alpha_s \cup \alpha_v \cup \{ \\
&\quad (\forall X \,.\, \mathsf{rd}(\widehat{s}, X) = \mathsf{tt} \Rightarrow \mathsf{rd}(u, X) = \widehat{v}), \\
&\quad (\forall X \,.\, \mathsf{rd}(\widehat{s}, X) = \mathsf{ff} \Rightarrow \mathsf{rd}(u, X) = \mathsf{null})\}) && (9.12)\\
\mathsf{T}(\mathsf{im}(f, v_1, v_1', \ldots, v_n, v_n')) &:= (u, \alpha_f \cup (\bigcup_{1 \leqslant i \leqslant n} \alpha_{v_i} \cup \alpha_{v_i'}) \cup \\
&\quad \bigcup_{1 \leqslant i \leqslant n} \{\forall X \,.\, \mathsf{rd}(\widehat{f}, X) = \widehat{v_i} \Rightarrow \mathsf{rd}(u, X) = \widehat{v_i'}\} \cup \\
&\quad \{\forall X \,.\, (\bigwedge_{1 \leqslant i \leqslant n} \mathsf{rd}(\widehat{f}, X) \neq \widehat{v_i}) \Rightarrow \mathsf{rd}(u, X) = \mathsf{null}\}) && (9.13)\\
\mathsf{F}(f \in s \to \{v_1, \ldots, v_n\}) &:= ((\forall X \,.\, \mathsf{rd}(\widehat{s}, X) = \mathsf{tt} \Rightarrow \bigvee_{1 \leqslant i \leqslant n} \mathsf{rd}(\widehat{f}, X) = \widehat{v_i}) \wedge \\
&\quad (\forall X \,.\, \mathsf{rd}(\widehat{s}, X) = \mathsf{ff} \Rightarrow \mathsf{rd}(\widehat{f}, X) = \mathsf{null}), \\
&\quad \alpha_f \cup \alpha_s \cup \bigcup_{1 \leqslant i \leqslant n} \alpha_{v_i}) && (9.14)
\end{aligned}
$$

où $\widehat{w}$ est une constante (resp. variable) fraîche de sorte VALUE si w est une constante (resp. variable), $\widehat{g}$ est une constante (resp. variable) fraîche de sorte ARRAY si g est une constante (resp. variable), f est un terme de sorte FUNC, e est une constante ou une variable de sorte ELEM telle que $\mathsf{T}(e) = (i, \emptyset)$, v est un terme de sorte STATEVALUE, u est une constante fraîche de sorte ARRAY, $\mathsf{T}(k) = (\widehat{k}, \alpha_k)$, pour $k \in \{f, v, s, v', v_1, \ldots, v_n, v_1', \ldots, v_n'\}$ et X est une variable de sorte INDEX.

FIG. 9.4 – Extension de T et F aux fonctions totales

(*i*) plus proche de la structure de raisonnement (la théorie des tableaux) que les ensembles, elle nécessite des traductions moins complexes ; (*ii*) permettant de représenter chaque état du système de transitions comme une constante de sorte STATEVALUE, les distinctions entre ces constantes qu'elle engendre dans chaque obligation de preuve ne sont plus que des littéraux sur des éléments et donc directement traduisibles dans $\mathcal{BA}_s^e$.

La démarche de traduction de *SFUNC* vers $\mathcal{BA}_s^e$ présentée étend celle du chapitre précédent aux opérateurs et prédicats de cette théorie.

Le chapitre suivant exploite une caractéristique des systèmes distribués pour simplifier la démarche de traduction vue dans ce chapitre et aboutir alors aux systèmes dits de tableaux.

Chapitre 10

Substitutions de tableaux

En présentant deux démarches de traduction vers une logique équationnelle, les deux chapitres précédents ont montré l'adéquation d'une telle logique avec notre cadre de vérification. Néanmoins, ces traductions peuvent paraître insuffisantes puisqu'elles génèrent des formules complexes soit en raison du choix de structure du modèle initial (les ensembles), soit parce qu'elle mélangent plusieurs structures (ensemblistes et fonctionnelles).

Or la classe de systèmes uniformes distribués présentée au chapitre 3 possède une propriété simplificatrice qui n'est pas exploitée et que l'on présente ici : le système global est composé d'un nombre d'entités constant. La destruction d'une entité est représentée par un état ρ du système de transitions de l'entité où elle est dite inactive, endormie,... (cet état n'intervient généralement pas dans l'expression de la propriété de sûreté), et, de même, la génération d'une entité consiste à extraire de cet état ρ l'entité en question.

Intuitivement, si l'ensemble des entités est représenté par un ensemble abstrait A et qu'aucune opération du système ne modifie A, toute contrainte $a \in A$ d'appartenance à cet ensemble est valide donc peut être abstraite. Il en résulte une traduction simplifiée (et donc plus rapide) en des formules (de taille encore plus réduite) exprimées dans une logique équationnelle pour laquelle nous avons établi de nouveaux résultats de décidabilité.

Ce chapitre est organisé comme suit : la première partie démontre que, dans certains cas, certains prédicats introduits par la traduction donnée au chapitre précédent peuvent être abstraits, et remplacés par vrai ($\top$) sans modifier la propriété de satisfaisabilité. La partie 10.2 présente le langage des substitutions de tableaux, permettant de décrire les systèmes de manière concise et pour lesquelles les obligations de preuve engendrées sont traitables par raisonnement équationnel. La partie 10.3 présente comment réécrire le symbole fonctionnel block correspondant à une modification par bloc (et donc globale) de l'état général du système. Enfin, la partie 10.4 montre quels résultats de décidabilité sont garantis selon la nature des obligations de preuve à vérifier.

10.1 Abstraire le domaine d'une fonction totale

Comme au chapitre précédent, on considère un système uniforme distribué dont l'état général est mémorisé à l'aide d'une fonction totale $f : s \to \{v_1, \ldots, v_n\}$. On montre dans cette partie comment abstraire certains prédicats ensemblistes relatifs à l'appartenance au domaine s de f. Intuitivement, comme ceci est mentionné dans l'introduction, aucune opération ne modifie ce domaine. Celui-ci peut donc être vu comme un ensemble universel et être donc

abstrait.

Plus formellement, on montre dans un premier temps (lemme 1) que f est invariablement une fonction totale de $s \to$ dans $\{v_1, \ldots, v_n\}$ pour les actions considérées ici. On démontre ensuite (lemme 2 et corollaire 3) que les prédicats d'appartenance à s peuvent être abstraits.

Lemme 1 (Invariance du type des fonctions totales) Soit $f : s \to \{v_1, \ldots, v_n\}$ la fonction totale représentant l'état général d'un système dont les opérations sont spécifiées selon les règles de la partie 9.2. On suppose de plus que la fonction totale est initialisée par une affectation de la forme $f := s \times \{v_i\}$ pour $1 \leqslant i \leqslant n$. Alors le prédicat $f : s \to \{v_1, \ldots, v_n\}$ est invariant.

PREUVE. Ce prédicat est établi à l'initialisation d'après l'axiome (9.4). Il reste alors à établir la preuve pour chacune des trois familles d'opérations données à la partie 9.2. Supposons par hypothèse d'induction que f vérifie cette propriété. Alors $f \Leftarrow \{p \mapsto v_m\}$, $f \Leftarrow \{p \mapsto v_m\} \Leftarrow \{p' \mapsto v_p\}$ et $\mathsf{im}(f, v_1, v'_1, \ldots, v_n, v'_n) \Leftarrow \{p \mapsto v_m\}$ vérifient cette propriété d'après les axiomes (9.2) et (9.3) et (9.5). □

Ainsi, sous les hypothèses du lemme 1, lorsque l'invariant *Inv* s'écrit sous la forme $(f \in s \to \{v_1, \ldots, v_n\}) \wedge \varphi$, l'obligation de preuve (5.1) à vérifier pour garantir que l'état initial est inclus dans l'invariant se réécrit

$$\bigwedge_{1 \leqslant j < k \leqslant n} v_j \neq v_k \Rightarrow \varphi(s \times \{v_i\}/f). \tag{10.1}$$

La formule (5.3) à vérifier pour garantir que l'invariant est stable par toute opération du programme se réécrit, selon la nature de l'opération,

$$\begin{array}{l} \left(\bigwedge_{1 \leqslant j < k \leqslant n} v_j \neq v_k \wedge f \in s \to \{v_1, \ldots, v_n\} \wedge \varphi\right) \Rightarrow \\ \quad (\forall\, p \,.\, (p \in s \wedge f(p) = v_l) \Rightarrow \varphi(f \Leftarrow \{p \mapsto v_m\}/f))\,, \end{array} \tag{10.2}$$

$$\begin{array}{l} \left(\bigwedge_{1 \leqslant j < k \leqslant n} v_j \neq v_k \wedge f \in s \to \{v_1, \ldots, v_n\} \wedge \varphi\right) \Rightarrow \\ \quad (\forall\, p, q \,.\, (p \in s \wedge q \in s \wedge f(p) = v_l \wedge f(q) = v_o) \Rightarrow \\ \qquad \varphi(f \Leftarrow \{p \mapsto v_m\} \Leftarrow \{q \mapsto v_p\}/f)) \end{array} \tag{10.3}$$

ou

$$\begin{array}{l} \left(\bigwedge_{1 \leqslant j < k \leqslant n} v_j \neq v_k \wedge f \in s \to \{v_1, \ldots, v_n\} \wedge \varphi\right) \Rightarrow \\ \quad (\forall\, p \,.\, (p \in s \wedge f(p) = v_l) \Rightarrow \varphi(\mathsf{im}(f, v_1, v'_1, \ldots, v_n, v'_n)/f)) \end{array} \tag{10.4}$$

où les obligations de preuve (10.2), (10.3) et (10.4) correspondent respectivement à la stabilité de φ par rapport aux opérations modélisant une transition locale, un rendez-vous et un broadcast.

Le lemme suivant montre que certains prédicats non triviaux d'une telle formule peuvent être abstraits sans modifier la satisfaisabilité de la formule.

Lemme 2 Soit S un ensemble de clauses d'un langage $\mathcal{L}$ possédant un symbole prédicatif p tel que

1. p est d'arité 1, $\sigma(p) = \tau$ et il n'existe pas de symbole fonctionnel $f \in \mathcal{L}$ d'arité non nulle avec $\sigma(f) = (\ldots, \tau)$,

2. $p(t)$ n'apparaît que dans des clauses positives unitaires de S lorsque le terme t est une constante,
3. S contient au moins une clause $p(t)$ où t est une constante.

Alors S est satisfaisable si et seulement si S' l'est où S' est l'ensemble de clauses S dans lequel on a remplacé chaque occurrence de $p(t)$ par $\top$.

PREUVE. Le si ($\Leftarrow$) est évident : soit $\mathcal{I}'$ un modèle de S'. On construit l'interprétation $\mathcal{I}$ comme le prolongement de $\mathcal{I}'$ tel que $\mathcal{I}$ interprète systématiquement le prédicat p à vrai. Par définition de $\mathcal{I}'$, l'interprétation $\mathcal{I}$ est un modèle pour S.

Étudions le seulement si ($\Rightarrow$). L'ensemble S étant satisfaisable, il existe une interprétation $\mathcal{I}$ qui est un modèle pour S. On construit $\mathcal{I}'$ identique à $\mathcal{I}$ sauf pour les prédicats p, qu'$\mathcal{I}'$ interprète toujours à $\top$. Si $\mathcal{I}'$ n'est pas un modèle de S, alors il existe une clause $C = C_1 \vee \ldots \vee C_n$ telle que $I'(C) = \bot$. Donc $I'(C_i) = \bot$ pour tout i avec $1 \leqslant i \leqslant n$. Or, par hypothèse sur $\mathcal{I}$, il existe un littéral C_i, qui est interprété à vrai par $\mathcal{I}$.

Cette différence d'interprétation implique la présence du symbole p dans C_i et plus précisément de $p(t)$ où t est une constante ou bien de $p(X)$ où X est une variable universellement quantifiée de sorte τ, d'après l'hypothèse 1. Si $p(t)$ apparaît dans C_i, d'après l'hypothèse 2, $C = p(t)$. Or, $\mathcal{I}'$ interprétant $p(t)$ par $\top$, on obtient une contradiction. Si $C_i = p(X)$, il est interprété à vrai par $\mathcal{I}'$, ce qui est une contradiction. Si $C_i = \neg p(X)$, alors, d'après l'hypothèse 3 du lemme, la clause unitaire $p(t)$ étant présente dans S et interprétée à $\top$ par $\mathcal{I}$, on a $\neg p(t)$ qui est interprétée à faux par $\mathcal{I}$, d'où la contradiction. □

Le corollaire suivant simplifie les formules (10.1), (10.2), (10.3) et (10.4) en y abstrayant le prédicat d'appartenance à s.

Corollaire 3 *Soit $f : s \to \{v_1, \ldots, v_n\}$ et φ une formule de la forme normale (9.7). Chaque obligation de preuve $\Phi \in \{(10.1), (10.2), (10.3), (10.4)\}$ d'invariance de φ est valide modulo SFUNC si et seulement si $\widehat{\Phi}'$ est valide modulo $\mathcal{BA}_s^e \cup \alpha'$ où*

- $\mathsf{F}(\Phi) = (\widehat{\Phi}, \alpha)$,
- $(\widehat{\Phi}', \alpha')$ *est la paire $(\widehat{\Phi}, \alpha)$ dans lequel chaque littéral $\mathsf{rd}(u_s, \_) = \mathsf{tt}$ (respectivement $\mathsf{rd}(u_s, \_) = \mathsf{ff}$) est remplacé par $\top$ (respectivement par $\bot$).*

PREUVE. On se limite à présenter la démonstration de ce corollaire pour une obligation de preuve de la forme (10.2). Soit Φ la formule obtenue en remplaçant φ par (9.7) dans (10.2). On pose alors $\mathsf{F}(\Phi) = (\widehat{\Phi}, \alpha)$. Un simple calcul montre que $\alpha = \{\delta_{u_s}\} = \{\forall X \,.\, \mathsf{rd}(u_s, X) = \mathsf{tt} \vee \mathsf{rd}(u_s, X) = \mathsf{ff}\}$ et $\widehat{\Phi}$ est

$$(\bigwedge_{1 \leqslant j < k \leqslant n} v_j \neq v_k \wedge (10.6) \wedge (10.7)) \Rightarrow (10.8) \tag{10.5}$$

où

$$(\forall X \,.\, \mathsf{rd}(u_s, X) = \mathsf{tt} \Rightarrow \bigvee_{1 \leqslant i \leqslant n} \mathsf{rd}(u_f, X) = v_i) \wedge (\forall X \,.\, \mathsf{rd}(u_s, X) = \mathsf{ff} \Rightarrow \mathsf{rd}(u_f, X) = \mathsf{null}) \tag{10.6}$$

est la traduction de $f \in s \to \{v_1, \ldots, v_n\}$,

$$\forall\, i_a, i_b\,.\,(\mathsf{rd}(u_s, i_a) = \mathsf{tt} \wedge \mathsf{rd}(u_s, i_b) = \mathsf{tt} \wedge i_a \neq i_b \wedge \mathsf{rd}(u_f, i_b) = v_p) \Rightarrow \\ \mathsf{rd}(u_f, i_a) \neq v_q) \tag{10.7}$$

est la traduction de (9.7) et

$$\forall\, i_p\,.\,(\mathsf{rd}(u_s, i_p) = \mathsf{tt} \wedge \mathsf{rd}(u_f, i_p) = v_l) \;\Rightarrow \\ (\forall\, i_{a'}, i_{b'}\,.\,(\mathsf{rd}(u_s, i_{a'}) = \mathsf{tt} \wedge \mathsf{rd}(u_s, i_{b'}) = \mathsf{tt} \wedge i_{a'} \neq i_{b'} \wedge \\ \mathsf{rd}(\mathsf{wr}(u_f, i_p, v_m), i_{a'}) = v_p) \Rightarrow \mathsf{rd}(\mathsf{wr}(u_f, i_p, v_m), i_{b'}) \neq v_q) \tag{10.8}$$

est la traduction de $\forall\, p\,.\,(p \in s \wedge f(p) = v_l) \Rightarrow (9.7)(f \lhd \{p \mapsto v_m\}/f)$.

Or, vérifier la validité de Φ modulo *SFUNC* revient à vérifier la non satisfaisabilité de $\neg\Phi$ modulo *SFUNC*, soit encore à vérifier la non satisfaisabilité de $\neg\widehat{\Phi}$ modulo $Ax(\mathcal{BA}_s^e) \cup \alpha$ par application de la conjecture 1.

La mise en CNF de α et de $\neg\widehat{\Phi}$ est l'ensemble de clauses S défini par

$$\begin{array}{l}
\{\mathsf{rd}(u_s, X) = \mathsf{tt} \vee \mathsf{rd}(u_s, X) = \mathsf{ff}\} \\
\cup \\
\left\{\bigwedge_{1 \leqslant j < k \leqslant n} v_j \neq v_k\right\} \\
\cup \\
\left\{\mathsf{rd}(u_s, X) \neq \mathsf{tt} \vee \bigvee_{1 \leqslant i \leqslant n} \mathsf{rd}(u_f, X) = v_i, \mathsf{rd}(u_s, X) \neq \mathsf{ff} \vee \mathsf{rd}(u_f, X) = \mathsf{null}\right\} \\
\cup \\
\{\mathsf{rd}(u_s, X) \neq \mathsf{tt} \vee \mathsf{rd}(u_s, Y) \neq \mathsf{tt} \vee X = Y \vee \mathsf{rd}(u_f, X) \neq v_p \vee \mathsf{rd}(u_f, Y) \neq v_q\} \\
\cup \\
\left\{\mathsf{rd}(u_s, \widetilde{\imath}_p) = \mathsf{tt}, \mathsf{rd}(u_f, \widetilde{\imath}_p) = v_l, \mathsf{rd}(u_s, \widetilde{\imath}_{a'}) = \mathsf{tt}, \mathsf{rd}(u_s, \widetilde{\imath}_{b'}) = \mathsf{tt},\right. \\
\quad \left. \widetilde{\imath}_{a'} \neq \widetilde{\imath}_{b'} \wedge \mathsf{rd}(\mathsf{wr}(u_f, \widetilde{\imath}_q, v_m), \widetilde{\imath}_{a'}) = v_p, \mathsf{rd}(\mathsf{wr}(u_f, \widetilde{\imath}_q, v_m), \widetilde{\imath}_{b'}) = v_q\right\}
\end{array}$$

où les variables en majuscules sont universellement quantifiées et les termes décorés par $\widetilde{}$ sont des constantes fraîches. Nous sommes alors dans les hypothèses du lemme 2 en posant p tel que $\sigma(p) = \text{INDEX}$ et $p(X) =_{def} (\mathsf{rd}(u_s, X) = \mathsf{tt})$ pour toute variable X de sorte INDEX. En construisant S' comme l'ensemble S dans lequel on remplace toutes les occurrences de $\mathsf{rd}(u_s, \_) = \mathsf{tt}$ (resp. de $\mathsf{rd}(u_s, \_) = \mathsf{ff}$) par $\top$ (resp. par $\bot$) et en remarquant que S' correspond à la mise en forme CNF de $\widehat{\Phi} \vee \alpha$, on peut conclure. □

On déduit de cette partie qu'une méthode basée sur la syntaxe des tableaux et qui traiterait directement ces abstractions serait appréciée. La partie suivante présente le langage de substitutions de tableaux qui résulte de l'abstraction des littéraux de la forme $\mathsf{rd}(u_s, \_) = \mathsf{tt}$ et $\mathsf{rd}(u_s, \_) = \mathsf{ff}$ lorsque u_s est la constante de type tableau représentant l'ensemble abstrait s.

