# Inférence d'invariants pour le model checking de systèmes paramétrés

Alain Mebsout

▶ **To cite this version:**

Alain Mebsout. Inférence d'invariants pour le model checking de systèmes paramétrés. Autre [cs.OH]. Université Paris Sud - Paris XI, 2014. Français. NNT : 2014PA112188 . tel-01073980

**HAL Id: tel-01073980**
**https://theses.hal.science/tel-01073980**

Submitted on 11 Oct 2014

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Université Paris-Sud

École Doctorale d'Informatique
Laboratoire de Recherche en Informatique
Discipline : Informatique


## Thèse de doctorat

soutenue le 29 septembre 2014

par


**Alain Mebsout**


---

# Inférence d'Invariants pour le Model Checking de Systèmes Paramétrés

---

| | | |
|---|---|---|
| Directeur de thèse : | M. Sylvain Conchon | Professeur (Université Paris-Sud) |
| Co-encadrante : | Mme Fatiha Zaïdi | Maître de conférences (Université Paris-Sud) |

**Composition du jury :**

| | | |
|---|---|---|
| Président du jury : | M. Philippe Dague | Professeur (Université Paris-Sud) |
| Rapporteurs : | M. Ahmed Bouajjani | Professeur (Université Paris Diderot) |
| | M. Silvio Ranise | Chercheur (Fondazione Bruno Kessler) |
| Examinateurs : | M. Rémi Delmas | Ingénieur de recherche (ONERA) |
| | M. Alan Schmitt | Chargé de recherche (Inria Rennes) |

*À Magali.*

## Résumé

Cette thèse aborde le problème de la vérification automatique de systèmes paramétrés complexes. Cette approche est importante car elle permet de garantir certaines propriétés sans connaître *a priori* le nombre de composants du système. On s'intéresse en particulier à la sûreté de ces systèmes et on traite le côté paramétré du problème avec des méthodes symboliques. Ces travaux s'inscrivent dans le cadre théorique du *model checking modulo théories* et ont donné lieu à un nouveau model checker : Cubicle.

Une des contributions principale de cette thèse est une nouvelle technique pour inférer des invariants de manière automatique. Le processus de génération d'invariants est intégré à l'algorithme de model checking et permet de vérifier en pratique des systèmes hors de portée des approches symboliques traditionnelles. Une des applications principales de cet algorithme est l'analyse de sûreté paramétrée de protocoles de cohérence de cache de taille industrielle.

Enfin, pour répondre au problème de la confiance placée dans le model checker, on présente deux techniques de certification de notre outil Cubicle utilisant la plate-forme Why3. La première consiste à générer des certificats dont la validité est évaluée de manière indépendante tandis que la seconde est une approche par vérification déductive du cœur de Cubicle.



## Abstract

This thesis tackles the problem of automatically verifying complex parameterized systems. This approach is important because it can guarantee that some properties hold without knowing *a priori* the number of components in the system. We focus in particular on the safety of such systems and we handle the parameterized aspect with symbolic methods. This work is set in the theoretical framework of the *model checking modulo theories* and resulted in a new model checker: Cubicle.

One of the main contribution of this thesis is a novel technique for automatically inferring invariants. The process of invariant generation is integrated with the model checking algorithm and allows the verification in practice of systems which are out of reach for traditional symbolic approaches. One successful application of this algorithm is the safety analysis of industrial size parameterized cache coherence protocols.

Finally, to address the problem of trusting the answer given by the model checker, we present two techniques for certifying our tool Cubicle based on the framework Why3. The first consists in producing certificates whose validity can be assessed independently while the second is an approach by deductive verification of the heart of Cubicle.

# Remerciements

Mes remerciements vont en tout premier lieu à mes encadrants de thèse, qui m'ont accompagné tout au long de ces années. Ils ont su orienter mes recherches tout en m'accordant une liberté et une autonomie très appréciables. Les discussions aussi bien professionnelles que personnelles ont été très enrichissantes et leur bonne humeur a contribué à faire de ces trois (et cinq) années un plaisir. Merci à Fatiha pour sa disponibilité et son soutien. Je tiens particulièrement à remercier Sylvain d'avoir cru en moi et pour son optimisme constant. Il m'apparaît aujourd'hui que cette dernière qualité est essentielle à tout bon chercheur et j'espère en avoir tiré les enseignements.

Merci également à mes rapporteurs, Ahmed Bouajjani et Silvio Ranise, d'avoir accepté de relire une version préliminaire de ce document et pour leurs corrections et remarques pertinentes. Je remercie aussi les autres examinateurs, Philippe Dague, Rémi Delmas et Alan Schmitt, d'avoir accepté de faire partie de mon jury.

Le bon déroulement de cette thèse doit aussi beaucoup aux membres de l'équipe VALS (anciennement Proval puis Toccata). Ils ont su maintenir une ambiance chaleureuse et amicale tout au long des divers changements de noms, déménagements et remaniements politiques. Les galettes des rois, les gâteaux, les paris footballistiques, et les discussions du coin café resteront sans aucun doute gravés dans ma mémoire. Pour tout cela je les en remercie.

Je tiens à remercier tout particulièrement Mohamed pour avoir partagé un bureau avec moi pendant près de cinq années. J'espère avoir été un aussi bon co-bureau qu'il l'a été, en tout cas les collaborations et les discussions autour d'Alt-Ergo furent très fructueuses. Merci également à Régine pour son aide dans la vie de doctorant d'un laboratoire de recherche et pour sa capacité à transformer la plus intimidante des procédures administratives en une simple tâche.

Merci à Jean-Christophe, Guillaume, François, Andrei, Évelyne et Romain pour leurs nombreux conseils et aides techniques. Merci aussi à Sava Krstiċ et Amit Goel d'Intel pour leur coopération scientifique autour de Cubicle.

Enfin un grand merci à toute ma famille pour leur soutien moral tout au long de ma thèse. Je souhaite remercier en particulier mes parents qui ont cru en moi et m'ont soutenu dans mes choix. Merci également à mes sœurs pour leurs nombreux encouragements. Pour finir je tiens à remercier ma moitié, Magali, pour son soutien sans faille et pour son aide. Rien de ceci n'aurait été possible dans elle.

# Table des matières

**Table des matières** 1

**Table des figures** 5

**Liste des Algorithmes** 7

**1 Introduction** 9
- 1.1 Model checking . . . 10
  - 1.1.1 Systèmes finis . . . 11
  - 1.1.2 Systèmes infinis . . . 12
- 1.2 Model checking de systèmes paramétrés . . . 14
  - 1.2.1 Méthodes incomplètes . . . 14
  - 1.2.2 Fragments décidables . . . 15
- 1.3 Contributions . . . 16
- 1.4 Plan de la thèse . . . 17

**2 Le model checker Cubicle** 19
- 2.1 Langage d'entrée . . . 20
- 2.2 Exemples . . . 23
  - 2.2.1 Algorithme d'exclusion mutuelle . . . 23
  - 2.2.2 Généralisation de l'algorithme de Dekker . . . 25
  - 2.2.3 Boulangerie . . . 28
  - 2.2.4 Cohérence de Cache . . . 30
- 2.3 Non-atomicité . . . 33
- 2.4 Logique multi-sortée et systèmes à tableaux . . . 36
  - 2.4.1 Syntaxe des formules logiques . . . 37
  - 2.4.2 Sémantique de la logique . . . 39
  - 2.4.3 Systèmes de transition à tableaux . . . 41
- 2.5 Sémantique . . . 44
  - 2.5.1 Sémantique opérationnelle . . . 44
  - 2.5.2 Atteignabilité . . . 46
  - 2.5.3 Un interpréteur de systèmes à tableaux . . . 47

**3 Cadre théorique : model checking modulo théories** **49**
3.1 Analyse de sûreté des systèmes à tableaux . . . 50
3.1.1 Atteignabilité par chaînage arrière . . . 50
3.1.2 Correction . . . 53
3.1.3 Effectivité . . . 54
3.2 Terminaison . . . 57
3.2.1 Indécidabilité de l'atteignabilité . . . 57
3.2.2 Conditions pour la terminaison . . . 59
3.2.3 Exemples . . . 64
3.3 Gardes universelles . . . 68
3.3.1 Travaux connexes . . . 68
3.3.2 Calcul de pré-image approximé . . . 69
3.3.3 Exemples . . . 70
3.3.4 Relation avec le modèle de panne franche et la relativisation des quantificateurs . . . 71
3.4 Conclusion . . . 74
3.4.1 Exemples sans existence d'un bel ordre . . . 74
3.4.2 Résumé . . . 76
3.4.3 Discussion . . . 77

**4 Optimisations et implémentation** **79**
4.1 Architecture . . . 80
4.2 Optimisations . . . 82
4.2.1 Appels au solveur SMT . . . 82
4.2.2 Tests ensemblistes . . . 85
4.2.3 Instantiation efficace . . . 88
4.3 Suppressions a posteriori . . . 91
4.4 Sous-typage . . . 96
4.5 Exploration parallèle . . . 99
4.6 Résultats et conclusion . . . 102

**5 Inférence d'invariants** **105**
5.1 Atteignabilité approximée avec retour en arrière . . . 107
5.1.1 Illustration sur un exemple . . . 107
5.1.2 Algorithme abstrait . . . 111
5.1.3 Algorithme complet pour les systèmes à tableaux . . . 113
5.2 Heuristiques et détails d'implémentation . . . 117
5.2.1 Oracle : exploration avant bornée . . . 117
5.2.2 Extraction des candidats invariants . . . 119
5.2.3 Retour en arrière . . . 121

5.2.4 Invariants numériques . . . 123
5.2.5 Implémentation dans Cubicle . . . 124
5.3 Évaluation expérimentale . . . 127
5.4 Étude de cas : Le protocole FLASH . . . 130
5.4.1 Description du protocole FLASH . . . 131
5.4.2 Vérification du FLASH : État de l'art . . . 133
5.4.3 Modélisation dans Cubicle . . . 135
5.4.4 Résultats . . . 137
5.5 Travaux connexes sur les invariants . . . 139
5.5.1 Génération d'invariants . . . 139
5.5.2 Cutoffs . . . 140
5.5.3 Abstraction . . . 141
5.6 Conclusion . . . 142

**6 Certification** **145**
6.1 Techniques de certification d'outils de vérification . . . 145
6.2 La plateforme de vérification déductive Why3 . . . 147
6.3 Production de certificats . . . 147
6.3.1 Invariants inductifs pour l'atteignabilité arrière . . . 147
6.3.2 Invariants inductifs et BRAB . . . 152
6.4 Preuve dans Why3 . . . 156
6.5 Discussion . . . 158

**7 Conclusion et perspectives** **161**
7.1 Résumé des contributions et conclusion . . . 161
7.2 Perspectives . . . 162

**A Syntaxe et typage des programmes Cubicle** **165**
A.1 Syntaxe . . . 165
A.2 Typage . . . 166

**Bibliographie** **173**

**Index** **187**

# Table des figures


2.1 Algorithme d'exclusion mutuelle . . . . . 24
2.2 Code Cubicle du mutex . . . . . 25
2.3 Graphe de Dekker pour le processus $i$ . . . . . 26
2.4 Code Cubicle de l'algorithme de Dekker . . . . . 27
2.5 Code Cubicle de l'algorithme de la boulangerie de Lamport . . . . . 29
2.6 Diagramme d'état du protocole German-*esque* . . . . . 31
2.7 Code Cubicle du protocole de cohérence de cache German-*esque* . . . . . 32
2.8 Évaluation non atomique des conditions globales par un processus $i$ . . . . 34
2.9 Encodage de l'évaluation non atomique de la condition globale $\forall j \neq i.\ c(j)$ 35
2.10 Grammaire de la logique . . . . . 38

3.1 Machine de Minsky à deux compteurs . . . . . 58
3.2 Traduction d'un programme et d'une machine de Minsky dans un système à tableaux . . . . . 59
3.3 Définition des configurations $\mathcal{M}$ et $\mathcal{N}$ . . . . . 61
3.4 Plongement de $\mathcal{M}$ vers $\mathcal{N}$ . . . . . 61
3.5 Séquence finie d'idéaux inclus calculée par l'algorithme 4 . . . . . 64
3.6 Trace fallacieuse pour un système avec gardes universelles . . . . . 71
3.7 Transformation avec modèle de panne franche . . . . . 72
3.8 Trace fallacieuse pour la transformation avec modèle de panne franche . . 73
3.9 Relations entre les différentes approches pour les gardes universelles . . . 74
3.10 Séquence infinie de configurations non comparables pour l'encodage des machines de Minksy . . . . . 75
3.11 Séquence infinie de configurations non comparables pour une relation binaire quelconque . . . . . 76
3.12 Différences de restrictions entre la théorie du model checking modulo théories et Cubicle . . . . . 78

4.1 Architecture de Cubicle . . . . . 81
4.2 Benchmarks et statistiques pour une implémentation naïve . . . . . 83
4.3 Arbre préfixe représentant $\mathcal{V}$ . . . . . 87
4.4 Benchmarks pour tests ensemblistes . . . . . 87

4.5 Benchmarks pour l'instantiation efficace . . . 91
4.6 Graphes d'atteignabilité arrière pour différentes stratégies d'exploration sur l'exemple Dijkstra . . . 93
4.7 Suppression *a posteriori* . . . 94
4.8 Préservation de l'invariant après suppression d'un nœud . . . 95
4.9 Benchmarks pour la suppression *a posteriori* . . . 95
4.10 Système Cubicle annoté avec les contraintes de sous-typage . . . 97
4.11 Benchmarks pour l'analyse de sous-typage . . . 98
4.12 Une mauvaise synchronisation des tests de subsomption effectués en parallèle 100
4.13 Utilisation CPU pour les versions séquentielle et parallèle de Cubicle . . . 101
4.14 Benchmarks . . . 103

5.1 Système de transition à tabeaux du protocole German-*esque* . . . 108
5.2 Exécution partielle de BRAB sur le protocole German-*esque* . . . 110
5.3 Transition sur un état avec variable non-initialisée . . . 119
5.4 Apprentissage à partir d'une exploration supplémentaire avant redémarrage 122
5.5 Architecture de Cubicle avec BRAB . . . 125
5.6 Résultats de BRAB sur un ensemble de benchmarks . . . 128
5.7 Architecture FLASH d'une machine et d'un nœud . . . 130
5.8 Structure d'un message dans le protocole FLASH . . . 132
5.9 Description des transitions du protocole FLASH . . . 134
5.10 Résultats pour la vérification du protocole FLASH avec Cubicle . . . 138
5.11 Protocoles de cohérence de cache hiérarchiques . . . 143

6.1 Invariants inductifs calculés par des analyses d'atteignabilité avant et arrière 148
6.2 Vérification du certificat Why3 de German-*esque* par différents prouveurs automatiques . . . 153
6.3 Invariant inductif calculé par BRAB . . . 154
6.4 Vérification par différents prouveurs automatiques du certificat (Why3) de German-*esque* généré par BRAB . . . 154
6.5 Vérification de certificats sur un ensemble de benchmarks . . . 155
6.6 Informations sur le développent Why3 . . . 158
6.7 Aperçu d'une technique d'extraction . . . 159

A.1 Grammaire des fichiers Cubicle . . . 167
A.2 Règles de typage des termes . . . 169
A.3 Règles de typage des formules . . . 170
A.4 Règles de typage des actions des transitions . . . 170
A.5 Vérification de la bonne formation des types . . . 171
A.6 Règles de typage des déclarations . . . 171

# Liste des Algorithmes


1 Code de Dekker pour le processus $i$ . . . . . 26
2 Pseudo-code de la boulangerie de Lamport pour le processus $i$ . . . . . 29
3 Interpréteur d'un système à tableaux . . . . . 47

4 Analyse d'atteignabilité par chaînage arrière . . . . . 53
5 Test de satisfiabilité naïf . . . . . 56

6 Analyse d'atteignabilité abstraite avec approximations et retour en arrière (BRAB) . . . . . 112
7 Analyse d'atteignabilité avec approximations et retour en arrière (BRAB) . . 115
8 Oracle : Exploration avant limitée en profondeur . . . . . 116

# 1

# Introduction

## Sommaire


**1.1 Model checking . . . . . 10**
1.1.1 Systèmes finis . . . . . 11
1.1.2 Systèmes infinis . . . . . 12
**1.2 Model checking de systèmes paramétrés . . . . . 14**
1.2.1 Méthodes incomplètes . . . . . 14
1.2.2 Fragments décidables . . . . . 15
**1.3 Contributions . . . . . 16**
**1.4 Plan de la thèse . . . . . 17**


Les systèmes informatiques sont aujourd'hui omniprésents, aussi bien dans les objets anodins de la vie courante que dans les systèmes critiques comme les contrôleurs automatiques utilisés par l'industrie aéronautique. Tous ces systèmes sont généralement très complexes, il est donc particulièrement difficile d'en construire qui ne comportent pas d'erreurs. On peut notamment constater le succès récent des architectures multi-cœurs, multi-processeurs et distribuées pour les serveurs haut de gamme mais aussi pour les terminaux mobiles personnels. Un des composants les plus complexes de telles machines est leur protocole de cohérence de cache. En vaut pour preuve le célèbre dicton :

> *« Il y a seulement deux choses compliquées en informatique : l'invalidation des caches et nommer les choses. »*
>
> — Phil Karlton

En effet pour fonctionner de manière optimale chaque composant qui partage la mémoire (processeur, cœur, *etc.*) possède son propre cache (une zone mémoire temporaire) lui permettant de conserver les données auxquelles il a récemment accédé. Le protocole en question assure que tous les caches du système se trouvent dans un état cohérent, ce qui en

fait un élément vital. Les méthodes les plus courantes pour garantir la qualité des systèmes informatiques sont le *test* et la *simulation*. Cependant, ces techniques sont très peu adaptées aux programmes concurrents comme les protocoles de cohérence de cache.

Pour garantir une efficacité optimale, ces protocoles sont souvent implantés au niveau matériel et fonctionnent par échanges de messages, de manière entièrement asynchrone. Le moment auquel un message arrivera ou un processeur accédera à la mémoire centrale est donc totalement imprévisible [36]. Pour concevoir de tels systèmes on doit alors considérer de nombreuses « courses critiques » (ou *race conditions* en anglais), c'est-à-dire des comportements qui dépendent de l'ordre d'exécution ou d'arrivée des messages. Ces scénarios sont par ailleurs très compliqués, faisant intervenir plusieurs dizaines d'échanges de messages. Ainsi, la rareté de leurs apparitions fait qu'il est très difficile de reproduire ces comportements par des méthodes de test et de simulation.

Une réponse à ce problème est l'utilisation de *méthodes formelles* pouvant garantir certaines propriétés d'un système par des arguments mathématiques. De nombreuses techniques revendiquent leur appartenance à cette catégorie, comme le *model checking*, qui s'attache à considérer *tous les comportements possibles* d'un système afin d'en vérifier différentes propriétés.

## 1.1 Model checking

La technique du model checking a été inventée pour résoudre le problème difficile de *la vérification de programmes concurrents*. Avant 1982, les recherches sur ce sujet intégraient systématiquement l'emploi de la preuve manuelle [131]. Pnueli [137], Owicki et Lamport [132] proposèrent, vers la fin des années 70, l'usage de la *logique temporelle* pour spécifier des propriétés parmi lesquelles :

— la sûreté : un mauvais comportement ne se produit jamais, ou

— la vivacité : un comportement attendu finira par arriver.

Le terme *model* dans l'expression « model checking » peut faire référence à deux choses. Le premier sens du terme renvoie à l'idée qu'on va représenter un programme par un *modèle abstrait*. Appelons ce modèle $M$. Le deuxième sens du terme fait référence au fait qu'on va ensuite essayer de vérifier des propriétés de $M$, autrement dit que $M$ est un *modèle* de ces propriétés. Le model checking implique donc de définir « simplement » ce qu'est (1) un modèle, et (2) une propriété d'un modèle.

1. Un modèle doit répondre aux trois questions suivantes :
    a) À tout instant, qu'est-ce qui caractérise *l'état* d'un programme ?
    b) Quel est l'état initial du système ?
    c) Comment passe-t-on d'un état à un autre ?

2. Les propriétés qu'on cherche à vérifier sont diverses : Est-ce qu'un mauvais état peut être atteint (à partir de l'état inital) ? Est-ce qu'un certain état sera atteint quoiqu'il advienne ? Plus généralement, on souhaite vérifier des propriétés qui dépendent de la manière dont le programme (ou le modèle) évolue. On exprime alors ces formules dans des logiques temporelles.

### 1.1.1 Systèmes finis

Les travaux fondateurs de Clarke et Emerson [38] et de Queille et Sifakis [140] intègrent cette notion de logique temporelle à l'exploration de l'ensemble des états du système (espace d'état). Dans leur article de 1981 [38], Clarke et Emerson montrent comment synthétiser des programmes à partir de spécifications dans une logique temporelle (appelée CTL, pour Computational Tree Logic). Mais c'est surtout la seconde partie de l'article qui a retenu l'attention de la communauté. Dans celle-ci, ils construisent une procédure de décision fondée sur des calculs de points-fixes pour vérifier des propriétés temporelles de programmes finis.

L'avantage principal du model checking par rapport aux autres techniques de preuve est double. C'est d'abord un processus automatique et rapide, de sorte qu'il est souvent considéré comme une technique « presse-bouton ». C'est aussi une méthode qui fonctionne déjà avec des spécifications partielles. De cette façon, l'effort de vérification peut être commencé tôt dans le processus de conception d'un système complexe. Un de ses désavantages, qui est aussi inhérent à toutes les autres techniques de vérification, est que la rédaction des spécifications est une tâche difficile qui requiert de l'expertise. Mais l'inconvénient majeur du model checking est apparu dans les années 80, et est connu sous le nom du phénomène d'*explosion combinatoire de l'espace d'états*. Seuls les systèmes avec un nombre petit d'états (de l'ordre du million) pouvaient être analysés à cette époque, alors que les systèmes réels en possèdent beaucoup plus. Par conséquent, un important corpus de recherche sur le model checking aborde le problème du passage à l'échelle.

Une avancée notoire pour le model checking a été l'introduction de techniques symboliques pour résoudre ce problème d'explosion. Alors que la plupart des approches avant 1990 utilisaient des représentations explicites, où chaque état individuel est stocké en mémoire, Burch *et al.* ont montré qu'il était possible de représenter de manière symbolique et compacte des ensembles d'états [29]. McMillan reporte dans sa thèse [116] une représentation de la relation des états de transition avec BDD [27] (diagrammes de décision binaire) puis donne un nombre d'algorithmes basés sur des graphes dans le langage du $\mu$-calcul. L'utilisation de structures de données compactes pour représenter de larges ensembles d'états permet de saisir certaines des régularités qui apparaissent naturellement dans les circuits ou autres systèmes. La complexité en espace du model checking symbolique a été grandement diminuée et, pour la première fois, des systèmes avec $10^{20}$ états ont été vérifiés. Cette limite a encore été repoussée avec divers raffinements du model checking

symbolique.

Le point faible des techniques symboliques utilisant des BDD est que la quantité d'espace mémoire requise pour stocker ces structures peut augmenter de façon exponentielle. Biere *et al.* décrivent une technique appelée le *model checking borné* (ou BMC pour *Bounded Model Checking*) qui sacrifie la correction au profit d'une recherche d'anomalies efficace [19]. L'idée du BMC est de chercher les contre-exemples parmi les exécutions du programme dont la taille est limitée à un certain nombre d'étapes $k$. Ce problème peut être encodé efficacement en une formule propositionnelle dont la statisfiabilité mène directement à un contre-exemple. Si aucune erreur n'est trouvée pour des traces de longueur inférieure à $k$, alors la valeur de la borne $k$ est augmentée jusqu'à ce qu'une erreur soit trouvée, ou que le problème devienne trop difficile à résoudre [1]. La force de cette technique vient principalement de la mise à profit des progrès fait par les solveurs SAT (pour la **sat**isfiabilité booléenne) modernes. Elle a rencontré un succès majeur dans la vérification d'implémentations de circuits matériels et fait aujourd'hui partie de l'attirail standard des concepteurs de tels circuits. Comme l'encodage vers des contraintes propositionnelles capture la sémantique des circuits de manière précise, ces model checkers ont permis de découvrir des erreurs subtiles d'implémentation, dues par exemple à la présence de débordements arithmétiques.

Une extension de cette technique pour la preuve de propriétés, plutôt que la seule recherche de bogues, consiste à ajouter une étape d'induction. Cette extension de BMC s'appelle la *k-induction* et diffère d'un schéma d'induction classique par le point suivant : lorsqu'on demande à vérifier que la propriété d'induction est préservée, on suppose qu'elle est vraie dans les $k$ étapes précédentes plutôt que seulement dans l'étape précédente [52, 151].

Une autre vision du model checking est centrée sur la théorie des automates. Dans ces approches, spécifications et implémentations sont toutes deux construites avec des automates. Vardi et Wolper notent une correspondance entre la logique temporelle et les automates de Büchi [164] : chaque formule de logique temporelle peut être considérée comme un automate à états fini (sur des mots infinis) qui accepte précisément les séquences satisfaites par la formule. Grâce à cette connexion, ils réduisent le problème du model checking à un test de vacuité de l'automate $A_M \cap A_{\neg\varphi}$ (où $A_M$ est l'automate du programme $M$ et $A_{\neg\varphi}$ est l'automate acceptant les séquences qui violent la propriété $\varphi$) [161].

### 1.1.2 Systèmes infinis

Le problème d'explosion combinatoire est encore plus frappant pour des systèmes avec un nombre *infini* d'états. C'est le cas par exemple lorsque les types des variables du

1. Dans certains cas un majorant sur la borne $k$ est connu (le *seuil de complétion*) qui permet d'affirmer que le système vérifie la propriété donnée.

programme sont infinis (*e.g.* entiers mathématiques) ou les structures de données sont infinies (*e.g.* buffers, files d'attente, mémoires non bornés). Pour traiter de tels systèmes, deux possibilités s'offrent alors : manipuler des représentations d'ensembles d'états infinis directement, ou construire une *abstraction finie* du système.

Dans la première approche, des représentations symboliques adaptées reposent par exemple sur l'utilisation des formules logiques du premier ordre. Pour cela, les techniques utilisant précédemment les solveurs SAT (traitant exclusivement de domaines finis encodés par de la logique propositionnelle) ont été peu à peu migrées vers les solveurs SMT (Satisfiabilité Modulo Théories). Ces derniers possèdent un moteur propositionnel (un solveur SAT) couplé à des méthodes de combinaison de théories. La puissance de ces solveurs vient du grand nombre de théories qu'ils supportent en interne, comme la théorie de l'arithmétique linéaire (sur entiers mathématiques), la théorie de l'égalité et des fonctions non interprétées, la théorie des tableaux, la théorie des vecteurs de bits, la théorie des types énumérés, *etc.* Certains solveurs SMT supportent même les quantificateurs. Par exemple, une application récente de la technique de $k$-induction aux programmes Lustre (un langage synchrone utilisé notamment dans l'aéronautique) utilisant des solveurs SMT est disponible dans le model checker Kind [85, 86].

La seconde approche pour les systèmes infinis – qui peut parfois être utilisée en complément de la technique précédente – consiste à sacrifier la précision de l'analyse en simplifiant le problème afin de se ramener à l'analyse d'un système fini. La plupart de ces idées émergent d'une certaine manière du cadre de l'*interprétation abstraite* dans lequel un programme est interprété sur un domaine abstrait [46, 47]. Une forme d'abstraction particulière est celle de l'*abstraction par prédicats*. Dans cette approche, la *fonction d'abstraction* associe chaque état du système à un ensemble prédéfini de prédicats. L'utilisateur fournit des prédicats booléens en nombre fini pour décrire les propriétés possibles d'un système d'états infinis. Ensuite une analyse d'accessibilité est effectuée sur le modèle d'états finis pour fournir l'invariant le plus fort possible exprimable à l'aide de ces prédicats [81].

Généralement, les abstractions accélèrent la procédure de model checking lorsqu'elles sont les plus générales possibles. Parfois trop grossières, ces abstractions peuvent empêcher l'analyse d'un système pourtant sûr en exposant des contre-exemples qui n'en sont pas dans le système réel. Il devient alors intéressant de *raffiner* le domaine abstrait utilisé précédemment de manière automatique afin d'éliminer ces mauvais contre-exemples. Cette idée a donné naissance à la stratégie itérative appelée CEGAR (pour *Counter-Example Guided Abstraction Refinement*, *i.e.* raffinement d'abstraction guidé par contre-exemples) [13, 35, 147]. Elle est utilisée par de nombreux outils et plusieurs améliorations ont été proposées au fil des années. On peut mentionner par exemple les travaux de Jahla *et al.* [89, 94] sur *l'abstraction paresseuse* implémentés dans le model checker Blast [18] ainsi que ceux de McMillan [119] qui combine cette technique avec un raffinement par interpolation de Craig [48] permettant ainsi de capturer les relations entre variables utiles à la preuve de la propriété souhaitée.

Un système peut aussi être infini, non pas parce que ses variables sont dans des domaines infinis, mais parce qu'il est formé d'un nombre non borné de composants. Par exemple, un protocole de communication peut être conçu pour fonctionner quelque soit le nombre de machines qui y participent. Ces systèmes sont dits *paramétrés*, et c'est le problème de leur vérification qui nous intéresse plus particulièrement dans cette thèse.

## 1.2 Model checking de systèmes paramétrés

Un grand nombre de systèmes réels concurrents comme le matériel informatique ou les circuits électroniques ont en fait un nombre fini d'états possibles (déterminés par la taille des registres, le nombre de bascules, *etc.*). Toutefois, ces circuits et protocoles (*e.g.* les protocoles de bus ou les protocoles de cohérence de cache) sont souvent conçus de façon paramétrée, définissant ainsi une infinité de systèmes. Un pour chaque nombre de composants.

Souvent le nombre de composants de tels systèmes n'est pas connu à l'avance car ils sont conçus pour fonctionner quelque soit le nombre de machines du réseau, quelque soit le nombre de processus ou encore quelque soit la taille des buffers (mémoires tampons). En vérifier les propriétés d'une manière *paramétrée* est donc indispensable pour s'assurer de leur qualité. Dans d'autres cas ce nombre de composants est connu mais il est tellement grand (plusieurs milliers) que les techniques traditionnelles sont tout bonnement incapables de raisonner avec de telles quantités. Il est alors préférable dans ces circonstances de vérifier une version paramétrée du problème.

Apt et Kozen montrent en 1986 que, de manière générale, savoir si un programme $P$ paramétré par $n$, $P(n)$ satisfait une propriété $\varphi(n)$, est un problème indécidable [10]. Pour mettre en lumière ce fait, ils ont simplement créé un programme qui simule $n$ étapes d'une machine de Turing et change la valeur d'une variable booléenne à la fin de l'exécution si la machine simulée ne s'est pas encore arrêtée [2]. Ce travail expose clairement les limites intrinsèques des systèmes paramétrés. Même si le résultat est négatif, la vérification automatique reste possible dans certains cas. Face à un problème indécidable, il est coutume de restreindre son champ d'application en imposant certaines conditions jusqu'à tomber dans un fragment décidable. Une alternative consiste à traiter le problème dans sa globalité, mais avec des méthodes non complètes.

### 1.2.1 Méthodes incomplètes

Le premier groupe à aborder le problème de la vérification paramétrée fut Clarke et Grumberg [39] avec une méthode fondée sur un résultat de correspondance entre des

2. Ce programme est bien paramétré par $n$. Bien qu'il ne fasse pas intervenir de concurrence, le résultat qui suit peut être aussi bien obtenu en mettant $n$ processus identiques $P(n)$ en parallèle.

systèmes de différentes tailles. Notamment, ils ont pu vérifier avec cette technique un algorithme d'exclusion mutuelle en mettant en évidence une bisimulation entre ce système de taille $n$ et un système de taille 2.

Toujours dans l'esprit de se ramener à une abstraction finie, de nombreuses techniques suivant ce modèle ont été développées pour traiter les systèmes paramétrés. Par exemple Kurshan et McMillan utilisent un unique processus $Q$ qui agrège les comportements de $n$ processus $P$ concurrents [105]. En montrant que $Q$ est invariant par composition parallèle asynchrone avec $P$, on déduit que $Q$ représente bien une abstraction du système paramétré.

Bien souvent l'enjeu des techniques utilisées pour vérifier des systèmes paramétrés est de trouver une représentation à même de caractériser des familles infinies d'états. Par exemple si la topologie du système (*i.e.* l'organisation des processus) n'a pas d'importance, il est parfois suffisant de « compter » le nombre de processus se trouvant dans un état particulier. C'est la méthode employée par Emerson *et al.* dans l'approche dite d'*abstraction par compteurs* [64]. Si l'ordre des processus importe (*e.g.* un processus est situé « à gauche » d'un autre) une représentation adaptée consiste à associer à chaque état un mot d'un langage régulier. Les ensembles d'états sont alors représentés par les expressions régulières de ce langage et certaines relations de transition peuvent être exprimées par des *transducteurs finis* [1, 99]. Une généralisation de cette idée a donné naissance au cadre du *model checking régulier* [24] qui propose des techniques d'accélération pour calculer les clôtures transitives [96, 128]. Des extensions de ces approches ont aussi été adaptées à l'analyse de systèmes de topologies plus complexes [22].

Plutôt que de chercher à construire une abstraction finie du système paramétré dans son intégralité, l'approche dite de *cutoff* (coupure) cherche à découvrir une borne supérieure dépendant à la fois du système et de la propriété à vérifier. Cette borne $k$, appelée la valeur de *cutoff*, est telle que si la propriété est vraie pour les système plus petits que $k$ alors elle est aussi vraie pour les systèmes de taille supérieure à $k$ [62]. Parfois cette valeur peut-être calculée statiquement à partir des caractéristiques du système. C'est l'approche employée dans la technique des *invariants invisibles* de Pnueli *et al.* alliée à une génération d'invariants inductifs [12, 138]. L'avantage de la technique de *cutoff* est qu'elle s'applique à plusieurs formalismes et qu'elle permet simplement d'évacuer le côté paramétré d'un problème. En revanche le calcul de cette borne $k$ donne souvent des valeurs rédhibitoires pour les systèmes de taille réelle. Pour compenser ce désavantage, certaines techniques ont été mises au point pour découvrir les bornes de *cutoff* de manière dynamique [4, 98].

### 1.2.2 Fragments décidables

Il est généralement difficile d'identifier un fragment qui soit à la fois décidable et utile en pratique. Certaines familles de problèmes admettent cependant des propriétés qui en font des cadres théoriques intéressants. Partant du constat que les systèmes *synchrones* sont souvent plus simples à analyser, Emerson et Namjoshi montrent que certaines propriétés

temporelles sont en fait *decidables* pour ces systèmes [63]. Leur modèle fonctionne pour les systèmes composés d'un processus de contrôle et d'un nombre arbitraire de processus homogènes. Il ne s'applique donc pas, par exemple, aux algorithmes d'exclusion mutuelle. L'écart de méthodologie qui existe entre les méthodes non complètes et celles qui identifient un fragment décidable du model checking paramétré n'est en réalité pas si grand. Dans bien des cas, les chercheurs qui mettent au point ces premières mettent aussi en évidence des restrictions suffisantes sur les systèmes considérés pour rendre l'approche complète et terminante. C'est par exemple le cas des techniques d'accélération du model checking régulier ou encore celui du *model checking modulo théories* proposé par Ghilardi et Ranise [77, 79].

À la différence des techniques pour systèmes paramétrés mentionnées précédemment, cette dernière approche ne construit pas d'abstraction finie mais repose sur l'utilisation de *quantificateurs* pour représenter les ensembles infinis d'états de manière symbolique. En effet lorsqu'on parle de systèmes paramétrés, on exprime naturellement les propriétés en quantifiant sur l'ensemble des éléments du domaine paramétré. Par exemple, dans un système où le nombre de processus n'est pas connu à l'avance, on peut avoir envie de garantir que *quelque soit* le processus, sa variable locale $x$ n'est jamais nulle. Le cadre du model checking modulo théories définit une classe de systèmes paramétrés appelée *systèmes à tableaux* permettant de maîtriser l'introduction des quantificateurs. La sûreté de tels systèmes n'est pas décidable en général mais il existe des restrictions sur le problème d'entrée qui permettent de construire un algorithme complet et terminant. Un autre avantage de cette approche est de tirer parti de la puissance des solveurs SMT et de leur grande versatilité.

Toutes ces techniques attaquent un problème intéressant et important. Celles qui sont automatiques (model checking) ne passent pourtant pas à l'échelle sur des problèmes réalistes. Et celles qui sont applicables en pratique demandent quant à elles une expertise humaine considérable, ce qui rend le processus de vérification très long [33, 49, 118, 134, 135]. Le problème principal auquel répond cette thèse est le suivant.

> *Comment vérifier* automatiquement *des propriétés de sûreté de systèmes paramétrés complexes ?*

Pour répondre à cette question, on utilise le cadre théorique du model checking modulo théories pour concevoir de nouveaux algorithmes qui infèrent des invariants de manière automatique. Nos techniques s'appliquent avec succès à l'analyse de sûreté paramétrée de protocoles de cohérence de cache conséquents, entre autres.

## 1.3 Contributions

Nos contributions sont les suivantes :

— Un model checker open source pour systèmes paramétrés : Cubicle. Cubicle implémente les techniques présentées dans cette thèse et est librement disponible à l'adresse suivante : `http://cubicle.lri.fr`.

— Un ensemble de techniques pour l'implémentation d'un model checker reposant sur un solveur SMT.

— Un nouvel algorithme pour inférer des invariants de qualité de manière automatique : BRAB. L'idée de cet algorithme est d'utiliser les informations extraites d'un modèle fini du système afin d'inférer des invariants pour le cas paramétré. Le modèle fini est en fait mis à contribution en tant qu'oracle dont le rôle se limite à émettre un jugement de valeur sur les candidats invariants qui lui sont présentés. Cet algorithme fonctionne bien en pratique car dans beaucoup de cas les instances finies, même petites, exhibent déjà la plupart des comportements intéressants du système.

— Une implémentation de BRAB dans le model checker Cubicle.

— L'application de ces techniques à la vérification du protocole de cohérence de cache de l'architecture multi-processeurs FLASH. À l'aide des invariants découverts par BRAB, la sûreté de ce protocole a pu être vérifiée *entièrement automatiquement* pour la première fois.

— Deux approches pour la certification de Cubicle à l'aide de la plate-forme Why3. La première est une approche qui fonctionne par certificats (ou traces) et vérifie le résultat produit par le model checker. Ce certificat prend la forme d'un invariant inductif. La seconde approche est par vérification déductive du cœur de Cubicle. Ces deux approches mettent en avant une autre qualité de BRAB : faciliter ce processus de certification, par réduction de la taille des certificats pour l'une, et grâce à son efficacité pour l'autre.

## 1.4 Plan de la thèse

Ce document de thèse est organisé de la façon suivante : Le chapitre 2 introduit le langage d'entrée du model checker Cubicle à travers différents exemples d'algorithmes et de protocoles concurrents. Une seconde partie de ce chapitre présente la représentation formelle des systèmes de transitions utilisés par Cubicle ainsi que leur sémantique. Le chapitre 3 présente le cadre théorique du model checking modulo théories conçu par Ghilardi et Ranise. Les résultats et théorèmes associés sont également donnés et illustrés dans ce chapitre, qui constitue le contexte dans lequel s'inscrivent nos travaux. Le chapitre 4 donne un ensemble d'optimisations nécessaires à l'implémentation d'un model checker reposant sur un solveur SMT comme Cubicle. Nos travaux autour de l'inférence d'invariants sont exposés dans le chapitre 5. On y présente et illustre l'algorithme BRAB. Les détails pratiques de son fonctionnement sont également expliqués et son intérêt est appuyé par une

évaluation expérimentale sur des problèmes difficiles pour la vérification paramétrée. En particulier on détaille le résultat de la preuve du protocole de cohérence de cache FLASH. Le chapitre 6 présente deux techniques de certification que nous avons mis en œuvre pour certifier le model checker Cubicle. Ces deux techniques reposent sur la plate-forme de vérification déductive Why3. Enfin le chapitre 7 donne des pistes d'amélioration et d'extension de nos travaux et conclut ce document.