10.2 Spécification de tableaux

On présente dans cette partie le langage des substitutions généralisées de tableaux.

$$
\begin{array}{rcl}
expr & ::= & array \mid val \mid index \\
array & ::= & \mathsf{wr}(array, index, val) \mid \mathsf{const}(val) \\
 & & \mid \mathsf{block}(a, \{(\mathsf{v}_1, v_1'), \ldots, (\mathsf{v}_n, v_n')\}) \\
 & & \mid a \mid \mathsf{z} \\
val & ::= & \mathsf{rd}(array, index) \mid \mathsf{v} \\
index & ::= & i \mid \mathsf{j}
\end{array}
$$

où $a \in V_{array}$, $\mathsf{z} \in \mathsf{C}_{array}$, $\mathsf{v} \in \mathsf{C}_{val}$, $v_1', \ldots, v_n'$ sont à valeur dans C_{val}, $i \in V_{index}$ et $\mathsf{j} \in \mathsf{C}_{index}$.

FIG. 10.1 – Expressions de tableaux.

$$\mathsf{rd}(\mathsf{const}(E), I) = E \tag{10.9}$$

$$\bigwedge_{1 \leq l \leq n} (\mathsf{rd}(A, I) = \mathsf{v}_l \Rightarrow \mathsf{rd}(\mathsf{block}(A, \{(\mathsf{v}_1, V_1), \ldots, (\mathsf{v}_n, V_n)\}), I) = V_l) \tag{10.10}$$

où les variables, en majuscule, sont universellement quantifiées, V, V_1, $\ldots$, V_n sont de sorte VALUE et à valeur dans C_{val}, A est de sorte ARRAY et I est de sorte INDEX.

FIG. 10.2 – Extension de la théorie $\mathcal{A}_s^e$

10.2.1 Expressions de tableaux

Si précédemment l'état général du système était mémorisé par une fonction totale, c'est ici directement une variable de sorte ARRAY qui effectue ce travail. Cette variable est indexée par un ensemble fini et non vide d'indices de sorte INDEX et elle prend ses valeurs dans l'ensemble des valeurs des états d'un protocole, abstrait par la sorte VALUE. Les indices de sorte INDEX ne sont distinguables que par leur nom considéré comme une constante de l'ensemble C_{index}.

Le langage des expressions est généré par la grammaire de la figure 10.1, paramétré par les trois ensembles disjoints et finis C_{array}, C_{val} et C_{index} de constantes de sorte ARRAY, VALUE et INDEX respectivement, et par deux ensembles disjoints V_{array} et V_{index} de variables de sorte ARRAY et INDEX respectivement. On suppose que l'ensemble des valeurs des états d'un protocole est fini et de cardinalité n. On a alors $\mathsf{C}_{val} = \{\mathsf{v}_1, \mathsf{v}_2, \ldots, \mathsf{v}_n\}$.

Le terme $\mathsf{const}(v)$ définit le tableau constant dont la valeur en chaque indice est le même élément v de sorte VALUE. Un tableau constant est typiquement utilisé lorsqu'il est nécessaire de placer tous les processus dans un même état, comme souvent lors de l'initialisation.

Dans le terme $\mathsf{block}(a, \{(\mathsf{v}_1, v_1'), \ldots, (\mathsf{v}_n, v_n')\})$, la sorte VALUE apparaît $2n$ fois. Le tableau $\mathsf{block}(a, \{(\mathsf{v}_1, v_1'), \ldots, (\mathsf{v}_n, v_n')\})$ est obtenu à partir de a en remplaçant chaque valeur v_l par v_l'. En d'autres termes, sa valeur à l'indice i est v_l' si et seulement si la valeur de a en i est v_l. Comme im qu'il simule au niveau des substitutions généralisées de tableaux, l'opérateur block est utilisé notamment pour traduire les actions de broadcast.

Toutes ces définitions sont formalisées par les axiomes de la figure 10.2, exprimés dans une

$$
\begin{array}{rcl}
subst & ::= & \mathsf{skip} \mid affec \mid choix \mid pred \Longrightarrow subst \\
affec & ::= & x := expr \mid x := expr \mid\mid affec \\
choix & ::= & subst[]subst \mid (@j \ . \ subst)
\end{array}
$$

FIG. 10.3 – Syntaxe des substitutions généralisées de tableaux.

$$
\begin{array}{rcl}
pred & ::= & \mathsf{p} \mid litt \mid pred \wedge pred \mid pred \vee pred \mid (quant\ j \ . \ pred) \\
litt & ::= & equa \mid \neg equa \\
equa & ::= & val = val \mid index = index \\
quant & ::= & \forall \mid \exists
\end{array}
$$

où p est une constante propositionnelle et j est une variable de sorte INDEX.

FIG. 10.4 – Logique équationnelle du langage des substitutions de tableaux.

logique équationnelle du premier ordre multi-sortes. La syntaxe des prédicats de cette logique est détaillée dans la partie suivante.

Par convention, on nomme $Ax(\mathcal{A}_s^c) = Ax(\mathcal{A}_s) \cup \{(10.9)\}$, $Ax(\mathcal{A}_s^{ce}) = Ax(\mathcal{A}_s^e) \cup \{(10.9)\}$ et $Ax(\mathcal{A}_s^{xe}) = Ax(\mathcal{A}_s^{ce}) \cup \{(10.10)\}$.

10.2.2 Opérations et prédicats

La figure 10.3 définit la partie opérationnelle du langage comme une simplification du langage de substitutions généralisées donné à la figure 4.2. Cette figure est paramétrée par l'élément syntaxique *expr* de cette partie. On précise de plus que $j \in V_{index}$ est seulement une variable de sorte INDEX. Ce langage est alors nommé langage des *substitutions de tableaux*.

Les prédicats considérés sont définis par l'élément syntaxique *pred* de la figure 10.4. La syntaxe des quantifications ($\forall$, $\exists$) et du choix borné @ ne mentionne pas la sorte quantifiée puisque celle-ci est toujours INDEX. Cette restriction est importante car elle conditionne les résultats de décidabilité de la partie 10.4.

Pour simplifier l'exposé de notre méthode de preuve dans la partie 10.4 on note que

- les substitutions de la forme if then else du langage de la figure 4.2 doivent être réécrits comme un choix borné entre deux substitutions gardées et
- les prédicats sont exprimés en NNF ; en d'autres termes, les négations ne concernent que les égalités (*equa*). Ainsi, chaque quantificateur ($\forall$, $\exists$) a la même polarité (universelle ou existentielle) que dans la forme prénexe du prédicat.

Dans la suite, on considère deux langages d'assertions. Le langage de tous les prédicats définis par l'élément syntaxique *pred* de la figure 10.4 et le langage restreint des *contraintes* définies comme les prédicats sans quantificateur universel et sans terme avec block.

La partie suivante montre que ce langage permet d'exprimer de manière concise les systèmes uniformes distribués.

$$
\begin{array}{rl}
s_{write} =_{def} & (@j \,.\, \mathsf{rd}(a,j) = \mathsf{e} \Longrightarrow s := \mathsf{wr}(a,j,\mathsf{m})) \\
s_{inv} =_{def} & (@j \,.\, \mathsf{rd}(a,j) = \mathsf{s} \Longrightarrow s := \mathsf{wr}(\mathsf{const(i)},j,\mathsf{e})) \\
s_{read} =_{def} & (@j \,.\, \mathsf{rd}(a,j) = \mathsf{i} \Longrightarrow \\
& \quad s := \mathsf{wr}(\mathsf{block}(a,\{(\mathsf{s},\mathsf{s}),(\mathsf{e},\mathsf{s}),(\mathsf{m},\mathsf{s}),(\mathsf{i},\mathsf{i})\}),j,\mathsf{s})
\end{array}
$$

FIG. 10.5 – Substitutions du MESI.

10.2.3 Actions des systèmes uniformes distribués

Pour décrire un état du système global, on considère la variable a de type ARRAY, dont le domaine de type INDEX est l'ensemble des identifiants des processus. La valeur de a à l'indice i est l'état du processus p_i d'identifiant i. Chaque action qui implique un, deux ou plusieurs processus se traduit globalement par une transformation du contenu du tableau a. Nous donnons la traduction des trois catégories d'actions dans le langage des substitutions de tableaux.

Une action interne modélisant la transition d'un processus p_i, qui passe d'un état v_l à un état v_m, éventuellement sous une condition P, se traduit par la substitution

$$(@p.\mathsf{rd}(a,i) = v_l \wedge P \Longrightarrow a := \mathsf{wr}(a,i,v_m))$$

où le processus p concerné est choisi de manière indéterministe.

Un rendez-vous qui synchronise une transition d'un processus p_i de v_l à v_m avec une transition d'un processus p_j de v_o à v_p, dépendant éventuellement de conditions notées P et P' se traduit par la substitution

$$
\begin{array}{l}
(@i \,.\, \mathsf{rd}(a,i) = v_l \wedge P \Longrightarrow \\
\quad (@j \,.\, \mathsf{rd}(a,j) = v_o \wedge P' \Longrightarrow a := \mathsf{wr}(\mathsf{wr}(a,i,v_m),j,v_p)))
\end{array}
$$

où P et P' sont des prédicats.

Une synchronisation par broadcast entre un processus p_i franchissant la transition de v_l à v_m éventuellement sous une condition P, tandis que tous les processus dans les états $v_1,\ldots,v_n$, passent respectivement dans les états $v'_1,\ldots,v'_n$, est représentée par la substitution :

$$
\begin{array}{l}
(@p \,.\, \mathsf{rd}(a,i) = v_l \wedge P \Longrightarrow \\
\quad a := \mathsf{wr}(\mathsf{block}(a,\{(v_1,v'_1),\ldots,(v_n,v'_n)\}),i,v_m)))
\end{array}
$$

Dans ce langage, le patron de propriétés énoncé à l'équation (3.4) s'écrit

$$\forall\, i_a, i_b \,.\, (i_a \neq i_b \wedge \mathsf{rd}(a,i_a) = v_p) \Rightarrow \mathsf{rd}(a,i_b) \neq v_q \tag{10.11}$$

Retour à l'exemple MESI. La partie opérationnelle du MESI est représentée par les substitutions de la figure 10.5. Celles-ci utilisent une seule variable $a \in V_{array}$. Le fait qu'un processus p_j soit dans l'état $t \in \mathsf{C}_{val} = \{\mathsf{m},\mathsf{e},\mathsf{s},\mathsf{i}\}$ est représenté par $\mathsf{rd}(a,j) = t$.

La substitution d'initialisation est alors $Source =_{def} a := \mathsf{const(i)}$. La propriété de cohérence de cache relative à une lecture (cf partie 3.2.2) est

$$\forall\, j_1, j_2 \,.\, (j_1 \neq j_2 \wedge \mathsf{rd}(a,j_1) = \mathsf{s}) \Rightarrow \mathsf{rd}(a,j_2) \neq \mathsf{m} \tag{10.12}$$

tandis que celle relative à une écriture est

$$\forall\, j_1, j_2 \,.\, (j_1 \neq j_2 \wedge \mathsf{rd}(a,j_1) = \mathsf{m}) \Rightarrow \mathsf{rd}(a,j_2) \neq \mathsf{m} \tag{10.13}$$

10.2.4 Nature des conditions d'évolution

Les obligations de preuve d'invariance dans le cadre de substitutions généralisées de tableaux sont définies comme les traductions de leurs homologues dans le cas de fonctions totales. On se place ainsi dans le cadre du semi-algorithme présenté à la figure 4.6. Comme on l'a déjà présenté à la partie 5.3, on remplace $I \Rightarrow J_i$ par $[Source]J_i$ où J_i est l'assertion candidate pour être invariante, et où l'ensemble des états initiaux est défini par une assertion I ou caractérisé par une substitution généralisée *Source* initialisant les variables d'état. Comme à la partie 10.1, on considère que la propriété J de cet algorithme s'écrit sous la forme

$$(f \in s \rightarrow \{v_1, \ldots, v_n\}) \wedge \varphi$$

où φ suit le patron de l'équation (10.11).

On détaille les deux classes d'obligations de preuve.

– l'assertion φ contient l'ensemble des états initiaux si et seulement si

$$B_{\{v_1,\ldots,v_n\}} \Rightarrow [Source]\varphi$$

est valide, où $B_{\{v_1,\ldots,v_n\}} =_{def} \left(\bigwedge_{1 \leqslant i < j \leqslant n} v_i \neq v_j\right)$, c'est à dire si et seulement si

$$B_{\{v_1,\ldots,v_n\}} \wedge \neg[Source]\varphi \tag{10.14}$$

n'est pas satisfaisable.

– φ est stable par la substitution S de tableaux si et seulement si l'obligation de preuve

$$\left(\left(\forall j \,.\, \bigvee_{1 \leqslant i \leqslant n} \mathsf{rd}(a, j) = v_i\right) \wedge B_{\{v_1,\ldots,v_n\}} \wedge \varphi\right) \Rightarrow [S]\varphi$$

est valide c'est à dire si et seulement si

$$\left(\forall j \,.\, \bigvee_{1 \leqslant i \leqslant n} \mathsf{rd}(a, j) = v_i\right) \wedge B_{\{v_1,\ldots,v_n\}} \wedge \varphi \wedge \neg[S]\varphi \tag{10.15}$$

n'est pas satisfaisable.

Par rapport au semi-algorithme de la figure 4.6, pour simplifier le raisonnement par la suite, φ est remplacé par $\varphi \wedge [S]\varphi$ à chaque itération du semi-algorithme.

La partie suivante montre comment éliminer le symbole fonctionnel block dans ces formules sans modifier la satisfaisabilité.

10.3 Gestion des diffusions multiples

Étant d'arité variable et nécessitant une axiomatisation spécifique, l'opérateur block est problématique. On donne donc dans cette partie une démarche permettant de le traduire dans une extension de $\mathcal{A}_s^{ce}$.

La figure 10.6 définit les fonctions $\mathcal{T}$ et $\mathcal{F}$ dont les signatures sont les suivantes : $\mathcal{T}$ transforme chaque terme de $\mathcal{A}_s^{xe}$ en une paire dont le premier membre est un terme de $\mathcal{A}_s^{ce}$ et le second un ensemble de formules exprimées dans $\mathcal{A}_s^{ce}$. $\mathcal{F}$ transforme chaque formule dont les termes

$$\begin{aligned}
\mathcal{T}(\mathsf{block}(a,\{(\mathsf{v}_1,\mathsf{v}'_1),\ldots,(\mathsf{v}_n,\mathsf{v}'_n)\})) &:= (\widetilde{a},\alpha_a \cup \bigcup_{1\leqslant l\leqslant n}\{\forall X\,.\,\mathsf{rd}(\widehat{a},X)=\mathsf{v}_l \\
&\qquad \Rightarrow \mathsf{rd}(\widetilde{a},X)=\mathsf{v}'_l\}) && (10.16)\\
\mathcal{T}(t) &:= (t,\emptyset) \text{ sinon} && (10.17)\\
\mathcal{F}(\varphi_1 \diamond \varphi_2) &:= (\widehat{\varphi_1} \diamond \widehat{\varphi_2}, \alpha_{\varphi_1}\cup\alpha_{\varphi_2}) && (10.18)\\
\mathcal{F}(\neg\varphi_1) &:= (\neg\widehat{\varphi_1}, \alpha_{\varphi_1}) && (10.19)\\
\mathcal{F}(t_1 = t_2) &:= (\widehat{t_1}=\widehat{t_2}, \alpha_{t_1}\cup\alpha_{t_2}) && (10.20)\\
\mathcal{F}(\forall j\,.\,\varphi_1) &:= (\forall j\,.\,\alpha'_1 \Rightarrow \widehat{\varphi_1}, \emptyset) && (10.21)\\
\mathcal{F}(\exists j\,.\,\varphi_1) &:= (\exists j\,.\,\alpha'_1 \wedge \widehat{\varphi_1}, \emptyset) && (10.22)
\end{aligned}$$

où a est un terme de sorte ARRAY tel que $\mathcal{T}(a) = (\widehat{a},\alpha_a)$, $\widetilde{a}$ est une constante fraîche de sorte ARRAY, v, v_1, …, v_n, v'_1, …, v'_n sont des constantes dans C_{val}, X est une variable de sorte INDEX, t est un terme de $\mathcal{A}_s$, $\diamond \in \{\vee,\wedge\}$, $\mathcal{F}(\varphi_i) = (\widehat{\varphi_i},\alpha_{\varphi_i})$, $\mathcal{T}(t_i) = (\widehat{t_i},\alpha_{t_i})$ pour $1 \leqslant i \leqslant 2$ et $\alpha'_1 = \bigwedge_{d\in\alpha_{\varphi_1}} d$

FIG. 10.6 – Règles de suppression du symbole block

sont dans $\mathcal{A}_s^{xe}$ en une paire dont le premier membre est une formule portant sur des termes de $\mathcal{A}_s^{ce}$ et le second un ensemble de formules exprimées dans $\mathcal{A}_s^{ce}$.

Le théorème suivant établit une relation d'équisatisfaisabilité entre une formule φ exprimée dans $\mathcal{A}_s^{xe}$ et le résultat $\mathcal{F}(\varphi)$ de la traduction de φ selon $\mathcal{F}$.

Théorème 9 *Soit φ une formule du langage des prédicats de la figure 10.4. Alors φ est satisfaisable modulo $\mathcal{A}_s^{xe}$ si et seulement si $\widehat{\varphi}$ est satisfaisable modulo $\mathcal{A}_s^{e} \cup (10.9) \cup \alpha$ où $\mathcal{F}(\varphi) = (\widehat{\varphi},\alpha)$.*

PREUVE. La preuve se fait par induction sur la structure de la formule.

Pour le seulement si ($\Rightarrow$), on considère un modèle $\mathcal{I} = (D, I)$ de φ et de $\mathcal{A}_s^{xe}$. On construit ensuite l'interprétation $\mathcal{I}' = (D', I')$ identique à $\mathcal{I}$ sur les termes de sorte INDEX et VALUE et telle $I'(\widetilde{a}) = I(\mathsf{block})(a,\{(\mathsf{v}_1,\mathsf{v}'_1),\ldots,(\mathsf{v}_n,\mathsf{v}'_n)\})$ si

$$\mathcal{T}(\mathsf{block}(a,\{(\mathsf{v}_1,\mathsf{v}'_1),\ldots,\mathsf{v}_n,(\mathsf{v}'_n)\})) = (\widetilde{a},\alpha_a \cup \bigcup_{1\leqslant l\leqslant n}\{\forall X\,.\,\mathsf{rd}(\widehat{a},X)=\mathsf{v}_l \Rightarrow \mathsf{rd}(\widetilde{a},X)=(\mathsf{v}'_l)\}).$$

$\mathcal{I}'$ interprète ainsi $\widehat{\varphi}$ à vrai par hypothèse sur $\mathcal{I}$. Comme chaque axiome de $\mathcal{A}_s^{xe}$ est interprété à vrai, par construction de $I(\widetilde{a})$, chaque α est interprété à vrai.

Pour le si ($\Leftarrow$), on considère le modèle $\mathcal{I}' = (D', I')$ de $\widehat{\varphi}$ et de $\mathcal{A}_s^{xe} \cup \alpha$. Pour construire un modèle de φ et de $\mathcal{A}_s^{xe}$, on considère l'interprétation de Herbrand $\mathcal{I}$ sur les domaines $\mathbb{H}_{\text{ARRAY}}$, $\mathbb{H}_{\text{INDEX}}$ et $\mathbb{H}_{\text{VALUE}}$. On pose alors

$$\begin{aligned}
I(=)(t_1,t_2) &= \top \text{ si } I'(=)(\widehat{t_1},\widehat{t_2}) = \top \text{ et si } I'(\alpha_1) = I'(\alpha_2) = \top\\
&\quad \bot \text{ sinon.}
\end{aligned}$$

où $\mathsf{T}(t_1) = (\widehat{t_1},\alpha_1)$ et $\mathsf{T}(t_2) = (\widehat{t_2},\alpha_2)$. Il est alors aisé de voir que φ et l'axiome (10.10) sont interprétés à vrai par $\mathcal{I}$ et de conclure. □

$$
\begin{array}{rcl}
\mathsf{T}(x) & := & \big(x', \emptyset\big) \text{ si } x \text{ est une constante ou une variable} \\
 & & \text{de sorte ELEM, STATEVALUE ou FUNC} \\
\mathsf{T}(f \lhd \{e \mapsto v\}) & := & \Big(\mathsf{wr}(\widehat{f}, i, \widehat{v}), \alpha_f \cup \alpha_v\Big) \\
\mathsf{T}(f(e)) & := & \Big(\mathsf{rd}(\widehat{f}, i), \alpha_f \cup\Big) \\
\mathsf{T}(s \times \{v\}) & := & (\mathsf{const}(\widehat{v}), \alpha_s \cup \alpha_v) \\
\mathsf{T}(\mathsf{im}(f, v_1, v_1', \ldots, v_n, v_n')) & := & \big(u, \alpha_f \cup (\bigcup_{1 \leqslant i \leqslant n} (\alpha_{v_i} \cup \alpha_{v_i'})) \cup \\
 & & \bigcup_{1 \leqslant i \leqslant n} \{\forall X \,.\, \mathsf{rd}(\widehat{f}, X) = \widehat{v_i} \Rightarrow \mathsf{rd}(u, X) = \widehat{v_i'}\}\big) \\
\mathsf{T}(t) & := & t \text{ pour tout autre terme } t \\
 & & \\
\mathsf{F}(\forall\, y \,.\, \phi_1) & := & \Big(\forall\, \widehat{y} \,.\, \alpha' \Rightarrow \widehat{\phi_1}, \emptyset\Big) \ y \text{ est une variable} \\
\mathsf{F}(\exists\, y \,.\, \phi_1) & := & \Big(\exists\, \widehat{y} \,.\, \alpha' \wedge \widehat{\phi_1}, \emptyset\Big) \ y \text{ est une variable} \\
\mathsf{F}(\neg \phi_1) & := & \Big(\neg\, \widehat{\phi_1}, \alpha\Big) \\
\mathsf{F}(\phi_1 \circ \phi_2) & := & \Big(\widehat{\phi_1} \circ\ \widehat{\phi_2}, \alpha_1 \cup \alpha_2\Big) \text{ pour } \circ \in \{\vee, \wedge, \Rightarrow \Leftrightarrow\} \\
\mathsf{F}(e_1 = e_2) & := & (\widehat{e_1} = \widehat{e_2}, \alpha_{e_1} \cup \alpha_{e_2}) \\
\mathsf{F}(f \in s \to \{v_1, \ldots, v_n\}) & := & (\forall X \,.\, \bigvee_{1 \leqslant i \leqslant n} \mathsf{rd}(\widehat{f}, X) = \widehat{v_i}, \alpha_s \cup \bigcup_{1 \leqslant i \leqslant n} \alpha_{v_i}) \\
 & & \text{si } f \text{ est une variable de sorte FUNC} \\
\mathsf{F}(\varphi) & := & \top \text{ sinon}
\end{array}
$$

où x' est une constante (resp. une variable) fraîche de sorte INDEX si x est une constante (resp. une variable) de sorte ELEM, x' est une constante (resp. une variable) fraîche de sorte ARRAY si x est une constante (resp. une variable) fraîche de sorte FUNC, x' est une constante sorte VALUE si x est une constante de sorte STATEVALUE, X est de sorte INDEX, $\mathsf{T}(k) = (\widehat{k}, \alpha_k)$, pour $k \in \{f, v, s, v', x, e_1, e_2\}$, $\mathsf{T}(e) = (i, \emptyset)$, $\mathsf{T}(y) = (\widehat{y}, \emptyset)$, $\mathsf{F}(\phi_i) = (\widehat{\phi_i}, \alpha_i)$ pour $i = 1, 2$ et $\alpha' = \bigwedge_{d \in \alpha_1} d$.

FIG. 10.7 – T et F : traduction abstraite de formules avec fonctions totales

La figure 10.7 remonte au niveau des formules avec fonctions totales les simplifications de traduction exposées et démontrées dans cette partie. Ces règles sont implantées dans un des outils détaillé au chapitre 11.

Ainsi, seules six classes de formules sont traduites dans $\mathcal{A}_s^{ce}$: les autres sont abstraites par $\top$. Pour optimiser ensuite la traduction des termes, seules sept classes de termes sont traduites dans la théorie $\mathcal{A}_s^{ce}$; les autres sont laissés en l'état car ils interviennent dans des formules qui sont ensuite abstraites par traduction.

La partie suivante montre l'intérêt des simplifications apportées par ce chapitre en termes de procédures de décision.

10.4 Décider les conditions d'évolution

La partie précédente a montré comment réécrire une formule contenant le symbole fonctionnel block en préservant sa satisfaisabilité. On s'intéresse à présent à la décidabilité de la satisfaisabilité modulo $\mathcal{A}_s^{ce}$ de formules qui ne contiennent plus cet opérateur. Cette décidabilité n'est étudiée que pour les formules (10.14) et (10.15).

On montre dans la partie 10.4.1 que l'axiome d'extensionnalité peut être supprimé. Des résultats de décidabilité sont ensuite établis d'abord pour les contraintes (partie 10.4.2), puis pour une classe plus générale de formules (partie 10.4.3). Dans ces deux cas, des restrictions sur les termes en block sont posées.

10.4.1 Suppression de l'axiome d'extensionnalité

Les résultats de décidabilité établis sur $\mathcal{A}_s^e$ nécessitent que l'axiome d'extensionnalité soit supprimé. On montre dans cette partie la faisabilité de ce pré-requis.

Théorème 10 *Soit φ une formule du langage des prédicats de la figure 10.4 sans symbole fonctionnel* block. *Alors φ est satisfaisable modulo $\mathcal{A}_s^{ce}$ ssi φ est satisfaisable modulo $\mathcal{A}_s^{c}$.*

PREUVE. La preuve est une application de [ARR03, théo. 7.1], en remarquant que φ ne contient pas d'égalités entre deux termes de sorte ARRAY et donc que la démarche présentée dans [ARR03, Sec. 7] s'étend aux tableaux constants défini par (10.9) (cf.[ARR03, p. 153]).□

On en déduit le corollaire suivant :

Corollaire 4 *Soit une formule ψ de la forme* (10.14) *ou* (10.15) *où φ est une formule exprimée dans le langage des prédicats de la figure 10.4 et S est une substitution de tableaux. La formule ψ est alors satisfaisable modulo $\mathcal{A}_s^{xe}$ ssi $\widehat{\psi}$ est satisfaisable modulo $\mathcal{A}_s^{c} \cup \alpha$ où $\mathcal{F}(\psi) = (\widehat{\psi}, \alpha)$.*

PREUVE. Il convient d'abord de constater que le langage des prédicats de la figure 10.4 est stable par calcul de plus faible précondition appliqué aux substitutions de tableaux. On poursuit en remarquant que la fonction $\mathcal{F}$ n'introduit pas d'égalités entre termes de sorte ARRAY. On conclut en appliquant successivement les théorèmes 9 et 10. □

Les raisonnements qui suivent visent à décider de la satisfaisabilité modulo $\mathcal{A}_s^c$ de formules exprimées dans le langage des contraintes, c'est à dire celui des prédicats sans quantificateur universel ni opérateur block.

10.4.2 Cas des contraintes

Le théorème suivant est à la base de la preuve de décidabilité du cas des contraintes :

Théorème 11 ([ARR03, Lem. 7.4]) *Soit S une conjonction de littéraux sans quantificateurs. S est satisfaisable modulo $\mathcal{A}_s^c$ si et seulement si est S est satisfaisable modulo $\mathcal{A}^c$ qui est la théorie engendrée par $Ax(\mathcal{A}_s^c)$ en oubliant les sortes.*

On en déduit le corollaire suivant :

Corollaire 5 *Si φ est une contrainte, alors la satisfaisabilité de φ modulo $\mathcal{A}_s^c$ est décidable par superposition.*

PREUVE. La première étape consiste à skolémiser φ pour obtenir la formule ψ, qui est satisfaisable modulo $\mathcal{A}_s^c$ (noté aussi $\mathcal{A}_s^c$-satisfaisable) si et seulement si φ l'est. Pour vérifier si ψ est $\mathcal{A}_s^c$-satisfaisable, on construit sa forme normale disjonctive, θ, et on applique la procédure de décision par superposition précédente pour chaque conjonction de littéraux de θ. On remarque que la skolemization de φ n'introduit pas de symbole fonctionnel non constant puisque φ ne contient pas de quantificateur universel. □

Pour une théorie $\mathcal{T}$ différente de $\mathcal{A}_s^c$, le calcul par superposition fournit une procédure de semi-décision qui termine si φ n'est pas $\mathcal{T}$-satisfaisable. Au cas où la formule φ est $\mathcal{T}$-satisfaisable, la terminaison n'est pas garantie.

Théorème 12 *Si le prédicat φ est sans* block *et ne contient que des quantificateurs universels, si les gardes de la substitution S sont toutes des contraintes et si S ne contient pas de symbole* block*, alors $[S]\,\varphi$ ne contient que des quantificateurs universels sans le symbole fonctionnel* block*. La satisfaisabilité de $\neg[S]\,\varphi$ modulo $\mathcal{A}_s^c$ est alors décidable par superposition.*

PREUVE. De la figure 4.5, on déduit que les quantificateurs dans $[S]\varphi$ ne proviennent que de φ, de l'application $[S]$ sur un @ de S ou sur une garde de S. Tous ces cas engendrent des quantifications universelles lorsqu'on les exprime en NNF. On en déduit que $\neg[S]\,\varphi$ est une contrainte, et donc que sa satisfaisabilité est décidable par superposition (corollaire 5). L'absence du symbole block est garantie par son absence dans φ et dans S. □

Dans le contexte de substitutions de tableaux, le semi-algorithme de la figure 4.7, qui vérifie uniquement qu'une propriété de sûreté n'est pas établie, ne vérifie que la satisfaisabilité de (10.14). Le corollaire suivant établit des conditions garantissant que ce semi-algorithme ne va pas diverger lors d'un appel au prouveur par superposition.

Corollaire 6 *Dans la formule (10.14) si le prédicat φ est sans* block *et ne contient que des quantificateurs universels, si les gardes des substitutions Source, $S_1, \ldots, S_p$ sont des contraintes, et si ni Source, ni S_1, ..., ni S_p ne contiennent de symbole* block*, alors la satisfaisabilité de* (10.14) *est décidable par superposition à chaque itération.*

PREUVE. Dans les hypothèses du corollaire, par induction sur le nombre d'itérations, on montre que la nouvelle valeur de φ demeure universellement quantifiée et sans le symbole fonctionnel block : puisqu'elle construite comme la conjonction entre son ancienne valeur et la formule $[S]\,\varphi$, la preuve est une conséquence du théorème 12. A nouveau d'après le théorème 12, $\neg[Source]\varphi$ est une contrainte à chaque itération et sa satisfaisabilité est donc décidable par superposition. □

On reprend l'exemple du MutEx (cf. partie 3.1.4) en guise d'application de ce corollaire. Les substitutions qui correspondent à ce système sont données dans la figure 10.8 ; elles ne définissent que des actions locales gardées sur les variables $a \in V_{array}$ et $t \in V_{index}$.

$$\begin{aligned}
s_{ask} &=_{def} (@p\ .\mathsf{rd}(a,p) = \mathsf{sl} \Longrightarrow a := \mathsf{wr}(a,p,\mathsf{r})) \\
s_{get} &=_{def} (@p\ .\mathsf{rd}(a,p) = \mathsf{r} \wedge \mathsf{rd}(a,t) = \mathsf{sl} \Longrightarrow t := p) \\
s_{active} &=_{def} (@p\ .t = p \wedge \mathsf{rd}(a,p) = \mathsf{r} \Longrightarrow a := \mathsf{wr}(a,p,\mathsf{a})) \\
s_{sleep} &=_{def} (@p\ .\mathsf{rd}(a,p) = \mathsf{a} \Longrightarrow a := \mathsf{wr}(a,p,\mathsf{sl}))
\end{aligned}$$

FIG. 10.8 – Substitutions du MutEx.

L'ensemble initial est caractérisé par l'assertion

$$Source_{\text{MutEx}} =_{def} @q\ .\ t := q \ ||\ a := \mathsf{const}(\mathsf{sl})$$

précisant que tous les processus sont initialement endormis et que la variable t est initialisée avec un indice quelconque associé à un processus. Une initialisation erronée pourrait être

$$Source_{\text{bug}} =_{def} @q\ .\ t := q$$

qui n'initialise pas le tableau. Lorsqu'on écrit la propriété d'exclusion mutuelle sous la forme

$$\forall\, p_1, p_2\ .\ (p_1 \neq p_2 \wedge \mathsf{rd}(a,p_1) = \mathsf{a}) \Rightarrow \mathsf{rd}(a,p_2) \neq \mathsf{a}$$

on entre dans les conditions du corollaire.

Cependant, ne savoir décider que des contraintes sans le symbole fonctionnel block n'est pas suffisant pour prétendre décider dans toutes les situations de la satisfaisabilité des conditions d'évolution. Par exemple, cette restriction ne permet pas de décider de la satisfaisabilité de la condition (10.15).

En effet, si φ contient au moins un quantificateur universel, alors la satisfaisabilité de (10.15) n'est pas nécessairement détectée. La partie suivante établit un résultat de décidabilité pour une plus grande classe de formules.

10.4.3 Procédures de décision

La procédure de décision donnée ici est basée sur le corollaire 2, énoncé au chapitre 2 et appliqué au langage des expressions de tableaux sans l'opérateur block.

Théorème 13 *Soit une formule close φ de $\mathcal{A}_s^c$ dans laquelle aucune quantification existentielle n'est sous la portée d'une quantification universelle. Alors la satisfaisabilité de φ modulo $\mathcal{A}_s^c$ est décidable.*

PREUVE. On sait que $\sigma(\mathsf{rd}) = (\text{ARRAY}, \text{INDEX}, \text{VALUE})$, que $\sigma(\mathsf{wr}) = (\text{ARRAY}, \text{INDEX}, \text{VALUE}, \text{ARRAY})$ et que $\sigma(\mathsf{const}) = (\text{VALUE}, \text{ARRAY})$ et

On constate donc qu'il n'y a pas de symbole fonctionnel d'arité $\geqslant 2$ se terminant par INDEX. Ainsi les conditions d'application du corollaire 2 sont réunies.

La skolemization de φ ne peut introduire que des constantes puisque cette dernière ne contient pas de quantificateur existentiel sous la portée d'un quantificateur universel. L'hypothèse du corollaire 2 portant sur les sortes des symboles fonctionnels est donc toujours respectée après cette étape de skolemization.

On nomme ψ la forme de Skolem de φ écrite sous la forme prénexe ; ψ est

$$\forall\, x_1 \ldots \forall\, x_k \ .\Phi(x_1, \ldots, x_k)$$

pour $x_1, \ldots, x_k$ ($k \geqslant 0$) des variables de sorte INDEX et $\Phi(x_1, \ldots, x_k)$ une formule sans quantificateur. La preuve se fait alors en suivant le corollaire 2 par induction sur k. □

Considérons à présent la décidabilité des conditions d'évolution (10.14) et (10.15).