# 2

# Le model checker Cubicle

## Sommaire


**2.1 Langage d'entrée . . . . . 20**
**2.2 Exemples . . . . . 23**
2.2.1 Algorithme d'exclusion mutuelle . . . . . 23
2.2.2 Généralisation de l'algorithme de Dekker . . . . . 25
2.2.3 Boulangerie . . . . . 28
2.2.4 Cohérence de Cache . . . . . 30
**2.3 Non-atomicité . . . . . 33**
**2.4 Logique multi-sortée et systèmes à tableaux . . . . . 36**
2.4.1 Syntaxe des formules logiques . . . . . 37
2.4.2 Sémantique de la logique . . . . . 39
2.4.3 Systèmes de transition à tableaux . . . . . 41
**2.5 Sémantique . . . . . 44**
2.5.1 Sémantique opérationnelle . . . . . 44
2.5.2 Atteignabilité . . . . . 46
2.5.3 Un interpréteur de systèmes à tableaux . . . . . 47


L'outil issu des travaux présentés dans ce document est un model checker pour systèmes paramétrés, dénommé Cubicle. Le but de ce chapitre est de familiariser le lecteur avec l'outil, tout particulièrement son langage d'entrée et sa syntaxe concrète. Quelques exemples variés de tels systèmes sont formulés dans ce langage afin d'offrir un premier aperçu des possibilités du model checker. On donne également une description plus formelle de la sémantique des systèmes décrits dans le langage de Cubicle.

## 2.1 Langage d'entrée

Cubicle est un model checker pour systèmes paramétrés. C'est-à-dire qu'il permet de vérifier statiquement des propriétés de sûreté d'un programme concurrent pour un nombre quelconque de processus [1]. Pour représenter ces programmes, on utilise des *systèmes de transition* car ils permettent facilement de modéliser des comportements asynchrones ou non déterministes. Ces systèmes décrivent les transitions possibles d'un état du programme à un autre. Ils peuvent être considérés comme un langage bas niveau pour la vérification.

Étant donné que les systèmes manipulés par Cubicle sont paramétrés, les transitions qui le composent le sont également. Dans cette section, on présente de manière très informelle le langage d'entrée de Cubicle [2].

La description d'un système commence par des déclarations de types, de variables globales et de tableaux. Cubicle connaît quatre types en interne : le type des entiers (`int`), le type des réels (`real`), le type des booléens (`bool`) et le type des identificateurs de processus (`proc`). Ce dernier est particulièrement important car c'est par les éléments de ce type que le système est paramétré. Le *paramètre* du système est la cardinalité du type `proc`. L'utilisateur a aussi la liberté de déclarer ses propres types abstraits ou ses propres types énumérés. L'exemple suivant définit un type énuméré `state` à trois constructeurs `Idle`, `Want` et `Crit` ainsi qu'un type abstrait `data`.

```
type state = Idle | Want | Crit
type data
```

Les tableaux et variables représentent l'état du système ou du programme. Tous les tableaux ont la particularité d'être uniquement indexés par des éléments du type `proc`, leur taille est par conséquent inconnue. C'est une limitation de l'implémentation actuelle du langage de Cubicle qui existe pour des raisons pratiques et rien n'empêcherait d'ajouter la possibilité de déclarer des tableaux indexés par des entiers. Dans ce qui suit on déclare une variable globale `Timer` de type `real` et trois tableaux indexés par le type `proc`.

```
var Timer : real
array State[proc] : state
array Chan[proc] : data
array Flag[proc] : bool
```

Les tableaux peuvent par exemple être utilisés pour représenter des variables locales ou des canaux de communication.

1. Un programme concurrent est souvent paramétré par son nombre de processsus ou threads mais ce paramètre peut aussi être la taille de certains buffers ou le nombre de canaux de communication par exemple.
2. Une description plus précise de la syntaxe et de ce langage est faite en annexe A de ce document.

Les états initiaux du système sont définis à l'aide du mot clef `init`. Cette déclaration qui vient en début de fichier précise quelles sont les valeurs initiales des variables et tableaux pour *tous* leurs indices. Notons que certaines variables peuvent ne pas être initialisées et on a le droit de mentionner seulement les relations entres elles. Dans ce cas tout état dont les valeurs des variables et tableaux respectent les contraintes fixées dans la ligne `init` sera un état initial correct du système. Par exemple, la ligne suivante définit les états initiaux comme ceux ayant leurs tableaux `Flag` à false et `State` à Idle pour tout processus `z`, ainsi que leur variable globale `Timer` valant `0.0`. Le paramètre `z` de `init` représente tous les processus du système.

```
init (z) { Flag[z] = False && State[z] = Idle && Timer = 0.0 }
```

*Remarque.* On ne précise pas le type des paramètres car seuls ceux du type `proc` sont autorisés

Le reste du système est donné comme un ensemble de transitions de la forme garde/action. Chaque transition peut être paramétrée (ou non) par un ou plusieurs identificateurs de processus comme dans l'exemple suivant.

```
transition t (i j)
requires { i < j && State[i] = Idle && Flag[i] = False &&
           forall_other k. (Flag[k] = Flag[j] || State[k] <> Want) }
{
  Timer := Timer + 1.0;
  Flag[i] := True;
  State[k] := case
              | k = i : Want
              | State[k] = Crit && k < i : Idle
              | _ : State[k];
}
```

Dans cet exemple, la transition `t` est déclenchable s'il existe deux processus *distincts* d'identificateurs `i` et `j` tels que `i` est inférieur à `j`. Les tableaux `State` et `Flag` doivent contenir respectivement la valeur Idle et la valeur false à l'indice `i`. En plus de cette garde *locale* (aux processus `i` et `j`), on peut mentionner l'état des autres processus du système. La partie *globale* de la garde est précédée du mot clef `forall_other`. Ici, on dit que tous les autres processus (sous-entendu différents de `i` et `j`) doivent soit avoir la même valeur de `Flag` que `j`, soit contenir une valeur différente de Want dans le tableau `State`.

*Remarque.* Dans Cubicle, les processus sont différenciés par leur identificateurs. Ici, les paramètres `i` et `j` de la transition sont implicitement quantifiés existentiellement et doivent

être des identificateurs de processus deux à deux distincts. L'ensemble des identificateurs de processus est seulement muni d'un ordre total. La présence de cet ordre induit une topologie linéaire sur les processus. La comparaison entre identificateurs est donc autorisée et on peut par exemple écrire `i < j` dans une garde.

La garde de la transition est donnée par le mot clef `requires`. Si elle est satisfaite, les actions de la transition sont exécutées. Chaque action est une mise à jour d'une variable globale ou d'un tableau. La sémantique des transitions veut que l'ensemble des mises à jour soit réalisé de manière atomique et chaque variable qui apparaît à droite d'un signe `:=` dénote la valeur de cette variable avant la transition. L'ordre des affectations n'a donc pas d'importance. La première action `Timer := Timer + 1.0` incrémente la variable globale de 1 lorsque la transition est prise. Les mises à jour de tableaux peuvent être codées comme de simples affectations si une seule case est modifiée. L'action `Flag[i] := True` modifie le tableau `Flag` à l'indice `i` en y mettant la valeur true. Le reste du tableau n'est pas modifié. Si la transition modifie plusieurs cases d'un même tableau, on utilise une construction `case` qui précise les nouvelles valeurs contenues dans le tableau à l'aide d'un filtrage. `State[k] := case ...` mentionne ici les valeurs du tableau `State` pour tous les indices `k` (`k` doit être une variable fraîche) selon les conditions suivantes. Pour chaque indice `k`, le premier cas possible du filtrage est exécuté. Le cas

```
| k = i : Want
```

nous demande de vérifier tout d'abord si `k` vaut `i`, c'est-à-dire de mettre de la valeur Want à l'indice `i` du tableau `State`. Le deuxième cas

```
| State[k] = Crit && k < i : Idle
```

est plus compliqué : si le premier cas n'est pas vérifié (*i.e.* `k` est différent de `i`), que `State` contenait Crit à l'indice `k`, et que `k` est inférieur à `i` alors la nouvelle valeur de `State[k]` est Idle. Plus simplement, cette ligne change les valeurs `Crit` du tableau `State` se trouvant à gauche de `i` (`k < i`) en Idle. Enfin tous les filtrages doivent se terminer par un cas par défaut noté `_`. Dans notre exemple, le cas par défaut du filtrage

```
| _ : State[k]
```

dit que toutes les autres valeurs du tableau restent inchangées (on remet l'ancienne valeur de `State[k]`).

La relation de transition, décrite par l'ensemble des transitions, définit l'exécution du système comme une boucle infinie qui à chaque itération :

1. choisit de manière non déterministe une instance de transition dont la garde est vraie dans l'état courant du système ;
2. met à jour les variables et tableaux d'état conformément aux actions de la transition choisie.

Les propriétés de sûreté à vérifier sont exprimées sous forme négative, c'est-à-dire qu'on caractérise les états *dangereux* du système. On les exprime dans Cubicle à l'aide du mot clef `unsafe`, éventuellement suivi d'un ensemble de variables de processus distinctes. La formule *dangereuse* suivante exprime que les mauvais états du système sont ceux où il existe au moins deux processus distincts `x` et `y` tels que le tableau `State` contienne la valeur `Crit` à ces deux indices.

```
unsafe (x y) { State[x] = Crit && State[y] = Crit }
```

On dira qu'un système est *sûr* si aucun des états dangereux ne peut être atteint à partir d'un des états initiaux.

*Remarque.* Bien que Cubicle soit conçu pour vérifier des systèmes paramétrés, il est tout de même possible de l'utiliser pour vérifier des systèmes dont le nombre de processus est fixé à l'avance. Pour cela, il suffit d'inclure la ligne "`number_procs` *n*" dans le fichier, où *n* est le nombre de processus. Dans ce cas, on pourra mentionner explicitement les processus 1 à *n* en utilisant les constantes `#1` à `#`*n*.

*Remarque.* Le langage d'entrée de Cubicle est une version moins riche mais paramétrée du langage de Mur$\varphi$ [57] et similaire à Uclid [28]. Bien que limité pour l'instant, il est assez expressif pour permettre de décrire aisément des systèmes paramétrés conséquents (75 transitions, 40 variables et tableaux pour le protocole FLASH [106] par exemple).

## 2.2 Exemples

Dans cette section, on montre comment utiliser Cubicle et son expressivité pour modéliser différents algorithmes et protocoles de la littérature. Ces exemples sont donnés à titre didactique et sont choisis de manière à illustrer les caractéristiques fondamentales du langage. Le but de cette section est de donner au lecteur une bonne intuition des possibilités offertes par l'outil de manière à pouvoir modéliser et expérimenter le model checker sur ses propres exemples.

### 2.2.1 Algorithme d'exclusion mutuelle

L'exclusion mutuelle est un problème récurrent de la programmation concurrente. L'algorithme décrit ci-dessous résout ce problème, *i.e.* il permet à plusieurs processus de partager une ressource commune à accès unique sans conflit, en communiquant seulement au travers de variables partagées. C'est une version simplifiée de l'algorithme de Dekker (présenté en 2.2.2) qui fonctionne pour un nombre arbitraire de processus identiques.

Sur la figure 2.1, on matérialise *n* processus concurrents qui exécutent tous le même protocole. Chaque processus est représenté par le graphe de transition entre ses états : Idle,

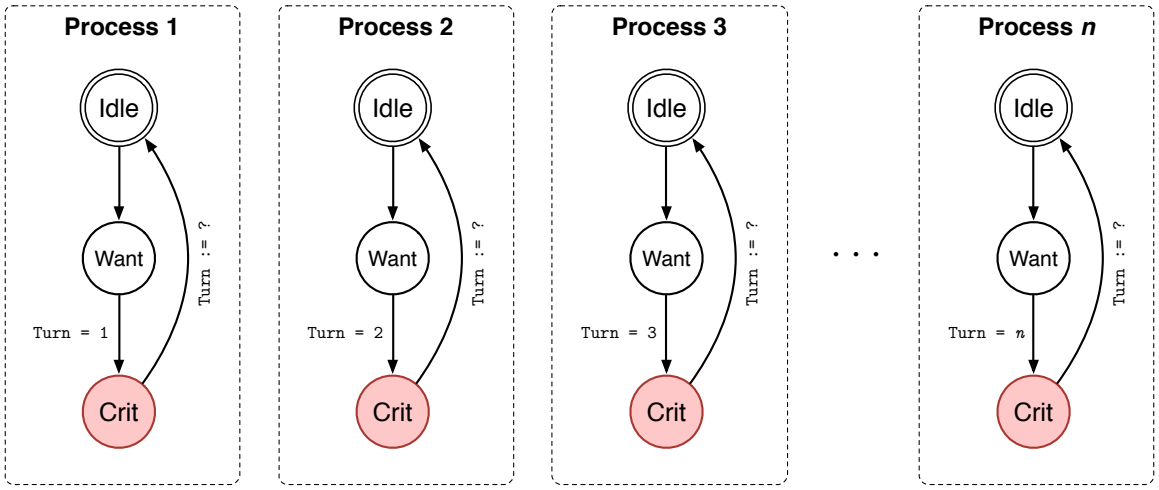



FIGURE 2.1 – Algorithme d'exclusion mutuelle

Want, et Crit, l'état initial étant Idle. Un processus peut demander l'accès à la section critique à tout moment en passant dans l'état Want. La synchronisation est effectuée au travers de la seule variable partagée `Turn`. La priorité est donnée au processus dont l'identifiant $i$ est contenu dans la variable `Turn`, qui peut dans ce cas passer en section critique. Un processus en section critique peut en sortir sans contrainte si ce n'est celle de « donner la main » à un de ses voisins en changeant la valeur de `Turn`. Cette dernière est modifiée de façon *non-déterministe* (`Turn := ?` dans le schéma), donc rien n'interdit à un processus de se redonner la main lui-même. La propriété qui nous intéresse ici est de vérifier la sûreté du système, c'est-à-dire qu'à tout moment, *au plus un* processus est en section critique.

Le code Cubicle correspondant à ce problème est donné ci-dessous en figure 2.2. Pour modéliser l'algorithme on a choisi de représenter l'état des processus par un tableau `State` contenant les valeur `Idle`, `Want`, ou `Crit`. On peut aussi voir `State[`$i$`]` comme la valeur d'une variable locale au processus d'identificateur $i$. La variable partagée `Turn` est simplement une variable globale au système. On définit les états initiaux comme ceux où tous les processus sont dans l'état `Idle`, quelque soit la valeur initiale de la variable `Turn`.

Prenons l'exemple de la transition `req`. Elle correspond au moment où un processus demande l'accès à la section critique, passant de l'état `Want` à `Idle`. Elle se lit : la transition `req` est possible s'il existe un processus d'identifiant `i` tel que `State[i]` a pour valeur `Idle`. Dans ce cas, la case `State[i]` prend pour valeur `Want`.

La formule `unsafe` décrit les mauvais états du système comme ceux dans lesquels il existe (au moins) deux processus distincts en section critique (*i.e.* pour lesquels la valeur de `State` est `Crit`). Autrement dit, on veut s'assurer que la formule suivante est un invariant du système :

$$\forall i, j.\ (i \neq j \wedge \texttt{State}[i] = \texttt{Crit}) \implies \texttt{State}[j] \neq \texttt{Crit}$$

*mutex.cub*

```
type state = Idle | Want | Crit

array State[proc] : state
var Turn : proc

init (z) { State[z] = Idle }

unsafe (z1 z2) { State[z1] = Crit &&
                 State[z2] = Crit }

transition req (i)
requires { State[i] = Idle }
{ State[i] := Want }

transition enter (i)
requires { State[i] = Want &&
           Turn = i }
{ State[i] := Crit; }

transition exit (i)
requires { State[i] = Crit }
{ Turn := ? ;
  State[i] := Idle; }
```

FIGURE 2.2 – Code Cubicle du mutex

Pour vérifier que le système décrit dans le fichier `mutex.cub` précédent est sûr, la manière la plus simple d'invoquer Cubicle est de lui passer seulement le nom de fichier en argument :

```
cubicle mutex.cub
```

On peut voir ci-après la trace émise par le model checker sur la sortie standard. Lorsque ce dernier affiche sur la dernière ligne "`The system is SAFE`", cela signifie qu'il a été en mesure de vérifier que l'état `unsafe` n'est jamais atteignable dans le système.

```
node 1: unsafe[1]                                    5 (2+3) remaining
node 2: enter(#2) -> unsafe[1]                       7 (2+5) remaining
node 3: req(#2) -> enter(#2) -> unsafe[1]            8 (1+7) remaining

The system is SAFE
```

### 2.2.2 Généralisation de l'algorithme de Dekker

L'algorithme de Dekker est en réalité la première solution au problème d'exclusion mutuelle. Cette solution est attribuée au mathématicien Th. J. Dekker par Edsger W. Dijkstra qui en donnera une version fonctionnant pour un nombre arbitraire de processus [54]. En 1985, Alain J. Martin présente une version simple de l'algorithme généralisé à $n$ processus [115]. C'est cette version qui est donnée ici sous forme d'algorithme (Algorithme 1) et sous forme de graphe de flot de contrôle (Figure 2.3).

La variable booléenne $x(i)$ est utilisée par un processus pour signaler son intérêt à entrer en section critique. La principale différence avec l'algorithme original est l'utilisation d'une valeur spéciale pour réinitialiser $t$. Dans Cubicle, les constantes de processus ne sont pas explicites donc on ne peut pas matérialiser l'identifiant spécial 0 dans le type `proc`. Pour

**Algorithme 1 :** Code de Dekker pour le processus $i$ [115]

**Variables** :
$x(i)$ : variable booléenne, initialisée à false
$t$ : variable partagée, initialisée à 0

$p(i)$ :
NCS **while** *true* **do**
  $x(i)$ := true;
WANT **while** $\exists j \neq i.\ x(j)$ **do**
AWAIT  $x(i)$ := false;
  **await** [ $t = 0 \vee t = i$ ];
TURN  $t$ := $i$;
  $x(i)$ := true;
CS // **Section critique**;
  $x(i)$ := false;
  $t$ := 0;

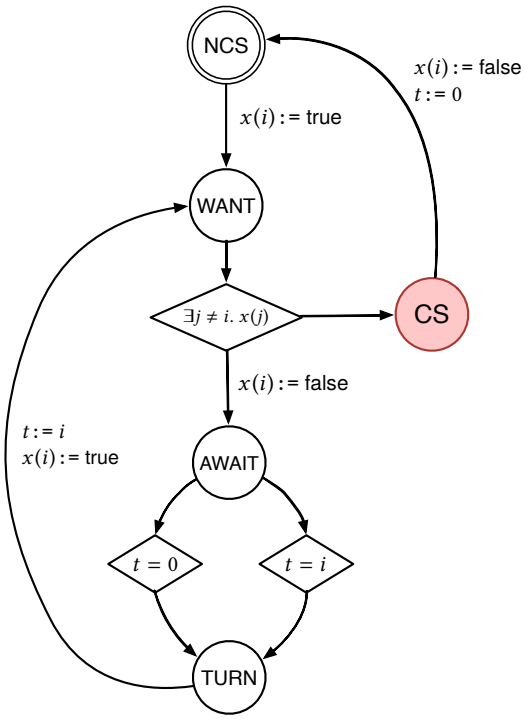



FIGURE 2.3 – Graphe de Dekker pour le processus $i$

modéliser $t$, on a choisit d'utiliser deux variables globales : `T` représente $t$ lorsque celle-ci est non nulle et la variable booléenne `T_set` vaut `False` lorsque $t$ a été réinitialisée (à 0). Le tableau `P` est utilisé pour représenter le compteur de programme et peut prendre les valeurs des étiquettes de l'algorithme 1. On peut remarquer que dans notre modélisation, la condition de la boucle while est évaluée de manière atomique. Les transitions `wait` et `enter` testent en une seule étape l'existence (ou non) d'un processus $j$ dont la variable $x(j)$ est vraie. De la même manière que précédemment, on veut s'assurer qu'il y ait au plus un processus au point de programme correspondant à l'étiquette CS, *i.e.* en section critique.

Cubicle est capable de prouver la sûreté de ce système. À vue d'œil, la correction de l'algorithme n'est pas triviale lorsqu'un nombre arbitraire de processus s'exécutent de manière concurrente. Pour illustrer ce propos, admettons que le concepteur ait fait une erreur et qu'un processus $i$ puisse omettre de passer sa variable $x(i)$ à true lorsqu'il arrive à l'étiquette TURN. Il suffit alors de rajouter la transition suivante au système pour modéliser les nouveaux comportements introduits :

*dekker_n.cub*

```
type location =
       NCS | WANT | AWAIT | TURN | CS

array P[proc] : location
array X[proc] : bool
var T : proc
var T_set : bool

init (i) {
    T_set = False &&
    X[i] = False && P[i] = NCS
}

unsafe (i j) { P[i] = CS && P[j] = CS }

transition start (i)
requires { P[i] = NCS }
{ P[i] := WANT;
  X[i] := True; }

transition wait (i j)
requires { P[i] = WANT && X[j] = True }
{ P[i] := AWAIT;
  X[i] := False; }

transition enter (i)
requires { P[i] = WANT &&
    forall_other j. X[j] = False }
{ P[i] := CS }

transition awaited_1 (i)
requires { P[i] = AWAIT &&
           T_set = False }
{ P[i] := TURN }

transition awaited_2 (i)
requires { P[i] = AWAIT &&
           T_set = True && T = i }
{ P[i] := TURN }

transition turn (i)
requires { P[i] = TURN }
{ P[i] := WANT;
  X[i] := True;
  T := i; T_set := True; }

transition loop (i)
requires { P[i] = CS }
{ P[i] := NCS;
  X[i] := False;
  T_set := False; }
```

FIGURE 2.4 – Code Cubicle de l'algorithme de Dekker

```
transition turn_buggy (i)
requires { P[i] = TURN }
{ P[i] := WANT;
  T := i; T_set := True; }
```

Si on exécute Cubicle sur le fichier maintenant obtenu, il nous fait savoir que la propriété n'est plus vérifiée en exposant une *trace d'erreur*. On peut voir sur la trace suivante qu'un mauvais état est atteignable en *dix* étapes avec deux processus.

```
Error trace: start(#2) -> enter(#2) -> start(#1) -> wait(#1, #2) ->
             loop(#2) -> awaited_1(#1) -> turn_buggy(#1) -> enter(#1) ->
             start(#2) -> enter(#2) -> unsafe[1]
```

```
UNSAFE !
```

Des constantes sont introduites pour chaque processus entrant en jeu dans la trace et sont notées #1, #2, #3, .... La notation "... `wait(#1, #2) ->` ..." signifie que pour reproduire la trace, il faut prendre la transition `wait` instanciée avec les processus #1 et #2.

*Remarque.* Un état dangereux de ce système est en réalité atteignable en seulement huit étapes mais il faut pour cela qu'au moins trois processus soient impliqués. On peut s'en rendre compte en forçant Cubicle à effectuer une exploration purement en largeur grâce à l'option `-postpone 0`. De cette façon, on est assuré d'obtenir la trace d'erreur la plus courte possible :

```
Error trace: start(#1) -> start(#3) -> wait(#1, #3) -> awaited_1(#1) ->
             turn_buggy(#1) -> enter(#1) -> start(#2) -> enter(#2) ->
             unsafe[1]
```

### 2.2.3 Boulangerie

L'algorithme de la boulangerie a été proposé par Leslie Lamport en réponse au problème d'exclusion mutuelle [110]. Contrairement aux approches précédentes, la particularité de cet algorithme est qu'il est *résistant aux pannes* et qu'il fonctionne même lorsque les opérations de lecture et d'écriture ne sont pas atomiques. Son pseudo-code est donné dans l'algorithme 2 (voir page suivante).

On peut faire le parallèle entre le principe de base de l'algorithme et celui d'une boulangerie aux heures de pointe, d'où son nom. Dans cette boulangerie, chaque client (matérialisant un processus) choisi un numéro en entrant dans la boutique. Le client qui souhaite acheter du pain ayant le numéro le plus faible s'avance au comptoir pour être servi. La propriété d'exclusion mutuelle de cette boulangerie se manifeste par le fait que son fonctionnement garantit qu'un seul client est servi à la fois. Il est possible que deux clients choisissent le même numéro, celui dont le nom (unique) est avant l'autre a alors la priorité.

La modélisation faite dans Cubicle est reprise de la version modélisée dans PFS par Abdulla et *al.* [3] et est donnée ci-dessous. C'est une version *simplifiée* de l'algorithme original de Lamport dans laquelle le calcul du maximum à l'étiquette Choose et l'ensemble des tests de la boucle for à l'étiquette Wait sont *atomiques*. Les lectures et écritures sont aussi considérées comme étant instantanées.

Cet algorithme est *tolérant aux pannes*, c'est à dire qu'il continue de fonctionner même si un processus s'arrête. Un processus $i$ a le droit de tomber en panne et de redémarrer en section non critique tout en réinitialisant ses variables locales. Ce comportement peut être modélisé dans Cubicle en rajoutant un point de programme spécial Crash représentant le fait qu'un processus est en panne. On ajoute une transition sans garde qui dit qu'un processus peut tomber en panne à tout moment, ainsi qu'une transition lui permettant de redémarrer (voir page 30).

**Algorithme 2 :** Pseudo-code de la boulangerie de Lamport pour le processus $i$ [110]

**Variables** :
`choosing`$[i]$ : variable booléenne, initialisée à false
`number`$[i]$ : variable entière initialisée à 0

$p(i)$ :
NCS **begin**
  `choosing`$[i]$ := true;
Choose `number`$[i]$ := 1 + max(`number`$[1], \ldots,$ `number`$[N]$);
  `choosing`$[i]$ := false;
  **for** $j = 1$ **to** $N$ **do**
Wait  **await** [ ¬ `choosing`$[j]$ ];
    **await** [ `number`$[j] = 0 \vee$ (`number`$[i], i$) < (`number`$[j], j$) ];
CS  // **Section critique**;
  `number`$[i]$ := 0;
  **goto** NCS;

*bakery_lamport.cub*

```
type location = NCS | Choose | Wait | CS

array PC[proc] : location
array Ticket[proc] : int
array Num[proc] : int
var Max : int

init (x) { PC[x] = NCS && Num[x] = 0 &&
           Max = 1 && Ticket[x] = 0 }

invariant () { Max < 0 }

unsafe (x y) { PC[x] = CS && PC[y] = CS }

transition next_ticket ()
{
  Ticket[j] := case | _ : Max;
  Max := Max + 1;
}

transition take_ticket (x)
requires { PC[x] = NCS &&
    forall_other j. Num[j] < Max }
{
  PC[x] := Choose;
  Ticket[x] := Max;
}

transition wait (x)
requires { PC[x] = Choose }
{
  PC[x] := Wait;
  Num[x] := Ticket[x];
}

transition turn (x)
requires { PC[x] = Wait &&
  forall_other j.
  (PC[j] <> Choose && Num[j] = 0 ||
   PC[j] <> Choose && Num[x] < Num[j] ||
   PC[j] <> Choose &&
   Num[x] = Num[j] && x < j) }
{
  PC[x] := CS;
}

transition exit (x)
requires { PC[x] = CS }
{
  PC[x] := NCS;
  Num[x] := 0;
}
```

FIGURE 2.5 – Code Cubicle de l'algorithme de la boulangerie de Lamport

```
type location =
     NCS | Choose | Wait | CS | Crash
...

transition fail (x)
{ PC[x] := Crash }
```

```
transition recover (x)
requires { PC[x] = Crash }
{
  PC[x] := NCS;
  Num[x] := 0;
}
```

### 2.2.4 Cohérence de Cache

Dans une architecture multiprocesseurs à mémoire partagée, l'utilisation de *caches* est requise pour réduire les effets de la latence des accès mémoire et pour permettre la coopération. Les architectures modernes possèdent de nombreux caches ayant des fonctions particulières, mais aussi plusieurs niveaux de cache. Les caches qui sont les plus près des unités de calcul des processeurs offrent les meilleures performances. Par exemple un accès en lecture au cache (*succès de cache* ou *cache hit*) L1 consomme seulement 4 cycles du processeur sur une architecture Intel Core i7, alors qu'un accès mémoire (*défaut de cache* ou *cache miss*) consomme en moyenne 120 cycles [113]. Cependant l'utilisation de caches introduit le problème de *cohérence de cache* : toutes les copies d'un même emplacement mémoire doivent être dans des états compatibles. Un *protocole de cohérence de cache* fait en sorte, entre autres, que les écritures effectuées à un emplacement mémoire partagé soient visibles de tous les autres processeurs tout en garantissant une absence de conflits lors des opérations.

Il existe plusieurs types de protocoles de cohérence de cache :

— cohérence par *espionnage*[3] : la communication se fait au travers d'un bus central sur lequel les différentes transactions sont visibles par tous les processeurs.

— cohérence par *répertoire*[4] : chaque processeur est responsable d'une partie de la mémoire et garde trace des processeurs ayant une copie locale dans leur cache. Ces protocoles fonctionnent par envoi de messages sur un réseau de communication.

Bien que plus difficile à mettre en œuvre, cette dernière catégorie de protocoles est aujourd'hui la plus employée, pour des raisons de performance et de passage à l'échelle. Les architectures réelles sont souvent très complexes, elles implémentent plusieurs protocoles de cohérence de manière hiérarchique, et utilisent plusieurs dizaines voire centaines de variables. Néanmoins leur fonctionnement repose sur le même principe fondamental. Un protocole simple mais représentatif de cette famille à été donné par Steven German à

3. Ces protocoles sont aussi parfois qualifiés de *snoopy*.
4. On qualifie parfois ces protocoles par le terme anglais *directory based*.

la communauté académique [138]. L'exemple suivant dénommé German-*esque* est une version simplifiée de ce protocole.

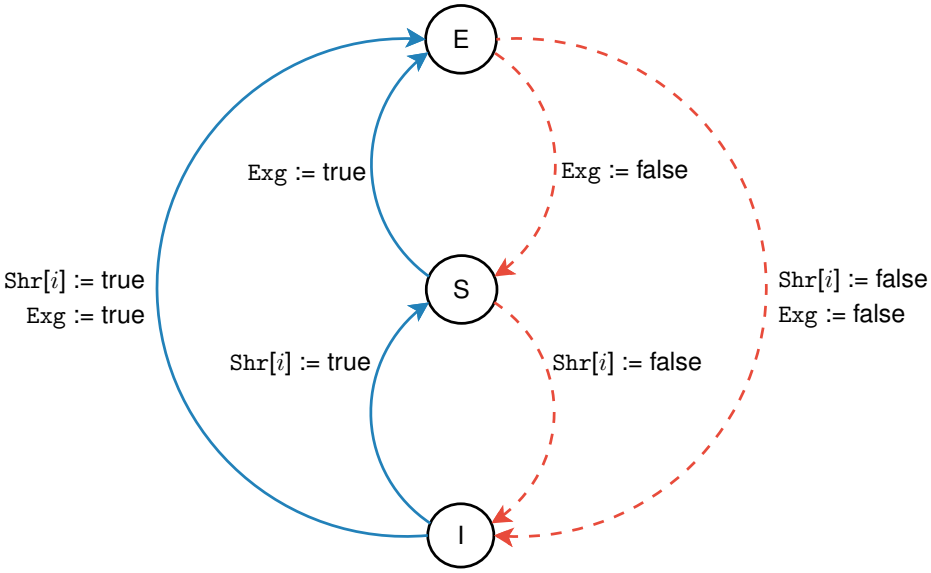



FIGURE 2.6 – Diagramme d'état du protocole German-*esque*

L'état d'un processeur $i$ est donné par la variable $\texttt{Cache}[i]$ qui peut prendre trois valeurs : (E)xclusive (accès en lecture et écriture), (S)hared (accès en lecture seulement) ou (I)nvalid (pas d'accès à la mémoire). Les clients envoient des requêtes au responsable lorsqu'un défaut de cache survient : RS pour un accès partagé (défaut de lecture), RE pour un accès exclusif (défaut d'écriture). Le répertoire contient quatre informations : une variable booléene Exg signale par la valeur true qu'un des clients possède un accès exclusif à la mémoire principale, un tableau de booléens Shr, tel que $\texttt{Shr}[i]$ est vrai si le client $i$ possède une copie (avec un accès en lecture ou écriture) de la mémoire dans son cache, Cmd contient la requête courante ($\epsilon$ marque l'absence de requête) dont l'émetteur est enregistré dans Ptr.

Les états initiaux du système sont représentés par la formule logique suivante :

$$\forall i.\ \texttt{Cache}[i] = \mathsf{I} \ \wedge\ \neg\texttt{Shr}[i] \ \wedge\ \neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \epsilon$$

signifiant que les caches de tous les processeurs sont invalides, aucun accès n'a encore été donné et il n'y a pas de requête à traiter.

La Figure 2.6 donne une vue assez haut niveau de l'évolution d'un seul processeur. Les flèches pleines montrent l'évolution du cache du processeur selon ses propres requêtes, alors que les flèches pointillées représentent les transitions résultant d'une requête d'un autre client. Par exemple, un cache va de l'état I à S lors d'un défaut de lecture : le répertoire lui accorde un accès partagé tout en enregistrant cette action dans le tableau $\texttt{Shr}[i] := \mathsf{true}$. De

***germanesque.cub***

```
type msg = Epsilon | RS | RE
type state = I | S | E

(* Client *)
array Cache[proc] : state

(* Directory *)
var Exg : bool
var Cmd : msg
var Ptr : proc
array Shr[proc] : bool

init (z) {
    Cache[z] = I && Shr[z] = False &&
    Exg = False && Cmd = Epsilon
}

unsafe (z1 z2) {
    Cache[z1] = E && Cache[z2] <> I
}

transition request_shared (n)
requires { Cmd = Epsilon &&
           Cache[n] = I }
{
  Cmd := RS;
  Ptr := n ;
}

transition request_exclusive (n)
requires { Cmd = Epsilon &&
           Cache[n] <> E }
{
  Cmd := RE;
  Ptr := n;
}

transition invalidate_1 (n)
requires { Shr[n]=True && Cmd = RE }
{
  Exg := False;
  Cache[n] := I;
  Shr[n] := False;
}

transition invalidate_2 (n)
requires { Shr[n]=True &&
           Cmd = RS && Exg=True }
{
  Exg := False;
  Cache[n] := S;
  Shr[n] := True;
}

transition grant_shared (n)
requires { Ptr = n &&
           Cmd = RS && Exg = False }
{
  Cmd := Epsilon;
  Shr[n] := True;
  Cache[n] := S;
}

transition grant_exclusive (n)
requires {
    Cmd = RE && Exg = False &&
    Ptr = n && Shr[n] = False &&
    forall_other l. Shr[l] = False }
{
  Cmd := Epsilon;
  Exg := True;
  Shr[n] := True;
  Cache[n] := E;
}
```

FIGURE 2.7 – Code Cubicle du protocole de cohérence de cache German-*esque*

façon similaire, si un défaut en écriture survient dans un autre cache, le répertoire *invalide* tous les clients enregistrés dans `Shr` avant de donner l'accès exclusif. Cette invalidation générale a pour effet de passer les caches dans les états E et S à l'état I.

La modélisation qui est faite de ce protocole est assez immédiate et est donnée ci-dessous. On peut toutefois remarquer qu'on s'intéresse ici seulement à la partie *contrôle* du protocole et on oublie les actions d'écriture et lecture réelles de la mémoire. La seule propriété qu'on souhaite garantir dans ce cas est que si un processeur a son cache avec accès exclusif alors tous les autres sont invalides :

$$\forall i, j.\ (i \neq j \wedge \texttt{Cache}[i] = \mathsf{E}) \implies \texttt{Cache}[j] = \mathsf{I}$$

Le responsable du répertoire est abstrait et on ne s'intéresse qu'à une seule ligne de mémoire donc l'état du répertoire est représenté avec des variables globales.

## 2.3 Non-atomicité

Dans la plupart des travaux existants sur les systèmes paramétrés, on fait la supposition que l'évaluation des conditions globales est *atomique*. La totalité de la garde est évaluée en une seule étape. Cette hypothèse est raisonnable lorsque les conditions globales masquent des détails d'implémentation qui permettent de faire cette évaluation de manière atomique. En revanche, beaucoup d'implémentations réelles d'algorithmes concurrents et de protocoles n'évaluent pas ces conditions instantanément mais plutôt par une itération sur une structure de donnée (tableau, liste chaînée, *etc.*). Par exemple, l'algorithme de Dekker (voir Section 2.2.2) implémente le test $\exists j \neq i.x(j)$ de l'étiquette WANT par une boucle qui recherche un $j$ tel que $x(j)$ soit vrai. De même, on a modélisé la boucle d'attente de l'algorithme de la boulangerie (Section 2.2.3, algorithme 2, étiquette Choose) par une garde universelle dans la transition `turn`.

En supposant que des conditions sont atomiques, on simplifie le problème mais cette approximation n'est pas conservatrice. En effet, l'évaluation d'une condition peut être entrelacée avec les actions des autres processus. Il est possible par exemple, qu'un processus change la valeur de sa variable locale alors que celle-ci à déjà été comptabilisée pour une condition globale avec son ancienne valeur. Il peut dans ce cas exister dans l'implémentation des configurations qui ne sont représentées par aucun état du modèle atomique. Si on veut prendre en compte ces éventualités dans notre modélisation, on se doit de refléter tous les comportements possibles dans le système de transition.

La vérification de propriétés d'exclusion mutuelle dans des protocoles paramétrés comme l'algorithme de la boulangerie à déjà été traitée par le passé [32, 114]. Ces preuves ne sont souvent que partiellement automatisées et font usage d'abstractions *ad-hoc*. La première approche permettant une vérification automatique de protocoles avec gardes non atomiques a été développée en 2008 [5]. Les auteurs utilisent ici un protocole annexe de *raffinement*

pour modéliser l'évaluation non atomique des conditions globales. L'approche qu'on présente dans la suite de cette section est identique en substance à celle de [5]. On montre cependant qu'on peut rester dans le cadre défini par Cubicle pour modéliser ces conditions non atomiques.

Comme les tests non atomiques sont implémentés par des boucles, on choisit de modéliser les implémentations de la figure 2.8. La valeur $N$ représente ici le paramètre du système,

| $\forall j \neq i.\ c(j)$ | $\exists j \neq i.\ c(j)$ |
|---|---|
| $j := 1$ ;<br>**while** $(j \leq N)$ **do**<br>  **if** $j \neq i \wedge \neg c(j)$ **then return** *false* ;<br>  $j := j + 1$<br>**end** ;<br>**return** *true* | $j := 1$ ;<br>**while** $(j \leq N)$ **do**<br>  **if** $j \neq i \wedge c(j)$ **then return** *true* ;<br>  $j := j + 1$<br>**end** ;<br>**return** *false* |

FIGURE 2.8 – Évaluation non atomique des conditions globales par un processus $i$

c'est-à-dire la cardinalité du type `proc`. Pour une condition universelle, on parcourt tous les éléments (de 1 à $N$) et on s'assurent qu'ils vérifient la condition $c$. Si on trouve un processus qui ne respecte pas cette condition on sort de la boucle et on renvoie *false*. Malheureusement dans Cubicle, on ne peut pas mentionner ce paramètre $N$ explicitement (car le solveur SMT ne peut pas raisonner sur la valeur de la cardinalité des modèles). On ne peut dès lors pas utiliser de compteur entier. Pour s'en sortir, on construit une boucle avec un compteur abstrait pour lequel on peut seulement tester si sa valeur est 0 ou $N$ [5]. On peut également incrémenter, décrémenter, et affecter ce compteur à ces mêmes valeurs.

Dans Cubicle on modélise ce compteur par un tableau de booléens représentant un encodage unaire de sa valeur entière. Le nombre de cellules contenant la valeur true correspond à la valeur du compteur. Incrémenter ce compteur revient donc à passer une case de la valeur false à la valeur true. Pour modéliser l'évaluation non atomique de la condition $\forall j \neq i.\ c(j)$ [6], on utilise un compteur `Cpt`. Comme on veut qu'il soit local à un processus, on a besoin d'un tableau bi-dimensionnel. Le tableau en `Cpt`$[i]$ matérialise le compteur local au processus $i$. Le résultat de l'évaluation de cette condition sera stocké dans la variable `Res`$[i]$ à valeurs dans le type énuméré $\{\mathsf{E}, \mathsf{T}, \mathsf{F}\}$. `Res`$[i]$ contient E initialement ce qui signifie que la condition n'est pas finie d'être évaluée. Lorsqu'elle contient T alors la condition globale a été évalué à vrai, et si elle contient F la condition globale est fausse. Les transitions correspondantes sont données figure 2.9.

Avec cette modélisation, on spécifie juste que la condition locale $c(j)$ doit être vérifiée

5. On peut généraliser à tester si sa valeur est $k$ ou $N - k$ où k est une *constante* positive entière.
6. L'encodage de l'évaluation de la condition duale $\exists j \neq i.\ c(j)$ est symétrique.

| Transition | Commentaire |
|---|---|
| `type result = E \| T \| F`<br>`array Cpt[proc,proc] : bool`<br>`array Res[proc] : result` | |
| `transition start (i)`<br>`requires { ... }`<br>`{ Res[i] := E`<br>`  Cpt[x,y] := case`<br>`  \| x=i : False`<br>`  \| _ : Cpt[x,y] }` | On initialise le compteur à 0 et on signale que la condition est en cours d'évaluation avec la variable locale `Res`. |
| `transition iter (i j)`<br>`requires { Cpt[i,j] = False && c(j) }`<br>`{ Cpt[i,j] := True }` | La condition $c(j)$ n'a pas encore été vérifiée. Comme elle est vraie on incrémente le compteur. |
| `transition abort (i j)`<br>`requires { Cpt[i,j] = False && ¬c(j) }`<br>`{ Res[i] := F }` | La condition $c(j)$ n'a pas encore été vérifiée mais elle est fausse. On sort de la boucle, la condition globale est fausse. |
| `transition exit (i)`<br>`requires { forall_other j.`<br>`  Cpt[i,j] = True }`<br>`{ Res[i] := T }` | Toutes les conditions $c(j)$ ont étés vérifiées. On sort de la boucle, la condition globale est vraie. |

FIGURE 2.9 – Encodage de l'évaluation non atomique de la condition globale $\forall j \neq i.\ c(j)$

pour tous les processus, indépendamment de l'ordre. Cette sous-spécification reste conservatrice. Il est toutefois possible d'ajouter des contraintes d'ordre sur la variable $j$ si c'est important pour la propriété de sûreté.

On peut remarquer qu'on se sert d'une garde universelle dans la transition `exit`. Ceci n'influe en rien la « non-atomicité » de l'évaluation de la condition car le quantificateur est seulement présent pour encoder le test $\texttt{Cpt}[i] = N$.

*Remarque.* Avec cette garde universelle, on risque de tomber dans le cas d'une fausse alarme à cause du traitement détaillé en section 3.3 du chapitre suivant. L'intuition apportée par le modèle de panne franche, nous permet d'identifier ces cas problématiques. Si la sûreté dépend du fait qu'un processus puisse disparaître du système pendant l'évaluation d'une condition globale, alors on risque de tomber dans ce cas défavorable.

Les versions non atomiques des algorithmes sont bien plus compliquées que leurs homologues atomiques. Les raisonnements à mettre en œuvre pour dérouler la relation de transition (aussi bien en avant qu'en arrière) sont plus longs. C'est ce qui en fait des candidats particulièrement coriaces pour les procédures de model checking.

## 2.4 Logique multi-sortée et systèmes à tableaux

On a montré dans la section précédente que le langage de Cubicle permet de représenter des algorithmes d'exclusion mutuelle ou des protocoles de cohérence de cache à l'aide de systèmes de transitions paramétrés. Le lecteur intéressé par la définition précise de la syntaxe concrète et des règles de typage de ce langage trouvera leur description en annexe A.

Dans cette section, on définit formellement la sémantique des programmes Cubicle. Pour ce faire, on utilise le formalisme des systèmes à tableaux conçu par Ghilardi et Ranise [77]. Ainsi, on montre dans un premier temps que chaque transition d'un système paramétré peut simplement être interprétée comme une formule dans un fragment de la logique du premier ordre multi-sortée. Ce fragment est défini par l'ensemble des types et variables déclarés au début du programme (ainsi que ceux prédéfinis dans Cubicle). Dans un deuxième temps, on donne une sémantique opérationnelle à ces formules à l'aide d'une relation de transition entre les modèles logiques de ces formules.

On rappelle ici brièvement les notions usuelles de la logique du premier ordre multi-sortée et de la théorie des modèles qui sont utilisées pour définir les systèmes à tableaux de Cubicle. Des explications plus amples peuvent être trouvées dans des manuels traitant du sujet [92, 152]. Le lecteur familier avec ces concepts peut ne lire que la section 2.4.3 qui présente plus en détail la construction des ces systèmes.

### 2.4.1 Syntaxe des formules logiques

La logique du premier ordre multi-sortée est une extension classique qui possède essentiellement les mêmes propriétés que la logique du premier ordre sans sortes [149]. Son intérêt est qu'elle permet de partitionner les éléments qu'on manipule selon leur type.

**Définition 2.4.1.** Une *signature* $\Sigma$ est un tuple $(S, F, R)$ où :

- $S$ est un ensemble non vide de symboles de sorte
- $F$ est un ensemble de symboles de fonction, chacun étant associé à un type de la forme
  - $s$ pour les symboles d'arité 0 (zéro) avec $s \in S$. On appelle ces symboles des *constantes*.
  - $s_1 \times \ldots \times s_n \to s$ pour les symboles d'arité $n$, avec $s_1, \ldots, s_n, s \in S$
- $R$ est un ensemble de symboles de relation (aussi appelé *prédicats*). On associe à chaque prédicat d'arité $n$ un type de la forme $s_1 \times \ldots \times s_n$ avec $s_1, \ldots, s_n \in S$.

Dans la suite on supposera que le symbole d'égalité = est inclus dans toutes les signatures qui seront considérées et qu'il a les types $s \times s$ quelque soit la sorte $s \in S$[7]. On notera par exemple par $f : s_1 \times \ldots \times s_n \to s$ un symbole de fonction $f$ dans $F$ dont le type est $s_1 \times \ldots \times s_n \to s$.

L'en-tête du fichier Cubicle est en fait une façon de donner cette signature. Par exemple l'en-tête du code de l'exemple du mutex de la section 2.2.1 correspond à la définition de la signature $\Sigma$ suivante :

| **en-tête Cubicle** | **signature** $\Sigma = (S, F, R)$ |
|---|---|
| `type state = Idle \| Want \| Crit`<br><br>`array State[proc] : state`<br>`var Turn : proc` | $S = \{\texttt{state}, \texttt{proc}\}$<br>$F = \{\texttt{Idle} : \texttt{state}, \texttt{Want} : \texttt{state},$<br>$\texttt{Crit} : \texttt{state},$<br>$\texttt{State} : \texttt{proc} \to \texttt{state},$<br>$\texttt{Turn} : \texttt{proc}\}$<br>$R = \{= : \texttt{state} \times \texttt{state}$<br>$: \texttt{proc} \times \texttt{proc}\}$ |

*Remarque.* Le mot-clef `var` ne permet pas d'introduire une variable dans la logique mais permet de déclarer une constante logique avec son type. Le choix des mots-clefs `var` et `array` reflète une vision plus intuitive des systèmes Cubicle comme des programmes.

7. On pourrait éviter cette formulation avec un symbole d'égalité polymorphe mais cela demande d'introduire des variables de type.

*Remarque.* Cubicle connaît en interne les symboles de sorte `proc`, `bool`, `int` et `real` ainsi que les symboles de fonction $+$, $-$, les constantes $0, 1, 2, \ldots, 0.0, 1.0, \ldots$ et les relations $<$ et $\leq$. Par convention, ils apparaîtront dans la signature $\Sigma$ seulement s'ils sont utilisés dans le système.

On distinguera parmi ces signatures celles qui ne permettent pas d'introduire de nouveaux termes car il est important pour Cubicle de maîtriser la création des processus.

**Définition 2.4.2.** Une signature est dite *relationnelle* si elle ne contient pas de symboles de fonction.

**Définition 2.4.3.** Une signature est dite *quasi-relationnelle* si les symboles de fonction qu'elle contient sont tous des constantes.

On définit les $\Sigma$-*termes*, $\Sigma$-*atomes* (ou $\Sigma$-*formules atomiques*), $\Sigma$-*littéraux* et $\Sigma$-*formules* comme les expressions du langage défini par la grammaire décrite en figure 2.10. Les types des variables quantifiées ne sont pas indiqués par commodité.

| | | | | |
|---|---|---|---|---|
| $\Sigma$-terme : | $\langle t \rangle$ | ::= | $c$ | où $c$ est une constante de $\Sigma$ |
| | | \| | $f(\langle t \rangle, \ldots, \langle t \rangle)$ | où $f$ est un symbole de fonction de $\Sigma$ d'arité $> 0$ |
| | | \| | $ite(\langle \varphi \rangle, \langle t \rangle, \langle t \rangle)$ | |
| $\Sigma$-atome : | $\langle a \rangle$ | ::= | false \| true | |
| | | \| | $p(\langle t \rangle, \ldots, \langle t \rangle)$ | où $p$ est un symbole de relation de $\Sigma$ |
| $\Sigma$-littéral : | $\langle l \rangle$ | ::= | $\langle a \rangle \mid \neg\langle a \rangle$ | |
| $\Sigma$-formule : | $\langle \varphi \rangle$ | ::= | $\langle l \rangle \mid \langle \varphi \rangle \wedge \langle \varphi \rangle \mid \langle \varphi \rangle \vee \langle \varphi \rangle \mid \forall i.\, \langle \varphi \rangle \mid \exists i.\, \langle \varphi \rangle$ | |

FIGURE 2.10 – Grammaire de la logique

En plus de cette restriction syntaxique, on impose que les $\Sigma$-termes et $\Sigma$-atomes soient *bien typés* en associant des sortes aux termes :

1. Chaque constante $c$ de sorte $s \in S$ est un terme de sorte $s$.
2. Soient $f$ un symbole de fonction de type $s_1 \times \ldots \times s_n \to s$, $t_1$ un terme de sorte $s_1$, …, et $t_n$ un terme de sorte $s_n$ alors $f(t_1, \ldots\ t_n)$ est un terme de sorte $s$.
3. Soit $p$ un symbole de relation de type $s_1 \times \ldots \times s_n$. L'atome $p(t_1, \ldots, t_n)$ est bien typé ssi $t_1$ est un terme de sorte $s_1$, …, et $t_n$ est un terme de sorte $s_n$.

On appelle une Σ-*clause* une disjonction de Σ-littéraux. On dénote par, Σ-*CNF* (*resp.* Σ-*DNF*) une conjonction de disjonction de littéraux (*resp.* une disjonction de conjonction de littéraux).

#### Conventions et notations

Pour éviter la lourdeur de la terminologie on dénotera par *termes*, *atomes* (ou *formules atomiques*), *littéraux*, *formules*, *clauses*, *CNF* et *DNF* respectivement les Σ-*termes*, Σ-*atomes* (ou Σ-*formules atomiques*), Σ-*littéraux*, Σ-*formules*, Σ-*clauses*, Σ-*CNF* et Σ-*DNF* lorsque le contexte ne permet pas d'ambiguïté.

Les termes de la forme $ite(\varphi_{cond}, t_{then}, t_{else})$ correspondent à la construction conditionnelle classique `if` $\varphi_{cond}$ `then` $t_{then}$ `else` $t_{else}$. On suppose l'existence d'une fonction `elim_ite` qui prend en entrée une formule sans quantificateurs dont les termes peuvent contenir le symbole $ite$ et renvoie une formule *équivalente* sans $ite$. De plus on suppose que `elim_ite` renvoie une formule en forme normale disjonctive (Σ-DNF). Par exemple,

$$\texttt{elim_ite}(x = ite(\varphi, t_1, t_2)) \;=\; (\varphi \wedge x = t_1) \;\vee\; (\neg\varphi \wedge x = t_2)$$

On notera par $\bar{i}$ une séquence $i_1, i_2, \ldots, i_n$. Pour les formules quantifiées, on écrira $\exists x, y.\ \varphi$ (*resp.* $\forall x, y.\ \varphi$) pour $\exists x \exists y.\ \varphi$ (*resp.* $\forall x \forall y.\ \varphi$). En particulier, on écrira $\exists \bar{i}.\ \varphi$ pour $\exists i_1 \exists i_2 \ldots \exists i_n.\ \varphi$.

### 2.4.2 Sémantique de la logique

**Définition 2.4.4.** Une Σ-*structure* $A$ est une paire $(D, \mathcal{I})$ où $D$ est un ensemble appelé le *domaine* de $A$ (ou l'*univers* de $A$) et dénoté par $dom(A)$. Les éléments de $D$ sont appelés les *éléments* de la structure $A$. On notera $|dom(A)|$ le cardinal du domaine de $A$ et on dira qu'une structure $A$ est finie si $|dom(A)|$ est fini. $\mathcal{I}$ est l'interprétation qui :

1. associe à chaque sorte $s \in S$ de Σ un sous ensemble non vide de $D$.
2. associe à chaque constante de Σ de type $s$ un élément du sous-domaine $\mathcal{I}(s)$.
3. associe à chaque symbole de fonction $f \in F$ d'arité $n > 0$ et de type $s_1 \times \ldots \times s_n \to s$ une fonction totale $\mathcal{I}(f) : \mathcal{I}(s_1) \times \ldots \times \mathcal{I}(s_n) \to \mathcal{I}(s)$.
4. associe à chaque symbole de relation $p \in R$ d'arité $n > 0$ et de type $s_1 \times \ldots \times s_n$ une fonction totale $\mathcal{I}(p) : \mathcal{I}(s_1) \times \ldots \times \mathcal{I}(s_n) \to \{\mathsf{true}, \mathsf{false}\}$.

Cette interprétation peut être étendue de manière homomorphique aux Σ-termes et Σ-formules – elle associe à chaque terme $t$ de sorte $s$ un élément $\mathcal{I}(t) \in \mathcal{I}(s)$ et à chaque formule $\varphi$ une valeur $\mathcal{I}(\varphi) \in \{\mathsf{true}, \mathsf{false}\}$.

**Définition 2.4.5.** On appelle une Σ-*théorie* $\mathcal{T}$ un ensemble (potentiellement infini) de Σ-structures. Ces structures sont aussi appelées les *modèles* de $\mathcal{T}$.

**Définition 2.4.6.** On dit qu'un Σ-modèle $\mathcal{M} = (A, \mathcal{I})$ *satisfait* une formule $\varphi$ ssi $\mathcal{I}(\varphi) =$ true, dénoté par $\mathcal{M} \models \varphi$.

**Définition 2.4.7.** Une formule $\varphi$ est dite *satisfiable* dans une théorie $\mathcal{T}$ (ou $\mathcal{T}$*-satisfiable*) ssi il existe un modèle $\mathcal{M} \in \mathcal{T}$ qui satisfait $\varphi$.

Une formule $\varphi$ est dite *conséquence logique* d'un ensemble $\Gamma$ de formules dans une théorie $\mathcal{T}$ (et noté par $\Gamma \models_{\mathcal{T}} \varphi$) ssi tous les modèles de $\mathcal{T}$ qui satisfont $\Gamma$ satisfont aussi $\varphi$.

**Définition 2.4.8.** Une formule $\varphi$ est dite *valide* dans une théorie $\mathcal{T}$ (ou $\mathcal{T}$*-valide*) ssi sa négation est $\mathcal{T}$*-insatisfiable*, dénoté par $\mathcal{T} \models \varphi$ ou $\emptyset \models_{\mathcal{T}} \varphi$.

**Définition 2.4.9.** Soient $A$ et $B$ deux Σ-structures. $A$ est une *sous-structure* de $B$, noté $A \subseteq B$ si $dom(A) \subseteq dom(B)$.

**Définition 2.4.10.** Soient $A$ une Σ-structure et $X \subseteq dom(A)$, alors il existe une unique plus petite sous-structure $B$ de $A$ tel que $dom(B) \subseteq X$. On dit que $B$ est la sous-structure de $A$ *générée* par $X$ et on note $B = \langle X \rangle_A$.

Les deux définitions suivantes seront importantes pour caractériser la théorie des processus supportée par Cubicle.

**Définition 2.4.11.** Une Σ-théorie $\mathcal{T}$ est *localement finie* ssi Σ est finie, chaque sous ensemble fini d'un modèle de $\mathcal{T}$ génère une sous-structure finie.

*Remarque.* Si Σ est relationnelle ou quasi-relationnelle alors toute Σ-théorie est localement finie.

**Définition 2.4.12.** Une Σ-théorie $\mathcal{T}$ est *close par sous-structure* ssi chaque sous-structure d'un modèle de $\mathcal{T}$ est aussi un modèle de $\mathcal{T}$.

**Exemple.** La théorie ayant pour modèle la structure de domaine $\mathbb{N}$ et signature ($\{$`int`$\}$, $\{0, 1\}$, $\{=, \leq\}$) où ces symboles sont interprétés de manière usuelle (comme dans la théorie de l'arithmétique de Presburger) est localement finie et close par sous-structure. Si on étend cette signature avec le symbole de fonction + alors elle n'est plus localement finie mais reste close par sous-structure. Une théorie ayant pour modèle une structure finie, avec une signature $(_, \emptyset, \{=, R\})$ où $R$ est interprétée comme une relation binaire qui caractériserait un anneau est localement finie mais n'est pas close par sous-structure.

Le problème de la *satisfiabilité modulo* une Σ-théorie $\mathcal{T}$ (SMT) consiste à établir la satisfiabilité de formules closes sur une extension arbitraire de Σ (avec des constantes). Une extension de ce problème, beaucoup plus utile en pratique, est d'établir la satisfiabilité *modulo la combinaison de deux (ou plus) théories.*

**Exemples de théories.** La théorie de l'égalité (aussi appelée théorie *vide*, ou *EUF*) est la théorie qui a comme modèles tous les modèles possibles pour une signature donnée. Elle n'impose aucune restriction sur l'interprétation faite de ses symboles (ses symboles sont dits non-interprétés). Les fonctions non-interprétées sont souvent utilisées comme technique d'abstraction pour s'affranchir d'une complexité ou de détails inutiles.

La théorie de l'arithmétique est une autre théorie omniprésente en pratique. Elle est utilisée pour modéliser l'arithmétique des programmes, la manipulation de pointeurs et de la mémoire, les contraintes de temps réels, les propriétés physiques de certains systèmes, etc. Sa signature est $\{0, 1, \ldots, +, -, *, /, \leq\}$ étendue à un nombre arbitraire de constantes, et ses symboles sont interprétés de manière usuelle sur les entiers et les réels.

Une théorie de types énumérés est une théorie ayant une signature $\Sigma$ quasi-relationnelle contenant un nombre fini de constantes (constructeurs). L'ensemble de ses modèles consiste en une unique $\Sigma$-structure dont chaque symbole est interprété comme un des constructeurs de $\Sigma$. Dans ce qui va suivre on verra que ces théories seront utiles pour modéliser les *points de programme* de processus et les messages échangés par ces processus dans les systèmes paramétrés.

### 2.4.3 Systèmes de transition à tableaux

Cette section introduit le formalisme des systèmes à tableaux conçu par Ghilardi et Ranise [77]. Il permet de représenter une classe de systèmes de transition paramétrés dans un fragment restreint de la logique du premier ordre multi-sortée. Pour ceci on aura besoin des théories suivantes :

- une théorie *des processus* $\mathcal{T}_P$ sur signature $\Sigma_P$ localement finie dont le seul symbole de type est `proc`, et telle que la $\mathcal{T}_P$-satisfiabilité est décidable sur le fragment sans quantificateurs
- une théorie *d'éléments* $\mathcal{T}_E$ sur signature $\Sigma_E$ localement finie dont le seul symbole de type est `elem`, et telle que la $\mathcal{T}_E$-satisfiabilité est décidable sur le fragment sans quantificateurs. $\mathcal{T}_E$ peut aussi être l'union de plusieurs théories $\mathcal{T}_E = \mathcal{T}_{E_1} \cup \ldots \cup \mathcal{T}_{E_k}$, dans ce cas $\mathcal{T}_E$ a plusieurs symboles de types $\texttt{elem}_1$, …, $\texttt{elem}_k$.
- la théorie *d'accès* $\mathcal{T}_A$ sur signature $\Sigma_A$, obtenue en combinant la théorie $\mathcal{T}_P$ et la théorie $\mathcal{T}_E$ de la manière suivante. $\Sigma_A = \Sigma_P \cup \Sigma_E \cup Q$ où $Q$ est un ensemble de symboles de fonction de type $\texttt{proc} \times \ldots \times \texttt{proc} \rightarrow \texttt{elem}_i$ (ou constantes de type $\texttt{elem}_i$). Étant donnée une structure $S$, on note par $S_{|\texttt{ty}}$ la structure $S$ dont le domaine est restreint aux éléments de type `ty`. Les modèles de $\mathcal{T}_A$ sont les structures $S$ où $S_{|\texttt{proc}}$ est un modèle de $\mathcal{T}_P$ et $S_{|\texttt{elem}}$ est un modèle de $\mathcal{T}_E$ et $S_{|\texttt{proc}\times\ldots\times\texttt{proc}\rightarrow\texttt{elem}}$ est l'ensemble des fonctions totales de $\texttt{proc} \times \ldots \times \texttt{proc} \rightarrow \texttt{elem}$.

On suppose dans la suite que les théories $\mathcal{T}_E$ et $\mathcal{T}_P$ ne partagent pas de symboles, *i.e.* $\Sigma_E \cap \Sigma_P = \{=\}$ (seule l'égalité apparaît dans toutes les signatures).

**Définition 2.4.13.** Un *système (de transition) à tableaux* est un triplet $\mathcal{S} = (Q, I, \tau)$ avec $Q$ partitionné en $Q_0, \ldots, Q_m$ où

- $Q_i$ est un ensemble de symboles de fonction d'arité $i$. Chaque $f \in Q_i$ a comme type $\underbrace{\texttt{proc} \times \ldots \times \texttt{proc}}_{i \text{ fois}} \to \texttt{elem}_f$. Les fonctions d'arité 0 représentent les variables globales du système. Les fonctions d'arité non nulle représentent quand à elles les tableaux (indicés par des processus) du système.
- $I$ est une formule qui caractérise les *états initiaux* du système (où les variables de $Q$ peuvent apparaître libres).
- $\tau$ est une relation de transition.

La relation $\tau$ peut être exprimée sous la forme d'une disjonction de formules quantifiées existentiellement par zéro, une, ou plusieurs variables de type `proc`. Chaque composante de cette disjonction est appelée une transition et est paramétrée par ses variables existentielles. Elle met en relation les variables globales et tableaux d'états avant et après exécution de la transition. Si $x \in Q$ est un tableau (ou une variable globale), on notera par $x'$ la valeur de $x$ après exécution de la transition et par $Q'$ l'ensemble des variables et tableaux après exécution de la transition. La forme générale des transitions qu'on considère est la suivante :

$$t(Q, Q') = \exists \bar{i}.\ \underbrace{\gamma(\bar{i}, Q)}_{\text{garde}} \wedge \underbrace{\bigwedge_{x \in Q} \forall \bar{j}. x'(\bar{j}) = \delta_x(\bar{i}, \bar{j}, Q)}_{\text{action}}$$

où

- $\gamma$ est une formule sans quantificateurs [8] appelée la garde de $t$
- $\delta_x$ est une formule sans quantificateurs appelée la mise à jour de $x$

Les variables de $Q$ peuvent apparaître libres dans $\gamma$ et les $\delta_x$. Cette formule est équivalente à la variante suivante où les fonctions $x'$ sont écrites sous forme fonctionnelle avec un lambda-terme :

$$t(Q, Q') = \exists \bar{i}.\ \gamma(\bar{i}, Q) \wedge \bigwedge_{x \in Q} x' = \lambda \bar{j}.\ \delta_x(\bar{i}, \bar{j}, Q)$$

8. On autorise une certaine forme de quantification universelle dans $\gamma$ en section 3.3.

Intuitivement, une transition $t$ met en jeu un ou plusieurs processus (les variables quantifiées existentiellement $\bar{i}$, ses paramètres) qui peuvent modifier l'état du système (*cf.* Section 2.5). Ici $\gamma$ représente la garde de la transition et les $\delta_x$ sont les mises à jour des variables et tableaux d'état. Un tel système $\mathcal{S} = (Q, I, \tau)$ est bien décrit par la syntaxe concrète de Cubicle et de manière analogue, sa sémantique est une boucle infinie qui, à chaque tour, exécute une transition choisie arbitrairement dont la garde $\gamma$ est vraie, et met à jour les valeurs des variables de $Q$ en conséquence.

Soit un système $\mathcal{S} = (Q, I, \tau)$ et une formule $\Theta$ (dans laquelle les variables de $Q$ peuvent apparaître libres), le problème de la *sûreté* (ou de l'*atteignabilité*) est de déterminer s'il existe une séquence de transitions $t_1, \ldots, t_n$ dans $\tau$ telle que

$$I(Q^0) \wedge t_1(Q^0, Q^1) \wedge \ldots \wedge t_n(Q^{n-1}, Q^n) \wedge \Theta(Q^n)$$

est satisfiable modulo les théories mises en jeu. S'il n'existe pas de telle séquence, alors $\mathcal{S}$ est dit *sûr* par rapport à $\Theta$. Autrement dit, $\neg\Theta$ est une propriété de sûreté ou un invariant du système.

**Exemple** (Mutex)**.** On prend ici l'exemple d'un mutex simple paramétré par son nombre de processus (dans type `proc`) dont un aperçu plus détaillé a été donnée en section 2.2.1.

Pour cet exemple, on prendra $\mathcal{T}_P$ la théorie de l'égalité de signature $\Sigma_P = (\{\texttt{proc}\}, \emptyset, \{=\})$ ayant pour symbole de type `proc`. On considère comme théorie des éléments, l'union d'une théorie des types énumérés $\mathcal{T}_E$ de signature $\Sigma_E = (\{\texttt{state}\}, \{\mathsf{Idle}, \mathsf{Want}, \mathsf{Crit}\}, \{=\})$ où Idle, Want, et Crit en sont les constructeurs de type `state` et de la théorie $\mathcal{T}_P$. La théorie d'accès $\mathcal{T}_A$ est définie comme la combinaison de $\mathcal{T}_P$ et $\mathcal{T}_E$. Elle a pour signature $\Sigma_A = \Sigma_P \cup \Sigma_E \cup (_, \{\texttt{State}, \texttt{Turn}\}, \emptyset)$ où `State` est un symbole de fonction de type `proc` $\rightarrow$ `state`, et `Turn` est une constante de type `proc`.

Avec le formalisme décrit précédemment, le système $\mathcal{S} = (Q, I, \tau)$ représentant le problème du mutex s'exprime de la façon suivante. L'ensemble $Q$ contient les symboles `State` et `Turn`. Les états initiaux du système sont décris par la formule

$$I \equiv \forall i.\ \texttt{State}(i) = \mathsf{Idle}$$

La relation de transition $\tau$ est la disjonction $t_{req} \vee t_{enter} \vee t_{exit}$ avec :

$$\begin{aligned} t_{req} \ \equiv\ & \exists i.\ \texttt{State}(i) = \mathsf{Idle} \wedge \\ & \forall j.\ \texttt{State}'(j) = ite(i = j,\ \mathsf{Want},\ \texttt{State}(j)) \ \wedge\ \texttt{Turn}' = \texttt{Turn} \end{aligned}$$

$$\begin{aligned} t_{enter} \ \equiv\ & \exists i.\ \texttt{State}(i) = \mathsf{Want} \ \wedge\ \texttt{Turn} = i \wedge \\ & \forall j.\ \texttt{State}'(j) = ite(i = j,\ \mathsf{Crit},\ \texttt{State}(j)) \ \wedge\ \texttt{Turn}' = \texttt{Turn} \end{aligned}$$

$$\begin{aligned} t_{exit} \ \equiv\ & \exists i.\ \texttt{State}(i) = \mathsf{Crit} \wedge \\ & \forall j.\ \texttt{State}'(j) = ite(i = j,\ \mathsf{Idle},\ \texttt{State}(j)) \end{aligned}$$

Enfin la formule caractérisant les mauvais états du système est :

$$\Theta \equiv \exists i, j.\ i \neq j \ \wedge\ \texttt{State}(i) = \mathsf{Crit} \ \wedge\ \texttt{State}(j) = \mathsf{Crit}$$

La description de ce système faite dans la syntaxe concrète de Cubicle donnée en section 2.2.1 est l'expression directe de cette représentation. Pour plus de simplicité, on notera les transitions avec le sucre syntaxique (de Cubicle)

$$\texttt{transition}\ t\ (\bar{i})\ \texttt{requires}\ \{\ g\ \}\ \{\ a\ \}$$

dans la section suivante. Par exemple, la première transition $t_{req}$ sera notée par

$$\begin{array}{l}\texttt{transition}\ t_{req}(i)\\ \texttt{requires}\ \{\texttt{State}[i] = \mathsf{Idle}\}\\ \{\ \texttt{State}[j] := \texttt{case}\ i = j : \mathsf{Want} \mid _ : \texttt{State}'[j]\ \}\ .\end{array}$$

## 2.5 Sémantique

La sémantique d'un programme décrit par un système de transition à tableaux dans Cubicle est définie pour un *nombre de processus donné*. On fixe $n$ la cardinalité du domaine `proc`. C'est-à-dire que tous les modèles qu'on considère dans cette section interprètent le type `proc` vers un ensemble à $n$ éléments.

*Remarque.* On peut aussi voir un $\Sigma$-modèle $\mathcal{M}$ comme un dictionnaire qui associe les éléments de $\Sigma$ aux éléments du domaine de $\mathcal{M}$.

Soit $\mathcal{S} = (Q, I, \tau)$ un système à tableaux, et $\mathcal{T}_A$ la combinaison des théories comme définie en section 2.4.3. Dans la suite de cette section, on verra les modèles de $\mathcal{T}_A$ comme des dictionnaires associant *tous* les symboles de $\Sigma_A$ à des éléments de leur domaine respectif.

La première partie de ce chapitre donne au lecteur tous les éléments nécessaires pour réaliser un interprète du langage d'entrée de Cubicle et s'assurer qu'il soit correct.

### 2.5.1 Sémantique opérationnelle

On définit la sémantique opérationnelle du système à tableaux $\mathcal{S}$ pour $n$ processus sous forme d'un triplet $K^n = (S_n, I_n, \longrightarrow)$ où :

- $S_n$ est l'ensemble potentiellement infini des modèles de $\mathcal{T}_A$ où la cardinalité de l'ensemble des éléments de `proc` est $n$.
- $I_n = \{\mathcal{M} \in S_n \mid \mathcal{M} \models I\}$ est le sous ensemble de $S_n$ dont les modèles satisfont la formule initiale $I$ du système à tableaux $\mathcal{S}$.

— $\longrightarrow \subseteq S_n \times S_n$ est la relation de transition d'états définie par la règle suivante :

$$\frac{\begin{array}{c}\texttt{transition}\ t\ (\bar{i})\ \texttt{requires}\ \{\ g\ \}\ \{\ a\ \} \\ \sigma : \bar{i} \mapsto \texttt{proc} \qquad \mathcal{M} \models g\sigma \qquad A = \texttt{all}(a, \bar{i})\end{array}}{\mathcal{M} \longrightarrow \texttt{update}(A\sigma, \mathcal{M})}$$

où $A = \texttt{all}(a, \texttt{proc})$ est l'ensemble des actions de $a$ instanciées avec tous les éléments de `proc` et $\sigma$ est une substitution des paramètres $\bar{i}$ de la transition vers éléments de `proc`. L'application de substitution de variables sur les actions s'effectue sur les termes de droite. On note $A\sigma$ l'ensemble des actions de $A$ auxquelles on a appliqué la substitution $\sigma$ :

$$\texttt{x} := t \in A \iff x := t\sigma \in A\sigma$$

$$\texttt{A}[\bar{i}] := \begin{array}{ll}\texttt{case} & c_1 : t_1 \\ \mid & \ldots \\ \mid & c_m : t_m \\ \mid & _ : t_{m+1}\end{array} \in A \iff \texttt{A}[\bar{i}] := \begin{array}{ll}\texttt{case} & c_1\sigma : t_1\sigma \\ \mid & \ldots \\ \mid & c_m\sigma : t_m\sigma \\ \mid & _ : t_{m+1}\sigma\end{array} \in A\sigma$$

$\texttt{update}(A\sigma, \mathcal{M})$ est le modèle obtenu après application des actions de $A\sigma$. On définit une fonction d'évaluation $\vdash_{\mathcal{I}}$ telle que :

$$\mathcal{M} \vdash_{\mathcal{I}} e : v \text{ ssi l'interprétation de } e \text{ dans } \mathcal{M} \text{ est } v$$

L'union des dictionnaires $\mathcal{M}_1 \cup \mathcal{M}_2$ est définie comme l'union des ensembles des liaisons lorsque ceux-ci sont disjoints. Lorsque certaines liaisons apparaissent à la fois dans $\mathcal{M}_1$ et $\mathcal{M}_2$, on garde seulement celles de $\mathcal{M}_2$. On note la fonction `update` par le symbole $\vdash_{\texttt{up}}$ dans les règles qui suivent.

$$\frac{\mathcal{M} \vdash_{\texttt{up}} a_1 : \mathcal{M}_1 \qquad \mathcal{M} \vdash_{\texttt{up}} a_2 : \mathcal{M}_2}{\mathcal{M} \vdash_{\texttt{up}} a_1; a_2 : \mathcal{M}_1 \cup \mathcal{M}_2} \qquad \frac{\mathcal{M} \vdash_{\mathcal{I}} t : v}{\mathcal{M} \vdash_{\texttt{up}} \texttt{x} := t : \mathcal{M} \cup \{\texttt{x} \mapsto v\}}$$

$$\frac{\mathcal{M} \not\models c_1 \qquad \mathcal{M} \vdash_{\texttt{up}} \texttt{A}[\bar{i}] := \texttt{case}\ c_2 : t_2 \mid \ldots \mid _ : t_{n+1} : \mathcal{M}'}{\mathcal{M} \vdash_{\texttt{up}} \texttt{A}[\bar{i}] := \texttt{case}\ c_1 : t_1 \mid c_2 : t_2 \mid \ldots \mid _ : t_{n+1} : \mathcal{M}'}$$

$$\frac{\mathcal{M} \models c_1 \qquad \mathcal{M} \vdash_{\mathcal{I}} t_1 : v_1 \qquad \mathcal{M} \vdash_{\mathcal{I}} \texttt{A} : f_{\texttt{A}} \qquad \mathcal{M} \vdash_{\mathcal{I}} \bar{i} : \overline{v_i}}{\mathcal{M} \vdash_{\texttt{up}} \texttt{A}[\bar{i}] := \texttt{case}\ c_1 : t_1 \mid c_2 : t_2 \mid \ldots \mid _ : t_{n+1} : \mathcal{M} \cup \{\texttt{A} \mapsto f_{\texttt{A}} \cup \{\overline{v_i} \mapsto v_1\}\}}$$

$$\frac{\mathcal{M} \vdash_{\texttt{up}} \mathcal{M} \vdash_{\mathcal{I}} t : v \qquad \mathcal{M} \vdash_{\mathcal{I}} \texttt{A} : f_{\texttt{A}} \qquad \mathcal{M} \vdash_{\mathcal{I}} \bar{i} : \overline{v_i}}{\mathcal{M} \vdash_{\texttt{up}} \texttt{A}[\bar{i}] := \texttt{case}\ _ : t : \mathcal{M} \cup \{\texttt{A} \mapsto f_{\texttt{A}} \cup \{\overline{v_i} \mapsto v\}\}}$$

Exécuter l'action `x := t` revient à changer la liaison de `x` dans le modèle $\mathcal{M}$ par la valeur de $t$, comme montré par la deuxième règle. L'avant dernière règle définit quant à elle l'exécution d'une mise à jour du tableau `A` en position $\bar{i}$. Si la première condition $c_1$ de la construction `case` est satisfiable dans le modèle courant, l'interprétation de la fonction correspondant à `A` est changée pour qu'elle associe maintenant la valeur de $t$ aux valeurs de $\bar{i}$.

*Remarque.* La construction `case` du langage de Cubicle est similaire aux constructions traditionnelles de certains langages de programmation comme le branchement conditionnel `switch ... case ... break` de C, ou `match` de ML. En effet les instructions d'un cas sont exécutées seulement si toutes les conditions précédentes sont fausses. Le dernier cas `_` permet de s'assurer qu'au moins une instruction est exécutée. En utilisant la forme logique de la section 2.4.3, une mise à jour avec `case` s'exprime sous la forme de termes $ite$ (`if-then-else`) imbriqués. Une fois les $ite$ éliminés, l'ensemble des conditions obtenues forme une partition (*i.e.* elles sont mutuellement insatisfiables et leur disjonction est valide).

Ces règles montrent notamment qu'on évalue les termes des affectations dans le modèle précédent et non celui qu'on est en train de construire. Le caractère ; dans les actions ne représente pas une séquence car les actions d'une transition sont exécutées simultanément et de manière atomique.

Un système paramétré a alors toutes les sémantiques $\{K^n \mid n \in \mathbb{N}\}$. On peut aussi voir la sémantique d'un système à tableaux paramétré comme une fonction qui à chaque $n$ associe $K^n$ [9].

### 2.5.2 Atteignabilité

Ici on ne s'intéresse qu'aux propriétés de sûreté des systèmes à tableaux, ce qui revient au problème de l'atteignabilité d'un ensemble d'états.

Soit $K^n = (S_n, I_n, \longrightarrow)$ un triplet exprimant la sémantique d'un système à tableaux $\mathcal{S} = (Q, I, \tau)$ comme défini précédemment. On note par $\longrightarrow^*$ la clôture transitive de la relation $\longrightarrow$ sur $S_n$. On dit qu'un état $s \in S_n$ est atteignable dans $K^n$ et on note $Reachable(s, K^n)$ ssi il existe un état $s_0 \in I_n$ tel que

$$s_0 \longrightarrow^* s$$

Soit une propriété *dangereuse* caractérisée par la formule $\Theta$. On dira qu'un système à tableaux $\mathcal{S}$ est non sûr par rapport à $\Theta$ ssi

$$\exists n \in \mathbb{N}.\ \exists s \in S_n.\ s \models \Theta \ \wedge\ Reachable(s, K^n)$$

9. On peut remarquer que $K^n$ est en réalité une structure de *Kripke infinie* pour laquelle on omet la fonction de labellisation $L$ des états de la structure de Kripke car elle correspond exactement aux modèles, *i.e.* $L(\mathcal{M}) = \mathcal{M}$.