Corollaire 7 *Supposons que φ ne contienne que des quantifications universelles et que toutes les gardes des substitutions Source, $S_1, \ldots, S_p$ soient des prédicats clos dans lesquels aucune quantification existentielle n'est sous la portée d'un quantificateur universel. Alors la satisfaisabilité de* (10.14) *est décidable à chaque itération pour tous les semi-algorithmes du chapitre 4.*

PREUVE. D'après la figure 4.5, on remarque que le calcul de $[S]\varphi$ ne peut introduire que des quantificateurs provenant de @ dans S, ou des gardes de S, ou directement de φ. Dans tous les cas, seuls des quantificateurs universels sont introduits pendant ce calcul. On remarque de plus que la traduction du symbole block de (10.14) n'introduit que des quantificateurs universels (éventuellement sous la portée de quantificateurs existentiels). Il est ainsi aisé de voir, par induction sur le nombre d'itérations, que la condition d'atteignabilité (10.14) n'a pas de quantificateur existentiel sous un quantificateur universel. Le théorème 13 permet alors de conclure. □

Corollaire 8 *Supposons que φ soit universellement quantifiée et que toutes les gardes des substitutions $S_1, \ldots, S_p$ soient des prédicats clos, sans* block *et sans quantificateur universel. Alors la validité de la condition* (10.15) *est décidable à chaque itération.*

PREUVE. Comme dans la preuve précédente, on constate par induction qu'à chaque itération, les prédicats φ et $[S]\varphi$ ne contiennent que des quantificateurs universels. La traduction des block de $[S]\varphi$ n'introduit aussi que des quantifications universelles. La traduction de (10.15) satisfait les conditions du théorème 13 et sa satisfaisabilité est décidable.

Retour à l'exemple MESI. Les propriétés (10.12) et (10.13), la substitution *Source* et les gardes des substitutions de l'exemple du MESI satisfont les conditions des corollaires 7 et 8. Ainsi la vérification de toutes les conditions dans les semi algorithmes d'atteignabilité arrière est décidable pour la propriété de cohérence de cache considérée.

10.5 Résumé

Ce chapitre a présenté les systèmes définis à l'aide du langage des substitutions de tableaux. Ce langage présente deux avantages : d'un point de vue syntaxique, il permet d'exprimer de manière concise les systèmes distribués se synchronisant par rendez-vous ou par broadcast ; d'un point de vue pratique, il permet un déchargement à moindre frais des conditions d'évolution du calcul du point fixe.

La seconde moitié du chapitre s'est intéressée à la réécriture dans la théorie $\mathcal{A}_s^e$ de l'opérateur block de broadcast puis a exhibé des cas où la décidabilité des conditions d'évolution est garantie. Celle-ci dépend de la présence ou non de cet opérateur.

Le chapitre suivant présente les prototypes développés et une panoplie d'exemples de systèmes uniformes distribués permettant de valider les idées énoncées dans ce travail de thèse.

Chapitre 11

Développements et expérimentations

Le verbe expérimenter possède au moins deux sens d'après le dictionnaire de l'Académie Française : "*EXPÉRIMENTER. v. tr. 1. Vérifier expérimentalement les qualités de [...] 2. SC. Procéder à une expérience en vue d'analyser ses conséquences.*".

Dans une perspective d'ingénieur, il est utile de fournir un outil complet appliquant les meilleures idées d'une réflexion théorique : l'efficience de la méthode est alors mesurable par des temps de calcul, par la pertinence des verdicts donnés à l'utilisateur...

Depuis une perspective d'informaticien, pour un travail théorique donné, il est aussi excitant de trouver une démarche de conception permettant un développement fiable et efficace et assurant (autant que possible) la correction de l'implantation. Ces phases d'analyse et d'implantation font très souvent ressortir ce qu'on pensait être un détail technique et qui nécessite en fait, pour être traité autrement que par un patch, un important travail théorique. On en arrive alors à la vue du théoricien.

Depuis une perspective de théoricien, l'expérimentation est capitale à plusieurs titres : elle permet de mettre en avant les lacunes, même infimes, du travail théorique ceci, soit en exhibant une difficulté d'implantation, soit en proposant des résultats non conformes avec ce qui était attendu. Ces lacunes sont alors autant de perspectives de recherches théoriques à explorer pour compléter la démarche. De même, des résultats non conformes aux prévisions, et sous réserve que l'implantation soit correcte, permettent d'invalider des conjectures. L'expérimentation est aussi un moyen d'obtenir rapidement une batterie d'exemples, en l'occurrence ici des formules exprimées en logique équationnelle. L'analyse de ces formules issues de cas pratiques dans un objectif de mise sous une certaine forme normale permet alors d'orienter les travaux de recherche vers la décidabilité de ces formes normales.

Le chapitre s'organise comme suit : La partie 11.1 présente le prototype barvey qui pose les bases de la traduction d'obligations de preuve ensemblistes en logique équationnelle. Des considérations techniques visant à garantir la qualité du logiciel sont énoncées en partie 11.2. La partie 11.3 présente une collection de protocoles sur lesquels l'outil a été appliqué. La partie 11.4 donne les résultats numériques des expérimentations menées sur ces protocoles. Un résumé conclut ce chapitre.

11.1 barvey : une étude de faisabilité

Cette partie présente Barvey [CDGR04], le premier prototype déchargeant dans haRVey les obligations de preuve ensemblistes de cohérence d'une machine abstraite B. Ce prototype sert

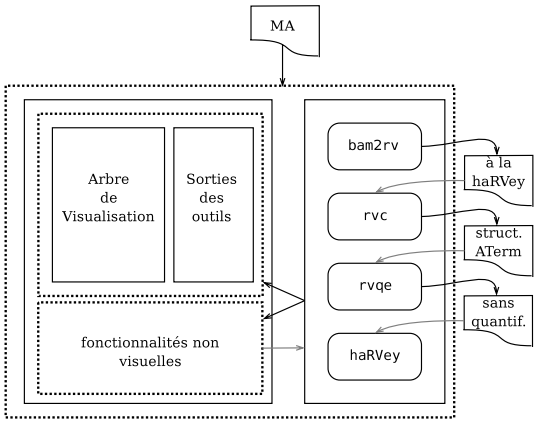


FIG. 11.1 – Architecture de Barvey

à valider l'intuition que la méthode de traduction en logique équationnelle est réaliste même pour des spécifications de grande taille. Cette partie présente en premier lieu les caractéristiques de l'outil, précise ensuite sommairement comment l'utiliser, donne quelques résultats expérimentaux et se termine par un bilan de cette expérience.

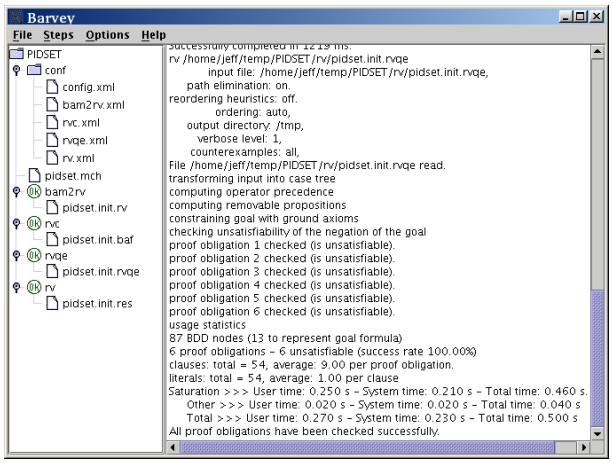
11.1.1 Bref aperçu de l'outil

Ce prototype est composé de deux modules (figure 11.1) qui sont la chaîne d'outils nécessaires à la gestion de la preuve (partie droite de la figure) et l'interface d'exécution (partie gauche de la figure). En plus des modules de haRVey(rvc et rv), ce prototype repose sur deux modules bam2rv et rvqe :

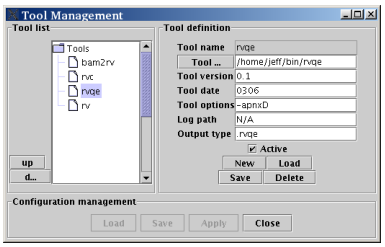
- bam2rv[6] prend en entrée une machine abstraite ensembliste et génère l'ensemble des OPs en logique équationnelle du premier ordre (cf figure 8.4 et 8.5), à partir du prédicat avant après correspondant à chaque substitution $S \in \mathit{Substs}$ (cf partie 4.2.2).
- rvc est un compilateur interne à haRVey. Il traduit chacune des OPs dans le langage des ATerm, structure optimisée pour la taille des formules.
- rvqe[7] gère alors les quantificateurs présents dans la formule pour la rendre traitable par haRVey en implantant des procédures de réduction de portée des quantificateurs, de skolémisation et de déplacement des sous-formules quantifiées résiduelles dans la théorie (cf figures 7.2, 7.3 et 7.4).
- rv, qui est le prouveur de haRVey, décide de la validité de chaque formule qui lui est fournie selon la démarche détaillée au chapitre 7.

L'interface (figure 11.2(a)) contient un espace de représentation du projet (sous la forme d'un arbre à gauche et d'une fenêtre de sortie texte à droite) et un ensemble de fonctionnalités accessibles au travers d'un menu.

[6] http://lifc.univ-fcomte.fr/~giorgett/Rech/Software/bam2rv/index.html
[7] http://lifc.univ-fcomte.fr/~couchot/rvqe/



(a) Interface utilisateur



(b) Interface de gestion des outils

FIG. 11.2 – Barvey

11.1.2 Utilisation de l'outil

Après avoir défini la machine abstraite à vérifier, chaque outil peut être successivement activé en utilisant Steps/Execute next et ce jusqu'à terminer la chaîne. Pour chaque exécution, l'interface graphique donne à la fois une synthèse visuelle (sous la forme d'une icône) du résultat dans la fenêtre de gauche et, de manière plus précise, les messages de sortie de l'outil qui s'exécute dans la fenêtre de droite. L'utilisateur peut ainsi obtenir des informations complémentaires comme les durées d'exécution, la nature de l'échec...

Une interface de gestion des outils (figure 11.2(b)) a été intégrée pour permettre d'ajouter ultérieurement d'autres outils, d'ordonner ceux-ci et d'en préciser la configuration. Des options peuvent être ainsi passées aux outils internes. L'ensemble est sauvegardé au format XML.

11.1.3 Résultats expérimentaux

Le tableau de la figure 11.3 présente des durées (en secondes[8]) de vérification sur Barvey et sur l'Atelier-B. Le modèle du "Robot" est un dispositif de transport de pièces et "T=1" est un protocole de communication entre une carte à puce et son lecteur. Ces deux modèles sont sans paramètres et donc finis. Détaillé à la partie suivante, l'algorithme du PidSet est un protocole d'exclusion mutuelle paramétré.

Le modèle confidentiel (sous contrat) du "SCard" spécifie un terminal de lecture de carte à puce développé avec un partenaire industriel. La spécificité du modèle réside (*i*) dans la structure de l'unique opération qui possède 31 *if then else* emboîtés jusqu'à une profondeur de 8 (*ii*) dans le nombre et la taille des ensembles énumérés : il y a 9 ensembles énumérés dont un contenant 91 éléments distincts.

On remarque que barvey est beaucoup plus lent que l'Atelier-B, en grande partie à cause des prototypes bam2rv et rvqe, sur les deux exemples de taille finie (T=1 et robot) et 10 fois

[8]Mesures effectuées sur un centrino 1.5Ghz sous Linux avec 768 MO de RAM.

Modèle	bam2rv	rvqe	haRVey	Total	Atelier-B
Robot	0.53	1.28	1.30	3.11	0.1
T=1	0.64	1.6	8.83	11.2	0.46
PidSet	0.48	1.41	2.91	4.8	0.58
SCard	118	2.8	18.2	138.4	Out of memory

FIG. 11.3 – Résultats expérimentaux

plus lent pour la spécification paramétrée PidSet. Par contre, barvey termine pour chacun des exemples, ce qui n'est pas le cas de l'Atelier-B qui génère un dépassement de mémoire pour "SCard". L'introduction de la structure prédicative ite (cf partie 2.1 et 7.1) est pour beaucoup dans cette réussite puisqu'elle évite l'explosion combinatoire due aux imbrications des if then else.

Pour l'analyse des protocoles de communication, qui nécessitent plus d'appels au prouveur qu'une simple vérification de cohérence, on peut faire les remarques suivantes :

- développé en Java, rvqe est un goulet d'étranglement ; son action a été intégrée au compilateur rvc à partir de la version 0.4 de haRVey ;
- les obligations de preuve construites à l'aide des prédicats avant après ont été abandonnées au profit des OPs engendrées par calcul [] de plus faible précondition (cf partie 4.3) ;
- les structures de données ensemblistes ont été abandonnées au profit des fonctions totales (cf chapitre 9) introduisant des obligations de preuve de taille plus réduite.

La partie suivante présente l'outil qui intègre ces remarques pour effectuer la vérification d'invariant sur des systèmes uniformes distribués.

11.2 Démarche d'implantation

Cette partie présente le deuxième prototype développé pour vérifier des systèmes uniformes distribués. La première partie présente les flux d'informations internes à l'outil, la deuxième montre comment une démarche de génie logiciel a été mise en place pour garantir un développement de qualité. Les options d'utilisation du prototype sont détaillées en troisième partie.

11.2.1 Flux dans l'outil

Le prototype est conçu pour vérifier la cohérence de machines B avec fonctions totales. Lorsque l'invariant n'est pas établi, il entreprend un calcul de plus faible invariant.

Il implante pour cela les différents semi-algorithmes présentés au chapitre 4, chacun de ces algorithmes étant activable par un paramètre passé en ligne de commande (cf partie 11.2.3). Par défaut, c'est le semi-algorithme de la figure 4.8 qui est activé. A chaque itération et pour un ensemble $\mathcal{S}$ de substitutions, un ensemble d'obligations de preuve est construit à l'aide de l'opérateur de plus faible précondition $[S]$ (cf équations (5.1) et (5.3)).

Chaque obligation de preuve est traduite en logique équationnelle. Par défaut c'est la traduction dans la théorie $\mathcal{BA}_s^e$ des tableaux de booléens qui est retenue et qui implante les règles des figures 8.4, 8.5 et 9.4. Un paramètre passé en ligne de commande permet d'activer l'abstraction vue au 10 précédent et d'exécuter l'implantation des règles données à la figure 10.7.

La version équationnelle de l'obligation de preuve est fournie à haRVey qui en décide la validité. Le calcul de point fixe se poursuit en fonction du verdict donné par le prouveur. Lorsque l'invariant J_M a pu être calculé (cf partie 4.4), il est donné à l'utilisateur qui en déduit que la propriété de sûreté correspondante est établie. Dans le cas où l'outil s'arrête en raison de la non validité d'une condition de la forme (5.1), l'utilisateur est informé de l'existence d'une trace qui mène à un état violant la propriété. On note que la taille de l'invariant retourné par l'outil dépend du semi-algorithme utilisé : si la localité est utilisée, l'invariant est naturellement de plus petite taille.

La partie suivante s'intéresse aux structures de données internes à l'outil et montre comment une démarche de génie logiciel a été mise en place pour garantir la qualité du développement.

11.2.2 Structures d'arbres abstraits

Issu de l'outil CLPS-B [BLP02], le parseur utilisé génère un arbre de syntaxe abstrait (AST) représentant en mémoire l'intégralité du contenu de la machine B analysée. Le patron de conception de *Visitor* est retenu pour développer efficacement les traitements à effectuer sur cet AST. L'architecture est la suivante :

- le visiteur `ConsistancyPOGenerator` récupère dans l'AST le prédicat I de la clause INVARIANT et l'ensemble des substitutions de la machine ; il engendre alors les obligations de preuve en invoquant :
- le visiteur `WeakestPreconditionApplication` qui, pour une substitution généralisée S et un prédicat Q donné, calcule $[S]Q$ inductivement selon S ;
- le visiteur `RVReadWriteGenerator` (resp. `RVAbstractReadWriteGenerator`) assure la traduction de l'obligation de preuve en logique équationnelle sans abstraction (resp. avec abstraction) ;
- lorsqu'il est invoqué, le visiteur `FormulaPrinterVisitor` traduit textuellement tout prédicat B stocké sous la forme d'un AST

Il peut paraître paradoxal d'utiliser le patron de conception de visiteur : un visiteur doit définir une méthode de visite pour tous les nœuds de l'arbre, même s'il n'opère en fait que sur une infime partie de ceux-ci : le visiteur qui implante par exemple le calcul de plus faible précondition [] ne visite que les nœuds de la famille des substitutions généralisées auxquelles il s'applique. L'idée naturelle consiste alors à ne développer que la partie propre à chaque visiteur, la partie laissée inchangée étant obtenue à coût nul en héritant d'un visiteur générique. Nous avons construit automatiquement ce visiteur générique par analyse syntaxique du fichier `jjt` ayant généré le parseur B, nous mettant à l'abri d'une modification éventuelle du parseur en question.

11.2.3 Utilisation

A partir d'une machine B `machine.mch` dont on cherche à vérifier si la clause INVARIANT est réellement un invariant, on utilise la syntaxe suivante pour lancer le prototype

```
java -jar reacha.jar [OPTION...] machine.mch
```

où `[OPTION...]` contient éventuellement une ou plusieurs options détaillées à la figure 11.4. La vérification locale des prédécesseurs (option `-d2`) impose le calcul local des prédécesseurs (option `-d`) : la première option implante en effet un algorithme qui est un raffinement de l'algorithme correspondant à la seconde option.

option	sens	détail
`-nun`	Détection unique de violation de propriété	Figure 4.7
`-d`	Calcul des prédécesseurs locaux	Figure 4.9
`-d2`	Vérification locale de l'implication (impose `-d`)	Figure 4.10
`-a`	Traduction abstraite des OPs	Figure 10.7
`-i`	Exhibe l'invariant inductif	Chapitre 4

FIG. 11.4 – Options du prototype reacha

Retour à l'exemple MESI. On conclut cette partie en donnant à la figure 11.5 la sortie de l'exécution du prototype sur l'exemple du MESI, pour lequel l'invariant inductif a été indenté pour plus de lisibilité. L'option de vérification locale des prédécesseurs étant activée, le calcul local des prédécesseurs est implicite.

Les lignes

- 13 à 17 informent que la propriété d'invariant peut être découpée en deux propriétés P_0 et P_1.
- 19 à 21 informent qu'à la première itération, l'opération *write* viole les deux propriétés.
- 23 à 24 informent qu'à la seconde itération, l'opération *write* viole la propriété $[write]P_1$
- 26 à 30 informent qu'à la troisième itération l'invariant construit par cette méthode de renforcement est inductif
- 33 à 59 donnent alors l'invariant inductif.

La partie suivante présente une collection d'exemples auxquels la méthode a été appliquée.