Au contraire, on dira que $\mathcal{S}$ est sûr par rapport à $\Theta$ ssi

$$\forall n \in \mathbb{N}.\ \forall s \in S_n.\ s \models \Theta \implies \neg Reachable(s, K^n)$$

### 2.5.3 Un interpréteur de systèmes à tableaux

À l'aide de la sémantique précédente, on peut définir différentes analyses sur un système à tableaux. Par exemple, un interpréteur de systèmes à tableaux est un programme dont les états sont des états de $K^n = (S_n, I_n, \longrightarrow)$, qui démarre en un état initial de $I_n$ et qui « suit » une séquence de transitions de $\longrightarrow$.

Comme dans la section précédente, l'interpréteur pour un système à tableaux $\mathcal{S} = (Q, I, \tau)$ représente l'état du programme par un dictionnaire sur $Q$ (*i.e.* un modèle de $\mathtt{T}_A$). On suppose ici que $I_n$ est non vide, c'est-à-dire qu'il existe au moins un modèle de $I$ où `proc` est de cardinalité $n$. Dans ce cas, on note *Init* la fonction qui prend $n$ en argument et qui retourne un état de $I_n$ au hasard.

L'interpréteur consiste en une procédure `run` (Algorithme 3) prenant en argument le paramètre $n$ et la relation de transition $\tau$. Elle maintient l'état du système dans le dictionnaire $\mathcal{M}$ et exécute une boucle infinie qui modifie $\mathcal{M}$ en fonction des transitions choisies.

**Algorithme 3 :** Interpréteur d'un système à tableaux

**Variables** :
  $\mathcal{M}$ : l'état courant du programme (un modèle de $\mathcal{T}_A$)

**procedure** `run`$(n, \tau)$ : **begin**
  $\mathcal{M}$ := *Init*$(n)$;
  **while** *true* **do**
    **let** $L$ = `enabled`$(n, \tau, \mathcal{M})$ **in**
    **if** $L = \emptyset$ **then** `deadlock ()`;
    **let** $a, \sigma$ = `select`$(L)$ **in**
    $\mathcal{M}$ := `update`(`all`$(a,$ `proc`$)\sigma, \mathcal{M})$;

La fonction `enabled`$(n, \tau, \mathcal{M})$ renvoie l'ensemble des transitions de $\tau$ dont la garde est vraie pour $\mathcal{M}$. Autrement dit,

$$(\texttt{transition}\ t\ (\bar{i})\ \texttt{requires}\ \{\ g\ \}\ \{\ a\ \})\ \in\ \texttt{enabled}(n, \tau, \mathcal{M}) \implies \mathcal{M} \models \exists \bar{i}.\ g(\bar{i})$$

où $g$ est la garde de la transition $t$ et $\bar{i}$ en sont les paramètres. Si aucune des transitions n'est possible alors on sort de la boucle d'exécution avec la fonction `deadlock`. Si $\mathcal{M}$

correspond à un état final du système, le programme s'arrête. Sinon on est dans une situation d'*interblocage* (ou *deadlock*).

La fonction `select` choisit de manière aléatoire une des transitions de $L$ et retourne les actions $a$ correspondantes ainsi qu'une substitution $\sigma$ des arguments vers les éléments du domaine de `proc` telle que $\mathcal{M} \models g(\bar{i})\sigma$.

Ensuite la fonction $\texttt{update}(\texttt{all}(a, \texttt{proc})\sigma, \mathcal{M})$ retourne un nouveau dictionnaire où les valeurs des variables et tableaux sont mises à jour en fonction des actions de l'instance de la transition choisie, comme défini précédemment.

*Remarque.* On peut aussi ajouter une instruction $\texttt{assert}(\neg\mathcal{M} \models \Theta)$ à l'algorithme 3 qui lève une erreur dès qu'on passe par un état mauvais, *i.e.* lorsque $\mathcal{M} \models \Theta$.

**Exemple** (Mutex)**.** Prenons l'exemple du mutex de la section précédente avec deux processus $\#_1$ et $\#_2$. Une configuration de départ satisfaisant la formule initiale $I$ possible est le modèle $\mathcal{M}$ ci-dessous. En exécutant la transition $t_{req}$ pour le processus $\#_1$, on change la valeur de $\texttt{State}(\#_1)$ et obtient le nouveau modèle $\mathcal{M}'$.

$$\mathcal{M} = \begin{array}{|c|}\hline \boldsymbol{F} \quad \boldsymbol{dom}(\mathcal{M}) \\ \hline \texttt{Turn} \mapsto \#_2 \\ \texttt{State} \mapsto \begin{cases} \#_1 \mapsto \mathsf{Idle} \\ \#_2 \mapsto \mathsf{Idle} \end{cases} \\ \vdots \\ \hline \end{array} \xrightarrow{t_{req}[i\backslash\#_1]} \mathcal{M}' = \begin{array}{|c|}\hline \boldsymbol{F} \quad \boldsymbol{dom}(\mathcal{M}') \\ \hline \texttt{Turn} \mapsto \#_2 \\ \texttt{State} \mapsto \begin{cases} \#_1 \mapsto \mathsf{Want} \\ \#_2 \mapsto \mathsf{Idle} \end{cases} \\ \vdots \\ \hline \end{array}$$

Maintenant qu'il est aisé de se faire une idée du langage de description et de sa sémantique qu'on utilise pour les systèmes de transition paramétrés, on montre dans la suite de ce document comment construire un model checker capable de prouver la sûreté des exemples de la section 2.2. Les algorithmes présentés dans ce chapitre sont des systèmes jouets, dans le sens où ils représentent des problèmes de vérification formelle de taille assez réduite. Cependant les preuves manuelles de ses systèmes paramétrés ne sont pas pour autant triviales et les outils automatiques sont souvent mis au point sur ce genre d'exemples. On s'efforcera bien sûr d'expliquer comment développer des algorithmes qui passent à l'échelle sur des problèmes plus conséquents.

# 3

# Cadre théorique : model checking modulo théories

## Sommaire


**3.1 Analyse de sûreté des systèmes à tableaux . . . . . 50**
3.1.1 Atteignabilité par chaînage arrière . . . . . 50
3.1.2 Correction . . . . . 53
3.1.3 Effectivité . . . . . 54
**3.2 Terminaison . . . . . 57**
3.2.1 Indécidabilité de l'atteignabilité . . . . . 57
3.2.2 Conditions pour la terminaison . . . . . 59
3.2.3 Exemples . . . . . 64
**3.3 Gardes universelles . . . . . 68**
3.3.1 Travaux connexes . . . . . 68
3.3.2 Calcul de pré-image approximé . . . . . 69
3.3.3 Exemples . . . . . 70
3.3.4 Relation avec le modèle de panne franche et la relativisation des quantificateurs . . . . . 71
**3.4 Conclusion . . . . . 74**
3.4.1 Exemples sans existence d'un bel ordre . . . . . 74
3.4.2 Résumé . . . . . 76
3.4.3 Discussion . . . . . 77


Le cadre théorique sur lequel repose Cubicle est celui du *model checking modulo théories* proposé par Ghilardi et Ranise [77]. C'est un cadre déclaratif dans lequel les systèmes manipulent un ensemble de tableaux infinis d'où le nom de *systèmes à tableaux*. Un système est décrit par des formules logiques du premier ordre et des transitions gardées (voir

Section 2.4.3). Une des forces de cette approche réside dans l'utilisation des capacités de raisonnement des solveurs SMT et de leur support interne pour de nombreuses théories. On rappelle dans ce chapitre les fondements de cette théorie et certains résultats et théorèmes associés.

Dans la suite de ce chapitre on se donne un système à tableaux $\mathcal{S} = (Q, I, \tau)$ et une formule $\Theta$ représentant des états dangereux. Rappelons que la relation $\tau$ peut s'exprimer sous la forme d'une disjonction de transitions paramétrées de la forme :

$$t(Q,Q') = \exists \bar{i}.\ \underbrace{\gamma(\bar{i},Q)}_{\text{garde}} \wedge \underbrace{\bigwedge_{x \in Q} \forall \bar{j}.x'(\bar{j}) = \delta_x(\bar{i},\bar{j},Q)}_{\text{action}}$$

avec $\gamma$ et $\delta_x$ des formules *sans quantificateurs*. Ces restrictions sont très importantes car elles permettent à la théorie exposée dans ce chapitre de caractériser des conditions suffisantes pour lesquelles l'atteignabilité (ou la sûreté) est *décidable*.

Cubicle implémente le cadre du model checking modulo théories mais va au-delà en relâchant certaines contraintes, au prix de la complétude et de la terminaison, mais reste correct.

## 3.1 Analyse de sûreté des systèmes à tableaux

Cette section présente les conditions sous lesquelles l'analyse de sûreté d'un système à tableau peut être mise en œuvre de façon *effective*. En particulier on caractérise un fragment de la logique qui est clos par les opérations de l'algorithme d'atteignabilité arrière et on montre que cette approche est correcte.

### 3.1.1 Atteignabilité par chaînage arrière

Plusieurs approches sont possibles pour résoudre les instances du problème de sûreté (ou d'atteignabilité). Une première approche consiste à construire l'ensemble des états atteignables (par chaînage avant) à partir des états initiaux, une autre approche, celle qui sera adoptée dans la suite, consiste à construire l'ensemble des états qui peuvent atteindre les mauvais états du système (atteignabilité par chaînage arrière).

**Définition 3.1.1.** La *post-image* d'une formule $\varphi(X)$ par la relation de transition $\tau$ est définie par

$$\textsc{Post}_\tau(\varphi)(X') = \exists X.\ \varphi(X) \wedge \tau(X,X')$$

Elle représente les états atteignables à partir de $\varphi$ en une étape de $\tau$.

**Définition 3.1.2.** La *pré-image* d'une formule $\varphi(X')$ par la relation de transition $\tau$ est définie par

$$\textsc{Pre}_\tau(\varphi)(X) = \exists X'.\ \tau(X,X') \wedge \varphi(X')$$

De manière analogue la *pré-image* d'une formule $\varphi(X')$ par une transition $t$ est définie par

$$\textsc{Pre}_t(\varphi)(X) = \exists X'.\ t(X,X') \wedge \varphi(X')$$

et donc $\textsc{Pre}_\tau(\varphi)(X) = \bigvee_{t\in\tau} \textsc{Pre}_t(\varphi)(X)$.

La pré-image $\textsc{Pre}_\tau(\varphi)$ est donc une formule qui représente les états qui peuvent atteindre $\varphi$ en une étape de transition de $\tau$.

La clôture de la pré-image $\textsc{Pre}^*_\tau(\varphi)$ est définie par :

$$\begin{cases} \textsc{Pre}^0_\tau(\varphi) &= \varphi \\ \textsc{Pre}^n_\tau(\varphi) &= \textsc{Pre}_\tau(\textsc{Pre}^{n-1}_\tau(\varphi)) \\ \textsc{Pre}^*_\tau(\varphi) &= \bigvee_{k\in\mathbb{N}} \textsc{Pre}^k_\tau(\varphi) \end{cases}$$

La clôture de $\Theta$ par $\textsc{Pre}_\tau$ caractérise alors l'ensemble des états qui peuvent atteindre $\Theta$. Une approche générale pour résoudre le problème d'atteignabilité consiste alors à calculer cette clôture et vérifier si elle contient un état de la formule initiale. L'algorithme d'atteignabilité par chainage arrière présenté ci-après, fait exactement ceci de manière incrémentale. Si, pendant sa construction de $\textsc{Pre}^*_\tau(\Theta)$, on découvre qu'un état initital (un modèle de $I$) est aussi un modèle d'un des $\textsc{Pre}^n_\tau(\Theta)$, alors le système n'est *pas sûr* vis-à-vis de $\Theta$. Le système est *sûr* si, *a contrario*, aucune des pré-images ne s'intersecte avec $I$. L'algorithme peut décider une telle propriété seulement si cette clôture est aussi un point-fixe.

Plusieurs briques de base sont nécessaires pour construire un tel algorithme. La première est d'être capable de calculer les pré-images successives. Pour une recherche en avant, il faut pouvoir calculer $\textsc{Post}^n(I)$ ce qui est souvent impossible pour les systèmes paramétrés à cause de la présence de quantificateurs universels dans la formule initiale $I$. En revanche, le calcul du $\textsc{Pre}$ est effectif dans les systèmes à tableaux si on restreint la forme des formules exprimant les états dangereux.

**Définition 3.1.3.** On appelle *cube* une formule de la forme $\exists \bar{i}.(\Delta \wedge F)$, où les $\bar{i}$ sont des variables de la théorie $\mathcal{T}_P$, $\Delta$ est une conjonction de diséquations entre les variables de $\bar{i}$, et $F$ est une conjonction de littéraux.

**Proposition 3.1.1.** *Si $\varphi$ est un cube, la formule $\textsc{Pre}_\tau(\varphi)$ est équivalente à une disjonction de cubes.*

*Démonstration.* Soit $\varphi = \exists \overline{p}.(\Delta \wedge F)$ un cube, $\text{PRE}_\tau(\varphi)$ est la disjonction des $\text{PRE}_t(\varphi)$ pour chaque transition de $t$ de $\tau$.

Prenons une transition $t$ de $\tau$, $\text{PRE}_t(\varphi)(Q) = \exists Q'.\ t(Q,Q') \wedge \exists \overline{p}.(\Delta \wedge F(Q'))$ se réécrit en

$$\exists Q'.\ \exists \bar{i}.\ \gamma(\bar{i},Q) \wedge \bigwedge_{x \in Q} x' = \lambda \bar{j}.\ \delta_x(\bar{i},\bar{j},Q) \wedge \exists \overline{p}.(\Delta \wedge F(Q')) \tag{3.1}$$

Par la suite, on utilisera la notation traditionnellement employée en combinatoire $\binom{A}{k}$ pour l'ensemble des combinaisons de $A$ de taille $k$.

Soit $\bar{j} = j_1, \ldots\ j_k$. Étant donné $x \in Q$ et $x' = \lambda \bar{j}.\ \delta_x(\bar{i},\bar{j},Q)$ une des mises à jour de la transition $t$, on notera $\sigma_{x'}$ la substitution $x'(\overline{p_\sigma}) \backslash \delta_x(\bar{i},\overline{p_\sigma},Q)$ pour tout $\overline{p_\sigma} \in \binom{\bar{p}}{k}$. Maintenant, la formule $\exists \overline{p}.(\Delta \wedge F(Q'))[\sigma_{x'}]$ correspond en essence à la réduction de l'application d'une lambda expression apparaissant dans l'équation (3.1). La réduction de toutes les lambda expressions conduit à la formule suivante :

$$\exists Q'.\ \exists \bar{i}.\ \gamma(\bar{i},Q) \wedge \exists \overline{p}.(\Delta \wedge F(Q')[\sigma_{x'}, x' \in Q'])$$

$F(Q')[\sigma_x, x \in Q]$ n'est pas tout à fait sous forme de cube car les $\sigma_x$ peuvent contenir des termes *ite*. Notons $F_{d_1}(\bar{i},Q) \vee \ldots \vee F_{d_h}(\bar{i},Q) = \texttt{elim_ite}(F(Q')[\sigma_x, x \in Q])$ la disjonction résultant de l'élimination des *ite* des mises à jour. Les $F_d(\bar{i},Q)$ sont des conjonctions de littéraux. En sortant les disjonctions, on obtient :

$$\exists Q'. \bigvee_{F_d(\bar{i},Q)\ \in \texttt{elim_ite}(F(Q')[\sigma_x, x \in Q])} \exists \bar{i}.\ \gamma(\bar{i},Q) \wedge \exists \overline{p}.(\Delta \wedge F_d(\bar{i},Q))$$

et donc :

$$\text{PRE}_t(\varphi)(Q) = \bigvee_{F_d(\bar{i},Q)\ \in\ \texttt{elim_ite}(F(Q')[\sigma_x, x \in Q])} \exists \bar{i}. \exists \overline{p}.\ (\Delta \wedge \gamma(\bar{i},Q) \wedge F_d(\bar{i},Q))$$

□

Dans ce qui suit on supposera que la formule caractérisant les états dangereux $\Theta$ est un cube. La clôture construite par l'algorithme sera donc une disjonction de cubes et pourra être vue comme un ensemble de cubes $\mathcal{V}$.

On donne une analyse d'atteignabilité par chaînage arrière classique dans l'algorithme 4. Cet algorithme maintient une file à priorité $\mathcal{Q}$ de cubes à traiter et la clôture partielle ou l'ensemble des cubes *visités* $\mathcal{V}$ dont la disjonction des éléments caractérise les états pouvant atteindre $\Theta$. On démarre avec en ensemble $\mathcal{V}$ vide matérialisant la formule false et une file $\mathcal{Q}$ où seule la formule $\Theta$ est présente.

La fonction BWD calcule itérativement la clôture $\text{PRE}^*_\tau(\Theta)$ et si elle termine, renvoie un des deux résultats possibles :

**Algorithme 4 :** Analyse d'atteignabilité par chaînage arrière

**Entrées** : un système paramétré $\mathcal{S} = (Q, I, \tau)$ et un cube $\Theta$
**Variables** :
$\mathcal{V}$ : cubes visités
$\mathcal{Q}$ : file de travail

```
1  function BWD(S, Θ) : begin
2      V := ∅;
3      push(Q, Θ);
4      while not_empty(Q) do
5          φ := pop(Q);
6          if φ ∧ I satisfiable then
7              return unsafe
8          else if φ ⊭ V then
9              V := V ∪ {φ};
10             push(Q, PRE_τ(φ));
11     return safe
```

— ***unsafe*** lorsqu'un cube ne passe pas le test de *sûreté* (ligne 6) et donc s'intersecte avec la formule initiale.

— ***safe*** lorsque la file $\mathcal{Q}$ est vide. Dans ce cas on a atteint un point fixe global et $\mathcal{V}$ contient $\text{PRE}^*_\tau(\Theta)$.

### 3.1.2 Correction

La correction de l'algorithme BWD de l'algorithme 4 repose sur les deux invariants suivants :

1. $\mathcal{V}$ ne contient pas de cube *directement* atteignable, *i.e.*

$$\mathcal{V} \models \neg I \tag{3.2}$$

2. $\text{PRE}^*_\tau(\Theta)$ est calculé *incrémentalement* à l'aide de $\mathcal{V}$ et $\mathcal{Q}$, *i.e.*

$$\text{PRE}^*_\tau(\Theta) \models \mathcal{V} \vee \text{PRE}^*_\tau(\mathcal{Q} \backslash \mathcal{V}) \tag{3.3}$$

*Démonstration.*
(3.2) Trivial car lorsqu'on rajoute un nœud $\varphi$ à $\mathcal{V}$, le test de sûreté ligne 6 est faux, $\varphi \wedge I$ est insatisfiable.

(3.3) Supposons que $\text{Pre}^*_\tau(\Theta) \models \mathcal{V} \vee \text{Pre}^*_\tau(\mathcal{Q}\backslash\mathcal{V})$. Si $\varphi \models \mathcal{V}$ (point-fixe) alors $\mathcal{Q}\backslash\mathcal{V} \models (\mathcal{Q}\backslash\varphi)\backslash\mathcal{V}$ donc $\text{Pre}^*_\tau(\Theta) \models \mathcal{V} \vee \text{Pre}^*_\tau((\mathcal{Q}\backslash\varphi)\backslash\mathcal{V})$.

Dans tous les cas on peut calculer la pré-image d'un nœud pour le faire passer dans $\mathcal{V}$. On a besoin de la propriété suivante sur la fonction Pre :

$$\text{Pre}^*_\tau(\varphi) = \varphi \ \vee \ \text{Pre}^*_\tau(\text{Pre}_\tau(\varphi)) \tag{3.4}$$

$$\begin{aligned}
\text{On a } \text{Pre}^*_\tau(\mathcal{Q}\backslash\mathcal{V}) &= \text{Pre}^*_\tau((\mathcal{Q}\backslash\varphi \ \vee \ \varphi)\backslash\mathcal{V}) \ \text{ car } \varphi \models \mathcal{Q} \\
&= \text{Pre}^*_\tau((\mathcal{Q}\backslash\varphi)\backslash\mathcal{V}) \ \vee \ \text{Pre}^*_\tau(\varphi\backslash\mathcal{V}) \\
&\models \text{Pre}^*_\tau((\mathcal{Q}\backslash\varphi)\backslash\mathcal{V}) \ \vee \ (\text{Pre}^*_\tau(\varphi) \wedge \text{Pre}^*_\tau(\neg\mathcal{V})) \\
\text{avec (3.4), } &\models \text{Pre}^*_\tau((\mathcal{Q}\backslash\varphi)\backslash\mathcal{V}) \ \vee \ \text{Pre}^*_\tau(\text{Pre}_\tau(\varphi)\backslash\mathcal{V}) \vee (\varphi \wedge \text{Pre}^*_\tau(\neg\mathcal{V})) \\
&\models \text{Pre}^*_\tau((\mathcal{Q}\backslash\varphi)\backslash\mathcal{V}) \ \vee \ \text{Pre}^*_\tau(\text{Pre}_\tau(\varphi)\backslash\mathcal{V}) \vee \varphi \\
&\models \text{Pre}^*_\tau((\mathcal{Q}\backslash\varphi \vee \text{Pre}_\tau(\varphi))\backslash\mathcal{V}) \vee \varphi \\
&\models \text{Pre}^*_\tau((\mathcal{Q}\backslash\varphi \vee \text{Pre}_\tau(\varphi))\backslash(\mathcal{V} \vee \varphi)) \vee \varphi \\
\text{donc, } \text{Pre}^*_\tau(\Theta) &\models (\mathcal{V} \vee \varphi) \ \vee \ \text{Pre}^*_\tau((\mathcal{Q}\backslash\varphi \vee \text{Pre}_\tau(\varphi))\backslash(\mathcal{V} \vee \varphi))
\end{aligned}$$

□

**Théorème 3.1.2.** *Si* `BWD`$(\mathcal{S},\Theta)$ *renvoie* ***safe*** *alors* $\Theta$ *n'est pas atteignable dans le système* $\mathcal{S}$.

*Démonstration.* Lorsque `BWD` renvoie ***safe***, la boucle de la ligne 4 se termine avec la file $\mathcal{Q}$ vide. Maintenant, supposons par contradiction que $\Theta$ est atteignable. Par définition $\text{Pre}^*_\tau(\Theta) \wedge I$ est satisfiable. Comme $\mathcal{Q}$ est vide, l'invariant (3.3) devient $\text{Pre}^*_\tau(\Theta) \models \mathcal{V}$, et donc immédiatement, $\mathcal{V} \wedge I$ est aussi satisfiable. Cela contredit l'invariant (3.2).

□

*Remarque.* La preuve de la correction de `BWD` ne dépend pas du fragment de la logique qu'on choisit. En particulier, les restrictions sur les théories des indices et des éléments n'entrent pas en compte.

### 3.1.3 Effectivité

On peut remarquer que les tests qui sont effectués aux lignes `6` et `8` sont des tests de *satisfiabilité* et peuvent être déchargés par un solveur SMT. La deuxième brique de base de cet algorithme est ce solveur. Cependant, pour que son utilisation soit possible il faut que la satisfiabilité des formules qui nous intéressent, $\varphi \wedge I$ et $\varphi \wedge \neg\mathcal{V}$, reste décidable. Ici $\varphi$ est un cube et $\mathcal{V}$ est une disjonction de cubes. Le test de point fixe prend en réalité la forme suivante :

$$\exists\bar{i}.\ (\Delta \wedge F) \wedge \neg \bigvee \exists\overline{j_k}.\ (\Delta_k \wedge F_k)$$

En poussant les négations sous les quantificateurs on obtient

$$\exists \bar{i}.\ (\Delta \wedge F) \wedge \bigwedge \forall \overline{j_k}.\ (\neg \Delta_k \vee \neg F_k)$$

Enfin, en renommant les $\overline{j_k}$ pour éviter les captures, on peut facilement regrouper et faire ressortir les variables quantifiées universellement ($\cdot$ dénote ici la concaténation de séquence)

$$\exists \bar{i}. \forall \overline{j_1} \cdot \ldots \cdot \overline{j_{|\mathcal{V}|}}.\ (\Delta \wedge F) \wedge \bigwedge (\neg \Delta_k \vee \neg F_k)$$

Plus simplement, on a besoin de décider la satisfiabilité de formules de la forme $\exists \bar{i}. \forall \bar{j}. \psi$ où les éléments de $\bar{i}$ et $\bar{j}$ sont du type `proc` (dans $\mathcal{T}_P$) et $\psi$ est une formule sans quantificateurs de $\mathcal{T}_A$.

**Théorème 3.1.3** ([74, Théorème A.1])**.** *Une formule de la forme $\exists \bar{i}. \forall \bar{j}. \psi$ est satisfiable dans $\mathcal{T}_A$ ssi elle est satisfiable dans un modèle d'indices fini (un modèle $\mathcal{M} = (\mathcal{I}, D)$ de $\mathcal{T}_A$ tel que $\mathcal{I}(\texttt{proc})$ est fini).*

**Proposition 3.1.4** ([77, Théorème 3.3])**.** *Soit $\mathcal{T}_A$ la combinaison des théories $\mathcal{T}_P$ et $\mathcal{T}_E$ comme décrite en section 2.4.3. La satisfiabilité des formules $\exists \bar{i}. \forall \bar{j}. \psi$ dans $\mathcal{T}_A$ est décidable si :*

- *le problème $SMT(\mathcal{T}_P)$ est décidable*
- *le problème $SMT(\mathcal{T}_E)$ est décidable*
- *$\mathcal{T}_P$ est localement finie et close par sous-structures*

Plutôt que de donner la preuve de ce théorème (la proposition 3.1.4 est une conséquence du théorème 3.1.3, le lecteur intéressé par ces preuves est renvoyé vers [77]), on donne ici un algorithme naïf pour décider (sous les hypothèses de la proposition 3.1.4) la satisfiabilité des formules apparaissant dans les tests de point fixes.

La théorie $\mathcal{T}_A$ est la combinaison des théories $\mathcal{T}_P$, et $\mathcal{T}_E$ étendue avec un ensemble de fonctions non-interprétées ayant pour domaine `proc` et pour co-domaine `elem`. Comme les modèles qui nous intéressent sont ceux d'indices fini, il suffit de considérer toutes les classes d'équivalence possibles entre les constantes de type `proc` et de choisir un représentant pour chaque classe. Il y a un nombre fini d'arrangements possibles avec chacun un nombre fini de processus distincts. Les interprétations possibles des symboles de fonction et prédicats de $\mathcal{T}_P$ sont donc en nombre fini. Une fois que le modèle de $\mathcal{T}_P$ est fixé, il suffit d'appeler le solveur $SMT(\mathcal{T}_E)$ sur le problème résultant.

Cette combinaison est décidable par des techniques comme celles du cadre de Nelson-Oppen [127] ou de la méthode de Shostak [146, 153] et on peut donc vérifier la satisfiabilité de ces formules closes avec un solveur SMT. En revanche les formules qui nous intéressent sont de la forme $\exists \bar{i}. \forall \bar{j}. \psi$.

**Définition 3.1.4** (Forme normale de Skolem)**.** Une formule est en *forme normale de Skolem* si elle est sous forme *prénexe* et que tous ses quantificateurs sont *universels*.

*Remarque.* Il est toujours possible de mettre une formule en forme normale de Skolem grâce à un processus appelé la *Skolemisation*. La formule résultante n'est pas forcément équivalente à la première mais elle lui reste *équisatisfiable*. Prenons par exemple une formule en forme prénexe $\exists x.\forall y.\exists z.\forall w.F(x,y,z,w)$, la Skolemisation de cette formule est $\forall y.\forall w.F(\#_x, y, \#_z(y), w)$ où $\#_x$ est une *constante de Skolem* fraîche et $\#_z$ est une *fonction de Skolem* fraîche. Le terme obtenu par Skolemisation pour $z$ dépend de $y$ car $z$ était quantifiée existentiellement en dessous du quantificateur de $y$. Par la suite on gardera la notation $\#_i$ pour dénoter les constantes de Skolem.

Pour décider la satisfiabilité de $\exists \bar{i}.\forall \bar{j}.\psi$ la première étape est une Skolemisation à la suite de laquelle on obtient $\forall \bar{j}.\psi[i_1 \backslash \#_1, \ldots, i_n \backslash \#_n]$. Tout ce qu'il reste à faire pour se ramener à une formule close est d'instancier de manière exhaustive les variables quantifiées universellement avec les représentants des classes d'équivalences choisies pour $\mathcal{T}_P$. Appelons $p_1, \ldots, p_n$ ces représentants, et notons $\sigma$ l'ensemble de toutes les substitutions possibles associant aux $\bar{j}$ une combinaision des $p_1, \ldots, p_n$. Par le théorème 3.1.3 la formule de départ est satisfiable ssi la formule suivante est satisfiable

$$\bigwedge_{\sigma_j \in \sigma} \psi[i_1 \backslash \#_1, \ldots, i_n \backslash \#_n]\sigma_j$$

Pour résumer, ces tests de satisfiabilité peuvent être déchargés par l'algorithme naïf suivant.

**Algorithme 5 :** Test de satisfiabilité naïf

**Entrées** : une formule $\exists \bar{i}.\forall \bar{j}.\psi$ telle que $\psi$ est sans quantificateurs dans $\mathcal{T}_A$

```
1 function sat(∃ī.∀j̄.ψ) : begin
2     let ∀j̄.ψ' = skolemisation(∃ī.∀j̄.ψ) in
3     foreach 𝒞 ∈ classes(𝒯_P) do
4         let p_1, ..., p_n = représentants(𝒞) in
5         let σ = substitutions(j̄, (p_1, ..., p_n)) in
6         if SMT(⋀_{σ_j ∈ σ} ψ'σ_j) = sat then return sat
7     return unsat
```

On n'appelle pas de solveur SMT sur le problème avec quantificateurs de départ car ils ne sont ni complets ni efficaces sur ce fragment. L'algorithme 5 est fortement inefficace mais a le mérite de décider de manière simple la satisfiabilité de formules $\exists \bar{i}.\forall \bar{j}.\psi$ dans notre fragment choisi. Une version améliorée est donnée en section 4.2.3.

*Remarque.* Les tests de sûreté $\varphi \wedge I$ sont décidables sous les mêmes conditions et par le même algorithme car $I$ est supposé quantifié universellement. Les tests de sûreté s'effectuent donc sur des formules $\exists \bar{i}.(\Delta \wedge F) \wedge \forall \bar{j}.I'$ qui peuvent se réécrire sous la forme utilisée précédemment de la même manière.

## 3.2 Terminaison

La terminaison de l'algorithme 4 n'est pas garantie en général. Dans cette section, on explicite les conditions suffisantes sous lesquelles l'analyse d'atteignabilité termine. La condition de terminaison relativement simple fait des systèmes à tableaux un cadre facile à utiliser pour s'assurer que les problèmes d'atteignabilité qu'un utilisateur décrit sont décidables.

### 3.2.1 Indécidabilité de l'atteignabilité

Il est bien connu que dans le cas général, le problème de l'atteignabilité (ou de la sûreté) dans un système paramétré est un problème indécidable, comme l'ont montré Apt et Kozen en 1986 [10]. Ce problème est aussi appelé le problème de la *vérification uniforme*. En réalité, même dans les systèmes à tableaux qui respectent les limites sur les théories qu'on s'est fixé précédemment, ce problème est indécidable.

**Théorème 3.2.1** ([77, Théorème 4.1])**.** *Soit $U$ un cube, le problème de la sûreté d'un système à tableaux vis-à-vis de $U$ est* indécidable.

*Démonstration.* La preuve de ce théorème consiste à encoder l'arrêt d'une machine de Turing (indécidable) comme un problème d'atteignabilité dans un système à tableaux.

La machine qu'on utilise ici est une machine de Minsky [123] à deux registres. Elle est Turing-complète. Une telle machine est composée d'un compteur d'instruction (`PC`) et de deux registres (`R1` et `R2`) contenant des entiers naturels non bornés. Elle prend en argument un programme sous la forme d'une suite d'instructions. Les instructions sont de deux types `INC` et `JMP` présentés en figure 3.1 ($r$ désigne le registre `R1` ou `R2`).

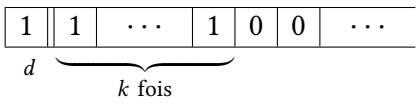
Lorsque la machine lit l'instruction $\texttt{INC}(r, l)$, elle incrémente le registre correspondant à $r$ et va en position $l$. L'instruction $\texttt{JMP}(r, l, l')$ correspond à un saut conditionnel. Si le registre dénoté par $r$ contient une valeur non nulle, alors elle est décrémentée et la machine va en $l$. Si, au contraire, le registre contient 0, la machine *saute* au point de programme $l'$. Il est bien connu que le problème de l'arrêt d'une machine de Minsky est indécidable [123]. C'est à dire, il est impossible de construire un algorithme qui, étant donné un programme $P$ et un point de ce programme $h$, dit si $P$ atteint $h$ ou non.

Dans notre cas, on montre qu'il est possible de prendre n'importe quel programme $P$ (utilisant les instructions décrites précédemment) et de construire un système à tableaux

| **instruction** | **actions** |
|---|---|
| $\texttt{INC}(r, l)$ | $r := r + 1$<br>$\texttt{PC} := l$ |
| $\texttt{JMP}(r, l, l')$ | **if** $r \neq 0$ **then** $r := r - 1$<br>$\texttt{PC} := l$<br>**else** $\texttt{PC} := l'$ |

FIGURE 3.1 – Machine de Minsky à deux compteurs

qui simule l'exécution de la machine de Minsky sur $P$. Une manière immédiate pour réaliser cette traduction est de choisir pour théories des éléments une théorie des types énumérés dont les constructeurs sont les points de programme de $P$ et la théorie de l'arithmétique de Presburger (domaine $\mathbb{N}$ et signature $(\{\texttt{int}\}, \{+, -, 0, 1\}, \{=\})$). Toutefois cette dernière n'est pas localement finie. Une astuce pour s'affranchir de cette restriction est d'encoder les nombres naturels sous forme *unaire* dans des tableaux de taille non bornée (d'une manière similaire à l'encodage fait en section 2.3). L'entier $k$ sera représenté par le tableau suivant :




La théorie des indices $\mathcal{T}_P$ a pour signature $\Sigma_P = (\{\texttt{proc}\}, \{d : \texttt{proc}\}, \{=, \lhd : \texttt{proc}\times\texttt{proc}\})$ où $d$ est une constante servant à marquer le début du tableau et $\lhd$ est une relation binaire matérialisant « être successeur. » $i \lhd j$ signifie $j$ est le successeur de $i$, et on note $i \lhd j \lhd k$ pour $i \lhd j \wedge j \lhd k$. La relation $\lhd$ est définie injective et respecte l'axiome $\neg\exists i.\ i \lhd d$ ($d$ marque le début du tableau et n'est donc le successeur de personne). On donne en figure 3.2 (voir page suivante) la traduction des instructions du programme $P$ concernant le registre R1 vers un système à tableaux. Le système obtenu simule l'exécution de la machine de Minksy à deux registres sur $P$ car à chaque état de la machine correspond un état du système. On donne aussi la formule représentant les états initiaux de la machine au point d'entrée du programme. La traduction des instructions pour le registre R2 est symétrique.

Remarquons que la formule $\Theta \equiv \texttt{PC} = h$ est bien un cube. Elle est atteignable dans le système à tableaux obtenu après la traduction de la figure 3.2 effectuée ssi le point de programme $h$ est atteignable dans $P$ par la machine de Minsky à deux registres. □

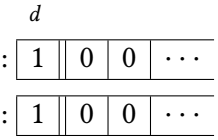
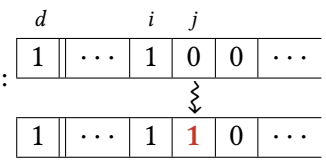
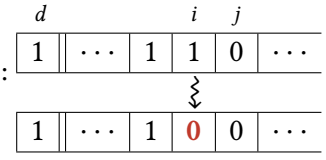
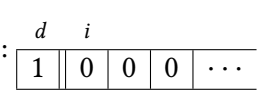
| instruction | transition/formule | schématique |
|---|---|---|
| point d'entrée $l$ : | $\mathtt{PC} = l \ \wedge \ \mathtt{R1}[d] = \mathtt{R2}[d] = 1 \ \wedge$ $\forall i.\ i \neq d \Rightarrow \mathtt{R1}[i] = \mathtt{R2}[i] = 0$ | $\mathtt{PC} : l$   |
| $l : \mathtt{INC}(\mathtt{R1}, m)$ | $\exists i, j.\ \mathtt{PC} = l \ \wedge \ i \lhd j \ \wedge$ $\mathtt{R1}[i] = 1 \ \wedge \ \mathtt{R1}[j] = 0 \ \wedge$ $\mathtt{PC}' = m \ \wedge \ \mathtt{R2}' = \mathtt{R2} \ \wedge$ $\mathtt{R1}' = \lambda k.\ ite(k = j,\ 1,\ \mathtt{R1}[k])$ | $\mathtt{PC} : l \rightsquigarrow \boldsymbol{m}$   |
| $l : \mathtt{JMP}(\mathtt{R1}, m, n)$ | $\exists i, j.\ \mathtt{PC} = l \ \wedge \ i \neq d \wedge i \lhd j \ \wedge$ $\mathtt{R1}[i] = 1 \ \wedge \ \mathtt{R1}[j] = 0 \ \wedge$ $\mathtt{PC}' = m \ \wedge \ \mathtt{R2}' = \mathtt{R2} \ \wedge$ $\mathtt{R1}' = \lambda k.\ ite(k = i,\ 0,\ \mathtt{R1}[k])$ | $\mathtt{PC} : l \rightsquigarrow \boldsymbol{m}$   |
| | $\exists i.\ \mathtt{PC} = l \ \wedge \ d \lhd i \ \wedge$ $\mathtt{R1}[i] = 0 \ \wedge$ $\mathtt{PC}' = n \ \wedge$ $\mathtt{R1}' = \mathtt{R1} \ \wedge \ \mathtt{R2}' = \mathtt{R2}$ | $\mathtt{PC} : l \rightsquigarrow \boldsymbol{n}$   |

FIGURE 3.2 – Traduction d'un programme et d'une machine de Minsky dans un système à tableaux

### 3.2.2 Conditions pour la terminaison

Un système à tableaux $\mathcal{S}$ a un nombre potentiellement infini d'états, et chaque état est un modèle de $\mathcal{T}_A$. Les conditions mises en évidence par Ghilardi et Ranise [77] permettent de se ramener à des preuves de terminaison reposant sur la notion de *bel ordre*. L'argument de terminaison est ensuite similaire à celui employé par Abdulla *et al.* pour la preuve de la décidabilité de l'analyse arrière pour les systèmes d'états infinis [6]. Cette approche est également utilisée dans le cadre des systèmes bien formés, aussi appelés WSTS [69, 70].

**Définition 3.2.1.** On appelle *configuration* de $\mathcal{S}$ un modèle de $\mathcal{T}_A$ (i.e. un état de $\mathcal{S}$) d'indices finis.

**Définition 3.2.2.** Un *pré-ordre* $\preceq$ est une relation (binaire) réflexive et transitive sur un

ensemble $D$. On notera $a \prec b$ si $a \preceq b$ et $b \not\preceq a$. On dit que $\preceq$ est *bien fondé* ssi il n'existe pas de séquence infinie décroissante $a_1 \succ a_2 \succ a_3 \succ \cdots$.

**Définition 3.2.3.** Un ensemble $I \subseteq D$ est un *idéal* (pour $\preceq$ dans $D$), si pour tout $a \in I$, $b \in D$, $a \preceq b \implies b \in I$. On définit la *clôture supérieure* (ou *section finissante*) de $A$, et on note $\uparrow A$, l'idéal $\{b \in D \mid \exists a \in A.\ a \preceq b\}$ généré par $A$.

**Définition 3.2.4.** Un pré-ordre $\preceq$ est un *bel ordre*, si pour toute séquence infinie $s_1, s_2, \ldots$, il existe $i < j$ tels que $s_i \preceq s_j$.

**Définition 3.2.5.** Un $\Sigma$-*plongement* $\mu$ d'une $\Sigma$-structure $A$ vers une $\Sigma$-structure $B$ est un homomorphisme $\mu : dom(A) \to dom(B)$ injectif ayant les propriétés suivantes :

1. pour toute constante $c$ de $\Sigma$, $\mu(c_A) = c_B$
2. pour tout symbole de relation $R$ de $\Sigma$ d'arité $n$, et $a_1, \ldots, a_n \in dom(A)$, $R_A(a_1, \ldots, a_n) \iff R_B(\mu(a_1), \ldots, \mu(a_n))$
3. pour tout symbole de fonction $f$ de $\Sigma$ d'arité $n$, et $a_1, \ldots, a_n \in dom(A)$, $\mu(f_A(a_1, \ldots, a_n)) = f_B(\mu(a_1), \ldots, \mu(a_n))$

où $c_A, f_A, R_A$ dénotent les interprétations des symboles dans le domaine de $A$ ($c_A = \mathcal{I}_A(c), f_A = \mathcal{I}_A(f), R_A = \mathcal{I}_A(R)$).

On peut maintenant définir un pré-ordre sur les configurations de la manière suivante. Soient $\mathcal{M}$ et $\mathcal{N}$ deux configurations, on définit un pré-ordre sur les configurations tel que $\mathcal{M} \preceq \mathcal{N}$ ssi il existe un $\Sigma_A$-plongement $\mu$ de $\mathcal{M}$ vers $\mathcal{N}$.

**Exemple.** Donnons nous comme théorie $\mathcal{T}_E$ une théorie des types énumérés à trois constructeurs $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ de sorte $\mathtt{t}$ et comme théorie $\mathcal{T}_P$ la théorie de l'ordre total ayant pour signature $\Sigma_P = (\{\mathtt{proc}\}, \emptyset, \{=, \leq\})$. Ses axiomes sont :

$$\forall i, j, k.\ i \leq j \wedge j \leq k \implies i \leq k \qquad \text{(transitivité)} \tag{3.5}$$

$$\forall i, j.\ i \leq j \wedge j \leq i \implies i = j \qquad \text{(antisymétrie)} \tag{3.6}$$

$$\forall i.\ i \leq i \qquad \text{(réflexivité)} \tag{3.7}$$

$$\forall i, j.\ i \leq j \vee j \leq i \qquad \text{(totalité)} \tag{3.8}$$

On se donne également deux symboles de fonction $x$ et $y$ d'arité un pour former $\mathcal{T}_A$. Soient $\mathcal{M}$ et $\mathcal{N}$ deux configurations de $\mathcal{T}_A$ telles que décrites dans le tableau en figure 3.3 (voir page suivante).

On a alors $\mathcal{M} \preceq \mathcal{N}$ car il existe bien un $\Sigma_A$-plongement $\mu : dom(\mathcal{M}) \to dom(\mathcal{N})$ avec $\mu(m_1) = n_1$, $\mu(m_2) = n_3$, $\mu(\mathsf{A}) = \mathsf{A}$, $\mu(\mathsf{B}) = \mathsf{B}$, $\mu(\mathsf{C}) = \mathsf{C}$, $\mu(\leq^{\mathcal{M}}) = \leq^{\mathcal{N}}$, $\mu(x^{\mathcal{M}}) = x^{\mathcal{N}}$ et $\mu(y^{\mathcal{M}}) = y^{\mathcal{N}}$. Le plongement $\mu$ est illustré en figure 3.4 (voir page précédente).

| | $\mathcal{M}$ | $\mathcal{N}$ |
|---|---|---|
| $\Sigma_A$ | $S = \{\ \mathtt{proc},\ \mathtt{t}\ \}$<br>$F = \{\ \mathsf{A} : \mathtt{t},\ \mathsf{B} : \mathtt{t},\ \mathsf{C} : \mathtt{t},\ x : \mathtt{proc} \to \mathtt{t},\ y : \mathtt{proc} \to \mathtt{t}\ \}$<br>$R = \{\ = : \ldots,\ \leq : \mathtt{proc} \times \mathtt{proc}\ \}$ | |
| *dom* | $\{m_1, m_2, \leq^{\mathcal{M}}, \mathsf{A}, \mathsf{B}, \mathsf{C}, x^{\mathcal{M}}, y^{\mathcal{M}}\}$ | $\{n_1, n_2, n_3, \leq^{\mathcal{N}}, \mathsf{A}, \mathsf{B}, \mathsf{C}, x^{\mathcal{N}}, y^{\mathcal{N}}\}$ |
| $\mathcal{I}$ | $\mathtt{proc} \longmapsto \{m_1, m_2\}$<br>$\mathtt{t} \longmapsto \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$<br>$\leq \longmapsto \leq^{\mathcal{M}}$<br>$\mathsf{A} \longmapsto \mathsf{A}$<br>$\mathsf{B} \longmapsto \mathsf{B}$<br>$\mathsf{C} \longmapsto \mathsf{C}$<br>$x \longmapsto x^{\mathcal{M}}$<br>$y \longmapsto y^{\mathcal{M}}$ | $\mathtt{proc} \longmapsto \{n_1, n_2, n_3\}$<br>$\mathtt{t} \longmapsto \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$<br>$\leq \longmapsto \leq^{\mathcal{N}}$<br>$\mathsf{A} \longmapsto \mathsf{A}$<br>$\mathsf{B} \longmapsto \mathsf{B}$<br>$\mathsf{C} \longmapsto \mathsf{C}$<br>$x \longmapsto x^{\mathcal{N}}$<br>$y \longmapsto y^{\mathcal{N}}$ |
| définitions | $m_1 \leq^{\mathcal{M}} m_2$<br>$x^{\mathcal{M}} : \begin{cases} m_1 \longmapsto \mathsf{A} \\ m_2 \longmapsto \mathsf{C} \end{cases}$<br>$y^{\mathcal{M}} : \begin{cases} m_1 \longmapsto \mathsf{A} \\ m_2 \longmapsto \mathsf{A} \end{cases}$ | $n_1 \leq^{\mathcal{N}} n_2,\ n_2 \leq^{\mathcal{N}} n_3,\ n_1 \leq^{\mathcal{N}} n_3$<br>$x^{\mathcal{N}} : \begin{cases} n_1 \longmapsto \mathsf{A} \\ n_2 \longmapsto \mathsf{A} \\ n_3 \longmapsto \mathsf{C} \end{cases}$<br>$y^{\mathcal{N}} : \begin{cases} n_1 \longmapsto \mathsf{A} \\ n_2 \longmapsto \mathsf{B} \\ n_3 \longmapsto \mathsf{A} \end{cases}$ |

FIGURE 3.3 – Définition des configurations $\mathcal{M}$ et $\mathcal{N}$

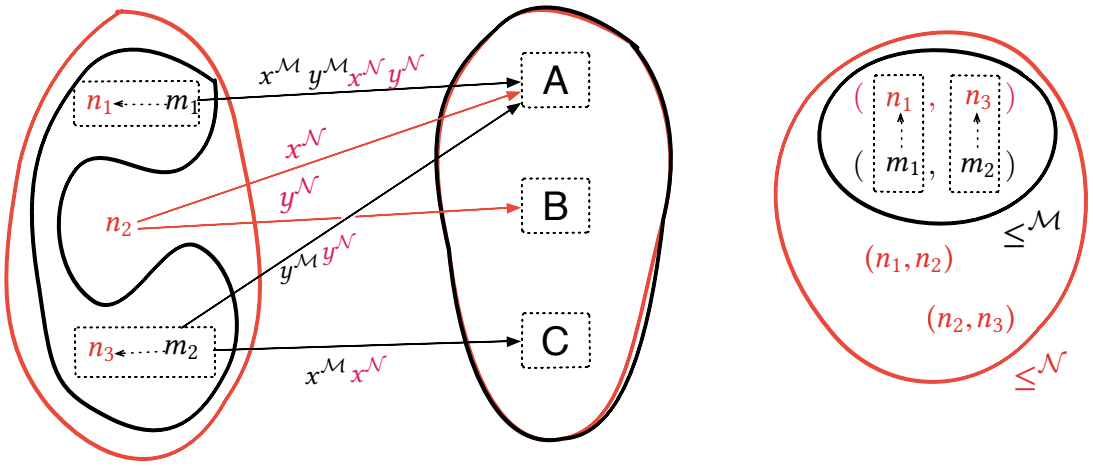



FIGURE 3.4 – Plongement de $\mathcal{M}$ vers $\mathcal{N}$

**Théorème 3.2.2.** *Soit $[\![U]\!] = \{\mathcal{M} \mid \mathcal{M} \models U\}$ l'ensemble des configurations de $U$. Pour tout cube $U$, l'ensemble $[\![U]\!]$ est un idéal pour l'ordre de plongement $\preceq$.*

*Démonstration.* On va en fait montrer qu'un plongement préserve la satisfiabilité des formules existentielles. C'est un résultat bien connu de théorie des modèles qu'on peut trouver par exemple dans [92, Théorème 2.4.1]. Soient $\mathcal{M}$ et $\mathcal{N}$ deux configurations telles que $\mathcal{M} \preceq \mathcal{N}$ et $\mathcal{M} \models \exists \bar{i}.\ \varphi(\bar{i})$, on veut montrer que $\mathcal{N} \models \exists \bar{i}.\ \varphi(\bar{i})$ aussi. Comme $\mathcal{M} \preceq \mathcal{N}$, alors il existe un plongement $\mu$ de $\mathcal{M}$ vers $\mathcal{N}$. Montrons tout d'abord que si $\varphi$ est une formule sans quantificateurs alors

$$\mathcal{M} \models \varphi \quad \iff \quad \mathcal{N} \models \mu(\varphi) \tag{3.9}$$

où $\mu$ est l'extension homomorphique du plongement $\mu$ aux termes et formules. Par induction sur la structure de $\varphi$, si $\varphi$ est un atome ($f(t_1, \ldots, t_n) = g(t'_1, \ldots, t'_m)$ ou $R(t_1, \ldots, t_n)$) alors (3.9) est vraie par la définition 3.2.5. Ensuite comme

$$\begin{array}{l}\mathcal{M} \models \neg\varphi_1 \text{ ssi } \mathcal{M} \not\models \varphi_1, \\ \mathcal{M} \models \varphi_1 \wedge \varphi_2 \text{ ssi } \mathcal{M} \models \varphi_1 \text{ et } \mathcal{M} \models \varphi_2, \text{ et enfin} \\ \mathcal{M} \models \varphi_1 \vee \varphi_2 \text{ ssi } \mathcal{M} \models \varphi_1 \text{ ou } \mathcal{M} \models \varphi_2,\end{array}$$

on a bien (3.9) pour les formules sans quantificateurs. Montrons maintenant que

$$\mathcal{M} \models \exists \bar{i}.\ \varphi(\bar{i}) \quad \implies \quad \mathcal{N} \models \exists \bar{i}.\ \mu(\varphi)(\bar{i}) \tag{3.10}$$

Si $\bar{i}$ est vide alors il n'y pas de quantificateurs et le résultat découle de (3.9). Sinon on sait qu'il y a des éléments $m_1, \ldots, m_n$ de $\mathcal{M}$ tels que $\mathcal{M} \models \varphi(m_1, \ldots, m_n)$. D'après (3.9), on a $\mathcal{N} \models \mu(\varphi)(\mu(m_1), \ldots, \mu(m_n))$, autrement dit $\mathcal{N} \models \exists \bar{i}.\ \mu(\varphi)(\bar{i})$.

□

**Proposition 3.2.3.** *Pour tous cubes $U_1, U_2$, $[\![U_1]\!] \subseteq [\![U_2]\!]$ ssi $\emptyset \models_{\mathcal{T}_A} U_1 \implies U_2$.*

*Démonstration.* Soient deux cubes $U_1$ et $U_2$. Triviallement, $\emptyset \models_{\mathcal{T}_A} U_1 \implies U_2$ implique $[\![U_1]\!] \subseteq [\![U_2]\!]$. Supposons maintenant par l'absurde que $\emptyset \not\models_{\mathcal{T}_A} U_1 \implies U_2$, ce qui veut dire que $U_1 \wedge \neg U_2$ est $\mathcal{T}_A$-satisfiable. On peut alors trouver une configuration $s$ telle que $s \models U_1$ et $s \models \neg U_2$, *i.e.* $s \in [\![U_1]\!]$ et $s \notin [\![U_2]\!]$, donc $[\![U_1]\!] \not\subseteq [\![U_2]\!]$.

□

**Lemme 3.2.4.** *Soit $\mathcal{S}$ un système à tableaux et $U$ un cube, alors $[\![\mathrm{PRE}_\tau(U)]\!]$ est un idéal.*

*Démonstration.* Comme $U$ est un cube alors par la proposition 3.1.1, $\mathrm{PRE}_\tau(U)$ est équivalente à une disjonction de cubes, i.e. $\mathrm{PRE}_\tau(U) \iff U_1 \vee \ldots \vee U_n$. Donc $[\![\mathrm{PRE}_\tau(U)]\!] = [\![U_1]\!] \cup \ldots \cup [\![U_n]\!]$, les $[\![U_i]\!]$ sont des idéaux par le théorème 3.2.2 et comme une union d'idéaux est un idéal alors $[\![\mathrm{PRE}_\tau(U)]\!]$ est un idéal.

□

**Théorème 3.2.5.** *Soit un bel ordre $\preceq$ et une séquence infinie d'idéaux $I_0 \subseteq I_1 \subseteq I_2 \subseteq \ldots$ alors il existe un $k$ tel que $I_k = I_{k+1}$.*

*Démonstration.* Supposons par l'absurde qu'il n'existe pas de tel $k$. Alors on peut trouver une séquence infinie d'éléments $s_0, s_1, s_2, \ldots$ telle que pout tout $i$, $s_i \in I_i$ et pour tout $j < i$, $s_i \notin I_j$. Et donc on a que $s_j \not\preceq s_i$ pour tous $j < i$ sinon $s_i \in I_j$ car $I_j$ est un idéal. Cette séquence qu'on a construite viole l'hypothèse de bel ordre faite sur $\preceq$.

□

Le théorème 3.2.5 dit essentiellement qu'il ne peut y avoir de suite infinie d'idéaux strictement croissante (dans le sens de l'inclusion). On va utiliser ce fait en remarquant que l'algorithme 4 construit dans $\mathcal{V}$ un idéal qui grossit strictement.

**Théorème 3.2.6** ([77, Corrolaire 4.7])**.** *L'algorithme 4 termine si le pré-ordre sur les configurations du système est un* bel-ordre*.*

*Démonstration.* Si le cube $U$ représente une formule dont un état est atteignable, alors il est atteignable en un nombre fini d'étapes et l'algorithme 4 termine. Supposons au contraire que $U$ ne soit pas atteignable. On note $Bad^n_\tau(U)$ la formule représentant les états pouvant atteindre $U$ en au plus $n$ étapes définie par :

$$\begin{cases} Bad^0_\tau(U) &= U \\ Bad^n_\tau(U) &= \textsc{Pre}_\tau(U) \vee Bad^{n-1}_\tau(U) \\ Bad^*_\tau(U) &= \bigvee_{k\in\mathbb{N}} Bad^k_\tau(U) \end{cases}$$

On a en fait $Bad^n_\tau(U) = \textsc{Pre}^n_\tau(U) \vee \textsc{Pre}^{n-1}_\tau(U) \vee \ldots \vee \textsc{Pre}_\tau(U) \vee U$ et surtout $Bad^*_\tau(U) = \textsc{Pre}^*_\tau(U)$. La formule $Bad^n_\tau(U)$ est donc la formule correspondant à la disjonction des éléments de $\mathcal{V}$ après $n$ tours de boucle.

Par le théorème 3.2.2 et le lemme 3.2.4, $Bad^n_\tau(U)$ est un idéal qu'on peut exprimer sous la forme $[\![Bad^n_\tau(U)]\!] = [\![\textsc{Pre}_\tau(U)]\!] \cup [\![Bad^{n-1}_\tau(U)]\!]$. En particulier, quelque soit $n \geq 1$, on a $[\![Bad^{n-1}_\tau(U)]\!] \subseteq [\![Bad^n_\tau(U)]\!]$. Il existe alors une séquence d'idéaux $[\![Bad^0_\tau(U)]\!] \subseteq [\![Bad^1_\tau(U)]\!] \subseteq [\![Bad^2_\tau(U)]\!] \subseteq \ldots$ (Figure 3.5) et le théorème 3.2.5 nous dit qu'il existe un $k$ tel que $[\![Bad^k_\tau(U)]\!] = [\![Bad^{k+1}_\tau(U)]\!]$ ce qui veut dire par la proposition 3.2.3 que $\emptyset \models_{\mathcal{T}_\mathcal{A}} Bad^k_\tau(U) \iff Bad^{k+1}_\tau(U)$. En particulier $\emptyset \models_{\mathcal{T}_\mathcal{A}} \textsc{Pre}^{k+1}\tau(U) \implies Bad^k_\tau(U)$, donc l'algorithme 4 termine.

□

Le théorème 3.2.6 est suffisant pour donner des conditions pour la terminaison de l'analyse d'atteignabilité par chaînage arrière sur les systèmes à tableaux. Toutefois Ghilardi et Ranise montrent le résultat plus fort suivant dans [77].

**Théorème 3.2.7** ([77, Théorème 4.6])**.** *Si $\mathcal{T}_E$ est localement finie et si $\Theta$ est inatteignable dans $\mathcal{S}$ alors l'algorithme 4 termine ssi $\textsc{Pre}^*_\tau(\Theta)$ est un idéal généré par un ensemble* fini*.*

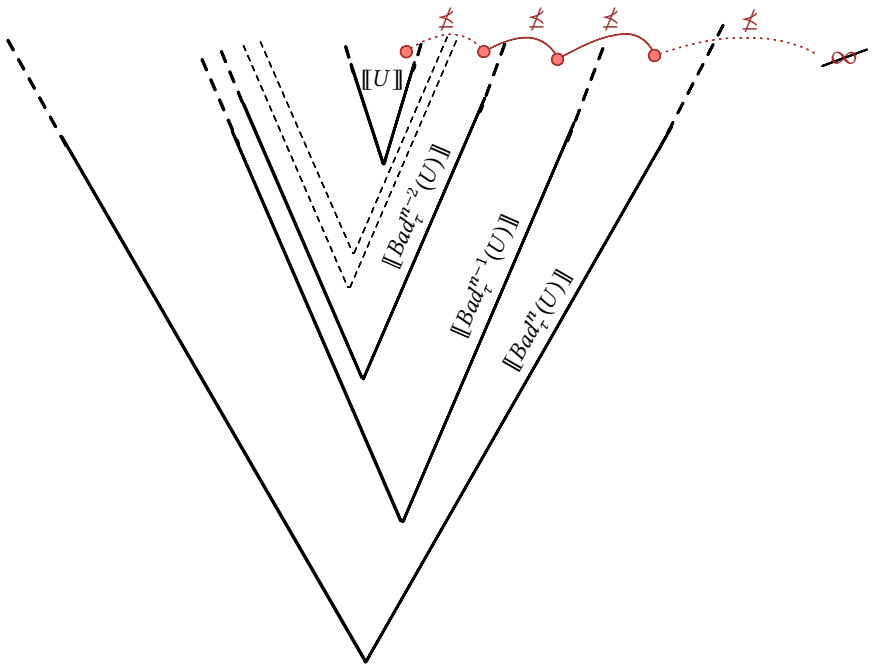



FIGURE 3.5 – Séquence finie d'idéaux inclus calculée par l'algorithme 4

### 3.2.3 Exemples

On a vu dans la section précédente que la terminaison de l'analyse d'atteignabilité arrière peut être obtenue dès qu'on est capable de prouver que l'ordre de plongement sur les configurations du système est un *bel ordre*. Dans cette section on donne quelques exemples de classes de systèmes pour lesquels on est assuré que l'algorithme 4 termine *en théorie*. La formalisation complète de certaines de ces classes de problèmes dans les systèmes à tableaux peut être trouvée dans [74].

#### Dickson

Prenons $\mathcal{T}_E$ une théorie des type énumérés et $\mathcal{T}_P$ la théorie de l'égalité. La signature de $\mathcal{T}_P$ contient seulement le symbole d'égalité et $\Sigma_E = \{=, C_1, \ldots, C_k\}$ où les $C_i$ sont les constructeurs. La seule structure de $\mathcal{T}_E$ est $\{C_1, \ldots, C_k\}$. Supposons aussi que les seuls symboles de fonctions $x_1, \ldots, x_n$ apportés par $\mathcal{T}_A$ soient d'arité un (on peut aussi avoir des symboles de constante $x_i$ car on peut toujours les voir comme des fonctions d'arité un constantes).

Les configurations de $\mathcal{T}_A$ sont de la forme $\mathcal{M} = (D, \mathcal{I})$ avec

$$D = \{m_1, \ldots, m_p,\ C_1, \ldots, C_k,\ x_1^{\mathcal{M}}, \ldots, x_n^{\mathcal{M}}\}$$

où $p$ est la cardinalité du modèle de $\mathcal{T}_P$. Les $x_i^{\mathcal{M}}$ sont des fonctions totales qui associent à chaque constante de processus $m$ un constructeur de $\mathcal{T}_E$.

À chaque configuration, on peut associer de manière unique un $k$-uplet $u$ de $n$-uplets d'entiers naturels (compris entre 1 et $p$) tel que $(z_1, \ldots, z_n)$ apparaisse dans $u$ en position $j$ ssi il y a $z_1$ éléments de $\{m_1, \ldots, m_p\}$ que la fonction $x_1^{\mathcal{M}}$ envoie sur $C_j$. De même pour $z_2$ et $x_2$, *etc.*

**Exemple.** Un exemple concret où on prend comme symboles de fonction unaire $x$ et constante $y$ ainsi que trois constructeurs $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ pour $\mathcal{T}_E$ associera un triplet d'éléments à deux composantes pour chaque configuration.

| | $\mathcal{M}$ | $\mathcal{N}$ |
|---|---|---|
| $\Sigma_A$ | $(\{\mathtt{proc}, \mathtt{t}\}, \{\mathsf{A} : \mathtt{t}, \mathsf{B} : \mathtt{t}, \mathsf{C} : \mathtt{t}, x : \mathtt{proc} \to \mathtt{t}, y : \mathtt{proc} \to \mathtt{t}\}, \{=\})$ | |
| *dom* | $\{m_1, m_2, \mathsf{A}, \mathsf{B}, \mathsf{C}, x^{\mathcal{M}}, y^{\mathcal{M}}\}$ | $\{n_1, n_2, n_3, \mathsf{A}, \mathsf{B}, \mathsf{C}, x^{\mathcal{N}}, y^{\mathcal{N}}\}$ |
| définitions | $x^{\mathcal{M}} : \begin{cases} m_1 \longmapsto \mathsf{A} \\ m_2 \longmapsto \mathsf{C} \end{cases}$ <br> $y^{\mathcal{M}} : \mathsf{B}$ | $x^{\mathcal{M}} : \begin{cases} n_1 \longmapsto \mathsf{C} \\ n_2 \longmapsto \mathsf{A} \\ n_3 \longmapsto \mathsf{A} \end{cases}$ <br> $y^{\mathcal{M}} : \mathsf{B}$ |
| triplet associé | $(\ (1,0),\ (0,2),\ (1,0)\ )$ | $(\ (2,0),\ (0,3),\ (1,0)\ )$ |

Ici le triplet $(\ (1,0),\ (0,2),\ (1,0)\ )$ dit que dans la configuration $\mathcal{M}$, il y a un processus pour lequel $x$ vaut $\mathsf{A}$, zéro processus pour lesquels $y$ vaut $\mathsf{A}$, et ainsi de suite.

**Lemme 3.2.8.** *On note $[\mathcal{M}]$ le $k$-uplet associé à la configuration $\mathcal{M}$. Sous les conditions définies précédemment, pour toutes configurations $\mathcal{M}$ et $\mathcal{N}$, on a $\mathcal{M} \preceq \mathcal{N}$ ssi $[\mathcal{M}] \leq [\mathcal{N}]$ où $\leq$ est l'extension de l'ordre point-à-point sur les $k$-uplets.*

*Démonstration.* Si $\mathcal{M} \preceq \mathcal{N}$, alors il existe un plongement $\mu$ de $\mathcal{M}$ vers $\mathcal{N}$. Soit $z$ la composante du $k$-uplet de $[\mathcal{M}]$ associée au constructeur $C$ et à la fonction $x$

$$(\ldots, \underbrace{(\ldots, \overbrace{z}^{x}, \ldots)}_{C}, \ldots)$$

donc il existe $m_1, \ldots, m_z$ tels que $x^{\mathcal{M}}(m_1) = C, \ldots, x^{\mathcal{M}}(m_z) = C$. Par la définition de $\mu$, $x^{\mathcal{N}}(\mu(m_1)) = C, \ldots, x^{\mathcal{N}}(\mu(m_z)) = C$ donc la composante du $k$-uplet de $[\mathcal{N}]$ associée

au constructeur $C$ et à la fonction $x$ et supérieure à $z$. Il en découle que $[\mathcal{M}] \leq [\mathcal{N}]$. Inversement si $[\mathcal{M}] \leq [\mathcal{N}]$, alors chaque composante fait associer $x$ à $C$ à un nombre $z'$ dans $[\mathcal{N}]$ plus grand que $z$ dans $[\mathcal{M}]$. Il suffit de prendre $\mu(m_1) = n_1$ jusqu'à $\mu(m_z) = n_z$ et de laisser $n_{z+1}, \ldots, n_{z'}$ hors du domaine de $\mu$. □

**Exemple.** Dans l'exemple concret précédent on a bien un plongement $\mu$ de $\mathcal{M}$ vers $\mathcal{N}$ (il suffit de prendre par exemple $\mu(m_1) = n_2$ et $\mu(m_2) = n_1$) et $(\ (1,0),\ (0,2),\ (1,0)\ ) \leq (\ (2,0),\ (0,3),\ (1,0)\ )$.

**Théorème 3.2.9** (Généralisation du lemme de Dickson [53])**.** *Soit $n \in \mathbb{N}$ et $D$ un ensemble muni d'un* bel ordre $\leq$*, alors l'extension point-à-point de $\leq$ aux $n$-uplets d'éléments de $D$ (i.e. $D^n$) est un* bel ordre.

Comme $\leq$ est un bel ordre sur les $n$-uplets d'entiers naturels, alors son extension $\leq$ est un bel ordre sur les $k$-uplets de $n$-uplets d'entiers naturels par le théorème 3.2.9. Il découle ensuite directement du lemme 3.2.8 que l'ordre de plongement $\leq$ est aussi un bel ordre.

#### Higman

Maintenant, on prend pour théorie $\mathcal{T}_E$ la théorie de l'égalité sur les entiers naturels $\mathbb{N}$, $\Sigma_E = (\{\texttt{int}\},\ \{0,1,2,3,\ldots\},\ \{=\})$ et comme théorie des processus $\mathcal{T}_P$ la théorie de l'ordre total (*c.f.* l'exemple en section 3.2.2) de signature $\Sigma_P = (\{\texttt{proc}\},\ \emptyset,\ \{=,\leq\})$. On suppose également que les seuls symboles de fonctions $x_1, \ldots, x_n : \texttt{proc} \to \texttt{int}$ apportés par $\mathcal{T}_A$ soient d'arité un.

Similairement au cas précédent, les $x_i^{\mathcal{M}}$ d'une configuration $\mathcal{M}$ associent à chaque constante $\{m_1, \ldots, m_p\}$ du modèle de $\mathcal{T}_P$ un entier de $\mathbb{N}$. Cependant, en plus de ça la configuration $\mathcal{M}$ doit aussi interpréter le symbole de relation $\leq$ sur $\{m_1, \ldots, m_p\}^2$ en satisfiant les axiomes (3.5 – 3.7) (voir page 60).

On ne peut plus associer de manière unique un tuple à chaque configuration à cause du domaine infini de $\mathcal{T}_E$ mais surtout à cause de la relation d'ordre sur les processus. Cependant il est possible d'associer à toute configuration et de façon unique un mot sur un alphabet (infini) de $n$-uplets d'entiers naturels.

On choisira le mot $(z_1^1, \ldots, z_n^1)\ \ldots\ (z_1^i, \ldots, z_n^i)\ \ldots\ (z_1^j, \ldots, z_n^j)\ \ldots\ (z_1^p, \ldots, z_n^p)$ si pour $\forall i,j.\ 1 \leq i < j \leq p \implies m_i \leq m_j$ et $\forall i,j.\ x_i^{\mathcal{M}}(m_j) = z_i^j$.

**Exemple.** Comme exemple concret on prend une théorie $\mathcal{T}_A$ avec un seul symbole de fonction supplémentaire unaire $x$. Pour chaque configuration on peut associer un mot sur l'alphabet $\mathbb{N}$.

| | $\mathcal{M}$ | $\mathcal{N}$ |
|---|---|---|
| $\Sigma_A$ | $(\{\texttt{proc}, \texttt{int}\},$ $\{x : \texttt{proc} \to \texttt{t}, 0 : \texttt{int}, 1 : \texttt{int}, 2 : \texttt{int}, 3 : \texttt{int}, \ldots\},$ $\{=, \leq : \texttt{proc} \times \texttt{proc}\})$ | |
| *dom* | $\{m_1, m_2, \leq^{\mathcal{M}}, x^{\mathcal{M}}, 0, 1, 2, 3, \ldots\}$ | $\{n_1, n_2, n_3, \leq^{\mathcal{N}}, x^{\mathcal{N}}, 0, 1, 2, 3, \ldots\}$ |
| définitions | $m_1 \leq^{\mathcal{M}} m_2$ $x^{\mathcal{M}} : \begin{cases} m_1 \longmapsto 2 \\ m_2 \longmapsto 8 \end{cases}$ | $n_1 \leq^{\mathcal{N}} n_2,\ n_2 \leq^{\mathcal{N}} n_3,\ n_1 \leq^{\mathcal{N}} n_3$ $x^{\mathcal{N}} : \begin{cases} n_1 \longmapsto 2 \\ n_2 \longmapsto 1 \\ n_3 \longmapsto 8 \end{cases}$ |
| mot associé | 2 8 | 2 1 8 |

**Lemme 3.2.10.** *On note* $[\mathcal{M}]$ *le mot associé à la configuration* $\mathcal{M}$*. Sous les conditions définies précédemment, pour toutes configurations* $\mathcal{M}$ *et* $\mathcal{N}$*, on a* $\mathcal{M} \preceq \mathcal{N}$ *ssi* $[\mathcal{M}] \leq [\mathcal{N}]$ *où* $\leq$ *est l'ordre* sous-mot.

*Démonstration.* Si $\mathcal{M} \preceq \mathcal{N}$, alors il existe un plongement $\mu$ de $\mathcal{M}$ vers $\mathcal{N}$. Soit $[\mathcal{M}] = \ldots\ (z_1^i, \ldots, z_n^i)\ \ldots\ (z_1^j, \ldots, z_n^j)\ \ldots$ alors on a

$$\begin{array}{l} x_1^{\mathcal{M}}(m_i) = z_1^i,\ \ldots,\ x_n^{\mathcal{M}}(m_i) = z_n^i \\ x_1^{\mathcal{M}}(m_j) = z_1^j,\ \ldots,\ x_n^{\mathcal{M}}(m_i) = z_n^j \\ m_i \leq^{\mathcal{M}} m_j \end{array}$$

par la définition du plongement on a également,

$$\begin{array}{l} x_1^{\mathcal{N}}(\mu(m_i)) = z_1^i,\ \ldots,\ x_n^{\mathcal{N}}(\mu(m_i)) = z_n^i \\ x_1^{\mathcal{N}}(\mu(m_j)) = z_1^j,\ \ldots,\ x_n^{\mathcal{N}}(\mu(m_i)) = z_n^j \\ \mu(m_i) \leq^{\mathcal{N}} \mu(m_j) \end{array}$$

donc $[\mathcal{N}] = \ldots\ (z_1^i, \ldots, z_n^i)\ \ldots\ (z_1^j, \ldots, z_n^j)\ \ldots$, c'est-à-dire $[\mathcal{M}]$ est un sous-mot de $[\mathcal{N}]$.

Inversement, supposons $[\mathcal{M}] \leq [\mathcal{N}]$. Prenons deux lettres $a$ et $b$ apparaissant dans $[\mathcal{M}]$ en positions respectives $i$ et $j$. Elles apparaissent donc aussi dans $[\mathcal{N}]$. Notons $q$ et $r$ leurs positions respectives. On a donc $i < j$ ssi $q < r$. Il existe alors un plongement $\mu$, par exemple en prenant $\mu(m_i) = n_q$ et $\mu(m_j) = n_r$. De plus comme $m_i \leq^{\mathcal{M}} m_j$ est équivalent au fait que $n_q \leq^{\mathcal{N}} n_r$ (car $\leq$ est total et transitif), le plongement tel que $\mu(\leq^{\mathcal{M}}) = \leq^{\mathcal{N}}$ préserve bien les relations.

□

**Exemple.** Dans l'exemple au dessus, le mot 2 8 est bien un sous mot de 2 1 8 et il y a un plongement $\mu$ de $\mathcal{M}$ vers $\mathcal{N}$ tel que $\mu(m_1) = n_1$ et $\mu(m_2) = n_3$.

**Théorème 3.2.11** (Lemme de Higman [90])**.** *Soit $D$ un ensemble muni d'un* bel ordre $\leq$*, alors l'ordre sous-mot $\leq^*$ est un* bel ordre *sur l'ensemble $D^*$ des mots de $D$.*

Grâce au lemme de Dickson (théorème 3.2.9) l'ordre point-à-point sur les tuples d'entiers naturels est un bel ordre. Le théorème 3.2.11 nous dit donc que l'ordre *sous-mot* sur les mots de tuples d'entiers naturels est lui aussi un bel ordre. Il découle ensuite directement du lemme 3.2.10 que l'ordre de plongement $\preceq$ est un bel ordre dans notre cas.

## 3.3 Gardes universelles

En regardant de plus près la syntaxe de Cubicle et les exemples de la section 2.2, on remarque que les gardes des transitions peuvent contenir des quantificateurs universels. Ces conditions, dites *globales*, sont souvent utilisées pour modéliser des actions atomiques dépendant des valeurs de certaines variables de tous les autres processus du système. On appelle aussi la partie quantifiée universellement de la garde, la *garde universelle*. On les trouve pratiquement dans tous les modèles réalistes de protocoles de communication de la littérature et en particulier dans les exemples traités par Cubicle.

La difficulté introduite par la présence de tels quantificateurs dans les gardes des transitions concerne le calcul des pré-images. La pré-image d'un cube par une transition avec garde universelle n'est malheureusement pas un cube. La classe des formules considérées précédemment n'est alors plus close par calcul de pré-image, l'algorithme 4 ne s'applique donc plus. Même si l'introduction de quantificateurs universels dans les gardes n'est pas permise dans le cadre strict de la théorie des systèmes à tableaux, on montre qu'il est possible d'autoriser une forme limitée de quantification au prix d'une perte de complétude de l'analyse de sûreté.

### 3.3.1 Travaux connexes

Dans la littérature, le problème est généralement en partie résolu en considérant une *abstraction* du système de transition dans laquelle ces quantificateurs universels n'apparaissent plus.

Abdulla *et al.* s'intéressent à des systèmes paramétrés dans lesquels les gardes des transitions sont exprimées à l'aide de contraintes propositionnelles ou différentielles (de la forme $x - y < c$ où $c$ est une constante entière). Ils proposent une transformation appelée *abstraction monotone* dans laquelle les gardes universelles sont remplacées par un opérateur spécial qui supprime les processus ne respectant pas la condition globale [3].

Les auteurs de MCMT utilisent une transformation très proche de celle proposée dans [3] mais plus syntaxique car elle consiste seulement en une modification du système de transition original qui reste dans le même fragment [9]. Leur transformation consiste à créer une version du système avec pannes et à *relativiser les quantificateurs* du problème au processus qui ne sont pas en panne.

L'approche adoptée dans Cubicle est plus légère car on définit seulement un calcul de pré-image *sur-approximé* qui préserve la forme (de cube) des formules. Contrairement à [9] on n'introduit aucune transition et aucun symbole de fonction supplémentaires. En revanche notre approche n'est pas plus précise. Elle est même équivalente car on montre qu'un système est prouvé sûr après la transformation effectuée par MCMT si et seulement si il est prouvé sûr par Cubicle avec calcul de pré-image approximé.

### 3.3.2 Calcul de pré-image approximé

Étant donné un système à tableaux $\mathcal{S} = (Q, I, \tau)$, on rappelle la définition de la pré-image d'une formule $\varphi(X')$ par la relation de transition $\tau$

$$\text{PRE}_\tau(\varphi)(X) = \exists X'.\ \tau(X, X') \wedge \varphi(X')$$

Soit $t$ une transition de $\tau$ de la forme $t \equiv \exists \bar{i}.\ \gamma(\bar{i}, Q) \wedge \bigwedge_{x \in Q} x' = \lambda \bar{j}.\ \delta_x(\bar{i}, \bar{j}, Q)$. La garde de $t$ est la formule $\gamma(\bar{i}, Q)$. Pour ajouter des gardes universelles au langage, on suppose sans perte de généralité que $\gamma(\bar{i}, Q)$ est de la forme $\gamma_L(\bar{i}, Q) \wedge \forall \bar{j}.\ \gamma_G(\bar{j}, \bar{i}, Q)$ où $\gamma_L$ (resp. $\gamma_G$) est une formule sans quantificateurs appelée la garde *locale* (resp. garde *universelle* ou *globale*). Dans ce cas la pré-image par la transition $t$ d'un cube $\varphi$ est équivalente à une formule de la forme $\exists\forall$ :

$$\text{PRE}_t(\varphi) = \exists \bar{i}.\ \forall \bar{j}.\ \phi(\bar{i}, \bar{j})$$

où $\phi$ est sans quantificateurs.

Pour une formule quantifiée universellement $\Psi = \forall \bar{j}.\ \psi(\bar{j})$, on note la conjonction des instances de $\phi$ sur l'ensemble $\bar{i}$ :

$$\mathit{Inst}(\Psi, \bar{i}) = \bigwedge_{\sigma \in \Sigma(\bar{j}, \bar{i})} \psi(\bar{j})\sigma$$

où $\Sigma(\bar{j}, \bar{i})$ est l'ensemble des substitutions de $\bar{j}$ vers $\bar{i}$. En particulier on a que $\Psi \models \mathit{Inst}(\Psi, \bar{i})$.

On définit la pré-image *approximée* par la transition $t$ d'un cube $\varphi$, notée par $\widetilde{\text{PRE}}_t(\varphi)$ :

$$\widetilde{\text{PRE}}_t(\varphi)(Q) = \exists \bar{i}.\ \mathit{Inst}(\phi(\bar{i}, \bar{j}), \bar{i})$$

**Proposition 3.3.1.** *Si $\varphi$ est un cube et $\tau$ est un ensemble de transitions avec gardes universelles, alors la formule $\widetilde{\text{PRE}}_\tau(\varphi)$ est équivalente à une disjonction de cubes.*

*Démonstration.* On sait grâce à la proposition 3.1.1 du chapitre 3 que si $\varphi$ est un cube, alors $\widetilde{\text{PRE}_t}(\varphi)(Q)$ est équivalente à une disjonction de cubes.