11.3 Études de protocoles

Cette partie présente six protocoles qui ont été traités par la présente étude. Présentés en premier, les protocoles du PidSet, du MuxSem, du MutEx et de Dijkstra relèvent de la classe des algorithmes d'exclusion mutuelle. Les deux derniers protocoles, celui de l'Université de l'Illinois et de S. German, sont des protocoles de cohérence de cache qui exploitent la diffusion multiple de messages.

11.3.1 PidSet

Dans le protocole du PidSet [DF93], chaque processus peut être en attente, prêt ou actif, ce qui est respectivement représenté dans le système de transitions de la figure 11.6 par W, R et A. Dans cette figure et par la suite, *gs* est la fonction qui à chaque processus associe son état. Tous les processus sont initialisés dans l'état W. Intuitivement un processus prêt est prioritaire sur un processus en attente pour accéder à la ressource partagée. Lorsqu'il souhaite utiliser la ressource, il demande un rendez-vous au processus utilisant celle-ci en émettant le message *swap*. La seule transition que peut alors franchir le processeur actif est celle qui le remet dans un état d'attente, ceci en consommant le message (noté *swap*?). Les autres transitions sont gardées par l'existence ou non d'autres processus dans les états A (resp. R) dont la traduction est donnée par un prédicat sur l'état de chaque processus noté $gs^{-1}(\mathsf{A}) = \emptyset$ (resp $gs^{-1}(\mathsf{R}) = \emptyset$). La figure 11.7 est la spécification B correspondante.

```
$ java -jar reacha.jar -a -d -i  mesi/mesi2.mch
Reachability with B(c) LIFC - http://lifc.univ-fcomte.fr
Any comments/suggestions: couchot@lifc.univ-fcomte.fr

  Parsing B file ........ ok
  Type checking .........
        Warning! Abstract Set:ind

option:--B-membership-abstraction
option:--local
option:--inductive-inv

P0 is:
(! (p1,p2).((p1:ind  &  p2:ind  &  gs(p1)=s) =>  gs(p2) /= m))

P1 is:
(! (p1,p2).((p1:ind  &  p2:ind  &  gs(p1)=m  &  gs(p2)=m) =>  p2=p1))

------ 1 -------
The operation write violates P0
The operation write violates P1

------ 2 -------
The operation write violates write.P1

------ 3 -------


Unreachable in 2759 ms. with 3 iterations
duration for rvc : 414 ms
duration for rv : 2240 ms



The inductive invariant is
(gs:(ind --> vAL)  &
 (! (p1,p2).((p1:ind  &  p2:ind  &  gs(p1)=s) =>  gs(p2) /= m))
 (! (p1,p2).((p1:ind  &  p2:ind  &  gs(p1)=m  &  gs(p2)=m) =>  p2=p1))
)
&
(! (q1).((q1:ind  &  gs(q1)=e)
          =>
          ((gs<+{q1|->m}):(ind --> vAL)  &
            (! (p1,p2).((p1:ind  &  p2:ind  &  (gs<+{q1|->m})(p1)=s)
                        =>  (gs<+{q1|->m})(p2) /= m)))))
&
(! (q2).((q2:ind  &  gs(q2)=e)
          =>
          ((gs<+{q2|->m}):(ind --> vAL)  &
            (! (p1,p2).((p1:ind  &  p2:ind  &
                          (gs<+{q2|->m})(p1)=m  &  (gs<+{q2|->m})(p2)=m)
                        =>  p2=p1)))))
&
(! (q3).((q3:ind  &  gs(q3)=e)
          => (! (q2).((q2:ind  &  (gs<+{q3|->m})(q2)=e)
                      =>
                       (((gs<+{q3|->m})<+{q2|->m}):(ind --> vAL)  &
                         (! (p1,p2).((p1:ind  &  p2:ind  &
                                      ((gs<+{q3|->m})<+{q2|->m})(p1)=m  &
                                      ((gs<+{q3|->m})<+{q2|->m})(p2)=m)
                                     =>
                                      p2=p1)))))))))
```

FIG. 11.5 – Exécution du prototype sur l'exemple du MESI

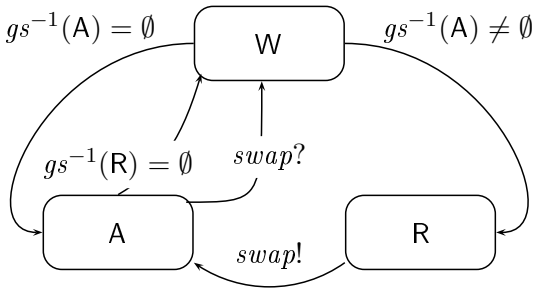


FIG. 11.6 – Système de transitions du PidSet

```
MACHINE
   pidset
SETS
  ind;
  status={a,r,w}
VARIABLES
  gs
INVARIANT
  gs : ind --> status &
  (!(p1,p2) . ((p1:ind & p2:ind & gs(p1)= a &
  gs(p2)=a) => p1=p2))
INITIALISATION
  gs := ind * {w}
OPERATIONS
  sleep =
    ANY pa WHERE pa : ind & gs(pa)= a  &
    (! pr. (pr : ind => gs(pr) /= r))
      THEN
        gs := gs <+ {pa|->w}
    END ;
  waitIfReady =
    ANY pa WHERE pa : ind & gs(pa)= a  &
      THEN
        ANY pr WHERE pr :ind & gs(pr)=  r
          THEN
    gs:= gs <+ {pa|->w,pr|->a}
        END
    END ;
  ativate =
    ANY pw WHERE pw : ind & gs(pw)= w  &
    (! pr. (pr : ind => gs(pr) /= a))
      THEN
        gs := gs <+ {pw|->a}
    END ;
  getReady =
    ANY pw WHERE pw : ind & gs(pw)= w  &
    (# pr. (pr : ind => gs(pr) = a))
    THEN gs := gs <+ {pw|->r}
    END
END
```

FIG. 11.7 – Spécification fonctionnelle B du PidSet

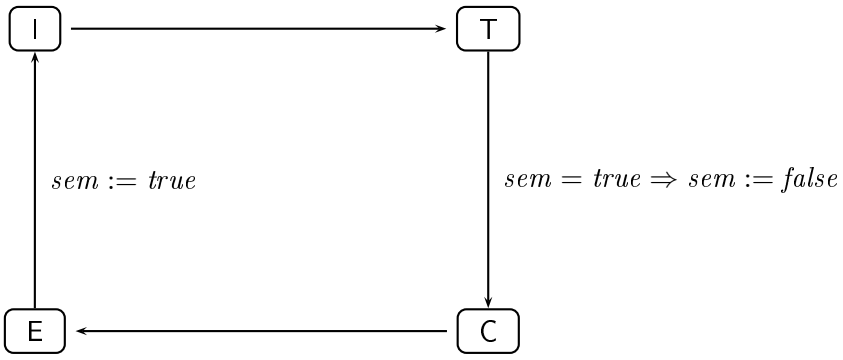


FIG. 11.8 – Système de transitions de MuxSem [PRZ01].

```
MACHINE
 muxsem

SETS
 status ={i,t,c,e}; semValues = {tr,fa};
 ind

VARIABLES
 gs, sem

INVARIANT
 gs : ind --> status &
 sem : semValues &
 (!(h1,h2).((h1: ind & h2 :ind & gs(h1)=c &
             gs(h2)=c) => h1=h2))

INITIALISATION
 gs := ind * {i} ||
 sem := tr

OPERATIONS
 sb =
  ANY p WHERE p : ind & gs(p) = i THEN
   gs := gs <+ {p |->t}
  END ;

 sc =
  ANY p WHERE p : ind & gs(p) = t &
                  sem = tr THEN
   gs := gs <+ {p |->c} ||
   sem := fa
  END;

 sd =
  ANY p WHERE p : ind & gs(p) = c THEN
   gs := gs <+ {p |->e}
  END ;

 se =
  ANY p WHERE p : ind & gs(p) = e THEN
   gs := gs <+ {p |->i} ||
   sem := tr
  END
END
```

FIG. 11.9 – Spécification fonctionnelle B du MuxSem

11.3.2 MuxSem

La figure 11.8 donne une représentation du système de transitions d'un processus quelconque de l'algorithme MuxSem [PRZ01]. Chaque processus peut être dans l'état inactif I, puis dans un état temporaire T avant d'entrer dans l'état C d'utilisation de la section critique. Il en termine alors avec la section critique en passant dans l'état E (pour *end*). L'unicité de l'accès à la section critique est assuré par un sémaphore représenté par la variable *sem*. Une spécification en langage B de ce protocole est donnée à la figure 11.9.

11.3.3 MutEx

La figure 11.10 donne une spécification en langage B de l'algorithme du MutEx dont la description a été effectuée dans la partie 3.1.4.

```
MACHINE
 mutex

SETS
 ind; status={a,r,s}

VARIABLES
 gs, turn

INVARIANT
 gs : ind --> status &
 turn : ind &
 (!(p1,p2) . ((p1:ind & p2:ind & gs(p1)= a &
               gs(p2)=a) => p1=p2))
INITIALISATION
 ANY ps WHERE ps : ind THEN turn := ps ||
 gs := ind * {s}
 END
OPERATIONS
 sleep =
  ANY p WHERE p : ind & gs(p)= a THEN
   gs := gs <+ {p|->s}
  END ;

 ask =
  ANY p WHERE p : ind & gs(p)= s THEN
   gs := gs <+ {p|->r}
  END ;

 mturn =
  ANY p WHERE p : ind & gs(p)= r &
   gs(turn)=s THEN
  turn := p
  END;

 activate =
  ANY p WHERE p : ind & gs(p)= r &
   turn = p THEN
    gs := gs <+ {p|->a}
  END
END
```

FIG. 11.10 – Spécification fonctionnelle B du MutEx

11.3.4 Dijkstra

Comme l'algorithme de Dijkstra [Dij65] dont il est issu, l'algorithme, dont le système de transitions est représenté à la figure 11.11, utilise une variable partagée *turn* de type indice mémorisant le processus ayant le droit d'accéder à la section critique. Par rapport à [Dij65], cet algorithme est obtenu en supprimant une variable partagée de type fonction totale ; les gardes de ses transitions s'en trouvent affaiblies et le nombre de ses comportements agrandi.

On note que son système de transitions est paramétré par p qui est l'identifiant du processus exécutant ce système de transition. Initialement en SI, chaque processus désirant accéder à la section critique entre dans la salle d'attente représentée par les trois états W_1,W_2 et W_3. Seul le processus identifié par *turn* peut se déplacer en W_3 , éventuellement via W_2 pour acquérir cette identification. S'il est seul dans W_3, le processus accède à la section critique A. Dans le cas contraire, il retourne en W_1. Une spécification B correspondante est donnée à la figure 11.12.

11.3.5 Université de l'Illinois

Le protocole de l'Université de l'Illinois[PP84] vise à garantir une propriété de cohérence de cache. Présenté à la figure 11.13, le système de transitions de chaque processeur possède les états M, E, S, I dont la sémantique est la même que pour l'algorithme du MESI (cf partie 3.3).

Par rapport au MESI, cet algorithme fournit plusieurs heuristiques :

- un cache c en I peut accéder directement à la mémoire en écriture en envoyant le message de broadcast *w2*. Les caches dans S et E réagissent à ce message en se déplacent en I tandis que celui en M réagit en mettant à jour la mémoire centrale et en retournant en I. Le cache c met à jour son contenu avec celui de la mémoire centrale, le modifie puis accède à M.
- lorsque le processeur dont le cache c est en I souahite avoir un accès à celui-ci en lecture, ce dernier
  - obtient une copie de la mémoire et passe dans l'état E si tous les autres caches sont en I

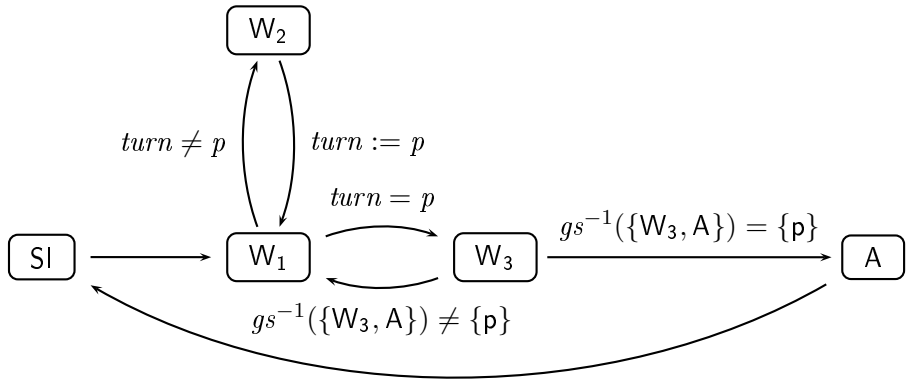


FIG. 11.11 – Système de transitions de Dijkstra

```
MACHINE
 dijkstra

SETS
 ind; status={s,w1,w2,w3,a}

VARIABLES
 gs,
 turn

INVARIANT
 gs:ind-->status &
 turn:ind &
 (!(p1,p2).((p1:ind & p2:ind & gs(p1)=a &
 gs(p2)=a)=>p1=p2))

INITIALISATION
 ANY ps WHERE ps:ind THEN turn:=ps ||
 gs:=ind * {s}
 END

OPERATIONS
 t2= ANY p WHERE p:ind & gs(p)=s THEN
   gs:=gs <+{p|->w1}
  END;

 t3= ANY p WHERE p:ind & gs(p)=w1 &
    turn/=p THEN
    gs:=gs <+{p|->w2}
  END;

 t4= ANY p WHERE p:ind & gs(p)=w2 THEN
   turn:=p ||
   gs:=gs <+{p|->w1}
 END;

 t5= ANY p WHERE p:ind & turn=p &
 gs(p)=w1 THEN
   gs:=gs <+{p|->w3}
 END;

 t6= ANY p WHERE p:ind & gs(p)=w3 &
   (#q.(q:ind & p/=q & (gs(q)=w3 or gs(q)=a))) THEN
     gs:=gs<+{p|->w1}
 END;

 t7= ANY p WHERE p:ind & gs(p)=w3 &
   (!q.((q:ind & p/=q) => (gs(q)/=w3 & gs(q)/=a)))
    THEN
     gs:=gs<+{p|->a}
 END;

 t8=
  ANY p WHERE p:ind & gs(p)=a THEN
   gs:=gs<+{p|->s}
  END
END
```

FIG. 11.12 – Spécification fonctionnelle B de l'algorithme de Dijkstra

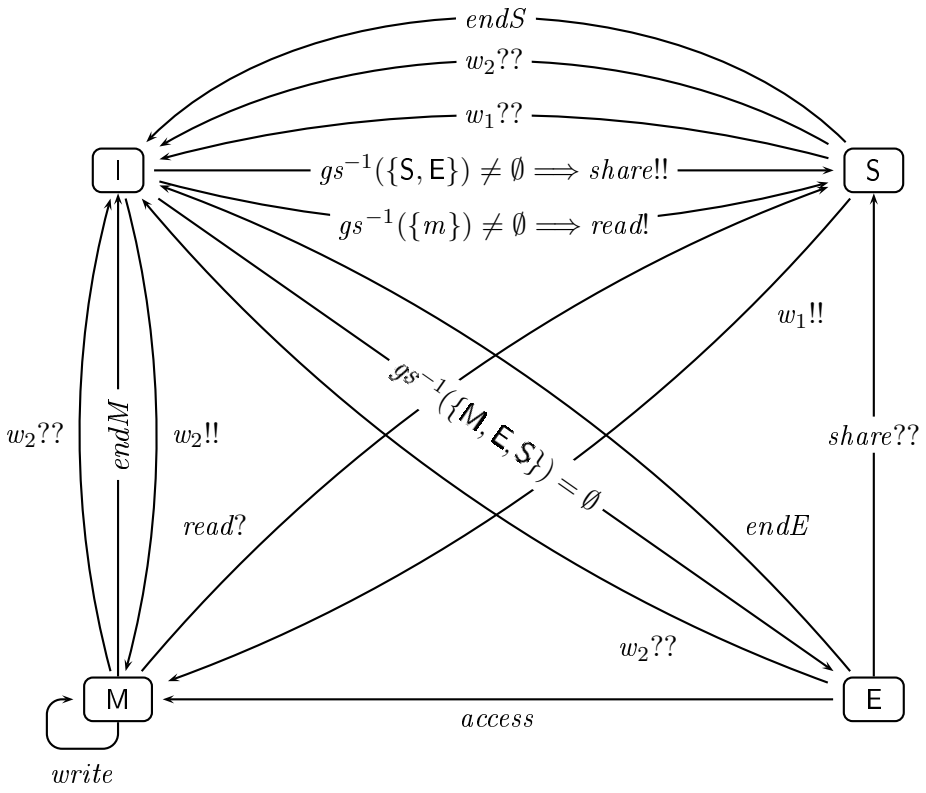


FIG. 11.13 – Système de transitions de l'Université de l'Illinois

```
MACHINE
  illinois

SETS
 ind; status={i,e,m,s}

VARIABLES
 gs

INVARIANT
 gs:ind-->status &
 (!(p1,p2).((p1:ind & p2:ind & gs(p1)=m
             & gs(p2)=m)=>p1=p2)) &
 (!(p1,p2).((p1:ind & p2:ind & gs(p1)=m)
             => gs(p2)/=s))

INITIALISATION
 gs:=ind*{i}

OPERATIONS
 firstOut= ANY p1 WHERE p1:ind & gs(p1)=i &
  (!qq . (qq:ind => gs(qq)=i)) THEN
   gs:=gs<+{p1|->e}
 END;

 out= ANY p1,p2 WHERE p1:ind & p2:ind &
  gs(p1)=i & gs(p2)=m THEN
   gs:=(gs<+{p1|->s})<+{p2|->s}
 END;

 share= ANY p1 WHERE p1:ind & gs(p1)=i THEN
  ANY p2 WHERE p2:ind &
   (gs(p2)=e or gs(p2)=s) THEN
    gs:=(dom(gs|>{i})*{i}\/
         dom(gs|>{e})*{s}\/
         dom(gs|>{s})*{s}\/
         dom(gs|>{m})*{m})<+{p1|->s}
  END
 END;

 access= ANY p1 WHERE p1:ind & gs(p1)=e THEN
  gs:=gs<+{p1|->m}
 END;

 w1= ANY p1 WHERE p1:ind & gs(p1)=s THEN
  gs:=(dom(gs|>{i})*{i}\/
       dom(gs|>{s})*{i}\/
       dom(gs|>{m})*{m}\/
       dom(gs|>{e})*{e})<+{p1|->m}
  END;

 w2= ANY p1 WHERE p1:ind & gs(p1)=i THEN
  gs:=(dom(gs|>{i})*{i}\/
       dom(gs|>{e})*{i}\/
       dom(gs|>{s})*{i}\/
       dom(gs|>{m})*{i})<+{p1|->m}
 END;

 endM= ANY p WHERE p:ind & gs(p)=m THEN
  gs:=gs<+{p|->i}
 END;

 endS= ANY p WHERE p:ind & gs(p)=s THEN
  gs:=gs<+{p|->i}
 END;

 endE= ANY p WHERE p:ind & gs(p)=e THEN
  gs:=gs<+{p|->i}
 END
END
```

FIG. 11.14 – Spécification fonctionnelle B de l'Université de l'illinois

- obtient une copie de la mémoire et passe dans l'état S si d'autres caches sont dans S ou E ; ceux de E rejoignent c en S à la réception du message *share* ;
- demande un rendez-vous par le message *read* avec le cache en M s'il y en a un ; celui ci met à jour la mémoire centrale puis va en S où il rejoint c qui aura mis à jour son contenu avec celui de la mémoire centrale.