□

**Lemme 3.3.2.** *Soit $\varphi$ un cube et $\tau$ est un ensemble de transitions avec gardes universelles, $\text{PRE}_\tau(\varphi) \models \widetilde{\text{PRE}_\tau}(\varphi)$.*

*Remarque.* En particulier, si $\tau$ ne contient pas de transitions avec gardes universelles alors $\widetilde{\text{PRE}_\tau}(\varphi) = \text{PRE}_\tau(\varphi)$.

**Théorème 3.3.3.** *Soit un système à tableaux $\mathcal{S} = (Q, I, \tau)$ avec gardes universelles et $\Theta$ une formule dangereuse sous forme de cube. Si $\text{PRE}^*_\tau(\Theta) \wedge I$ est satisfiable alors $\widetilde{\text{PRE}^*_\tau}(\Theta) \wedge I$ est satisfiable.*

*Démonstration.* Découle directement de la proposition 3.3.1 et du lemme 3.3.2. □

Ceci veut dire que si on remplace la fonction $\text{PRE}_\tau$ du calcul de pré-image de l'algorithme 4 par la fonction $\widetilde{\text{PRE}_\tau}$, on obtient un algorithme correct pour les systèmes à tableaux avec gardes universelles. Sous les mêmes conditions et pour les mêmes raisons que celles exposées dans le chapitre 3, cet algorithme termine cependant il n'est plus complet. En effet à cause de la sur-approximation faite lors du calcul de la pré-image, il est possible que l'algorithme expose une trace d'erreur fallacieuse. Une fausse alarme peut être déclenchée sur un système qui est en réalité sûr.

### 3.3.3 Exemples

L'exemple suivant donnée dans le langage de Cubicle (Figure 3.6) est un petit système très simple conçu pour mettre en évidence pourquoi l'analyse d'atteignabilité approchée (avec la fonction $\widetilde{\text{PRE}_\tau}$) peut lever une fausse alarme. La transition `t2` a une garde universelle qui demande que le tableau `X` contienne la valeur `B` pour tous les processus si on veut changer une case en `C`. La transition `t1` demande quant à elle qu'il existe deux processus différents pour lesquels `X` vaut `A` afin d'en passer une en `B`. Il est clair que lorsqu'une case de `X` vaut `B` alors au moins une autre case vaut `A`. Il existe un invariant du système qui est :

$$\forall i.\ \exists j.\ i \neq j \wedge (\texttt{X}[i] = \texttt{B} \implies \texttt{X}[j] = \texttt{A})$$

Malheureusement, cet invariant n'est pas exprimable (même en version négative) dans le langage de la théorie des tableaux. Seules les formules sous forme de cube $\exists i.\ \Theta(\bar{i})$ sont manipulables par l'algorithme d'atteignabilité, et on aurait besoin ici de traiter des formules de la forme $\exists i.\ \forall j.\ \Theta(\bar{i}\bar{j})$. On peut voir sur la partie droite de la figure 3.6 la trace mise en évidence par Cubicle sur ce système. Sur cette trace, il faut au moins deux processus pour atteindre l'état initial (en analyse arrière) alors que le calcul de pré-image par `t2` a

```
type t = A | B | C
array X[proc] : t

init (i) { X[i] = A }

unsafe (i) { X[i] = C }

transition t1 (i j)
requires { X[i] = A && X[j] = A }
{ X[i] := B }

transition t2 (i)
requires { X[i] = B &&
           forall_other j. X[j] = B }
{ X[i] := C }
```

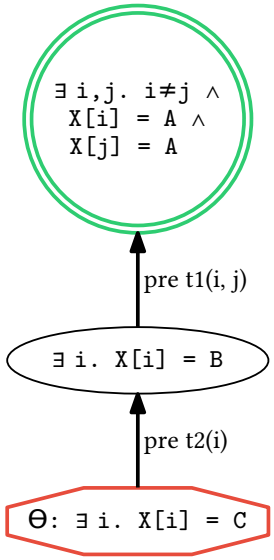



FIGURE 3.6 – Trace fallacieuse pour un système avec gardes universelles

approximé le résultat en considérant qu'il était possible que le système ne contienne qu'un seul processus pour prendre cette transition. On peut bien sûr rejouer la trace en fixant le nombre de processus du système pour se rendre compte qu'elle est fallacieuse. Dans ce cas Cubicle affichera le message suivant à l'utilisateur :

```
Spurious trace: t1(#1, #2) -> t2(#1) -> unsafe[1]
```

### 3.3.4 Relation avec le modèle de panne franche et la relativisation des quantificateurs

Alberti *et al.* montrent dans [9] que leur transformation syntaxique pour éliminer les gardes universelles revient à ajouter un modèle de panne franche au système considéré. Un problème récurrent traité dans la littérature sur les systèmes distribués consiste à concevoir et vérifier des systèmes *tolérant aux pannes* [159]. En effet, pour être utile, un système distribué doit pouvoir continuer de fonctionner même si certains de ces composants tombent en panne, ne répondent plus ou encore adoptent un comportement imprévu. On distingue les pannes selon le degré de gravité de la défaillance :

— *Panne franche* : le composant en panne cesse de fonctionner immédiatement et de façon permanente. Il n'envoie et ne reçoit plus aucun message.

— *Panne transitoire* ou *par omission* : un composant peut omettre d'envoyer un message (*omission d'envoi*) et/ou peut ignorer les messages qui lui sont destinés (*omission*

*générale*). Le système adopte ce comportement de façon temporaire et un composant peut reprendre un fonctionnement normal.

— *Panne temporelle* : le temps de réponse d'un composant peut dépasser ses spécifications.

— *Panne byzantine* : c'est la panne la plus dangereuse car un composant peut dévier de son comportement prévu de manière arbitraire et peut effectuer n'importe quelle action.

La transformation proposée dans [9] consiste à ajouter le comportement de panne franche au système pour se débarrasser des gardes universelles. Comme la panne franche est permanente, seuls les processus qui ne sont pas en panne peuvent participer aux transitions du système. Dans le cadre de la théorie des tableaux, cela revient à relativiser les quantificateurs des formules manipulées aux processus en fonctionnement (voir par exemple [92]). L'astuce pour éliminer les gardes universelles consiste à sur approximer les comportements du système en faisant « tomber en panne » tous les processus ne respectant pas la condition globale de la transition.

En pratique, on transforme un système à tableaux $\mathcal{S} = (Q, I, \tau)$ et une formule dangereuse $\Theta$ en un système $\widehat{\mathcal{S}} = (\widehat{Q}, \widehat{I}, \widehat{\tau})$ avec la formule dangereuse $\widehat{\Theta}$ comme décrit en figure 3.7.

| $\mathcal{S}$ | $\widehat{\mathcal{S}}$ |
|---|---|
| $Q$ | $\widehat{Q} \equiv Q \cup \{\texttt{Crash} : \texttt{proc} \mapsto \{\text{true}, \text{false}\}\}$ |
| $I \equiv \forall i.\ \mathit{Init}(i)$ | $\widehat{I} \equiv \forall i.\ \mathit{Init}(i) \wedge \texttt{Crash}[i] = \text{false}$ |
| $t \equiv \exists \bar{i}.\ \gamma_L(\bar{i}) \wedge \forall j.\ \gamma_G(\bar{i}, j) \wedge \bigwedge_{x \in Q} x' = \lambda \bar{j}.\ \delta_x(\bar{i}, \bar{j})$ | $\widehat{t} \equiv \exists \bar{i}.\ \bigwedge_{i \in \bar{i}} \texttt{Crash}[i] = \text{false} \wedge \gamma_L(\bar{i}) \wedge \bigwedge_{x \in Q} x' = \lambda \bar{j}.\ \delta_x(\bar{i}, \bar{j}) \wedge \texttt{Crash}' = \lambda j.\ \mathit{ite}(\gamma_G(\bar{i}, j), \texttt{Crash}[j], \text{true})$ |
| $\tau \equiv \bigvee t$ | $\widehat{\tau} \equiv \widehat{\tau_0} \vee t_{crash}$ avec $\widehat{\tau_0} \equiv \bigvee \widehat{t}$ et $t_{crash} \equiv \exists i.\ \bigwedge_{x \in Q} x' = x \wedge \texttt{Crash}' = \lambda j.\ \mathit{ite}(j = i, \text{true}, \texttt{Crash}[j])$ |
| $\Theta \equiv \exists \bar{i}.\ U(\bar{i})$ | $\widehat{\Theta} \equiv \exists \bar{i}.\ \bigwedge_{i \in \bar{i}} \texttt{Crash}[i] = \text{false} \wedge U(\bar{i})$ |

FIGURE 3.7 – Transformation avec modèle de panne franche

La correction de l'analyse d'atteignabilité arrière est préservée par la transformation précédente. On a donc le théorème suivant :

**Théorème 3.3.4** ([9, Proposition 5.1 et Théorème 5.1]). *Soit $\mathcal{S}$ un système à tableaux avec gardes universelles. Si $\widehat{\mathcal{S}}$ est sûr par rapport à $\widehat{\Theta}$, alors $\mathcal{S}$ est sûr par rapport à $\Theta$.*

```
type t = A | B | C
array X[proc] : t
array Crash[proc] : bool

init (i) { X[i] = A && Crash[i] = False }

unsafe (i) { X[i] = C && Crash[i] = False }

transition crash (n)
{ Crash[n] := True }

transition t1 (i j)
requires { X[i] = A && X[j] = A &&
   Crash[i] = False && Crash[j] = False }
{ X[i] := B }

transition t2 (i)
requires { X[i] = B && Crash[i] = False }
{
  X[i] := C;
  Crash[j] := case
              | X[j] <> B : True
              | _ : Crash[j];
}
```

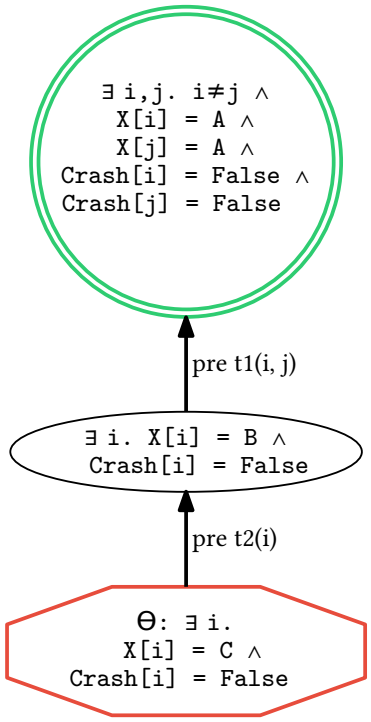



FIGURE 3.8 – Trace fallacieuse pour la transformation avec modèle de panne franche

L'exemple de la figure 3.6 est transformé en rajoutant le modèle de panne franche dans la figure 3.8. On peut voir qu'on obtient la même trace d'erreur avec ce modèle. Ce n'est pas par hasard, car comme suggéré précédemment les deux approches sont équivalentes.

**Lemme 3.3.5.** *Soit $\mathcal{S}$ un système à tableaux avec gardes universelles. Si il existe une trace d'erreur faisant intervenir la transition $t_{crash}$ alors il existe une trace d'erreur qui ne fait pas intervenir la transition $t_{crash}$. Autrement dit,*

$$PRE_{\widehat{\tau}}(\widehat{\Theta}) \wedge \widehat{I} \text{ satisfiable} \implies PRE_{\widehat{\tau_0}}(\widehat{\Theta}) \wedge \widehat{I} \text{ satisfiable}$$

*Démonstration.* Il suffit de montrer que seuls les processus $i$ tels que $\texttt{Crash}[i] = \mathsf{false}$ participent à la trace d'erreur. □

Dans la suite on note $\widetilde{\texttt{BWD}}$ l'algorithme d'atteignabilité arrière approximé BWD dans lequel la fonction Pre est remplacée par la fonction $\widetilde{\text{Pre}}$.

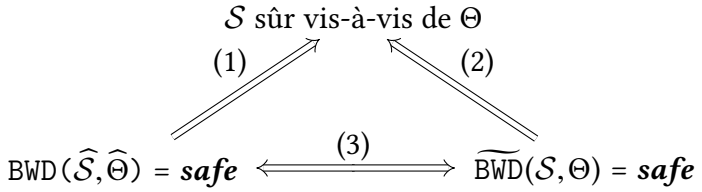



Figure 3.9 – Relations entre les différentes approches pour les gardes universelles

L'implication (1) de la figure 3.9 correspond au théorème 3.3.4 et l'implication (2) correspond au théorème 3.3.3. On a en réalité l'équivalence (3) qui correspond à la proposition suivante.

**Proposition 3.3.6.** *Soit $\mathcal{S}$ un système à tableaux avec gardes universelles et $\Theta$ un cube. $\widehat{\mathcal{S}}$ est sûr par rapport à $\widehat{\Theta}$ ssi $\widetilde{\texttt{BWD}}(\mathcal{S},\Theta)$ = **safe**.*

La preuve fait correspondre les formules obtenues par le calcul de pré-image approximé $\widetilde{\text{Pre}}_\tau$ et celles obtenues par la fonction $\text{Pre}_{\widehat{\tau_0}}$ du système sans transition sans `crash` grâce au lemme 3.3.5.

L'approche adoptée dans Cubicle correspond donc essentiellement à l'ajout du modèle de panne franche sans toutefois demander de modifier le système de départ en complexifiant les gardes et actions.

## 3.4 Conclusion

### 3.4.1 Exemples sans existence d'un bel ordre

Après avoir pris connaissance des caractéristiques et résultats de la théorie des systèmes à tableaux exposés dans cette section on est en droit de se demander ce qui nous empêcherait de trouver un bel ordre pour la classe de systèmes utilisée lors de l'encodage de la machine de Minsky.

Rappelons qu'on a utilisé comme théorie des éléments une théorie des types énumérés à deux constructeurs $\{0,1\}$. La théorie des processus est sur signature $\Sigma_P = (\{\texttt{proc}\}, \{d : \texttt{proc}\}, \{=, \lhd : \texttt{proc} \times \texttt{proc}\})$ où $d$ est un symbole de constante et $\lhd$ est un symbole de relation injectif. Les modèles de $\mathcal{T}_P$ doivent en plus respecter le fait que $\neg\exists i.\ i \lhd d$.

$\mathcal{T}_E$ et $\mathcal{T}_P$ sont toutes les deux localement finies et sur symboles disjoints. Pour montrer qu'on a pas de bel ordre il suffit de se donner un seul symbole de fonction $r$ des processus vers les éléments $\{0,1\}$. Une représentation graphique en figure 3.10 montre qu'on peut construire une suite infinie de configurations incomparables. On note par $ⓐ \longrightarrow ⓑ$ le fait que la configuration ait deux constantes de processus $p_1$ et $p_2$ telles que $p_1 \lhd p_2$ et que $r(p_1) = a$ et $r(p_2) = b$. Si un processus $p$ est représenté par $\bigcirc$ dans ce graphique, on dénote par $\bigcirc^d$ le fait que $d$ doit interprété par $p$ dans la configuration.

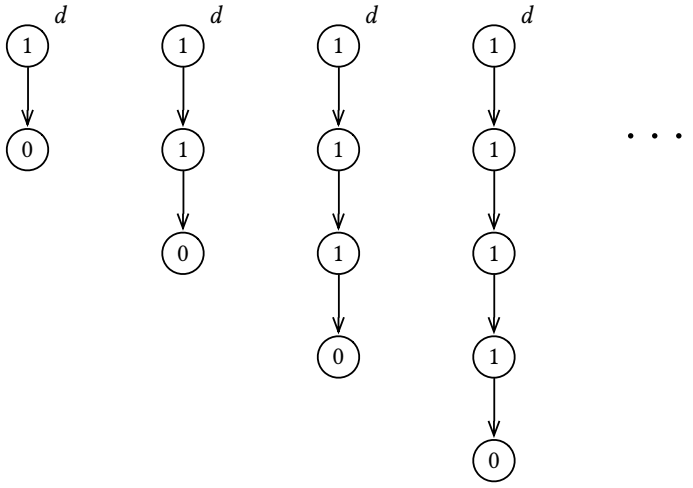



FIGURE 3.10 – Séquence infinie de configurations non comparables pour l'encodage des machines de Minksy

Un autre cas intéressant est celui où on a à notre disposition la seule théorie des processus munie d'une relation binaire quelconque, $\Sigma_A = \Sigma_P = (\{\texttt{proc}\}, \emptyset, \{=, R : \texttt{proc} \times \texttt{proc}\})$. Cette théorie est relationnelle donc clairement localement finie. On peut construire un ensemble infini de configurations incomparables si par exemple $R$ est utilisée pour représenter une topologie en anneau des processus. Cette fois on note par $\bigcirc \longrightarrow \bigcirc$ deux éléments distincts $p_1$ et $p_2$ de la configuration et tels que $(p_1, p_2) \in R$. La figure 3.11 montre une séquence infinie de configurations pour lesquelles on ne peut trouver de plongement.

*Remarque.* On peut aussi se ramener au cas de l'exemple précédent en enlevant le symbole de relation binaire $R$ de $\Sigma_P$ mais en autorisant cette fois un symbole de fonction d'arité deux ayant pour domaine les éléments d'un type énuméré à deux constructeurs. De cette façon on peut triviallement encoder la relation binaire avec ce nouveau symbole de fonction.

Si on relâche la restriction qui force les théories $\mathcal{T}_P$ et $\mathcal{T}_E$ à être sur symboles disjoints, on peut encoder cette même relation binaire avec un symbole de fonction unaire représentant un tableau indicé par les éléments de $\mathcal{T}_P$ et à valeur dans $\mathcal{T}_P$.

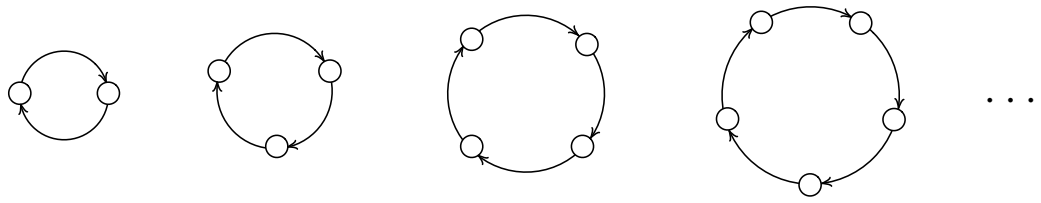


FIGURE 3.11 – Séquence infinie de configurations non comparables pour une relation binaire quelconque

### 3.4.2 Résumé

Nous avons exposé dans cette section le cadre théorique sur lequel est fondé Cubicle, à savoir celui des systèmes à tableaux. Une technique pour vérifier des propriétés de sûreté de tels systèmes est le *model checking modulo théories* conçu par Ghilardi et Ranise. Elle repose sur une boucle d'atteignabilité par chaînage arrière durant laquelle les tests logiques sont déchargés par un solveur SMT. L'algorithme est effectif lorsqu'il est possible de calculer de manière effective la pré-image de formules manipulées par le système et lorsque les tests de sûreté et de point fixe sont décidables. Pour ceci, on a besoin d'un système à tableau qu'il respecte les conditions suivantes :

- $\mathcal{T}_P$ est localement finie et close par sous-structures
- le problème $SMT(\mathcal{T}_P)$ est décidable
- le problème $SMT(\mathcal{T}_E)$ est décidable
- la formule décrivant les états initiaux du système est quantifiée universellement, *i.e.* est de la forme $\forall \bar{i}.\ \varphi(\bar{i})$
- la relation de transition du système $\tau$ est décrite par un ensemble de transitions $\tau = t_1 \vee t_2 \vee \ldots \vee t_n$ de la forme

$$t_k(Q, Q') = \exists \bar{i}.\ \gamma(\bar{i}, Q) \wedge \bigwedge_{x \in Q} \forall \bar{j}. x'(\bar{j}) = \delta_x(\bar{i}, \bar{j}, Q)$$

- la formule caractérisant les états mauvais du système est un cube (ou est équivalente à une disjonction de cubes), c'est-à-dire est de la forme

$$\exists \bar{i}.\ \Delta(\bar{i}) \wedge l_1(\bar{i}) \wedge l_2(\bar{i}) \wedge \ldots \wedge l_n(\bar{i})$$

À partir du moment où l'ordre de plongement sur les configurations (les états) du système est un *bel ordre*, on sait que l'algorithme d'atteignabilité arrière termine. La sûreté d'un système à tableaux est donc décidable sous ces conditions. Une technique pour autoriser une forme de quantification universelles dans les gardes, au prix de la complétude de l'approche, consiste à utiliser un calcul de pré-image approximé.

### 3.4.3 Discussion

Ce cadre théorique est très puissant dans le sens où il fournit un langage d'une grande expressivité englobant un sous-ensemble de la logique du premier ordre et permettant de décrire nombre de systèmes paramétrés. Un second point crucial de ce cadre est qu'il existe un algorithme effectif pour vérifier des propriétés de ces systèmes. En tirant parti des capacités de raisonnement des solveurs SMT modernes, il peut se décharger d'une partie de son travail au travers d'outils annexes en perpétuelle évolution. Les théories supportées par ces solveurs sont directement accessibles au model checker et à l'utilisateur (sous certaines conditions).

Bien que ce cadre général soit assez libéral, les restrictions imposées sur les théories pour garantir la terminaison sont tout de même limitantes en pratique. Par exemple la condition de non intersection des signatures des théories des éléments et des processus nous empêche formellement d'écrire un tableau indicé par des identificateurs de processus et contentant lui-même des identificateurs de processus. Ce cas arrive en pratique si on veut modéliser une variable locale à un processus ou un canal de communication dans lequel on enregistre le destinataire d'un éventuel message. L'exemple suivant sort du cadre des systèmes à tableaux :

```
type msg = M1 | M2 | M3

array Channel_msg[proc] : msg
array Channel_dest[proc] : proc
...

transition recieve (from to)
requires { Channel_dest[from] = to && Channel_msg[from] = M1 }
{
    (* Effectuer action correspondant à la réception de M1
       et accuser réception au processus from *)
}
```

Une autre limitation assez handicapante en pratique est la condition qui force $\mathcal{T}_E$ à être localement finie. Si on utilise la théorie des entiers naturels munis de l'addition de signature $\Sigma_E = \{=, +, 0, 1, 2, 3, \ldots\}$ on sort également du cadre des systèmes à tableaux et on ne pourra pas par exemple modéliser l'incrémentation d'un compteur.

```
var Counter : int
array Flag[proc] : bool
...

transition incr (i)
requires { Flag[i] = False }
{
  Flag[i] := True;
  Counter := Counter + 1;
}
```

Même si dans certains cas la terminaison de l'atteignabilité arrière est garantie à coup sûr, rien n'assure que le model checker n'aura pas besoin de « l'âge de l'univers » pour décider de la sûreté d'un système. Au contraire, il existe des exemples pour lesquels aucune garantie n'est faite quand à la décidabilité d'une propriété mais pour lesquels Cubicle atteint un point fixe global et termine quand même. Si on se trouve dans un cas défavorable, parce que l'algorithme ne termine pas ou bien parce que la complexité de l'analyse est trop grande, l'utilisateur ne pourra pas faire la différence. Pour toutes ces raisons, dans Cubicle on n'interdit pas l'utilisation de combinaisons particulières de théories du moment que la syntaxe les autorise. On résume dans le tableau en figure 3.12 les restrictions qui sont forcées (✓) par la théorie du model checking modulo théories et par Cubicle. Par exemple la théorie $\mathcal{T}_P$ est locallement finie dans Cubicle donc on ne fournit pas d'opération d'addition + sur les identificateurs de processus.

| **Restriction** | **Théorie** | **Cubicle** |
|---|---|---|
| $\mathcal{T}_P$ locallement finie | ✓ | ✓ |
| $\mathcal{T}_P$ close par sous-structures | ✓ | |
| $SMT(\mathcal{T}_P)$, $SMT(\mathcal{T}_E)$ décidables | ✓ | ✓ |
| $\Sigma_P \cap \Sigma_E = \emptyset$ | ✓ | |
| $I \equiv \forall \bar{i}.\ \varphi(\bar{i})$ | ✓ | ✓ |
| $\Theta \equiv$ disjonction de cubes | ✓ | ✓ |
| Gardes $\forall$ interdites | ✓ | |

FIGURE 3.12 – Différences de restrictions (✓) entre la théorie du model checking modulo théories et Cubicle

Les algorithmes 4 et 5 sont en réalité assez naïfs et bien trop inefficaces pour être utilisés sur des problèmes non triviaux. Le chapitre suivant détaille une implémentation effective *en pratique* de ce cadre théorique dans Cubicle.

# 4

# Optimisations et implémentation

## Sommaire


**4.1 Architecture . . . . . 80**
**4.2 Optimisations . . . . . 82**
4.2.1 Appels au solveur SMT . . . . . 82
4.2.2 Tests ensemblistes . . . . . 85
4.2.3 Instantiation efficace . . . . . 88
**4.3 Suppressions a posteriori . . . . . 91**
**4.4 Sous-typage . . . . . 96**
**4.5 Exploration parallèle . . . . . 99**
**4.6 Résultats et conclusion . . . . . 102**


Il est tout à fait possible de prendre tels quels les algorithmes présentés dans le chapitre 3 pour obtenir un model checker pour systèmes paramétrés correct. Cependant, un tel outil n'a que peu de chances de fonctionner en pratique. Dans cette partie, on décrit en détail l'architecture du model checker Cubicle. On justifie les choix qui ont été faits pour cette implémentation et on explicite également les optimisations les plus cruciales.

Plusieurs composants de l'algorithme peuvent faire l'objet d'une attention particulière. On redonne ci-dessous l'algorithme 4 dans son intégralité en précisant quelles parties section de ce chapitre traite plus spécifiquement.

**Algorithme 4 :** Analyse d'atteignabilité par chaînage arrière

**Entrées** : un système paramétré $\mathcal{S} = (Q, I, \tau)$ et un cube $\Theta$
**Variables** :
$\mathcal{V}$ : cubes visités
$\mathcal{Q}$ : file de travail

```
1  function BWD(S, Θ) : begin
2      V := ∅;
3      push(Q, Θ);
4      while not_empty(Q) do
5          φ := pop(Q);
6          if φ ∧ I satisfiable then
7              return unsafe
8          else if φ ⊭ V then
9              V := V ∪ {φ};
10             push(Q, Pre_τ(φ));
11     return safe
```

Section 4.4
Section 4.5
Section 4.2.1
Sections 4.2.2 et 4.2.3
Section 4.3

Les optimisations traitant des tests de satisfiabilité sont abordées dans la section 4.2. On développe en section 4.4 une analyse statique simple de sous-typage qui peut apporter des informations supplémentaires en début d'algorithme. L'ajout des nœuds dans l'ensemble $\mathcal{V}$ est potentiellement source de simplifications expliquées en section 4.3. On donne en outre une technique de parallélisation de la boucle d'atteignabilité (Section 4.5). Enfin, les expérimentations exposées en fin de chapitre confirment que les optimisations présentées ici permettent à Cubicle d'être compétitif avec les model checkers pour systèmes paramétrés de l'état de l'art. Une partie des travaux présentés dans ce chapitre ont étés publiés dans [43] et [45].

## 4.1 Architecture

La figure 4.1 présente l'architecture du model checker Cubicle sous forme de graphe de dépendance entre modules. Les modules en pointillés représentent des interfaces ou modules abstraits fournissant une signature. Les flèches pointillées dénotent les modules implémentant les signatures requises par ces interfaces.

Le module BWD contient l'algorithme d'atteignabilité (algorithme 4) de la section 3.1.1 du chapitre précédent. Il est paramétré par une structure de file à priorité PriorityQueue dont la signature est donnée ci-contre.

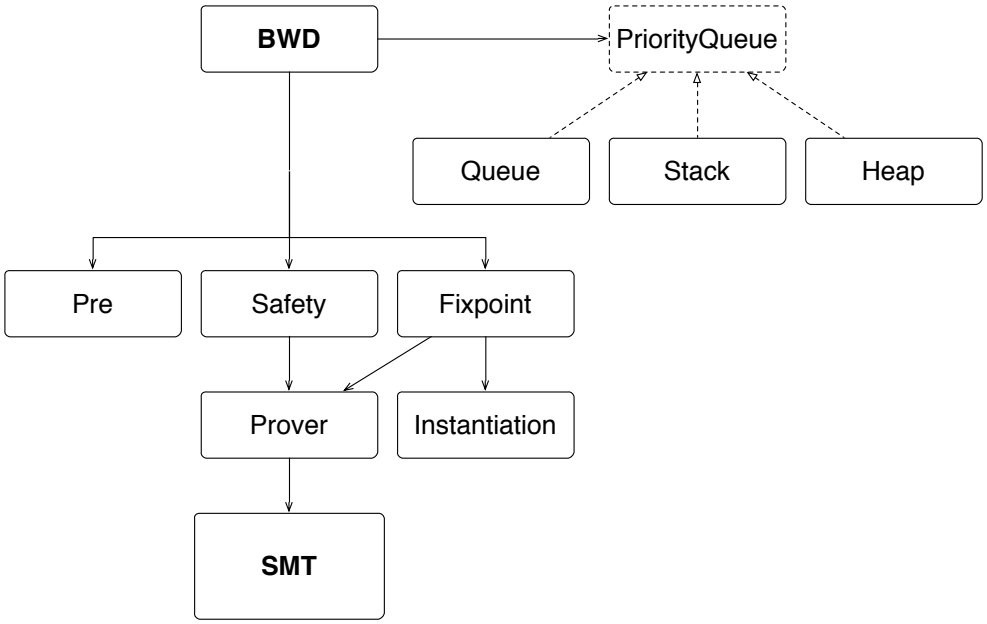



FIGURE 4.1 – Architecture de Cubicle

*priorityqueue.mli*

```
module type PriorityNodeQueue = sig
    type t
    val create : unit -> t
    val pop : t -> Node.t
    val push : Node.t -> t -> unit
    val push_list : Node.t list -> t -> unit
    val clear : t -> unit
    val length : t -> int
    val is_empty : t -> bool
end
```

Les structures de données qui implémentent cette signature doivent manipuler des nœuds (cubes riches) de type `Node.t` et définissent la stratégie d'exploration du graphe d'atteignabilité arrière. Par exemple une structure de file générera une exploration en largeur alors qu'une structure de pile générera une exploration en profondeur. Bien évidemment la taille du graphe d'atteignabilité dépend de la stratégie d'exploration utilisée. L'expérience montre qu'en pratique, une exploration en largeur (ou une variante) est souvent plus efficace qu'une exploration en profondeur pour les systèmes sûrs.

L'algorithme de BWD utilise entre autres un calcul de pré-image sur transitions para-

métrées (dans le module Pre), des tests de sûreté (module Safety) et des tests de points fixes (module Fixpoint). L'instantiation des quantificateurs, dont une variante naïve est présentée dans l'algorithme 5 (voir page 56), est réalisée de manière efficace par le module Instantiation. Enfin les tests logiques sont déchargés par un solveur SMT dont l'interface avec le model checker est faite au travers du module Prover.

## 4.2 Optimisations

Pour comprendre pourquoi il n'est pas suffisant d'implémenter l'algorithme 4 directement, on illustre l'efficacité d'une version naïve de Cubicle en figure 4.2. Les temps donnés ont été obtenus sur une machine 64 bits avec un processeur quadri-cœurs Intel® Xeon® cadencé à 3,2 GHz et comportant 24 Go de mémoire. On peut clairement voir sur les graphiques que la majorité du temps est passée dans le solveur SMT. Dans la suite de ce chapitre, on s'intéressera plus particulièrement aux exemples des deux dernières lignes du tableau. Le premier (German-*esque*++), assez simple mais non trivial, est une version évoluée du petit protocole de cohérence de cache présenté en section 2.2.4. Le second est un ordre de grandeur plus complexe. C'est une version du protocole de cohérence de cache proposé par Steven German avec plusieurs canaux de communication et plus de messages dont la formalisation peut être trouvée par exemple dans [16].

On peut voir que dans la majorité des exemples, le solveur SMT occupe la plus grande partie de la procédure de model checking. Même pour les exemples les plus simples, la taille des problèmes envoyé au solveur grandit et ce dernier se retrouve submergé.

### 4.2.1 Appels au solveur SMT

Quel est le problème avec le solveur SMT ? Pour répondre à cette question, intéressons-nous au test de point fixe effectué en ligne 8 de l'algorithme 4 :

$$\varphi \not\models \mathcal{V}$$

On a vu qu'il était possible de décider ces tests logiques de satisfiabilité avec l'algorithme 5 naïf et en s'aidant d'un solveur SMT. Rappelons que $\varphi$ est un cube et que $\mathcal{V}$ est une disjonction de cubes, le test de point fixe s'écrit :

$$\exists \bar{i}.\ \Delta \wedge F \not\models \bigvee_{0 \leq k \leq |\mathcal{V}|} \exists \overline{j_k}.\ \Delta(\overline{j_k}) \wedge F_k$$

Il suffit de vérifier la non satisfiabilité de la formule suivante :

$$\exists \bar{i}.\ \Delta \wedge F \wedge \bigwedge_{0 \leq k \leq |\mathcal{V}|} \forall \overline{j_k}.\ \neg\Delta(\overline{j_k}) \vee \neg F_k$$

| Benchmark | Statistiques | Répartition du temps |
|---|---|---|
| Bakery | **temps** : 0,012 s<br>**nœuds** : 2<br>**points fixes** : 7<br>**appels SMT** : 2 | |
| Dijkstra | **temps** : 1 m 30 s<br>**nœuds** : 77<br>**points fixes** : 1349<br>**appels SMT** : 2 | |
| Java Meta Lock | **temps** : 0,10 s<br>**nœuds** : 22<br>**points fixes** : 105<br>**appels SMT** : 49 | |
| Szymanski atomique (Talupur) | **temps** : 0,068 s<br>**nœuds** : 7<br>**points fixes** : 80<br>**appels SMT** : 14 | |
| Szymanski atomique | **timeout :** 2 h<br>**nœuds** : 775 (10187 restants)<br>**points fixes** : 4281<br>**appels SMT** : 779 | |
| German-*esque* | **temps** : 0,056 s<br>**nœuds** : 19<br>**points fixes** : 26<br>**appels SMT** : 21 | |
| German-*esque*+ | **temps** : 1,2 s<br>**nœuds** : 50<br>**points fixes** : 255<br>**appels SMT** : 51 | |
| German-*esque*++ | **temps** : 8 m 35 s<br>**nœuds** : 322<br>**points fixes** : 2087<br>**appels SMT** : 332 | |
| German (Baukus) | **timeout :** 2 h<br>**nœuds** : 2314 (16183 restants)<br>**points fixes** : 13209<br>**appels SMT** : 2321 | |

**Légende** :

SMT
Construction des formules
Substitutions
Pré-image
Autres

FIGURE 4.2 – Benchmarks et statistiques pour une implémentation naïve

Si on se place dans le cas simple où $\mathcal{T}_P$ est relationnelle alors il est possible d'éliminer les quantificateurs existentiels $\exists \bar{i}$ en introduisant des variables de Skolem $\overline{\#_i}$, et pour chaque $\overline{j_k}$, l'ensemble $\Sigma_k$ des substitutions de $\overline{j_k}$ vers les $\overline{\#_i}$ constitue une instantiation exhaustive des quantificateurs universels $\forall \overline{j_k}$. Dans ces conditions, la formule suivante est équisatisfiable :

$$\Delta(\overline{\#_i}) \wedge F(\overline{\#_i}) \wedge \bigwedge_{0 \leq k \leq |\mathcal{V}|} \bigwedge_{\sigma \in \Sigma_k} \neg(\Delta(\overline{j_k}))\sigma \vee \neg(F_k)\sigma$$

Supposons maintenant que l'algorithme ait visité 10 000 nœuds [1], et qu'en moyenne chaque cube de $\mathcal{V}$ soit quantifié par cinq variables distinctes, autrement dit $|\mathcal{V}| = 10000$ et $|\Sigma_k| = 120$. La conjonction $\bigwedge_{0 \leq k \leq |\mathcal{V}|} \bigwedge_{\sigma \in \Sigma_k} \ldots$ est dans ce cas constituée d'environ $1.10^6$ clauses. Chaque test de point fixe demande donc au solveur SMT de traiter des millions de clauses, et ce à chaque tour de boucle.

En raison du très grand nombre d'appels faits au solveur, son utilisation doit être la plus efficace possible. La plupart des solveurs SMT modernes sont construits avec des briques de bases (solveur propositionnel SAT, solveurs de théories, procédure de combinaison, *etc.*) incrémentales. Cette incrémentalité se retrouve dans l'interface du solveur et est donc directement exploitable par l'utilisateur final. C'est le cas du solveur SMT utilisé par Cubicle qui est une version allégée d'Alt-Ergo [42]. À chaque appel du solveur SMT dans Cubicle, le contexte est construit incrémentalement et sa cohérence est vérifié à chaque fois qu'une nouvelle clause est ajoutée.

L'interface du solveur permet de créer des instances qui ont la signature donnée par le code OCaml suivant :

***smt.mli***

```
exception Unsat of int list

module type Solver = sig
  ...
  val clear : unit -> unit
  val assume : id:int -> Formula.t -> unit
  val check : unit -> unit
end
```

L'interface est impérative. On efface le contexte à l'aide de la fonction `clear`, et on ajoute une formule au contexte avec la fonction `assume`. Enfin la fonction `check` peut être appelée à tout moment pour s'assurer que le contexte ainsi construit ne contient pas d'incohérences. Dans le cas contraire l'exception `Unsat [`$i_1$`; ...; `$i_n$`]` est levée, où $i_1, \ldots, i_n$ dénote le

1. C'est un nombre tout à fait raisonnable, en témoigne l'exemple German donné dans la figure 4.2.

*unsat core* composé des formules qui ont été ajoutées au contexte avec les identifiants correspondant. Un noyau d'insatisfiabilité ou *unsat core* d'un ensemble de formules dont la conjonction n'est pas satisfiable, est un sous ensemble dont la conjonction des éléments reste insatisfiable.

*Remarque.* La fonction `assume` peut aussi lever l'exception `Unsat` car elle fait la propagation propositionnelle des faits unitaires dans le solveur SAT.

Une utilisation traditionnelle du solveur SMT est montrée dans l'exemple ci-dessous. On ajoute au contexte le but à prouver dans son intégralité.

```
try
  SMT.clear ();
  SMT.assume ~id:0 ( Δ(#̄_i) ∧ F(#̄_i) ∧ ⋀_{0≤k≤|V|} ⋀_{σ∈Σ_k} ¬(Δ(j̄_k))σ ∨ ¬(F_k)σ );
  SMT.check ();
  false
with Unsat _ -> true
```

En ajoutant les clauses une à une et en vérifiant la cohérence du contexte à chaque étape, on s'assure que l'insatisfiabilité sera détectée dès que possible.

```
try
  SMT.clear ();
  SMT.assume ~id:0 (Δ(#̄_i) ∧ F(#̄_i));
  SMT.check ();
  SMT.assume ~id:1 (¬(Δ(j̄_1))σ_1 ∨ ¬(F_1)σ_1);
  SMT.check ();
  ...
  SMT.assume ~id:n (¬(Δ(j̄_k))σ_n ∨ ¬(F_k)σ_n);
  SMT.check ();
  false
with Unsat _ -> true
```

*Remarque.* Il est aussi important d'avoir accès au solveur SMT au travers d'une API (Application Programming Interface, ou interface de programmation) pour ne pas envoyer les buts sous forme de chaînes de caractères, ou encore pire, devoir écrire des fichiers sur le disque.

### 4.2.2 Tests ensemblistes

Comme les appels au solveur sont coûteux, il devient intéressant d'essayer d'en limiter au maximum le nombre. En réalité, beaucoup des tests de points fixes peuvent être déchargés

de manière quasi syntaxique.

Soit le cube traité par l'algorithme $\varphi \equiv \exists i, j.\ i \neq j \wedge \mathtt{A}[i] = 1 \wedge \mathtt{A}[j] = 2 \wedge \mathtt{B}[j] = 0$. Le point fixe $\varphi \models \mathcal{V}$ est trivial si une des composantes de $\mathcal{V}$ est le cube $\exists i, j.\ i \neq j \wedge \mathtt{A}[i] = 1 \wedge \mathtt{B}[j] = 0$. En effet, après Skolemisation et une instantiation, il devient évident que

$$\#_1 \neq \#_2 \wedge \mathtt{A}[\#_1] = 1 \wedge \mathtt{A}[\#_2] = 2 \wedge \mathtt{B}[\#_2] = 0$$

$$\implies \quad \#_1 \neq \#_2 \wedge \mathtt{A}[\#_1] = 1 \wedge \mathtt{B}[\#_2] = 0$$

Pour tout cube $\varphi$, on note par $(\!|\varphi|\!)$ l'ensemble des littéraux de la formule obtenue par la Skolemisation de $\varphi$. Pour le cube $\varphi$ du paragraphe précédent, on a $(\!|\varphi|\!) = \{\#_1 \neq \#_2,\ \mathtt{A}[\#_1] = 1,\ \mathtt{A}[\#_2] = 2,\ \mathtt{B}[\#_2] = 0\}$. L'ensemble $(\!|\varphi|\!)$ représente le cube de manière unique modulo équisatisfiabilité.

**Proposition 4.2.1.** *Pour tous cubes $\varphi$ et $\psi$, si $(\!|\psi|\!) \subseteq (\!|\varphi|\!)$ alors $\varphi \implies \psi$.*

*Démonstration.*

$$\begin{aligned} \text{Soient} \quad & \varphi \equiv \exists i_1, \ldots, i_k.\ \Delta(i_1, \ldots, i_k) \wedge l_1 \wedge \ldots \wedge l_n \\ \text{et} \quad & \psi \equiv \exists j_1, \ldots, j_l.\ \Delta(j_1, \ldots, j_l) \wedge l'_1 \wedge \ldots \wedge l'_m \end{aligned}$$

$(\!|\psi|\!) = (\!|\Delta(\#_1, \ldots, \#_l)|\!) \cup \{l'_1, \ldots, l'_m\}$ et $(\!|\varphi|\!) = (\!|\Delta(\#_1, \ldots, \#_k)|\!) \cup \{l_1, \ldots, l_n\}$. On suppose $(\!|\psi|\!) \subseteq (\!|\varphi|\!)$ donc $l \leq k$, $m \leq n$, car $(\!|\Delta(\#_1, \ldots, \#_k)|\!) \subseteq (\!|\Delta(\#_1, \ldots, \#_l)|\!)$ et $\{l'_1, \ldots, l'_m\} \subseteq \{l_1, \ldots, l_n\}$. À renommage près, on suppose que $l'_1 = l_1, \ldots$ et $l'_m = l_m$. Soit $\mathcal{M}$ un modèle de $\varphi$. En particulier, $\mathcal{M} \models l_1, \ldots$ et $\mathcal{M} \models l_m$, donc $\mathcal{M} \models l'_1 \wedge \ldots \wedge l'_m$. Similairement, $\mathcal{M} \models \Delta(\#_1, \ldots, \#_l)$, donc $\mathcal{M} \models \psi$.

□

Pour réaliser ces tests ensemblistes de manière rapide on représente l'ensemble $\mathcal{V}$ des nœuds visités sous forme d'un *arbre préfixe* (ou *trie*). Cette structure de donnée permet de partager les préfixes commun des cubes, étant donné un ordre sur les littéraux.

**Exemple.** Soient les cubes $c_1 = \exists \bar{i_1}.\ a \wedge b \wedge c \wedge d$, $c_2 = \exists \bar{i_2}.\ a \wedge b \wedge e$, $c_3 = \exists \bar{i_3}.\ b \wedge e$, et $c_4 = \exists \bar{i_4}.\ b \wedge e \wedge x$. On définit l'ordre total $<$ sur les littéraux tel que $a < b < c < d < e < x$. L'ensemble $\mathcal{V}$ dénotant la disjonction $c_1 \vee c_2 \vee c_3 \vee c_4$ est représenté par la structure d'arbre préfixe en figure 4.3.

Maintenant pour savoir s'il existe un sous ensemble de $(\!|\varphi|\!)$ dans $\mathcal{V}$, il suffit de parcourir l'arbre en largeur [17, 148].

*Remarque.* Il est aussi possible d'implémenter une représentation compacte de l'ensemble $\mathcal{V}$ avec un diagramme de décision binaire (BDD) [27] dans lequel les variables sont les littéraux. La taille du BDD obtenu dépend de l'ordre choisit sur les littéraux. Elle peut être exponentielle si on choisit mal cet ordre.

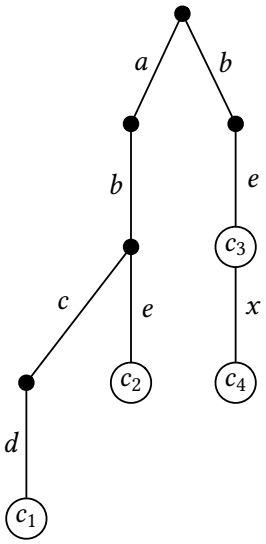



FIGURE 4.3 – Arbre préfixe représentant $\mathcal{V}$

| Benchmark | Statistiques | Répartition du temps |
|---|---|---|
| German-*esque*++ | **temps** : 25,4 s<br>**nœuds** : 322<br>**points fixes** : 2087<br>**appels SMT** : 281993 | |
| German (Baukus) | **timeout :** 2 h | — |

**Légende** :

SMT
Construction des formules
Substitutions
Tests ensemblistes
Pré-image
Autres

FIGURE 4.4 – Benchmarks pour tests ensemblistes

La figure 4.4 montre que l'utilisation de tests ensemblistes dans le model checker permet de se passer d'une bonne partie des appels au solveur SMT. On gagne ainsi presque un ordre de grandeur sur German-*esque*++ en passant de 2,6 millions à 280 000 appels au solveur. Cette optimisation fonctionne en pratique car les formules envoyées au solveur SMT ne sont pas anodines. Ce sont des cubes obtenus par calcul successifs de pré-images et les similarités entre ces formules sont grandes.

### 4.2.3 Instantiation efficace

Il est tout à fait concevable d'utiliser les mécanismes d'instantiation de quantificateurs internes aux solveurs SMT pour décharger les tests de points fixes. Traditionnellement, ces solveurs utilisent des *déclencheurs* pour filtrer les formules closes du contexte et obtenir des substitutions des variables quantifiées vers des termes clos. Ce filtrage est réalisé modulo la théorie de l'égalité libre [125] et chaque tour d'instantiation peut générer des termes de plus en plus gros et des instances de plus en plus nombreuses. Bien que ce mécanisme soit contrôlé par des stratégies limitant les instances possibles et des heuristiques favorisant certains termes pour le filtrage, cette technique d'instantiation reste incomplète et souvent peu efficace. Elle peut toutefois donner des résultats intéressants sur des problèmes dont le contexte (souvent composé d'axiomes avec quantificateurs universels) n'est pas trop important. C'est le cas par exemple des obligations de preuves venant de la vérification déductive de programmes. En revanche, les problèmes qui constituent nos tests de points fixes contiennent souvent plusieurs milliers de formules quantifiées ce qui en fait des candidats difficiles même pour les solveurs les plus modernes.

Il existe des théories axiomatiques pour lesquelles il est possible de calculer les déclencheurs des axiomes de façon à rendre l'instantiation par filtrage complète [60]. La preuve de complétude de la théorie devant être effectuée à chaque nouvelle théorie, il n'est pas possible de calculer automatiquement les déclencheurs pour chaque point fixe dans notre cas. Certains solveurs SMT comme Z3 utilisent une forme d'instantiation à partir de modèles [73] en complément des techniques classiques à déclencheurs. Cette approche permet de rendre le solveur plus complet mais n'améliore pas ses performances sur les exemples où le filtrage est suffisant.

Le principal problème rencontré dans Cubicle est en réalité l'efficacité des procédures d'instantiation. En effet, la restriction imposée sur les théories supportées par le model checker et la forme particulière des tests logiques nous permet de connaître toutes les instances à faire, grâce à l'algorithme 5 par exemple. Comme montré précédemment, c'est leur nombre qui devient rapidement rédhibitoire. On explique ici comment restreindre ce nombre de manière dramatique pour n'envoyer au solveur que les instances qui lui seront potentiellement utiles.

Considérons une nouvelle fois le test de point fixe. Il est vrai si la formule suivante est

insatisfiable :

$$\Delta(\overline{\#_i}) \wedge F(\overline{\#_i}) \wedge \bigwedge_{0 \leq k \leq |\mathcal{V}|} \bigwedge_{\sigma \in \Sigma_k} \neg(\Delta(\overline{j_k})\sigma \wedge (F_k)\sigma)$$

Pour réaliser ce test le plus efficacement possible il faut trouver quel est l'ensemble des instances des formules de $\mathcal{V}$ qui contredit $\Delta(\overline{\#_i}) \wedge F(\overline{\#_i})$. En général si on sait que pour deux formules $F$ et $G$, $F \wedge G$ est insatisifiable, alors la formule $F \wedge \neg G$ est satisfiable ssi $F$ est satisfiable. Intuitivement, $G$ n'apporte rien pour montrer l'insatisfiabilité de $F \wedge \neg G \wedge \ldots$. Dans notre cas, l'instance $\sigma : \{\overline{j_k}\} \mapsto \{\overline{\#_i}\}$ n'est pas utile si $\Delta(\overline{\#_i}) \wedge F(\overline{\#_i}) \wedge \neg(\Delta(\overline{j_k})\sigma \wedge (F_k)\sigma)$ est insatisfiable.

$F$ et $F_k$ sont des conjonctions de littéraux $F = l_1 \wedge \ldots \wedge l_n$, $F_k = k_1 \wedge \ldots \wedge k_m$. Sans perte de généralité on peut supposer que $F$ et $F_k$ sont chacune cohérentes de leur côté. Il suffit alors que deux littéraux $l_i \wedge k_j$ soient insatisfiables pour rendre $\Delta \wedge F \wedge \Delta \wedge F_k$ insatisfiables. De plus, lorsque deux littéraux $l_i \wedge k_j$ sont insatisfiables, un test syntaxique permet souvent de détecter cette insatisfiabilité. Par exemple, si $l_i \equiv x(\#_1) = 1$ et que $k_j \equiv x(\#_1) \neq 1$, syntaxiquement $l_i = \neg k_j$. Dans d'autres cas, un raisonnement simple modulo théorie est aussi possible et très peu coûteux. Par exemple, si $l_i \equiv x(\#_1) = 1$ et que $k_j \equiv x(\#_1) = 2$, comme la théorie de l'arithmétique nous dit que $1 \neq 2$ on peut trivialement déduire par simple transitivité de l'égalité que $l_i \wedge k_j$ est insatisfiable.

Ces premiers tests simples sur les littéraux des cubes instantiés nous permettent d'en éliminer une bonne partie. On doit tout de même, pour chaque cube de $\mathcal{V}$, construire toutes ses instances en appliquant les substitutions correspondantes. Cette opération est coûteuse car créer une instance demande d'allouer de la mémoire. En réalité, il est possible de faire ces tests avant même de construire les instances. Pour chaque cube $\exists\overline{j_k}.\ \Delta(\overline{j_k}) \wedge F_k$, on peut à moindre coût calculer l'ensemble $\Sigma'_k \subseteq \Sigma_k$ des substitutions correspondant aux instances potentiellement utiles pour le test de satisfiabilité qui nous intéresse. Pour cela, on calcule la substitution (partielle) *obligatoire*, et l'ensemble des substitutions *impossibles*.

On note $Obv_k$ la permutation obligatoire. S'il existe une variable globale $x$ telle que le littéral $x = \#_i$ apparaît dans $F$ et $x = j$ dans $F_k$ alors $(j \mapsto \#_i) \in Obv_k$.

On note $Impos_k$ l'ensemble des substitutions impossibles. Soit un symbole $A$ d'arité non nulle $p$, une constante $C$ (constructeur d'un type énuméré, constante entière, …) et une variable globale $x$. Le tableau suivant donne les conditions pour que $\{j'_1 \mapsto \#'_1, \ldots, j'_p \mapsto \#'_p\} \in Impos_k$.

| **littéral apparaît dans $F_k$** | **littéral apparaît dans $F$** | **conditions** |
|---|---|---|
| $A(j'_1,\ldots,j'_p) = t$ | $A(\#'_1,\ldots,\#'_p) \neq t$<br>ou $A(\#'_1,\ldots,\#'_p) < t$<br>ou $A(\#'_1,\ldots,\#'_p) > t$ | quelque soit le terme $t$ |
| $A(j'_1,\ldots,j'_p) \neq t$<br>ou $A(j'_1,\ldots,j'_p) < t$<br>ou $A(j'_1,\ldots,j'_p) > t$ | $A(\#'_1,\ldots,\#'_p) = t$ | quelque soit le terme $t$ |
| $A(j'_1,\ldots,j'_p) = C_1$ | $A(\#'_1,\ldots,\#'_p) = C_2$ | $C_1 \neq C_2$<br>et $C_1$ et $C_2$ sont des constantes (constructeurs d'un type énuméré, constantes entières, …) |

On définit ensuite l'ensemble des substitutions intéressantes :

$$\Sigma'_k = \left\{ \sigma \in \Sigma_k \;\middle|\; \begin{array}{l} \forall j \mapsto \# \in Obv_k.\ j \mapsto \#' \in Obv_k \implies \# = \#' \\ Obv_k \subseteq \sigma \\ \forall \sigma_I \in Impos_k.\ \sigma_I \not\subseteq Obv_k \\ \forall \sigma_I \in Impos_k.\ \sigma_I \not\subseteq \sigma \end{array} \right\}$$

*Remarque.* On détecte aussi les incohérences triviales entre les littéraux qui n'ont aucune variable quantifiée. Si pour un cube de $\mathcal{V}$, $\exists\overline{j_k}.\ \Delta(\overline{j_k}) \wedge F_k$ avec $\overline{j_k}$ non vide, alors lorsque $\Sigma'_k$ est vide cela signifie qu'aucune instance de ce cube n'est intéressante. Ce cube ne sera donc même pas considéré dans le test de point fixe.

L'ensemble des substitutions intéressantes $\Sigma'_k$ est calculé par la fonction `relevant` du module Instantiation de la figure 4.1. La satisfiabilité de la formule du test de point fixe est équivalente à celle-ci :

$$\Delta(\overline{\#_i}) \wedge F(\overline{\#_i}) \wedge \bigwedge_{0\leq k\leq|\mathcal{V}|} \bigwedge_{\sigma\in\Sigma'_k} \neg(\Delta(\overline{j_k})\sigma \wedge (F_k)\sigma)$$

De cette façon on envoie au solveur SMT seulement les instances des cubes de $\mathcal{V}$ qui seront potentiellement utiles pour décharger le test de point fixe. On applique également un raisonnement similaire pour découvrir immédiatement parmi ces instances, s'il en existe une qui est trivialement impliquée par le cube considéré.

*Remarque.* Une technique d'instantiation similaire est utilisée dans MCMT [76]. Les auteurs appellent cette technique un « filtrage modulo types énumérés ». D'autres heuristiques sont également mises en œuvre dans MCMT mais pas par Cubicle. Par exemple leur test de

point-fixe est fait de manière *chronologique*, *i.e.* les cubes de $\mathcal{V}$ ajoutés le plus récemment sont utilisés en priorité [76]. Dans nos expériences, on a remarqués qu'un tri fondé sur une notion de *distance entre cubes* donnaient de manière générale de meilleurs résultats. Ces heuristiques restent toutefois anecdotiques car leur utilité est négligeable face aux techniques de réduction du nombre d'instances.

| Benchmark | Statistiques | Répartition du temps |
|---|---|---|
| German-*esque*++ | **temps** : 0,46 s<br>**nœuds** : 322<br>**points fixes** : 2087<br>**appels SMT** : 5457 | |
| German (Baukus) | **temps** : 13 m 34 s<br>**nœuds** : 16287<br>**points fixes** : 320769<br>**appels SMT** : 4047581 | |

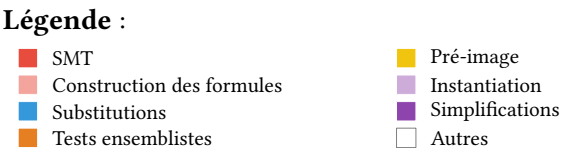



FIGURE 4.5 – Benchmarks pour l'instantiation efficace

La figure 4.5 reprend les expériences de la section précédente auxquelles on a ajouté cette nouvelle optimisation. On peut remarquer notamment que le nombre d'appels au solveur à été divisé par 50 sur l'exemple German-*esque*++ (5 457 appels contre 281 993 auparavant). On gagne au passage deux ordres de grandeur sur le temps de vérification de cet exemple et Cubicle est désormais capable de prouver la sûreté du protocole de cohérence de cache de German. Ces résultats montrent à quel point ce traitement particulier des instantiations est important pour l'efficacité du model checker. Si on voulait utiliser les mécanismes de gestion des quantificateurs internes aux solveurs SMT, il faudrait que ces derniers soient capables d'émuler ce type de raisonnement.

## 4.3 Suppressions a posteriori

Les optimisations présentées jusqu'ici ont principalement trait au test de point fixe. Les expériences montrent en effet que c'est un des points déterminants pour le bon fonctionnement du model checker. Un autre aspect sensible de Cubicle est sa stratégie d'exploration des nœuds du graphe d'atteignabilité arrière.

En comparant les graphes d'atteignabilité générés par Cubicle sur l'exemple Dijkstra pour différentes stratégies d'explorations (Figure 4.6), on peut constater qu'une exploration en profondeur (Figure 4.6(a)) considère 7 fois plus de nœuds et nécessite 1500 fois plus de temps qu'une exploration purement en largeur (Figure 4.6(b)). Ici, la stratégie la plus efficace est d'utiliser une forme d'exploration en largeur avec un calcul de priorité pour les cubes. On peut notamment remarquer sur la figure 4.6(c) que la branche la plus à gauche est explorée en profondeur alors que le reste est exploré en largeur.

Malheureusement cette dernière stratégie n'est pas forcément la bonne dans tous les cas. Pour remédier en partie à ce problème, on propose dans cette section une optimisation qui consiste en quelque sorte à réparer les mauvais choix faits par l'algorithme d'exploration. L'idéal est d'explorer les nœuds les plus généraux en premier, c'est-à-dire ceux qui représentent les ensembles d'états les plus grands. Par exemple, le cube $\varphi \equiv \exists i.\ x(i) = 1$ est plus général que $\psi \equiv \exists i.\ x(i) = 1 \wedge y = 2$ car $\psi$ implique $\varphi$. Si $\varphi$ est exploré après $\psi$, alors on peut en fait supprimer $\psi$ et le sous arbre enraciné en $\psi$ (dont une partie $\mathcal{V}_s$ est dans $\mathcal{V}$ et l'autre $\mathcal{Q}_s$ dans $\mathcal{Q}$). On montre cette opération en figure 4.7. On reprend ici le code couleur de la figure 4.6 pour l'ordre d'exploration, et on représente en bleu les nœuds qui se trouvent encore dans la file $\mathcal{Q}$.

**Proposition 4.3.1.** *L'opération de suppression* a posteriori *est correcte et ne change pas la valeur de retour de la fonction* `BWD` *de l'argorithme 4.*

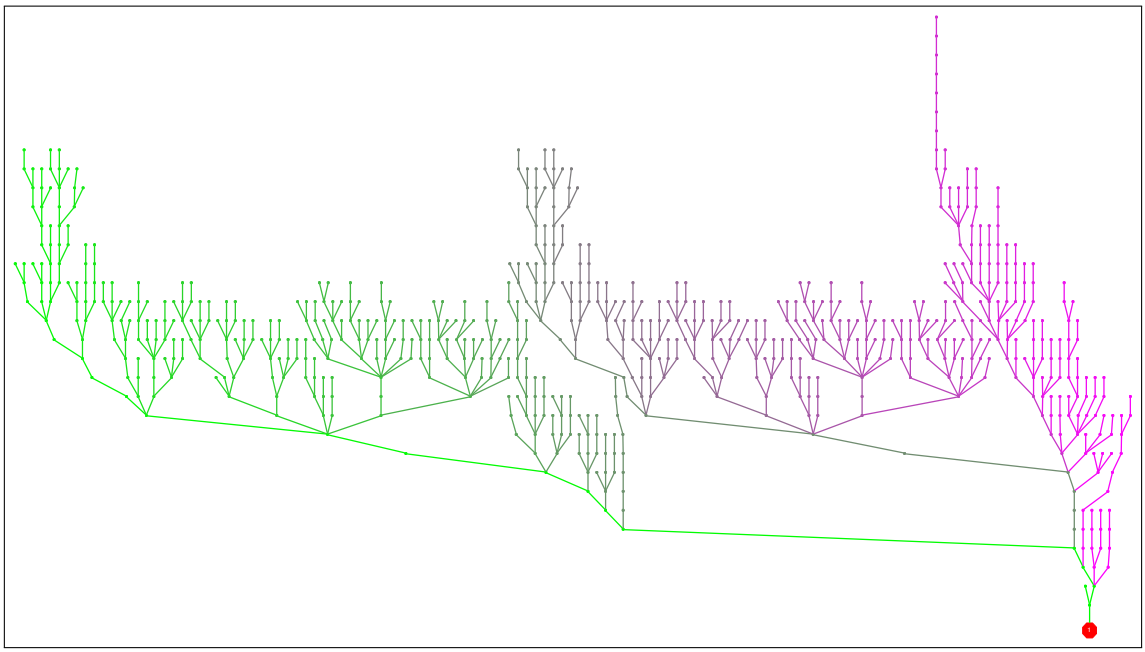
*Démonstration.* On va montrer que les invariants de boucle de la fonction `BWD` sont préservés par cette optimisation. On suppose qu'il existe un cube $\varphi$ dans $\mathcal{Q}$, et un cube $\psi$ de $\mathcal{V}$ tels que $\psi \models \varphi$. On suppose également que l'invariant (3.3) est vrai avant la suppression, *i.e.* $\textsc{Pre}^*_\tau(\Theta) \models \mathcal{V} \vee \textsc{Pre}^*_\tau(\mathcal{Q}\backslash\mathcal{V})$. On note $\mathcal{V}_s$ (resp. $\mathcal{Q}_s$) la partie de $\mathcal{V}$ (resp. $\mathcal{Q}$) obtenue par calculs de pré-image successifs à partir de $\psi$ (*cf.* Figure 4.8(a)). On souhaite montrer que $\textsc{Pre}^*_\tau(\Theta) \models \mathcal{V}' \vee \textsc{Pre}^*_\tau(\mathcal{Q}'\backslash\mathcal{V}')$ où $\mathcal{V}' = \mathcal{V}\backslash\mathcal{V}_s\backslash\psi$ et $\mathcal{Q}' = \mathcal{Q}\backslash\mathcal{Q}_s$.

On peut résumer les hypothèses par la liste suivante :

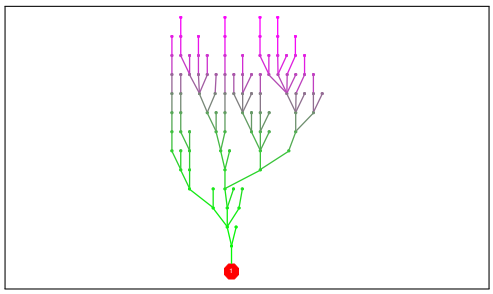
$$\textsc{Pre}^*_\tau(\Theta) \models \mathcal{V} \vee \textsc{Pre}^*_\tau(\mathcal{Q}\backslash\mathcal{V}) \tag{4.1}$$
$$\mathcal{V} = \mathcal{V}' \vee \mathcal{V}_s \vee \psi \tag{4.2}$$
$$\mathcal{Q} = \mathcal{Q}' \vee \mathcal{Q}_s \text{ et } \varphi \models \mathcal{Q}', \varphi \not\models \mathcal{Q}_s \tag{4.3}$$
$$\psi \models \varphi \tag{4.4}$$
$$\mathcal{V}_s \models \textsc{Pre}^*_\tau(\psi) \models \textsc{Pre}^*_\tau(\varphi) \tag{4.5}$$
$$\mathcal{Q}_s \models \textsc{Pre}^*_\tau(\psi) \models \textsc{Pre}^*_\tau(\varphi) \tag{4.6}$$

On a, grâce à 4.3,

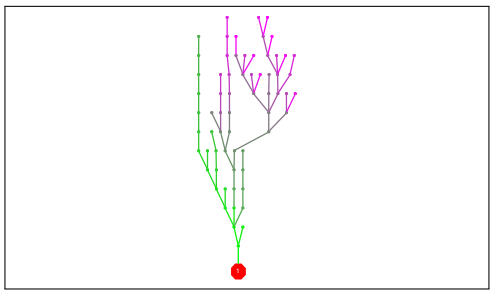
$$\textsc{Pre}^*_\tau(\mathcal{Q}\backslash\mathcal{V}) = \textsc{Pre}^*_\tau((\mathcal{Q}' \vee \mathcal{Q}_s)\backslash\mathcal{V}) = \textsc{Pre}^*_\tau(\mathcal{Q}'\backslash\mathcal{V}) \vee \textsc{Pre}^*_\tau(\mathcal{Q}_s\backslash\mathcal{V}) \tag{4.7}$$



(a) Exploration en profondeur (DFS) pure (3 m 22 s, 724 nœuds)



(b) Exploration en largeur (BFS) pure (0,34 s, 98 nœuds)



(c) Exploration en largeur avec priorité (0,13 s, 66 nœuds)

FIGURE 4.6 – Graphes d'atteignabilité arrière pour différentes stratégies d'exploration sur l'exemple Dijkstra. L'octogone rouge représente la formule *dangereuse* du système. Chaque nœud correspond à une formule de $\mathcal{V}$ et les arrêtes relient un nœud avec ses pré-images. Les nœuds et arrêtes coloriés en vert sont explorés en premier alors que ceux coloriés en magenta le sont en dernier.

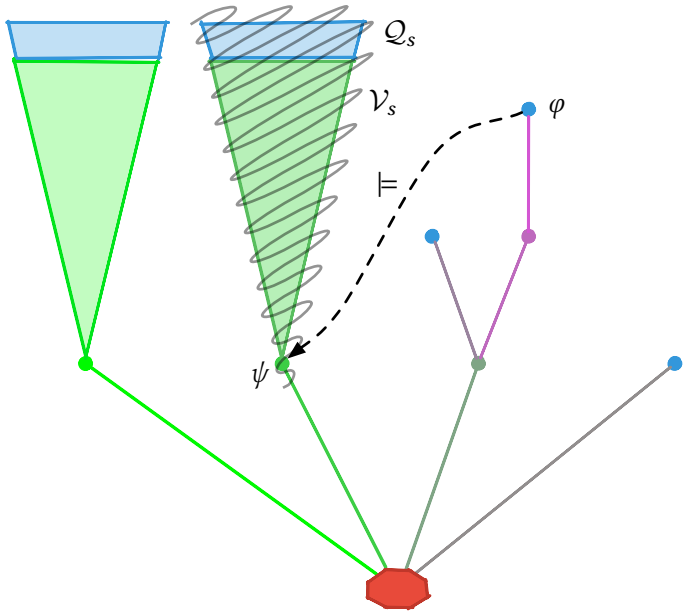



FIGURE 4.7 – Suppression *a posteriori*

Comme $\mathcal{Q}_s \backslash \mathcal{V} \models \mathcal{Q}_s$, alors $\text{PRE}^*_\tau(\mathcal{Q}_s \backslash \mathcal{V}) \models \text{PRE}^*_\tau(\psi)$ par (4.6).
D'autre part, $\mathcal{Q}' \backslash \mathcal{V} = \mathcal{Q}' \backslash (\mathcal{V}' \vee (\mathcal{V}_s \vee \psi)) = (\mathcal{Q}' \backslash \mathcal{V}') \wedge (\mathcal{Q}' \backslash (\mathcal{V}_s \vee \psi))$. Clairement $\mathcal{Q}' \backslash \mathcal{V} \models \mathcal{Q}' \backslash \mathcal{V}'$ donc $\text{PRE}^*_\tau(\mathcal{Q}' \backslash \mathcal{V}) \models \text{PRE}^*_\tau(\mathcal{Q}' \backslash \mathcal{V}')$. On peut alors réécrire (4.7) en

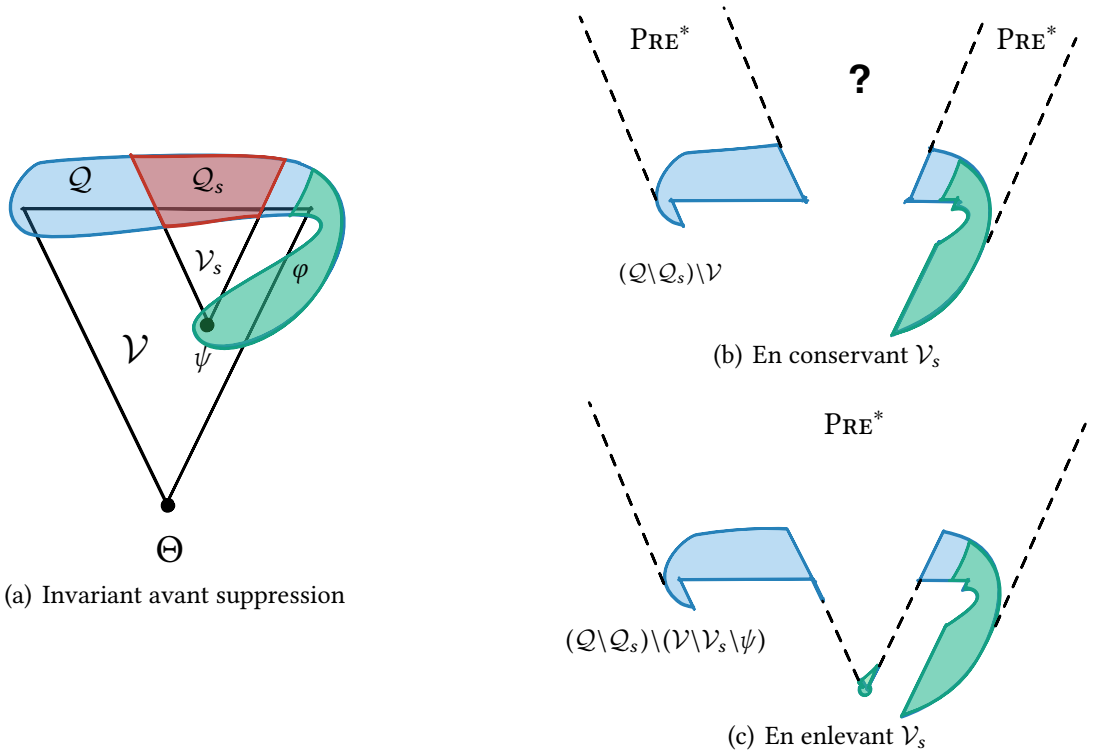
$$\begin{aligned}
& \text{PRE}^*_\tau(\mathcal{Q} \backslash \mathcal{V}) \models \text{PRE}^*_\tau(\mathcal{Q}' \backslash \mathcal{V}') \vee \text{PRE}^*_\tau(\psi) \\
\text{avec (4.1) et (4.2),} \quad & \text{PRE}^*_\tau(\Theta) \models \mathcal{V}' \vee (\mathcal{V}_s \vee \psi) \vee \text{PRE}^*_\tau(\mathcal{Q}' \backslash \mathcal{V}') \vee \text{PRE}^*_\tau(\psi) \\
\text{avec (4.4) et (4.5),} \quad & \text{PRE}^*_\tau(\Theta) \models \mathcal{V}' \vee \text{PRE}^*_\tau(\mathcal{Q}' \backslash \mathcal{V}') \vee \text{PRE}^*_\tau(\psi) \\
\text{comme } \mathcal{V}' = \mathcal{V} \backslash \mathcal{V}_s \backslash \psi, \quad & \text{PRE}^*_\tau(\Theta) \models \mathcal{V}' \vee \text{PRE}^*_\tau((\mathcal{Q}' \vee \psi) \backslash \mathcal{V}') \\
\text{et } \psi \models \varphi \models \mathcal{Q}' \text{ donc,} \quad & \text{PRE}^*_\tau(\Theta) \models \mathcal{V}' \vee \text{PRE}^*_\tau(\mathcal{Q}' \backslash \mathcal{V}')
\end{aligned}$$

La préservation de l'invariant (3.2) est triviale car $\mathcal{V}' \models \mathcal{V}$.

□

On voit sur la figure 4.8(c) qu'il faut bien supprimer tous les nœuds de $\mathcal{V}_s$ (obtenus par pré-images de $\psi$) sinon on risque de se retrouver dans la configuration figure 4.8(b) et risquer de ne pas explorer une partie de l'espace.

Lorsque $\varphi \models \mathcal{V}$, cette optimisation ne sert à rien, on ne la fait que dans le cas où le test de point-fixe n'est pas valide. En plus, pour que l'hypothèse (4.3) $\varphi \not\models \mathcal{Q}_s$ soit vérifiée, on fait attention à ne pas supprimer un nœud $\psi$ qui serait un *ancêtre* (dans le graphe d'atteignabilité arrière) de $\varphi$. En pratique on ne fait jamais appel au solveur SMT pour chercher les nœuds subsumés mais on préfère seulement utiliser les tests ensemblistes de la section 4.2.2.




(a) Invariant avant suppression

(b) En conservant $\mathcal{V}_s$

(c) En enlevant $\mathcal{V}_s$

FIGURE 4.8 – Préservation de l'invariant après suppression d'un nœud

| Benchmark | Statistiques | Répartition du temps |
|---|---|---|
| German-*esque*++ | **temps** : **0,44 s**<br>**nœuds** : 314<br>**points fixes** : 2052<br>**appels SMT** : 5138<br>**nœuds supprimés** : **18** | |
| German (Baukus) | **temps** : **11 m 10 s**<br>**nœuds** : 13451<br>**points fixes** : 264136<br>**appels SMT** : 3240926<br>**nœuds supprimés** : **1587** | |

**Légende** :

- SMT
- Construction des formules
- Substitutions
- Tests ensemblistes
- Pré-image
- Instantiation
- Simplifications
- Autres

FIGURE 4.9 – Benchmarks pour la suppression *a posteriori*

*Remarque.* Cette technique est similaire à l'*élimination de redondances arrière* utilisée dans les prouveurs par résolution [76]. Les instances des formules éliminées par cette technique n'ont plus besoin d'être considérées pour les tests de points-fixes ou d'intersection par MCMT. Ici, on s'autorise même à ne plus considérer de pré-images des formules « redondantes » en coupant effectivement l'arbre de recherche.

Cette optimisation permet de réduire la taille de l'arbre en général mais seulement d'un facteur assez faible. On ne supprime par exemple que 10% des nœuds pour l'exemple German (Figure 4.9) mais ça reste une simplification intéressante car son application restreinte n'est quasiment jamais désavantageuse. Dans le pire des cas aucun nœud ne sera supprimé et les tests ensemblistes ne pénaliseront pas le reste de l'exploration.

## 4.4 Sous-typage

Le langage d'entrée de Cubicle contient des informations de type. Le système décrit est donc en partie contraint par les types de ses variables et tableaux, ce qui interdit certains comportements (ceux dont les variables contiennent des valeurs du mauvais type). L'inférence des types est triviale (voir Section A.2) car le langage est très simple mais on peut tout de même considérer cette phase de typage comme une première analyse statique du programme.

Dans cette section on propose une analyse de sous-typage des systèmes de transitions décrits dans le langage de Cubicle. À l'issue de cette analyse, on obtient un *raffinement* des types des variables et tableaux du système. Les informations ainsi obtenues sont doublement utiles. Elles permettent à la fois de réduire l'espace de recherche du model checker et d'améliorer les performances du solveur en lui évitant certaines analyses par cas inutiles.

Dans la suite on écrit $\tau_1 <: \tau_2$ lorsque $\tau_1$ est un sous-type de $\tau_2$ et $\{C_1 \mid C_2 \mid \ldots \mid C_n\}$ désigne le type énuméré ayant pour constructeurs $C_1, C_2, \ldots, C_n$. Prenons comme exemple le système suivant décrit dans la syntaxe de Cubicle : Dans la figure 4.10, on note par $\tau_{\mathtt{X}}$ le type de la variable `X` et $\tau_{\mathtt{Y}}$ le type de la variable `Y`. Sur la droite de la figure on annote le programme avec les contraintes de sous-typage découvertes par l'analyse. Les déclarations des variables aux lignes 2 et 3 nous disent que les types de `X` et `Y` contiennent au plus les constructeurs `A`, `B`, `C` et `D`. La formule décrivant les états initiaux du système ligne 5 force les types des deux variables à contenir au moins `A`. Enfin on s'intéresse seulement aux actions des transitions car c'est la seule façon de modifier la valeur des variables et tableaux. Dans ce cas, on apprend que le type de `X` contient au moins les constructeurs `B` et `A`, que le type de `Y` contient `D`. L'affectation `Y := X` de la transition `t2` ajoute quand à elle la contrainte que le type de `Y` est un *sur-type* du type de `X`.

Un simple calcul de plus petit point-fixe sur cet ensemble de contraintes nous permet de

```
1  type t = A | B | C | D
2  var X : t                          τX <: {A | B | C | D}
3  var Y : t                          τY <: {A | B | C | D}
4
5  init () { X = A && Y = A }         τX :> {A} ∧ τY :> {A}
6
7  transition t1 ()
8  requires { ... }
9  { X := B; Y := D }                 τX :> {B} ∧ τY :> {D}
10
11 transition t2 ()
12 requires { ... }
13 { X := A; Y := X }                 τX :> {A} ∧ τY :> τX
```

FIGURE 4.10 – Système Cubicle annoté avec les contraintes de sous-typage

trouver la solution :

$$\tau_X = \{A \mid B\} \quad \wedge \quad \tau_Y = \{A \mid B \mid D\} \tag{4.8}$$

Les nouvelles informations ainsi obtenues peuvent ensuite être injectées dans le model checker sous forme d'invariants. Dans le cas de la figure 4.10, la solution donnée en (4.8) nous garantit que la formule suivante est un invariant du système :

$$X \neq C \wedge X \neq D \wedge Y \neq C$$

Le solveur SMT utilisé par Cubicle supporte directement la déclaration de sous-types pour les types énumérés. On peut donc gratuitement injecter les informations obtenues par l'analyse dans le solveur.