– un cache en S peut accéder directement (sans passer donc par E comme dans le MESI) à l'état M en envoyant le message de broadcast w_1 qui expédie dans I tous les caches présents aussi dans S.

On donne une spécification B de cet algorithme à la figure 11.14.

11.3.6 S. German

L'algorithme de S. German [Ger00, DB01, BLS02] vise à garantir des propriétés de cohérence de cache pour un ensemble de processeurs, qui sont contrôlés par un serveur.

Chaque processeur (resp. le serveur) suit le système de transitions donné à la figure 11.15 (resp. la figure 11.16).

Le système de transitions du serveur possède six états dont on donne alors le sens :

– I : initialisation ou attente de requête

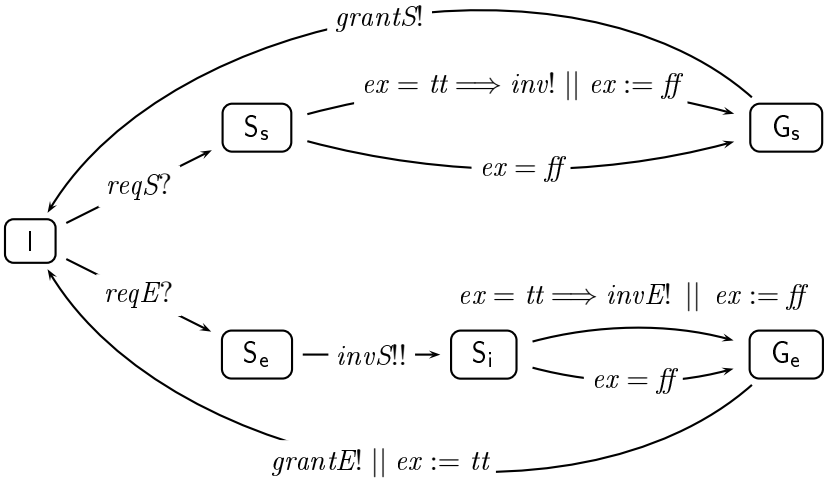


FIG. 11.15 – Système de transitions du serveur dans le protocole de S.German

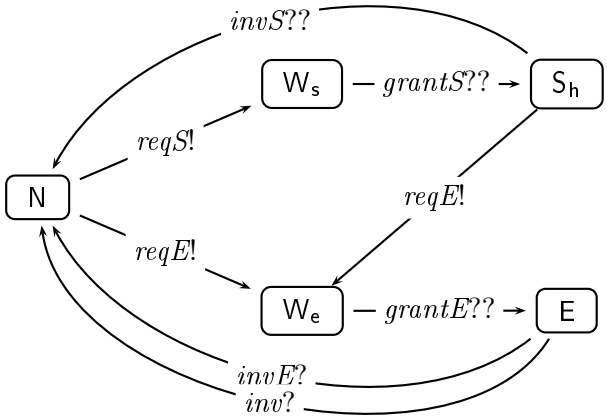


FIG. 11.16 – Système de transitions d'un client dans le protocole de S.German

– $\mathsf{S_e}$: a reçu une demande d'utilisation en exclusivité de la mémoire,
– $\mathsf{S_s}$: a reçu une demande d'accès à la mémoire en lecture,
– $\mathsf{S_i}$: a fait en sorte que chaque client qui partageait la mémoire l'a maintenant libérée,
– $\mathsf{G_e}$: est en position d'attribuer un droit d'accès exclusif à la mémoire et
– $\mathsf{G_s}$: est en position d'attribuer un droit d'accès en lecture à la mémoire.

Concernant le système de transitions du client, ce dernier possède cinq états :

- N : initialisation ou repos ,
- $\mathsf{W_e}$: attend d'obtenir le droit d'exclusivité sur la mémoire,
- $\mathsf{W_S}$: attend d'obtenir le droit d'accès en lecture dans la mémoire,
- E : possède le droit d'exclusivité sur la mémoire et
- $\mathsf{S_h}$: possède un droit de lecture dans la mémoire.

Cet algorithme utilise une variable booléenne partagée *ex* qui est vraie si un client possède actuellement un droit d'accès exclusif à la mémoire.

Lorsque le serveur se synchronise avec un client sur la requête *reqE* ou *reqS*, il passe dans un état "occupé à servir". Par exemple, supposons que le serveur doive servir une requête pour un accès exclusif : il envoie tout d'abord une requête *inv* par broadcast demandant à chaque client possédant une copie locale (et donc dans l'état S) de s'invalider (donc de revenir dans l'état N). Après cette étape, il envoie éventuellement (c.a.d. lorsque la variable *ex* est vraie) un message d'invalidation *invE* au processus qui est dans l'état E. Le serveur passe dans l'état $\mathsf{G_e}$. Si *ex* est faux, il passe directement dans l'état $\mathsf{G_e}$. Depuis $\mathsf{G_e}$, le serveur donne le droit d'accès exclusif au client de $\mathsf{W_e}$, qui passe dans l'état E en recevant le message $\mathsf{G_e}$ du serveur et le serveur mémorise qu'un client possède ce droit exclusif en fixant à vraie la variable *ex*. Une spécification en langage B de cet algorithme est donnée à la figure 11.17.

La partie suivante présente les temps de calcul pour vérifier les algorithmes présentés dans cette partie.

11.4 Résultats numériques

Le tableau de la figure 11.18 donne les temps de vérification pour les exemples présentés dans ce mémoire. Dans ce tableau, la seconde colonne indique le nombre d'opérations de la spécification ; la troisième reprend le nombre d'états du système de transitions qui est distribué ; dans le cas d'un algorithme avec plusieurs clients et un serveur (cf S. German), c'est le nombre d'états du client qui est donné. La quatrième colonne reprend le nombre d'itérations nécessaire pour établir la propriété. Les quatre dernières colonnes donnent des temps de calcul exprimés en secondes et mesurés sur un centrino 1.5 GHz avec 768 Mo de RAM. La colonne *Tr. Std.* donne le temps pour une traduction standard. A partir de la colonne *Tr. Abst.*, c'est la traduction abstraite qui est retenue. Dans la colonne *Pred. Locaux* les prédécesseurs sont en outre calculés localement et à la colonne *Verif. locales* toutes les vérifications se font localement. Dans ce tableau, le symbole ↑ signifie que le prouveur semble diverger : il lui faut plus de 30 secondes pour une des obligations de preuve. 4 [Del00] signifie que Delzanno a montré que la vérification de ce protocole nécessite 4 itérations.

En terme de comparaison avec le premier prototype barvey et les spécifications B ensemblistes, les durées de vérification sont divisées par 10 : 0.481 secondes suffisent pour vérifier le PidSet, alors qu'auparavant, 4.8 secondes étaient nécessaires pour barvey et 0.58 secondes pour l'atelier B.

```
MACHINE
 sgerman

SETS
 bbool={tt,ff}; ind;
 cstatus={n,we,ws,e,sh};
 sstatus={i,se,ss,si,ge,gs}

VARIABLES
 s, gsc, ex

INVARIANT
 s:sstatus &
 gsc:ind-->cstatus &
 ex:bbool &
 (!(p1,p2).((p1:ind & p2:ind & gsc(p1)=e)
             =>gsc(p2)/=sh)) &
 (!(p1,p2).((p1:ind & p2:ind & gsc(p1)=e
             & gsc(p2)=e)=>p1=p2))

INITIALISATION
 s:=i || gsc:=ind*{n} || ex:=ff

OPERATIONS
 reqE= ANY p WHERE p:ind & gsc(p)=n &
  s=i THEN
   s:=se || gsc:=gsc<+{p|->we}
 END;

 reqS= ANY p WHERE p:ind & gsc(p)=n &
  s=i THEN
   s:=ss || gsc:=gsc<+{p|->ws}
 END;

 invE= ANY p WHERE p:ind & s=si &
  ex=tt & gsc(p)=e THEN
   s:=ge || ex:=ff ||
   gsc:=gsc<+{p|->n}
 END;

 invS= ANY p WHERE p:ind & s=se &
  gsc(p)=sh THEN
   s:=si ||
   gsc:=dom(gsc|>{sh})*{n}\/
        dom(gsc|>{n})*{n}\/
        dom(gsc|>{we})*{we}\/
        dom(gsc|>{ws})*{ws}\/
        dom(gsc|>{e})*{e}
 END;

 inv= ANY p WHERE p:ind & s=ss &
  ex=tt & gsc(p)=e THEN
   s:=gs || ex:=ff
   gsc:=gsc<+{p|->n}||
  END;

 nonexE= SELECT ex=ff & s=si THEN
   s:=ge
 END;

 nonexS= SELECT ex=ff & s=ss THEN
   s:=gs
 END;

 grantE= ANY p WHERE p:ind & s=ge &
  gsc(p)=we THEN
   s:=i || ex:=tt
   gsc:=gsc<+{p|->e}||
 END;

 grantS= ANY p WHERE p:ind & s=gs &
  gsc(p)=ws THEN
   s:=i || gsc:=gsc<+{p|->sh}
 END;

 reqEb= ANY p WHERE p:ind & gsc(p)=sh &
  s=i THEN
   s:=se || gsc:=gsc<+{p|->we}
 END
END
```

FIG. 11.17 – Spécification fonctionnelle B du protocole de S.German

Exclusion Mutuelle							
Protocole	*Ops.*	*Etats*	*Itér.*	*Tr. Std.*	*Tr. Abst.*	*Pred. Locaux*	*Verif. locales*
PidSet	4	3	1	0.51	0.48	0.48	0.48
MuxSem	4	4	7	↑ après 3 it.	↑ après 3 it.	↑ après 4 it.	385
MutEx	4	3	3	2.27	1.88	1.95	2.48
Dijkstra	8	5	1	0.84	0.76	0.76	0.76
Cohérence de cache							
MESI	3	4	3	5.09	2.38	2.62	3.47
S. German	10	5	4	12	6.66	8.63	10.15
Illinois	9	4	4 [Del00]	↑ après 1 it.	↑ après 1 it.	↑ après 1 it.	↑ après 2 its.

FIG. 11.18 – Durées de vérification des protocoles

Une analyse de l'utilisation des options du prototype fait ressortir les constatations suivantes : Tout d'abord l'abstraction de traduction apporte un gain en temps de l'ordre de 30%. Les écarts sont plus faibles sur les spécifications avec peu d'itérations et d'opérations (comme le PidSet) et plus importantes sur les spécifications où ces deux critères sont plus élevés (on approche les 50% pour le MESI et l'algorithme de S. German).

On remarque ensuite que la localité ne réduit jamais les durées de calcul : elle les fait même augmenter de plus de 45% pour le MESI et S. German. Cette option correspond à une volonté de découper les obligations de preuve, plus pour permettre au prouveur de converger que pour réduire la durée totale de la vérification. Le prouveur décharge en plus grand nombre des OPs de taille plus réduite. Ceci est confirmé par l'algorithme de MuxSem, pour lequel cette localité permet de faire converger toutes les preuves et d'obtenir ainsi un verdict.

On remarque enfin que l'algorithme de l'Université de l'Illinois n'a pas pu être vérifié par la méthode, même avec les options de vérification locale : même si toutes les conditions d'application du corollaires 8 sont réunies, la convergence des semi-algorithmes n'est pas observée. Nous obtenons une différence pratique entre un résultat théorique basé sur les sortes et une implantation pratique basée sur la superposition qui ne les respecte pas. Cet exemple, à priori anecdotique, nous fournit une perspective d'étude comme l'annonçait l'introduction.

11.5 Résumé

Ce chapitre a présenté les prototypes qui ont permis de valider les démarches présentées dans ce mémoire : (*i*) la logique équationnelle est à même de traiter des obligations de preuve issues de spécifications ensemblistes avec une efficacité acceptable ; (*ii*) elle se présente alors comme très efficace pour traiter des obligations de preuve à base de fonctions totales. Néanmoins elle ne permet pas de décider tous les problèmes de la classe envisagée. Le chapitre suivant conclut le présent travail.

Chapitre 12

Conclusions et perspectives

La motivation de notre travail était guidée par la phrase de Dershowitz et Jouannaud dans [DJ90] : *Equations are ubiquitous in mathematics and the sciences [...]. These reasoning abilities are also important in many computer applications, including [...] automated theorem proving, program specification and verification, and high-level programming languages and environments.* De façon synthétique, notre principal objectif a été de réduire des problèmes de vérification d'invariant en des problèmes de satisfaction de clauses exprimées en logique équationnelle du premier ordre.

Dans ce chapitre, nous présentons dans la partie 12.1 une synthèse plus détaillée des travaux effectués. Nous établissons ensuite, dans la partie 12.2, un bilan critique de ceux-ci. Nous terminons enfin, dans la partie 12.3, par les perspectives ouvertes par cette étude.

12.1 Synthèse

Nous avons présenté une nouvelle démarche de vérification d'invariant par superposition adaptée aux systèmes paramétrés. Nous l'avons appliquée particulièrement à un exemple industriel et à des systèmes uniformes distribués.

L'objectif, qui était de savoir vérifier des exemples possédant les difficultés des deux classes, est à comparer avec l'ambivalence du prouveur haRVey auquel la démarche est associée : une face de gestion de la structure de la formule par une couche propositionnelle, une autre face de gestion des structures de données par superposition.

Notre travail s'est focalisé sur la théorie équationnelle des tableaux et il a montré qu'elle présente un compromis acceptable entre (*i*) le pouvoir d'expression qu'elle fournit, (*ii*) la simplicité d'utilisation de ses opérateurs et (*iii*) la simplicité des raisonnements à mettre en place pour établir, à son sujet, des preuves d'équisatisfaisabilité.

Concernant les deux premiers points, nous avons en effet démontré que cette théorie fournit un cadre suffisant pour exprimer les deux familles d'exemples, et ce à l'aide d'une formalisation des règles de traduction des formules ensemblistes ou des formules avec fonctions totales dans cette théorie des tableaux. Cette formalisation nous a permis de démontrer la correction de la traduction puis d'implanter rapidement les prototypes avec une conviction accrue de la qualité du code généré.

Concernant le dernier point, nous avons pu établir la décidabilité de plusieurs classes de formules dans cette théorie, formules engendrées lors des constructions d'invariants. Cette décidabilité exploite des résultats sur les logiques multi-sortes [FG03] et sur des techniques de

réécriture [ARR03]. Nous avons aussi montré les limites de tels raisonnements avec un cas où l'approche avec multi-sortes donne un résultat théorique différent de de l'expérimentation.

D'un point de vue pratique, nous avons entièrement implanté la démarche de vérification des deux familles d'exemples. Les prototypes sont disponibles en ligne, aux adresses `http://lifc.univ-fcomte.fr/~couchot/soft/barvey/` et `http://lifc.univ-fcomte.fr/~couchot/specs/`. Les temps de calcul montrent que l'approche décide ces exemples de grande taille et qu'elle décide des algorithmes d'exclusion mutuelle et de cohérence de cache paramétrés dans des temps acceptables.

12.2 Bilan

D'un point de vue théorique, nous avons prouvé la correction de la démarche et avons montré la décidabilité pour une sous-classe. Cette décidabilité est cependant à nuancer pour plusieurs raisons. La première est qu'elle n'est pas établie pour le contexte général qui était initialement fixé, mais seulement pour une sous-classe de programmes. La seconde est qu'elle emploie des techniques d'expansion des quantificateurs qu'il n'est pas judicieux de mettre en pratique pour des raisons évidentes de complexité algorithmique.

Une procédure alternative existe peut-être directement par superposition, mais nous n'avons pas pu l'établir : nous avons, certes, identifié une classe de clauses correspondant aux conditions d'évolution des calculs de point fixe, mais l'étude (approfondie) de celle-ci ne nous a pas permis de conclure positivement pour plusieurs raisons. La première est que cette classe ne reste stable que par un calcul de superposition respectant les sortes, comme celui de SPASS [Wei01], ce qui n'est pas le cas de $\mathcal{SP}$ embarqué dans haRVey. La seconde concerne la finitude du nombre de clauses engendrées que nous n'avons pas pu établir formellement. La troisième réside dans l'absence d'outils permettant d'effectuer automatiquement cette tâche de vérification : pour une classe donnée, il s'agit de vérifier manuellement que cette classe reste stable en appliquant toutes les règles de superposition puis de trouver manuellement un critère permettant d'assurer la convergence du calcul. A chaque extension de la classe de clauses, tout le travail de stabilité est à refaire et la convergence doit être à nouveau étudiée. Pour toutes ces raisons, cette étude n'a pas pu être intégrée dans ce mémoire.

Suite à cet échec, nos recherches se sont orientées vers l'approche multi-sortes proposée dans ce travail, même si celle-ci ne correspond pas à l'implantation. Ce changement de direction nous a été profitable puisque nous avons pu, tout de même, établir des résultats de décidabilité et pu constater les limites de tels résultats. Expérimentalement, nous avons remarqué une corrélation forte entre les résultats théoriques et pratiques : à chaque fois que la procédure basée sur les sortes était applicable, la preuve terminait et lorsqu'elle ne l'était pas elle divergeait. La question d'un lien plus général entre la finitude d'une procédure par superposition sans sorte et la finitude du domaine de Herbrand de la sorte de la formule quantifiée se pose alors.

Si, par la suite, nos travaux devaient se poursuivre dans la direction des preuves de décidabilité d'un calcul par superposition, comme $\mathcal{SP}$, pour d'autre théories équationnelles ou d'autres classes de systèmes, nous envisagerions le problème différemment : nous outillerions ce raisonnement en conceptualisant puis en développant un outil capable d'engendrer automatiquement, à nouveau comme un calcul de point fixe, la classe des clauses stables par ce calcul pour un ensemble de formules initiales.

12.3 Perspectives

Les expériences et connaissances acquises pendant ce travail nous conduisent naturellement vers de nouvelles perspectives de recherche qui s'organisent comme suit. La partie 12.3.1 montre quelles heuristiques pourraient être appliquées sur les calculs de point fixe du chapitre 4 pour en accélérer la convergence. La seconde (partie 12.3.2) s'intéresse à l'applicabilité des démarches de preuve vis à vis de la technique d'abstraction de prédicats.

12.3.1 Heuristiques

Ce travail se focalisant sur les démarches de traduction dans la théorie des tableaux, nous avons intégré peu d'heuristiques établies dans d'autres cadres. Les appliquer dans celui-ci nous permettrait de retrouver, voire prolonger des résultats établis dans leur contexte initial et bénéficier de procédures plus efficaces.

Pour accélérer la convergence des semi-algorithmes du chapitre 4, plusieurs heuristiques pourraient ainsi être appliquées, parmi lesquelles on propose l'introduction d'un transformateur d'assertions dédié et un ordonnanceur de formules.

Transformateur d'assertions dédié

Automatique, la technique de renforcement d'invariant telle qu'elle est appliquée souffre d'engendrer rapidement des formules de grande taille, et par conséquent difficilement lisibles et parfois non traitables par le prouveur. Bartzis et Bultan [BB03] ont introduit des heuristiques guidant le calcul d'assertions en fonction de la nature des transitions. On peut s'en inspirer et introduire un transformateur d'assertion dédié pour les systèmes de transitions de cette étude.

Retour à l'exemple MESI. On se restreint à vérifier sur le MESI l'invariance de la propriété P_0 de la figure 11.5 (l'écriture dans la mémoire ne s'effectue pas simultanément avec la lecture dans un des caches). En posant $[S_{write}]P_0$ égal à

$$\begin{array}{l}\forall\, q_1 \,.\, (q_1 \in CACHE \wedge gs(q_1) = e) \Rightarrow \\ \quad (\forall\, p_3, p_4 \,.\, (p_3 \in CACHE \wedge p_4 \in CACHE \wedge (gs \lhd\!+ \{q_1 \mapsto m\})(p_3) = s) \Rightarrow \\ \quad\quad (gs \lhd\!+ \{q_1 \mapsto m\})(p_4) \neq m)),\end{array} \tag{12.1}$$

le premier renforcement $P_0 \wedge [S_{write}]P_0$ de P_0 selon *write* qui est

$$\begin{array}{l}gs \in CACHE \rightarrow vAL \wedge \\ (\forall\, p_1, p_2 \,.\, ((p_1 \in CACHE \wedge p_2 \in CACHE \wedge gs(p_1) = s) \Rightarrow gs(p_2) \neq m)) \\ \wedge\ (12.1)\end{array} \tag{12.2}$$

possède les défauts énoncés plus haut. L'annexe A montre que cette formule est équivalente à

$$\begin{array}{l}gs \in CACHE \rightarrow vAL \wedge \\ (\forall\, p_1, p_2 \,.\, p_1 \in CACHE \wedge p_2 \in CACHE \wedge gs(p_1) = s \Rightarrow (gs(p_2) \neq m \wedge gs(p_2) \neq e))\end{array} \tag{12.3}$$

qui est beaucoup plus simple. Un transformateur d'assertions dédié qui engendrerait automatiquement un tel invariant réduit, en tenant compte de la propriété et de la nature de la transition, diminuerait la charge du prouveur.

Ordonnancement des obligations de preuve

L'introduction de la vérification locale de l'implication (figure 4.10) nécessite le déchargement de l'obligation de preuve

$$\bigwedge_{m \in M} \bigvee_{j \in \mathcal{J}} (j \Rightarrow m)$$

où M et $\mathcal{J}$ sont deux ensembles de formules ce qui nécessite $card(M) \times card(\mathcal{J})$ appels au prouveur. Pour chaque $m \in M$ et à l'aide de critères syntaxiques, ordonner les éléments j de $\mathcal{J}$ par ordre décroissant de probabilité d'établir l'implication $j \Rightarrow m$ permettrait de réduire le nombre d'appels au prouveur [Gri00, Col05].

12.3.2 Abstraction de prédicats

Générique dans leur approche, les démarches à bases d'abstraction [GS97, Bau03] souffrent du nombre et de la complexité des preuves qu'elle doivent décharger dans le prouveur lors de leur mise en œuvre. Dans les deux cas précédents, il s'agit de MONA qui implante une procédure de décision pour la logique WS1S. Les expériences [DG03, CDGR03] menées avec ce prouveur dans le cadre de la vérification de spécifications ensemblistes industrielles nous ont convaincu de son inaptitude à construire en général l'automate acceptant une formule WS1S de grande taille. Un calcul par superposition apparaît dans ce cas bien plus adapté.

En suivant ce constat, une piste de recherche consisterait à remettre le prouveur à sa place en le comparant à d'autres procédures de décision. Ce travail théorique, qui sort du cadre de la vérification de logiciel, nécessite des compétences que nous avons acquises lors de nos essais de preuve de décidabilité par $\mathcal{SP}$. Cependant, le profit en vérification de logiciel sera à nuancer puisque nous ne proposerons, en cas de succès, qu'une alternative à MONA.

Annexe A

Preuve d'équivalence entre (12.2) et (12.3)

Pour montrer que (12.2) est équivalente à (12.3), c'est à dire que

$$\begin{array}{l} gs \in CACHE \rightarrow vAL \wedge \\ (\forall\, p_1, p_2 \,.\, ((p_1 \in CACHE \wedge p_2 \in CACHE \wedge gs(p_1) = s) \Rightarrow gs(p_2) \neq m)) \\ \wedge\ (12.1) \end{array}$$

est équivalente à

$$\begin{array}{l} gs \in CACHE \rightarrow vAL \wedge \\ (\forall\, p_1, p_2 \,.\, p_1 \in CACHE \wedge p_2 \in CACHE \wedge gs(p_1) = s \Rightarrow (gs(p_2) \neq m \wedge gs(p_2) \neq e)), \end{array}$$

on commence par simplifier (12.1), qui est

$$\begin{array}{l} \forall\, q_1 \,.\, (q_1 \in CACHE \wedge gs(q_1) = e) \Rightarrow \\ \quad (\forall\, p_3, p_4 \,.\, (p_3 \in CACHE \wedge p_4 \in CACHE \wedge (gs \lhd\!\!+ \{q_1 \mapsto m\})(p_3) = s) \Rightarrow \\ \quad\quad (gs \lhd\!\!+ \{q_1 \mapsto m\})(p_4) \neq m)). \end{array}$$

Pour cela, on note tout d'abord que lorsque s et m sont distincts, on a

$$(gs \lhd\!\!+ \{q_1 \mapsto m\})(p_3) = s \Leftrightarrow \big(gs(p_3) = s \wedge p_3 \neq q_1\big)$$

et que

$$(gs \lhd\!\!+ \{q_1 \mapsto m\})(p_4) \neq m \Leftrightarrow \big(gs(p_4) \neq m \wedge p_4 \neq q_1\big).$$

De plus, comme tous les quantificateurs de (12.1) sont universels$_{nf}$, celle-ci s'écrit en forme prénexe comme

$$\begin{array}{l} \forall\, q_1, p_3, p_4 \,.\, \big(q_1 \in CACHE \wedge p_3 \in CACHE \wedge p_4 \in CACHE \wedge \\ \quad gs(q_1) = e \wedge gs(p_3) = s \wedge p_3 \neq q_1\big) \Rightarrow \\ \quad\quad (gs(p_4) \neq m \wedge p_4 \neq q_1), \end{array}$$

qui se réécrit en

$$\begin{array}{l} \forall\, p_3 \,.\, (p_3 \in CACHE \wedge gs(p_3) = s) \Rightarrow \\ \quad (\forall\, q_1, p_4 \,.\, (q_1 \in CACHE \wedge p_4 \in CACHE \wedge p_3 \neq q_1 \wedge gs(q_1) = e) \Rightarrow \\ \quad\quad (gs(p_4) \neq m \wedge p_4 \neq q_1)). \end{array} \qquad (A.1)$$

On montre enfin que (A.1) est équivalente à

$$\forall\, p_3 \,.\, (p_3 \in CACHE \wedge gs(p_3) = s) \Rightarrow (\forall\, q_1 \,.\, q_1 \in CACHE \Rightarrow gs(q_1) \neq e) \tag{A.2}$$

En effet si $\forall\, q_1 \,.\, q_1 \in CACHE \Rightarrow gs(q_1) \neq e$ est interprétée à $\top$, alors le membre de gauche de la formule quantifiée en q_1 et p_4 de (A.1) est interprété à $\bot$ et (A.1) est vraie. Sinon, s'il existe q_1 tel que $q_1 \in CACHE \wedge gs(q_1) = e$, alors $q_1 \neq p_3$ puisque $gs(p_3) = s$ et $e \neq s$. Ainsi le membre de gauche de la formule quantifiée en q_1 et p_4 de (A.1) est interprété à $\top$. Or le littéral $p_4 \neq q_1$ de son membre droit est interprété à $\bot$ si le domaine d'interprétation des constantes de sorte INDEX est non vide ce qui est le cas ici.

Ainsi (12.2) bénéficie de cette simplification et se réécrit en

$$\begin{array}{l} gs \in CACHE \rightarrow vAL \wedge \\ (\forall\, p_1, p_2 \,.\, ((p_1 \in CACHE \wedge p_2 \in CACHE \wedge gs(p_1) = s) \Rightarrow gs(p_2) \neq m)) \wedge \\ (\forall\, p_1 \,.\, (p_1 \in CACHE \wedge gs(p_1) = s) \Rightarrow (\forall\, p_2 \,.\, p_2 \in CACHE \Rightarrow gs(p_2) \neq e)) \end{array}$$

qui se simplifie en

$$\begin{array}{l} gs \in CACHE \rightarrow vAL \wedge \\ (\forall\, p_1, p_2 \,.\, p_1 \in CACHE \wedge p_2 \in CACHE \wedge gs(p_1) = s \Rightarrow (gs(p_2) \neq m \wedge gs(p_2) \neq e)) \end{array}$$

c'est à dire (12.3).

Bibliographie

[ABB+05] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Heám, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

[Abr96] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AC03] J.-R. Abrial and D. Cansell. Click'n'prove : Interactive proofs within set theory. In David Basin and Burkhart Wolff, editors, *16th Intl. Conf. Theorem Proving in Higher Order Logics (TPHOLs'2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, septembre 2003.

[ACJT96] P. A. Abdulla, K. Cerans, B. Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS '96 : Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 313, Washington, DC, USA, 1996. IEEE Computer Society.

[ACM03] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3) :215–227, 2003.

[ADMP01] Pablo Argon, Giorgio Delzanno, Supratik Mukhopadhyay, and Andreas Podelski. Model checking for communication protocols. In Leszek Pacholski and Peter Ruzicka, editors, *Proceedings of the 28th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM-2001)*, volume 2234 of *Lecture Notes in Computer Science*, pages 160–170, Piestany, Slovak Republic, 2001. Springer.

[AJMd02] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. In *CAV '02 : Proceedings of the 14th International Conference on Computer Aided Verification*, pages 555–568, London, UK, 2002. Springer-Verlag.

[AK86] K.R. Apt and D. Kozen. Limits for automatic program verification of finite-state concurrent systems. *Information Processing Letters*, 22(6) :307–309, 1986.

[APR+01] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV '01 : Proceedings of the 13th International Conference on Computer Aided Verification*, pages 221–234, London, UK, 2001. Springer-Verlag.

[ARR03] A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Journal of Information and computation*, 183 :140–164, 2003. Special Issue on the 12th International Conference on Rewriting Techniques and Applications (RTA'01).

[Bö62] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. 1960 Int. Congr. for Logic, Methodology, and Philosophy of Science*, pages 1–1. Stanford Univ. Press, 1962.

[Bau03] Kai Baukus. *Abstraction-Based Verification of Parameterized Networks*. PhD thesis, Technishen Fakultät der Christian-Albrechts-Universität zu Kiel, May 2003.

[BB03] Constantinos Bartzis and Tevfik Bultan. Efficient image computation in infinite state model checking. In *Computer Aided Verification, 15th International Conference, CAV 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 249–261. Spring, 2003.

[BB04] Clark W. Barrett and Sergey Berezin. Cvc lite : A new implementation of the cooperating validity checker category b. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, Lecture Notes in Computer Science, pages 515–518. Springer, 2004.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. METEOR : A successful application of B in a large project. In *Proceedings of FM'99 : World Congress on Formal Methods*, pages 369–387, 1999.

[BBLS00] Kai Baukus, Saddek Bensalem, Yassine Lakhnech, and Karsten Stahl. Abstracting ws1s systems to verify parameterized networks. In *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS'00*, volume 1785 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 2000.

[BCJK01] Francoise Bellegarde, S. Chouali, Jacques Julliand, and O. Kouchnarenko. Comment limiter la spécification de l'équité dans les systèmes d'événements B. In *AFADL'2001*, pages 205–219, June 2001.

[BD01] Tevfik Bultan and Giorgio Delzanno. Constraint-based verification of client-server protocols. In *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP'01*, volume 2239 of *Lecture Notes in Computer Science*, pages 286–301. Springer, 2001.

[Beh96] Patrick Behm. Développement formel des logiciels sécuritaires de METEOR. In *Proceedingsof 1st Conference on the B method*, pages 3–10, 1996.

[BF02] Jean-Paul Bodeveix and Mamoun Filali. Type synthesis in B and the translation of B to PVS. In *ZB'2002 – Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 350–369, Grenoble, France, January 2002. LSR-IMAG.

[BG01] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 19–99. Elsevier and MIT Press, 2001.

[BGL03] S. Bensalem, S. Graf, and Y. Lakhnech. Abstraction as the key for invariant verification. In *Verification : Theory and Practice : Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 67–99. Springer, 2003.

[BJK02] Françoise Bellegarde, Jacques Julliand, and Olga Kouchnarenko. Synchronised parallel composition of events systems in B. In *ZB'2002 – Formal Specification and Development in Z and B*, pages 436–457, 2002.

[BJM99] Francoise Bellegarde, Jacques Julliand, and Hassan Mountassir. Model-based verification trough refinment of B event systems. In *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques*, pages 16–26, 1999.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *CAV '00 : Proceedings of the 12th International Conference on Computer Aided Verification*, pages 403–418, London, UK, 2000. Springer-Verlag.

[BLLP04] Eddy Bernard, Bruno Legeard, Xavier Luck, and Fabien Peureux. Generation of test sequences from formal specifications : Gsm 11-11 standard case study. *Softw., Pract. Exper.*, 34(10) :915–948, 2004.

[BLO98a] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, pages 319–331. Springer-Verlag, 1998.

[BLO98b] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Invest : A tool for the verification of invariants. In *Computer Aided Verification, 10th International Conference, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 505–510. Springer, 1998.

[BLP02] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - A Constraint Solver for B. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS2002*, pages 188–204, Grenoble, France, April 2002. LNCS 2280.

[BLS01] K. Baukus, Y. Lakhnech, and K. Stahl. Verification of Parameterized Protocols. *Journal of Universal Computer Science*, 7(2) :141–158, Février 2001.

[BLS02] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Parameterized verification of a cache coherence protocol : Safety and liveness. In *Verification, Model Checking, and Abstract Interpretation, Third International Workshop, VMCAI'02*, volume 2294 of *Lecture Notes in Computer Science*, pages 317–330. Springer, 2002.

[BP03] D. Bert and M.-L. Potet. La méthode B. École Jeunes chercheurs en programmation, May 2003.

[BP04] Y. Bertot and P.Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Springer Verlag, 2004.

[BS03] E. Borger and Robert F. Stark. *Abstract State Machines : A Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[Bul00] Tevfik Bultan. Action language : a specification language for model checking reactive systems. In *ICSE '00 : Proceedings of the 22nd international conference on Software engineering*, pages 335–344, New York, NY, USA, 2000. ACM Press.

[Bus98] Samuel R. Buss. An Introduction to Proof Theory. In Samuel R. Buss, editor, *Handbook of Proof Theory*, pages 2–78. Elsevier Science B.V., 1998.

[BYK01] Tevfik Bultan and Tuba Yavuz-Kahveci. Action language verifier. In *ASE '01 : Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 382, Washington, DC, USA, 2001. IEEE Computer Society.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CDD+03] J.-F. Couchot, F. Dadeau, D. Déharbe, A. Giorgetti, and S. Ranise. Proving and debugging set-based specifications. In *Electronic Notes in Theoretical Computer Science, proceedings of the Sixth Brazilian Workshop on Formal Methods (WMF'03)*, volume 95, pages 189–208, october 2003.

[CDGR03] J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Scalable automated proving and debugging of set-based specifications. *Journal of the Brazilian Computer Society*, 9(2) :17–36, November 2003. ISSN 0104-6500.

[CDGR04] J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Barvey : Vérification automatique de consistance de machines abstraites B. In *Sessions Outils, Congrès Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL'04*, pages 369–372, Besançon, France, jun 2004.

[CDK01] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems. Concepts and design.* Addison Wesley, third edition, 2001.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, 1986.

[CG04] J.-F. Couchot and A. Giorgetti. Analyse d'atteignabilité déductive. In J. Julliand, editor, *Congrès Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL'04*, pages 269–283, 2004.

[CGB86] E M Clarke, O Grumberg, and M C Browne. Reasoning about networks with many identical finite-state processes. In *PODC '86 : Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 1986. ACM Press.

[CGK05] J.-F. Couchot, A. Giorgetti, and N. Kosmatov. A uniform deductive approach for parameterized protocol safety. In *ASE '05 : Proceedings of the 20th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 2005. to appear.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CL93] René Cori and Daniel Lascar. *Logique mathématique. Cours et exercices. Tome 1 : Calcul propositionnel, algèbres de Boole, calcul des prédicats*. Dunod, 1993.

[Cle] ClearSy. site web de b4free. http ://www.b4free.com.

[Cle04] ClearSy. Manuel de référence du langage B v.1.8.5. téléchargeable à http ://www.atelierb.societe.com/ressources/manrefb.185.fr.pdf, 2004.

[CM84] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4) :632–646, 1984.

[Col05] S. Colin. *Procédures de recherche en génération de tests à partir de modéles de spécifications*. PhD thesis, LIFC - University of Franche-Comté, 2005.

[Cou04] J.F. Couchot. Vérification d'invariant par superposition. In *MAnifestation de JEunes Chercheurs STIC (MAJECSTIC'04)*, Calais, France, october 2004.

[DB01] Giorgio Delzanno and Tevfik Bultan. Constraint-based verification of client-server protocols. In T. Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *LNCS*, pages 286–301, Paphos, Cyprus, December 2001.

[Del00] Giorgio Delzanno. Automatic verification of parameterized cache coherence protocols. In *CAV '00 : Proceedings of the 12th International Conference on Computer Aided Verification*, pages 53–68, London, UK, 2000. Springer-Verlag.

[DEP99] Giorgio Delzanno, Javier Esparza, and Andreas Podelski. Constraint-based analysis of broadcast protocols. In *CSL*, pages 50–66, 1999.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe (FME'93)*, volume 670 of *LNCS*, pages 268–284. Springer-Verlag, 1993.

[DG03] F. Dadeau and A. Giorgetti. Vérification de machines abstraites B en logique monadique du second ordre. Rapport de Recherche RR2003-01, LIFC - Laboratoire d'Informatique de l'Université de Franche Comté, October 2003. 29 pages.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9) :569, 1965.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8) :453–457, 1975.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Formal Models and Sematics*, volume B, pages 243–320. MIT Press, 1990.

[DNR04] René David, Karim Nour, and Christophe Raffalli. *Introduction à la logique. Théorie de la démonstration*. Dunod, 2004.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify : a theorem prover for program checking. *J. ACM*, 52(3) :365–473, 2005.

[DP99] Giorgio Delzanno and Andreas Podelski. Model checking in clp. In *TACAS '99 : Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 223–239, London, UK, 1999. Springer-Verlag.

[DP01] Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *STTT*, 3(3) :250–270, 2001.

[DR02] D. Déharbe and S. Ranise. BDD-driven first-order satisfiability procedures (extended version). Rapport de recherche 4630, LORIA, 2002.

[DR03] D. Déharbe and S. Ranise. Applying light-weight theorem proving to debugging and verifying pointer programs. In Ingo Dahn and Laurent Vigneron, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.

[EFM99] Javier Esparza, Alain Finkel, and Richard Mayr. On the verification of broadcast protocols. In *LICS '99 : Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, page 352, Washington, DC, USA, 1999. IEEE Computer Society.

[EK04] E. Allen Emerson and Vineet Kahlon. Parameterized model checking of ring-based message passing systems. In *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 325–339. Springer, 2004.

[EN95] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *POPL '95 : Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 85–94, New York, NY, USA, 1995. ACM Press.

[EN96] E. Allen Emerson and Kedar S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *CAV '96 : Proceedings of the 8th International Conference on Computer Aided Verification*, pages 87–98, London, UK, 1996. Springer-Verlag.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic*. Ac. Press, Inc., 1972.

[FG03] Pascal Fontaine and E. Pascal Gribomont. Decidability of invariant validation for parameterized systems. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 97–112. Springer-Verlag, 2003.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp - a protocol validation and verification toolbox. In *CAV '96 : Proceedings of the 8th International Conference on Computer Aided Verification*, pages 437–440, London, UK, 1996. Springer-Verlag.

[Fon04] Pascal Fontaine. *Techniques for verification of concurrent systems with invariants*. PhD thesis, Institut Montefiore, Université de Liège, Belgium, September 2004.

[FS01] A. Finkel and Ph. Schnoebelen. Well structured transition systems everywhere ! *Theoretical Computer Science*, 256 :63–92, 2001.

[Ger00] S. M. German. Personal Communication, 2000.

[GG91] P. Gochet and P. Gribomont. *Logique*, volume 1. Hermès, 1991.

[GGT00] P. Gochet, P. Gribomont, and A. Thayse. *Logique 3. Méthodes pour l'intelligence artificielle*. Hermès, 2000.

[GLT97] H. Guyennet, J.-C. Lapayre, and M. Tréhel. The pilgrim : A new consistency protocol for distributed shared memory. In *Third IEEE International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP*, pages 253–264, Melbourne, Australia, December 1997.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL (A theorem-proving environment for higher order logic)*. Cambridge University Press, 1993.

[GN01] H. Ganzinger and R. Nieuwenhuis. Constraints and theorem proving. In H. Comon, C. Marché, and Treinen R., editors, *Contraints in Computational Logics, International Summer School, September 5–8, 1999, Gif-sur-Yvette, France, Edition*, volume 2002 of *Lecture Notes in Computer Science*, pages 159–201. Springer-Verlag, 2001.

[GR95] Pascal Gribomont and Didier Rossetto. CAVEAT : technique and tool for computer aided verification and transformation. In Uffe H. Engberg, Kim G. Larsen, and Arne Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for The Construction and Analysis of Systems, TACAS* (Aarhus, Denmark, 19–20 May, 1995), number NS-95-2 in Notes Series, pages 216–229, Department of Computer Science, University of Aarhus, May 1995. BRICS.

[Gri00] E. Pascal Gribomont. Simplification of boolean verification conditions. *Theor. Comput. Sci.*, 239(1) :165–185, 2000.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In *Conference on Computer Aided Verification CAV'97, Haifa, Israel*, volume 1254 of *LNCS*. Springer-Verlag, July 1997.

[GZ98] E. Pascal Gribomont and Guy Zenner. Automated verification of szymanski's algorithm. In *TACAS '98 : Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 424–438, London, UK, 1998. Springer-Verlag.

[Han93] Jim Handy. *The cache memory book*. Academic Press Professional, Inc., San Diego, CA, USA, 1993.

[HJJ+96] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona : Monadic second-order logic in practice. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *LNCS*, pages 89–110, 1996.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5) :279–295, 1997.

[HT01] A. Housni and M. Tréhel. A new distributed mutual exclusion algorithm for two groups. In *ACM Symposium on Applied Computing (SAC'2001)*, pages 531–538, Las Vegas, USA, March 2001.

[JB98] Jacques Julliand and Françoise Bellegarde. Extension de spécifications B par de la logique temporelle linéaire pour décrire des propriétés dynamiques de systèmes réactifs. In *AFADL'98*, pages 125–136, 1998.

[Kal01] J. Kalman. *Automated Reasoning with Otter*. Rinton Press, 2001.

[KEW+85] Randy H. Katz, Susan J. Eggers, David A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th Annual Symposium on Computer Architecture, ISCA'85*, pages 276–283, 1985.

[KL04] Olga Kouchnarenko and Arnaud Lanoix. Verifying invariants of component-based systems through refinement. In *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16,*

2004, Proceedings, Lecture Notes in Computer Science, pages 289–303. Springer, 2004.

[Kle67] Stephen Kleene. *Mathematical Logic*. John Wiley and Sons, 1967.

[KMM+97] Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. Symbolic model checking with rich ssertional languages. In *Computer Aided Verification, 9th International Conference, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 424–435. Springer, 1997.

[Lah04] Shuvendu K. Lahiri. *Unbounded System Verification Using Decision Procedure and Predicate Abstraction*. PhD thesis, Carnegie Mellon University, 2004.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8) :453–455, 1974.

[Lan96] Kevin Lano. *The B Language and Method : A guide to Practical Formal Development*. Springer Verlag London Ltd., 1996.

[LB03] Michael Leuschel and Michael Butler. ProB : A model checker for B. In Araki Keijiro, Gnesi Stefania, and Mandrioli Dino, editors, *FME 2003 : Formal Methods*, LNCS 2805, pages 855–874. Springer–Verlag, 2003.

[LB04] Shuvendu K. Lahiri and Randal E. Bryant. Constructing quantified invariants via predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004*, volume 2937 of *Lecture Notes in Computer Science*, pages 267–281. Springer, 2004.

[LP02] B. Legeard and F. Peureux. B-Testing-Tools : génération de tests aux limites à partir de spécifications B. *TSI (Technique et Science Informatiques*, 21(9) :1189–1218, 2002.

[LS04] Shuvendu K. Lahiri and Sanjit A. Seshia. The uclid decision procedure. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, Lecture Notes in Computer Science, pages 475–478. Springer, 2004.

[Mai01] Monika Maidl. A unifying model checking approach for safety properties of parameterized systems. In *Computer Aided Verification, 13th International Conference, CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 2001.

[Mar03] Pierre Marchand. *Mathématiques discrètes. Automates, langages, logique et décidabilité*. Dunod, 2003.

[McC64] J. McCarthy. A basis for a mathematical theory of computation. In P. Brafford and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. Amsterdam : North-Holland, 1964.

[McM92] K. L. McMillan. *The SMV system*. Carnegie-Mellon University, 1992.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., 1992.

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems : safety*. Springer-Verlag New York, Inc., 1995.

[NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.

[NW01] A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science, 2001.

[OR00] Sam Owre and Harald Rueß. Integrating WS1S with PVS. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 548–551, Chicago, IL, July 2000. Springer-Verlag.

[ORSSC98] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS : an experience report. In *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345. Springer-Verlag, 1998.

[OS88] F. Oppacher and E. Suen. Harp : a tableau-based theorem prover. *J. Autom. Reason.*, 4(1) :69–100, 1988.

[PD97] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Comput. Surv.*, 29(1) :82–126, 1997.

[PLT00] Laurent Py, Bruno Legeard, and Bruno Tatibouët. Évaluation de spécifications formelles en programmation logique avec contraintes ensemblistes – application à l'animation de spécifications formelles B. In *AFADL'2000*, pages 21–35, 2000.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS'77) Providence, Rhode Island, USA*, pages 46–57, 1977.

[PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84 : Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, New York, NY, USA, 1984. ACM Press.

[Pra79] V. R. Pratt. Process logic : preliminary report. In *POPL '79 : Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 93–100, New York, NY, USA, 1979. ACM Press.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*. Springer, 2001.

[Ray92] M. Raynal. *Synchronisation et état global dans les systèmes répartis*. Eyrolles, 1992.

[RB86] M. Raynal and D. Beeson. *Algorithms for mutual exclusion*. MIT Press, Cambridge, MA, USA, 1986.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1) :23–41, 1965.

[RS99] Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *TACAS '99 : Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 178–192, London, UK, 1999. Springer-Verlag.

[RSB+99] Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau, and Dominique Schoen. Applying formal proof techniques to

avionics software : A pragmatic approach. In *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume II*, Lecture Notes in Computer Science, pages 1798–1815. Springer, 1999.

[Rus89] Michael Rusinowitch. *Démonstration automatique : techniques de réécriture.* InterEditions, 1989.

[Rus03a] Michaël Rusinowitch. Automated analysis of security protocols. *Electr. Notes Theor. Comput. Sci.*, 86(3), 2003.

[Rus03b] Vlad Rusu. Compositional verification of an atm protocol. In *FME 2003 : Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, Lecture Notes in Computer Science, pages 223–243. Springer, 2003.

[RV02a] Tatiana Rybina and Andrei Voronkov. Brain : Backward reachability analysis with integers. In *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002*, volume 2422 of *Lecture Notes in Computer Science*, pages 489–494. Springer, 2002.

[RV02b] Tatiana Rybina and Andrei Voronkov. Using canonical representations of solutions to speed up infinite-state model checking. In *Computer Aided Verification, 14th International Conference, CAV'02*, Lecture Notes in Computer Science, pages 386–400. Springer, 2002.

[RV03a] Tatiana Rybina and Andrei Voronkov. Fast infinite-state model checking in integer-based systems (invited lecture). In *Computer Science Logic, 17th International Workshop, CSL'03*, volume 2803 of *Lecture Notes in Computer Science*, pages 546–573. Springer, 2003.

[RV03b] Tatiana Rybina and Andrei Voronkov. A logical reconstruction of reachability. In *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI'03*, volume 2890 of *Lecture Notes in Computer Science*, pages 222–237. Springer, 2003.

[RW69] G. A. Robinson and L. Wos. Paramodulation and theorem proving in first order theories with equality. *Machine Intelligence*, (4) :135–150, 1969.

[Sai97] Hassen Saidi. The invariant checker : Automated deductive verification of reactive systems. In *Computer Aided Verification, 9th International Conference, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 436–439. Springer, 1997.

[SBB+99] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, and A. Petit. *Vérification de logiciels : Techniques et outils du model-checking.* Vuibert, 1999.

[Sch04] S. Schulz. System Description : E 0.81. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd IJCAR, Cork, Ireland*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

[Sif82] Joseph Sifakis. A unified approach for studying the properties of transition systems. *Theor. Comput. Sci.*, 18 :227–258, 1982.

[SP89] E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *PODC '89 : Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 177–191, New York, NY, USA, 1989. ACM Press.

[Sri93] Divesh Srivastava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Ann. Math. Artif. Intell.*, 8(3-4) :315–343, 1993.

[Sti92] Colin Stirling. Modal and temporal logic. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science : Background - Computational Structures (Volume 2)*, pages 477–563. Clarendon Press, Oxford, 1992.

[Suz88] Ichiro Suzuki. Proving properties of a ring of finite-state machines. *Inf. Process. Lett.*, 28(4) :213–214, 1988.

[Szy88] B. K. Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In *ICS '88 : Proceedings of the 2nd international conference on Supercomputing*, pages 621–626, New York, NY, USA, 1988. ACM Press.

[TH01] M. Tréhel and A. Housni. Comparison of some techniques in prioritized mutual exclusion algorithms by groups. In *PDCAT 2001 (International Conference on Parallel and Distributed Computing Applications and Techniques), sponsored by ACM Taipei Chapter and IEEE Taipei Chapter*, pages 259–264, Tamkang University, Taipei County, Taiwan, July 2001.

[TS87] Charles P. Thacker and Lawrence C. Stewart. Firefly : a multiprocessor workstation. In *ASPLOS-II : Proceedings of the second international conference on Architectual support for programming languages and operating systems*, pages 164–172, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[TV01] B. Tatibouet and J.C. Voisinet. jBTools and B2UML : a platform and a tool to provide a UML class diagram since a B specification. In *ICSSEA 2001, 14th International Conference on Software & Systems Engineering and Their Applications, Session 8-3, Formal Methods*, volume 2, France, Paris, December 2001.

[WB95] Pierre Wolper and Bernard Boigelot. An automata-theoretic approach to presburger arithmetic constraints (extended abstract). In *SAS '95 : Proceedings of the Second International Symposium on Static Analysis*, pages 21–32, London, UK, 1995. Springer-Verlag.

[Wei99] Christoph Weidenbach. System description : Spass version 1.0.0. In *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction,1999*, volume 1632 of *Lecture Notes in Computer Science*, pages 378–382. Springer, 1999.

[Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. In *Handbook of Automated Reasoning*, pages 1965–2013. Elsevier and MIT Press, 2001.

[Wor96] J. B. Wordsworth. *Software engineering with B.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[YKBB05] Tuba Yavuz-Kahveci, Constantinos Bartzis, and Tevfik Bultan. Action language verifier, extended. In *Computer Aided Verification, 17th International Conference, CAV'05*, volume 3576 of *Lecture Notes in Computer Science*, pages 413–417. Springer, 2005.

[YKTB01] Tuba Yavuz-Kahveci, Murat Tuncer, and Tevfik Bultan. A library for composite symbolic representations. In *TACAS 2001 : Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 52–66, London, UK, 2001. Springer-Verlag.

[ZP04] Lenore D. Zuck and Amir Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Computer Languages, Systems & Structures*, 30(3-4) :139–169, 2004.

Résumé

Le prouveur de théorèmes haRVey implante une procédure de décision pour les formules équationnelles sans quantificateurs dans les signatures des théories comme la théorie des tableaux avec extensionnalité, la théorie des listes, ou leur combinaison. Ce travail étudie l'applicabilité de ce prouveur à la vérification, par construction d'invariants, de propriétés de sûreté exprimées sur des systèmes paramétrés.

Le développement d'un programme ne s'effectuant jamais en une seule étape, il est souvent pratique de commencer en introduisant des paramètres, quitte à les préciser dans des étapes ultérieures. La correction du programme ainsi paramétré doit être assurée pour toutes les valeurs potentielles de ses paramètres, interdisant les techniques classiques de model-checking en raison de l'infinitude du graphe d'exécution. Vérifier le programme pour toutes les valeurs possibles des paramètres peut s'entreprendre par un calcul de point fixe déductif dont les conditions d'évolution sont déchargées dans un prouveur supportant les formules quantifiées. Pour mettre en œuvre ce calcul symbolique, il est suffisant de disposer d'un calcul de plus faible précondition d'assertions et d'une procédure permettant de décider la négation et la conjonction des formules engendrées.

Graf et Saïdi ont ouvert cette voie en déchargeant leurs conditions d'évolution dans le prouveur PVS. D'autres personnes ont suivi en exprimant leur système à l'aide de contraintes arithmétiques linéaires obtenues par une abstraction de comptage. Néanmoins, l'introduction d'une telle structure linéaire n'est pas toujours naturelle et est limitée à certaines classes de problèmes bien spécifiques. Nous suggérons un cadre plus basique mais unificateur où les paramètres sont des ensembles finis sans structure précise. Nous montrons que ce cadre correspond à la fois à des exemples industriels et à des systèmes uniformes distribués.

Nous utilisons le langage ensembliste des machines abstraites de la méthode formelle B pour spécifier les programmes et exploitons les calculs de plus faible précondition déjà définis pour cette méthode. Nous introduisons ensuite un calcul de point fixe, obtenu par optimisations successives d'un calcul de point fixe trivial.

Nous proposons ensuite différentes traductions en logique équationnelle des formules correspondantes aux conditions d'évolution des calculs du point fixe, dont l'unique objectif est de faire converger le plus rapidement possible le prouveur haRVey qui les décharge.

Théoriquement, nous montrons la décidabilité de certaines obligations de preuve dans le cadre de systèmes uniformes distribués se synchronisant par rendez-vous ou par broadcast. Nos expérimentations menées sur un exemple industriel montrent que la démarche supplante celle basée sur la logique WS1S. Celles menées sur des algorithmes d'exclusion mutuelle ou de cohérence de caches viennent confirmer l'effectivité et l'efficacité de cette démarche.

Mots-clés: vérification, harvey, invariant, méthode B, superposition, paramètres, systèmes uniformes distribués