Cette analyse n'est pas toujours payante. On peut remarquer en figure 4.11 qu'il n'y a aucune différence pour l'exemple German-*esque*++. C'est parce que les types donnés aux variables et tableaux sont déjà les plus précis possibles. En revanche, sur le deuxième exemple, l'analyse améliore les performances de manière conséquente. On explore maintenant sept fois moins de nœuds et on gagne deux ordres de grandeur sur le temps d'exécution. Dans le fichier Cubicle, les variables du systèmes sont déclarées comme :

```
type state = Invalid | Shared | Exclusive
type msg = Empty | Reqs | Reqe | Inv | Invack | Gnts | Gnte

var Exgntd : bool
var Curcmd : msg
var CurClient : proc
```

| Benchmark | Statistiques | Répartition du temps |
|---|---|---|
| German-*esque*++ | **temps** : **0,44 s**<br>**nœuds** : 314<br>**points fixes** : 2052<br>**appels SMT** : 5138<br>**nœuds supprimés** : 18 | |
| German (Baukus) | **temps** : **5,88 s**<br>**nœuds** : 2259<br>**points fixes** : 30828<br>**appels SMT** : 41716<br>**nœuds supprimés** : 247 | |

**Légende** :

SMT
Construction des formules
Substitutions
Tests ensemblistes
Pré-image
Instantiation
Simplifications
Autres

FIGURE 4.11 – Benchmarks pour l'analyse de sous-typage

```
array Chan1[proc] : msg
array Chan2[proc] : msg
array Chan3[proc] : msg
array Cache[proc] : state
array Invset[proc] : bool
array Shrset[proc] : bool

...
```

L'analyse de sous-typage caclule les types plus précis suivants :

```
var   Curcmd      : Empty | Reqs | Reqe
array Chan1[proc] : Empty | Reqs | Reqe
array Chan2[proc] : Empty | Inv | Gnts | Gnte
array Chan3[proc] : Empty | Invack
array Cache[proc] : Invalid | Shared | Exclusive
```

Bien sûr, ces informations auraient pu être données par l'utilisateur si celui-ci avait défini trois types `msg1`, `msg2`, `msg3` contenant les bons constructeurs pour chacun des canaux `Chan1`, `Chan2` et `Chan3`. L'analyse de sous-typage ayant un coût quasi nul face au reste des opérations du model checker, il aurait été dommage de passer à côté des renseignements précieux qu'elle fournit dans certains cas.

Au vu des résultats potentiels de cette simple analyse statique, une piste d'exploration consisterait à combiner du model checking avec différentes techniques d'analyse statiques plus poussées comme l'interprétation abstraite [46, 47], l'exécution symbolique [101], la logique de Hoare [71, 91], ou encore l'analyse de flots de données [100]. Par exemple, Sistla et Zhou utilisent une combinaison d'analyse statique et de réduction par bi-simulation [155]. Brat et Visser présentent quand à eux une technique de model checking consistant à entrelacer et raffiner des phases de model checking et d'analyse statique pour obtenir des informations de *réduction d'ordre partiel* [26]. Dans le même esprit, on présente dans le chapitre 5 une technique qui mélange une première analyse statique consistant en du model checking par exploration avant avec une phase de model checking par analyse d'atteignabilité arrière.

## 4.5 Exploration parallèle

Une manière naturelle d'augmenter la rapidité d'un model checker (en fait de n'importe quel programme) est de paralléliser les tâches indépendantes gourmandes en temps de calcul pour tirer parti de la disponibilité grandissante des machines multi-cœurs, multi-processeurs ou des clusters de machines [14, 83, 122]. Dans le cadre de Cubicle, la boucle d'atteignabilité arrière peut être parallélisée à un certain degré. Une implémentation directe d'une boucle parallèle affecterait cependant l'orientation de l'exploration, casserait les heuristiques employées et pourrait même rendre certaines des optimisations précédentes incorrectes. De plus l'expérience montre qu'une exploration non déterministe de l'espace de recherche est souvent moins efficace qu'une recherche guidée.

Dans notre cas, les tâches qui consomment le plus de ressources sont les tests de point-fixe (ligne 8 de l'algorithme 4) qui peuvent être des problèmes difficiles même pour des démonstrateurs SMT modernes, principalement de par leur taille. Pour paralléliser Cubicle nous avons implémenté une version concurrente de la recherche en largeur (BFS) en se reposant sur la simple observation que tous les calculs à faire sur toutes les branches à un même niveau de l'arbre de recherche peuvent être effectués en parallèle. Cette implémentation utilise une architecture centralisée de type maître/esclave. Le maître attribue des tests de point fixe aux esclaves et une barrière de synchronisation est placée à la fin de chaque niveau de l'arbre pour conserver une exploration en largeur. Le maître calcule ensuite de manière asynchrone les pré-images des nœuds qui n'ont pas pu être vérifiés comme étant des points fixes par les esclaves. Maintenant, on ne veut pas se retrouver dans la situation décrite par la figure 4.12 lorsqu'on souhaite supprimer des nœuds subsumés *a posteriori* de $\mathcal{V}$ comme dans la section 4.3. Pour ce faire le maître doit simplement ignorer les résultats concernant les nœuds qui ont été supprimés pendant qu'un esclave vérifiait leur propriété de point fixe.

D'un point de vue technique, Cubicle fournit une exploration parallèle de l'espace de

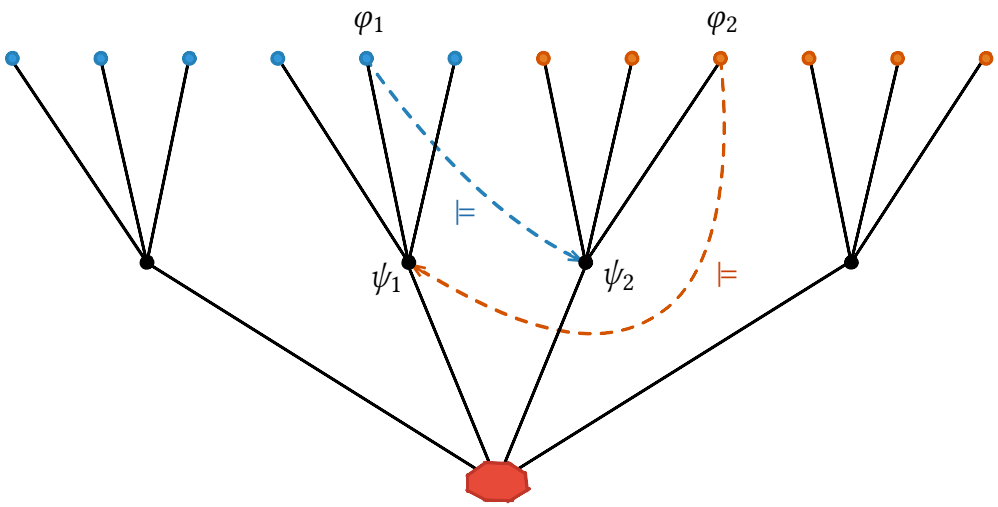
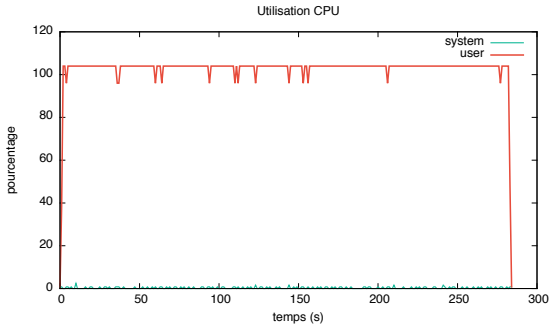



FIGURE 4.12 – Une mauvaise synchronisation des tests de subsomption effectués en parallèle (flèches en pointillés, bleue pour le processus 1 et orange pour le processus 2) peut supprimer des nœuds de manière incorrecte. Ici, une des deux suppressions par subsomption doit être ignorée pour garantir la correction.

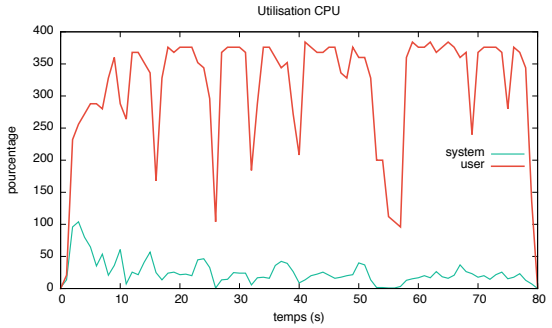
recherche en utilisant $n$ processus concurrents sur une architecture multi-cœurs lorsqu'il est invoqué avec l'option adéquate. L'implémentation utilise Functory [67], une bibliothèque OCaml fournissant une interface fonctionnelle riche et polymorphe permettant facilement de distribuer des calculs parallèles. Functory permet d'utiliser des architectures multi-cœurs ainsi que des réseaux de machines et fournit un mécanisme robuste de tolérance aux pannes.

Pour éviter de trop nombreuses commutations de contexte, on ne parallélise que les tests de point fixes qui ne sont pas triviaux. Les graphiques présentés en figure 4.13 comparent l'utilisation du processeur faite par les versions séquentielle et parallèle de Cubicle sur deux exemples qui demandent un travail significatif pour être vérifiés.

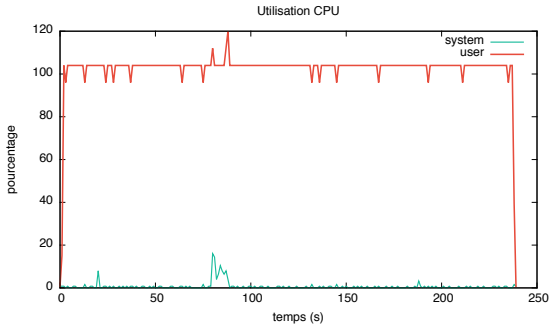
Bien entendu la version séquentielle utilise un cœur du processeur à 100% sur ces deux exemples (Figures 4.13(c) et 4.13(a)). En utilisant quatre cœurs du processeur pour la version parallèle de Cubicle, on obtient une accélération de 3,65 sur l'algorithme Szymanski (atomique) ce qui tout à fait satisfaisant. Cet exemple est particulièrement adapté à cette technique de parallélisation car il demande de faire beaucoup d'appels au solveur SMT et on peut voir sur la figure 4.13(b) que les quatre cœurs sont convenablement utilisés ($> 350\%$). En revanche l'accélération obtenue sur l'exemple German_pfs est seulement de 1,72 pour 4 cœurs. Ce résultat plus décevant est dû au fait que cet exemple fait moins appel au solveur SMT, beaucoup des tests de point fixes pouvant être déchargés avec les techniques



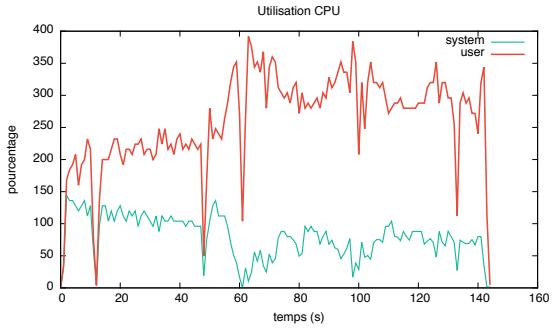

(a) Version séquentielle pour Szymanski (atomique)




(b) Version parallèle sur 4 cœurs pour Szymanski (atomique)




(c) Version séquentielle pour German_pfs




(d) Version parallèle sur 4 cœurs pour German_pfs

FIGURE 4.13 – Utilisation CPU pour les versions séquentielle et parallèle de Cubicle

d'optimisations de la section 4.2.2. On peut notamment remarquer sur la figure 4.13(d) que les cœurs du processeurs sont sous utilisés en particulier au début de l'exploration (< 250%). La parallélisation effectuée ici n'est donc pas efficace sur tous les exemples (en particulier ceux qui sont simples) mais peut porter ses fruits lorsque le solveur SMT est sollicité.

Bien que Cubicle ne fournisse pas aujourd'hui d'implémentation distribuée, c'est une des directions envisagées pour l'évolution du logiciel qui demanderait tout de même d'être capable de limiter la taille des données devant circuler sur le réseau lors des communications entre le maître et les esclaves. Ici c'est la taille de l'ensemble des nœuds visités $\mathcal{V}$ qui peut rapidement devenir un goulot d'étranglement dans une architecture distribuée reposant sur l'échange de messages. Une solution à ce problème serait que chaque esclave maintienne sa propre copie de $\mathcal{V}$ et que seules les mises à jour de cet ensemble soient transmises.

## 4.6 Résultats et conclusion

Nous avons évalué Cubicle sur des algorithmes d'exclusion mutuelle et des protocoles de cohérence de cache classiques mais aussi sur des protocoles plus difficiles à prouver (grand nombre de transitions, de variables, etc.). Une partie de ces exemples ont déjà été utilisés dans la section 4.2. Dans le tableau figure 4.14, on compare les performances de Cubicle avec d'autres model checkers existants pour systèmes paramétrés : MCMT (en version 2.5) qui implémente également le cadre théorique des systèmes à tableaux, PFS [88] et Undip [143]. Les expériences ont été conduites sur la même machine que précédemment avec une limite de 2h (on note T.O. au delà) et 20 Go de mémoire (on note O.M. au delà). Pour chaque outil, on donne les temps d'exécution obtenus avec les meilleurs réglages qu'il nous ait été donné de trouver. On a marqué par « / » les *benchmarks* qu'il nous a été impossible de traduire à cause de restrictions syntaxiques.

On peut conclure de ces résultats que Cubicle est compétitif sur ces types d'exemples. Même si la plupart sont des protocoles ou programmes académiques, il est nécessaire de mettre en place toutes les optimisations présentées précédemment pour arriver à de tels résultats. On peut par ailleurs remarquer que les temps d'exécution de Cubicle et MCMT sont du même ordre de grandeur pour la majorité des exemples, ce qui est dû au fait qu'ils utilisent la même technologie. Cependant il est intéressant de comparer les disparités entre ces deux outils qu'on aperçoit sur quelques benchmarks. Cubicle est plus performant sur le protocole d'exclusion mutuelle distribué de Ricart et Agrawala [145] car l'utilisation de tableaux multi-dimensionnels est particulièrement adaptée à la modélisation de canaux de communication. La version de ce protocole dans MCMT encode les canaux de communication entre les nœuds du réseau à l'aide de processus d'un type différent. MCMT est quant à lui beaucoup plus rapide sur la version atomique de l'algorithme de Szymanski car il possède un mécanisme de génération d'invariants lui permettant de

| **Benchmark** | **Cubicle** | MCMT [78] | Undip [143] | PFS [88] |
|---|---|---|---|---|
| Bakery | 0,01s | 0,01s | 0,04s | 0,01s |
| Dijkstra | 0,12s | 0,70s | **0,04s** | 0,26s |
| Java Meta Lock | 0,02s | 0,04s | 0,25s | 0,02s |
| Ricart Agrawala | 5,01s | 1m10s | **4,17s** | / |
| Szymanski_at | 4m41s | **0,24s** | 32,1s | T.O. |
| Berkeley | 0,01s | 0,01s | 0,01s | 0,01s |
| German_Baukus | **5,06s** | 33m15s | 9m43s | / |
| German_pfs | **2m45s** | 6m47s | T.O. | 36m05s |
| German_undip | **0,10s** | 0,46s | 1m32 | / |
| Illinois | 0,02s | 0,04s | 0,06s | 0,06s |
| Moesi | 0,01s | 0,01s | 0,01s | 0,01s |
| Szymanski_na | T.O. | / | / | / |
| Flash | T.O. | T.O. | / | / |

FIGURE 4.14 – Benchmarks

conclure quasi instantanément. Enfin les résultats particulièrement bons de Cubicle sur le protocole German sont essentiellement dûs à l'analyse de sous-typage présentée en section 4.4. Ces informations de sous typage peuvent aussi être exprimées sous la forme d'invariants mais le mécanisme de MCMT ne permet pas de les découvrir.

Pour finir, on ajoute à ce tableau deux exemples significatifs. Le premier est une version de l'algorithme de Szymanski dans laquelle les conditions globales sont évaluées de manière non atomique. La version pour Cubicle a été obtenue en utilisant l'encodage présenté en section 2.3. Le dernier est un protocole de cohérence de cache de taille industrielle conçu pour une architecture modulaire ce qui en fait un candidat particulièrement intéressant pour la vérification paramétrée. Malheureusement, la technique et l'implémentation réalisée dans Cubicle ne permettent pas de vérifier ce genre d'exemples plus réalistes. On essaye de répondre à ce problème dans le chapitre suivant en proposant une nouvelle technique pour inférer des invariants de manière automatique. La connaissance apportée par les invariants permet assurément de réduire l'espace de recherche, ils sont par conséquent très souvent mis à contribution dans les techniques de vérification.

# 5

# Inférence d'invariants

## Sommaire


**5.1 Atteignabilité approximée avec retour en arrière . . . . . 107**
5.1.1 Illustration sur un exemple . . . . . 107
5.1.2 Algorithme abstrait . . . . . 111
5.1.3 Algorithme complet pour les systèmes à tableaux . . . . . 113
**5.2 Heuristiques et détails d'implémentation . . . . . 117**
5.2.1 Oracle : exploration avant bornée . . . . . 117
5.2.2 Extraction des candidats invariants . . . . . 119
5.2.3 Retour en arrière . . . . . 121
5.2.4 Invariants numériques . . . . . 123
5.2.5 Implémentation dans Cubicle . . . . . 124
**5.3 Évaluation expérimentale . . . . . 127**
**5.4 Étude de cas : Le protocole FLASH . . . . . 130**
5.4.1 Description du protocole FLASH . . . . . 131
5.4.2 Vérification du FLASH : État de l'art . . . . . 133
5.4.3 Modélisation dans Cubicle . . . . . 135
5.4.4 Résultats . . . . . 137
**5.5 Travaux connexes sur les invariants . . . . . 139**
5.5.1 Génération d'invariants . . . . . 139
5.5.2 Cutoffs . . . . . 140
5.5.3 Abstraction . . . . . 141
**5.6 Conclusion . . . . . 142**


Ce chapitre permet de répondre au problème de la vérification de protocole de cohérence de cache de taille industrielle. Ces protocoles que l'on trouve dans les processeurs modernes sont des exemples particulièrement pertinents de systèmes concurrents car ils sont présents

dans presque toutes les machines modernes (ordinateurs personnels, serveurs, téléphones mobiles, *etc.*). De plus leur fonctionnement est très subtil car toutes les actions d'un tel protocole sont interdépendantes. Prenons par exemple l'architecture multi-processeurs FLASH développée à Stanford dans les années 90. Le système de transition décrivant son protocole de cohérence de cache contient déjà plus de 67 millions d'états possibles lorsque seulement quatre processeurs sont en compétition [33].

Vérifier ce protocole pour un plus grand nombre de processeurs en énumérant ses états s'avère être une tâche impossible. Bien sûr, il existe de nombreuses techniques de réduction, détection de symétries, compactage, *etc.* qui rendent l'exploration plus facile. Pourtant les gros problèmes mettent à défaut les model checkers énumératifs les plus modernes qui atteignent rapidement leurs limites en temps d'exécution mais surtout en occupation mémoire. Par exemple, le model checker Mur$\varphi$ très utilisé dans l'industrie [56], n'est pas capable de prouver la sûreté du protocole FLASH avec cinq processus en un temps imparti de 24 heures et un espace mémoire de 20 Go.

Ces problèmes seraient donc les candidats idéaux pour les techniques *paramétrées* ? En vérifiant une version paramétrée d'un protocole, on obtiendrait la certitude que le système est correct quelque soit le nombre de composants. Pourtant, la plupart des techniques utilisées pour vérifier des propriétés indépendamment du nombre de processus ne passent pas à l'échelle de manière satisfaisante. En effet, les model checkers paramétrés récents atteignent leurs limites sur des problèmes académiques : à titre d'exemple, la plupart des outils ont besoin de plusieurs minutes pour prouver la sûreté d'une version paramétrée du protocole German [138] (voir par exemple la figure 4.14 du chapitre précédent), bien que ce programme n'ait que 28 000 états atteignables pour quatre processus.

D'un autre côté, il existe des approches qui sont connues (et conçues) pour passer à l'échelle sur de larges problèmes : les techniques de model checking par abstraction et composition [37, 117, 118] ont été utilisées pour vérifier une version paramétrée du protocole FLASH [118, 158]. Un désavantage que présentent toutes ces techniques est qu'elles nécessitent que des experts humains fournissent des invariants mis au point à la main. Concevoir des algorithmes qui trouvent automatiquement ces invariants de qualité reste un défi et a donné naissance à un domaine de recherche actif [41, 82, 109, 120, 138].

Ce chapitre est consacré à un nouvel algorithme qui répond au problème de la vérification automatique de systèmes paramétrés conséquents. Cet algorithme peut être vu comme un mécanisme permettant d'inférer des invariants suffisamment puissants pour vérifier des protocoles complexes.

L'algorithme d'atteignabilité avec retour en arrière (BRAB, pour ***B**ackward **R**eachability with **A**pproximations and **B**acktracking*) est illustré et présenté en section 5.1. L'idée de cet algorithme est d'utiliser les informations extraites d'un modèle fini du système afin d'inférer des invariants pour le cas paramétré. Le modèle fini est en fait mis à contribution en tant qu'oracle dont le rôle se limite à émettre un jugement de valeur sur les candidats invariants qui lui sont présentés. À chaque tour de la boucle d'atteignabilité, BRAB calcule

une *sur-approximation* des états pouvant atteindre la formule dangereuse. Ce sont ces approximations qui constituent en réalité la source de candidats invariants. Si un candidat est un invariant valable pour l'oracle (*i.e.* si c'est un invariant du modèle fini) alors il est « model checké » en même temps que tous les autres candidats et propriétés originales du système. Pour garantir la complétude de l'approche, BRAB revient sur ses pas lorsqu'il détecte une trace d'erreur fallacieuse introduite par une approximation trop grossière.

Pour rendre la présentation la plus générale possible, la formalisation de BRAB est donnée pour un cadre symbolique générique dans lequel les états sont représentés par des formules logiques (Section 5.1.2). On exige seulement que la post-image soit calculable pour les instances finies et que l'atteignabilité par chaînage arrière soit décidable pour le cas paramétré (donc que la pré-image soit calculable). Sous ces conditions, on prouve la correction, la complétude et la terminaison de notre algorithme. On montre en plus que la correction de BRAB ne dépend pas de la correction de l'oracle ce qui laisse une grande liberté dans son choix.

On a implémenté BRAB dans le cadre de la théorie des système à tableaux du chapitre 3. Cette implémentation est disponible et librement accessible dans le model checker open source Cubicle. On montre l'intérêt de BRAB en comparant notre approche avec l'état de l'art des model checkers paramétrés et énumératifs sur un ensemble de problèmes réalistes (Section 5.3). À notre connaissance, Cubicle (avec BRAB) est le premier outil capable de prouver la sûreté du protocole FLASH de manière purement automatique (Section 5.4). Une partie des travaux présentés dans ce chapitre ont étés publiés dans [44].

## 5.1 Atteignabilité approximée avec retour en arrière

### 5.1.1 Illustration sur un exemple

#### Système à tableaux du protocole German-*esque*

Pour illustrer l'algorithme, on prend l'exemple du protocole de cohérence de cache donné en section 2.2.4. Comme mentionné précédemment, les états initiaux du système sont donnés par la formule logique

$$\forall i.\ \texttt{Cache}[i] = \mathsf{I} \ \wedge\ \texttt{Shr}[i] = \mathsf{false} \ \wedge\ \texttt{Exg} = \mathsf{false} \ \wedge\ \texttt{Cmd} = \epsilon$$

La description du système de transition a déjà été faite dans le chapitre 2, accompagnée du code Cubicle. On donne en figure 5.1 une version de la relation de transition sous forme logique comme fait dans le chapitre 2 avec la correspondance des noms des transitions du fichier Cubicle. Ici chaque transition est décrite par une formule logique qui relie les valeurs des variables d'états avant (*e.g.* `Cmd`) et après (*e.g.* `Cmd'`) exécution de la transition. Par exemple la transition $t_1$ se lit : s'il existe un processus $i$ dont le cache est invalide et

qu'il n'y a aucune commande à traiter, alors la variable `Ptr` est mise à jour à la valeur $i$ la variable `Cmd` passe à rs.

| **Nom Cubicle** | **Transition logique** |
|---|---|
| `request_shared` | $t_1 : \exists i.\ \mathtt{Cache}[i] = \mathsf{I} \wedge \mathtt{Cmd} = \epsilon \wedge$<br>$\mathtt{Ptr}' = i \wedge \mathtt{Cmd}' = \mathsf{rs} \wedge \mathtt{Exg}' = \mathtt{Exg} \wedge$<br>$\mathtt{Shr} = \mathtt{Shr}' \wedge \mathtt{Cache} = \mathtt{Cache}'$ |
| `request_exclusive` | $t_2 : \exists i.\ \mathtt{Cache}[i] \neq \mathsf{E} \wedge \mathtt{Cmd} = \epsilon \wedge$<br>$\mathtt{Ptr}' = i \wedge \mathtt{Cmd}' = \mathsf{re} \wedge \mathtt{Exg}' = \mathtt{Exg} \wedge$<br>$\mathtt{Shr} = \mathtt{Shr}' \wedge \mathtt{Cache} = \mathtt{Cache}'$ |
| `invalidate_1` | $t_3 : \exists i.\ \mathtt{Shr}[i] = \mathsf{true} \wedge \mathtt{Cmd} = \mathsf{re} \wedge$<br>$\mathtt{Ptr}' = \mathtt{Ptr} \wedge \mathtt{Cmd}' = \mathtt{Cmd} \wedge \mathtt{Exg}' = \mathsf{false} \wedge$<br>$\mathtt{Shr}' = \lambda j.\ ite(i = j, \mathsf{false}, \mathtt{Shr}'[j]) \wedge$<br>$\mathtt{Cache}' = \lambda j.\ ite(i = j, \mathsf{I}, \mathtt{Cache}'[j])$ |
| `invalidate_2` | $t_4 : \exists i.\ \mathtt{Shr}[i] \wedge \mathtt{Cmd} = \mathsf{rs} \wedge \mathtt{Exg} \wedge$<br>$\mathtt{Ptr}' = \mathtt{Ptr} \wedge \mathtt{Cmd}' = \mathtt{Cmd} \wedge \mathtt{Exg}' = \mathsf{false} \wedge$<br>$\mathtt{Shr}' = \mathtt{Shr} \wedge \mathtt{Cache}' = \lambda j.\ ite(i = j, \mathsf{S}, \mathtt{Cache}'[j])$ |
| `grant_shared` | $t_5 : \exists i.\ \mathtt{Ptr} = i \wedge \mathtt{Cmd} = \mathsf{rs} \wedge \neg\mathtt{Exg} \wedge$<br>$\mathtt{Ptr}' = \mathtt{Ptr} \wedge \mathtt{Cmd}' = \epsilon \wedge \mathtt{Exg}' = \mathtt{Exg} \wedge$<br>$\mathtt{Shr}' = \lambda j.\ ite(i = j, \mathsf{true}, \mathtt{Shr}'[j]) \wedge$<br>$\mathtt{Cache}' = \lambda j.\ ite(i = j, \mathsf{S}, \mathtt{Cache}'[j])$ |
| `grant_exclusive` | $t_6 : \exists i.\ \mathtt{Ptr} = i \wedge \mathtt{Cmd} = \mathsf{re} \wedge \neg\mathtt{Exg} \wedge \forall j.\ \neg\mathtt{Shr}[j] \wedge$<br>$\mathtt{Ptr}' = \mathtt{Ptr} \wedge \mathtt{Cmd}' = \epsilon \wedge \mathtt{Exg}' = \mathsf{true} \wedge$<br>$\mathtt{Shr}' = \lambda j.\ ite(i = j, \mathsf{true}, \mathtt{Shr}'[j]) \wedge$<br>$\mathtt{Cache}' = \lambda j.\ ite(i = j, \mathsf{E}, \mathtt{Cache}'[j])$ |

FIGURE 5.1 – Système de transition à tabeaux du protocole German-*esque*

On identifie la partie de la formule correspondant à la garde de la transition en noir, l'action en bleu et les variables et tableaux non modifiés sont écrits en gris pour faciliter la lecture. Ce protocole garantit que lorsqu'un cache d'un client a accès exclusif à la mémoire (E), aucun autre client n'a accès à la mémoire, *i.e.* ils sont invalidés (I). La formule dangereuse

$\Theta$ qui nous intéresse pour ce système est donc la suivante :

$$\Theta : \exists i, j.\ i \neq j \ \wedge\ \texttt{Cache}[i] = \mathsf{E} \ \wedge\ \texttt{Cache}[j] \neq \mathsf{I}$$

#### Oracle : Exploration avant pour deux caches

On considère une instance finie du protocole de la figure 5.1 avec deux caches. On donne en figure 5.2 (partie haute) un graphe partiel obtenu par une exploration avant. On démarre par l'état se trouvant dans le double cercle vert construit en instanciant la formule initiale $I$ avec deux constantes de processus distinctes #1 et #2. Chaque arc représente une transition entre deux états et est annoté par l'instance de la transition choisie. Par exemple $t_5(\#_2)$ dénote l'instance de la transition $t_5$ avec le processus $\#_2$.

#### Analyse d'atteignabilité arrière

Une fois le modèle pour l'instance finie du système construit, on lance une analyse d'atteignabilité arrière comme décrite dans les chapitres 3 et 4. En partant de la formule dangereuse $\Theta$ (octogone rouge), on calcule itérativement ses pré-images (nœuds ovales) pour toutes les transitions. Les pré-images qui sont subsumées par des nœuds déjà visités (arc pointillés) ne sont plus étendues davantage. Ce processus se termine soit lorsqu'un nœud intersecte la formule initiale $I$ ou lorsqu'il n'y a plus de pré-image à calculer.

Pour accélérer ce procédé, on essaye de réduire l'espace d'états à explorer en trouvant des sur-approximations des pré-images. Comme l'ensemble des possibilités est très grand, on restreint le choix de l'approximation aux formules qui représentent des états qui ne sont pas atteignables dans l'instance finie et qui sont des sous-formules syntaxiques de la pré-image considérée.

Si on réussit à extraire un candidat approprié, l'approximation ainsi trouvée (rectangles bleus reliés par un arc pointillé gras) remplace la formule originale. Pour illustrer ce propos, on montre un graphe partiel représentant l'exécution de BRAB sur le bas de la figure 5.2.

On commence par la formule dangereuse $\exists i \neq j.\ \texttt{Cache}[i] = \mathsf{E} \ \wedge\ \texttt{Cache}[j] \neq \mathsf{I}$, la pré-image par la transition $t_5$ retourne le nœud $\exists i \neq j.\ \neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \mathsf{rs} \ \wedge\ \texttt{Ptr} = j \ \wedge\ \texttt{Cache}[i] = \mathsf{E}$. Ce nœud peut être approximé par $\neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \mathsf{rs}$. Mais en regardant de plus près, on s'aperçoit que $\neg\texttt{Exg} \ \wedge\ \texttt{Cmd} = \mathsf{rs}$ est déjà atteignable sur le graphe du haut (Figure 5.2) car il est satisfait par un état concret de l'instance finie (arc à double flêche annoté par le symbole $\models$). De toute évidence, ce n'est pas une bonne approximation. En revanche, aucune instance de $\exists i.\ \neg\texttt{Exg} \ \wedge\ \texttt{Cache}[i] = \mathsf{E}$ n'est atteignable sur le graphe du haut. Cette approximation est insérée dans la boucle d'atteignabilié arrière qui continue comme à l'ordinaire.

Comme on peut le voir sur le graphe en figure 5.2, les sous-graphes de certaines approximations sont partagés. Comme ces sous-graphes dépeignent la preuve de l'inatteignabilité de chaque approximation, cela signifie que ces preuves sont factorisées, d'où l'intérêt de les

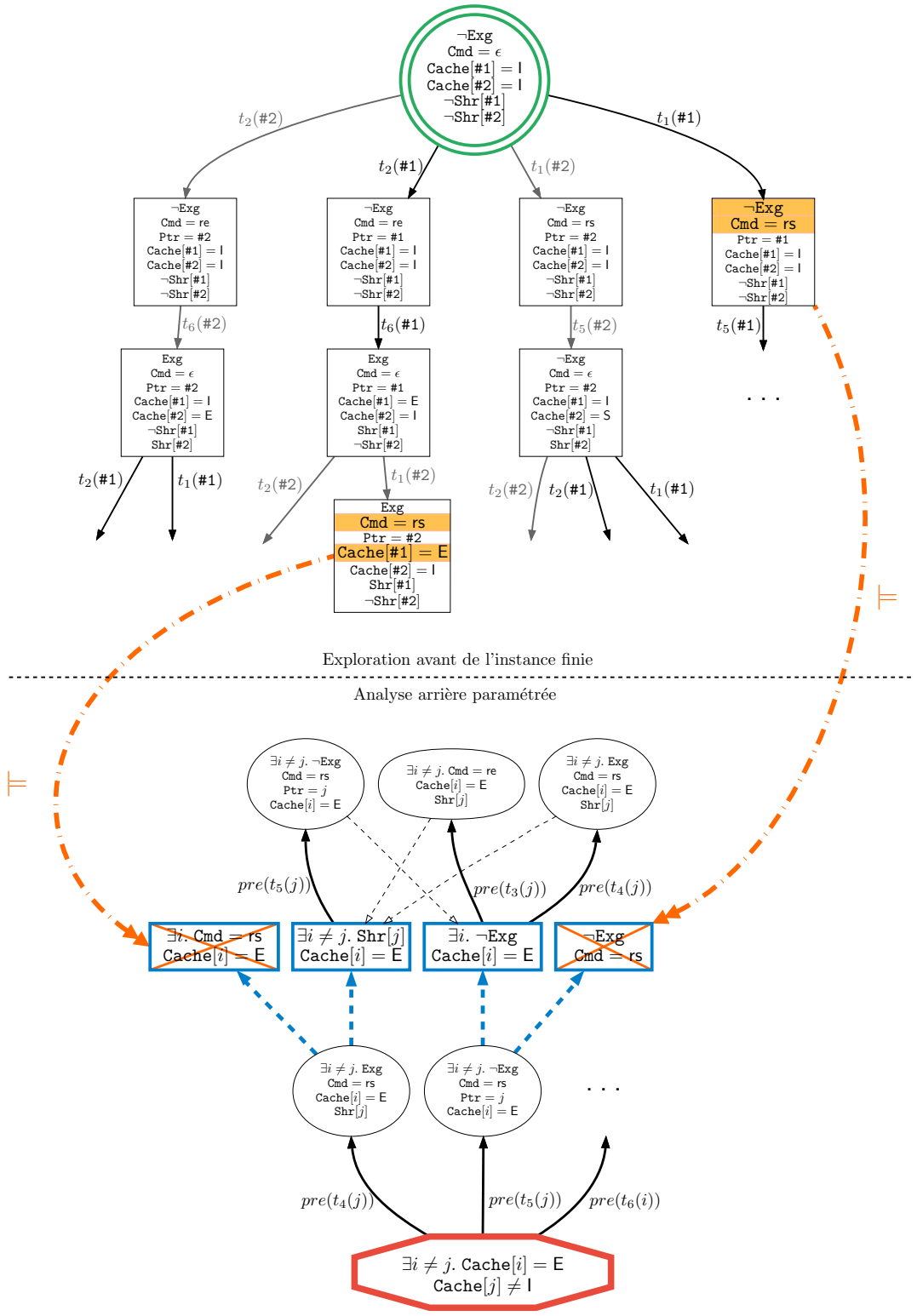



FIGURE 5.2 – Exécution partielle de BRAB sur le protocole German-*esque*

considérer toutes en même temps. Par exemple, une partie de la pré-image de la première approximation est subsumée par la seconde approximation (et *vice versa*).

Par nature, les approximations peuvent introduire des traces d'erreur fallacieuses. Lorsqu'une de celles-ci est démasquée, BRAB reprend la construction du graphe d'atteignabilité en partant de zéro, tout en se souvenant de ses précédents choix d'approximations pour éviter de refaire les mêmes erreurs autant que possible. Si, par exemple, on avait choisi comme oracle une instance finie avec seulement une constante de processus (*i.e.* un cache), rien ne nous aurait empêché de considérer la mauvaise approximation $\exists i.\texttt{Cmd} = \mathsf{rs} \wedge \texttt{Cache}[i] = \mathsf{E}$.

Dans notre exemple, en utilisant une instance finie avec deux processus, aucune approximation n'introduit de trace d'erreur et le système est prouvé sûr. Par conséquent, chaque nœud du graphe dénote un ensemble d'états inatteignables (que ce soit un nœud issu d'un calcul de pré-image ou une sur-approximation) et sa négation constitue un invariant du système. En particulier, les approximations donnent lieu aux invariants non triviaux suivants ($P_3$ n'est pas montré sur la figure 5.2 par souci de lisibilité) :

$$P_1 : \forall i, j.\ i \neq j \ \wedge\ \texttt{Cache}[i] = \mathsf{E} \implies \texttt{Shr}[j] = \mathsf{false}$$
$$P_2 : \forall i.\ \texttt{Cache}[i] = \mathsf{E} \implies \texttt{Exg} = \mathsf{true}$$
$$P_3 : \forall i.\ \texttt{Cache}[i] \neq \mathsf{I} \implies \texttt{Shr}[i] = \mathsf{true}$$

### 5.1.2 Algorithme abstrait

Après avoir vu le principe de cette technique, on donne dans cette section un algorithme abstrait qui implémente une boucle de retour en arrière autour d'une procédure d'atteignabilité générique (Algorithme 6). On explicite ici les conditions nécessaires pour pouvoir construire un tel algorithme ainsi que les conditions suffisantes pour ne perdre aucune des propriétés de l'algorithme d'atteignabilité original.

Notre algorithme est construit autour d'un algorithme d'atteignabilité arrière BWD pour systèmes paramétrés. Ici on fait apparaître clairement la fonction de pré-image utilisée par cet algorithme dans la signature de la fonction BWD car on la modifie légèrement.

La fonction `BRAB` constitue le point d'entrée de l'algorithme. Elle utilise un ensemble $\mathcal{B}$ pour garder en mémoire les mauvaises approximations effectuées (si toutefois il y en a eu). Sa boucle principale en ligne 8 fait appel à l'algorithme BWD dans lequel on a remplacé la fonction de calcul de pré-image Pre, par une version approximée $\textsc{Pre} \circ \texttt{Approx}$. Si BWD renvoie *safe* alors on est sûr que la formule $\Theta$ n'est pas atteignable dans $\mathcal{S}$, donc `BRAB` termine à son tour avec la valeur de retour *safe*. En revanche si BWD renvoie *unsafe*, il faut analyser l'erreur. Si la trace d'erreur mène bien de $I$ à $\Theta$ alors $\Theta$ est réellement atteignable. Au contraire, si l'erreur vient d'une mauvaise approximation on ne peut encore rien conclure quant à l'atteignabilité de $\Theta$. Ce dernier cas survient lorsqu'on a fait une

**Algorithme 6 :** Analyse d'atteignabilité abstraite avec approximations et retour en arrière (BRAB)

**Entrées** :

- $\mathcal{S} = (Q, I, \tau)$ : un système paramétré
- $\Theta$ : une formule dangereuse
- BWD : une fonction qui prend en arguments une fonction de calcul de pré-image Pre, un système paramétré $\mathcal{S}$ et une formule dangereuse $\Theta$, et qui renvoie *safe* seulement si $\Theta$ n'est pas atteignable ou (*unsafe*, $\bar{t}$) avec $\bar{t}$ la trace d'erreur calculée

**Variables** :

- $\mathcal{B}$ : comptabilisation des mauvaises approximations

```
1  function Approx(φ) : begin
2      foreach ψ ∈ candidates(φ) do
3          if ψ ∉ B ∧ Oracle(ψ) = Good then return ψ;
4      return φ
5
6  function BRAB(S, Θ) : begin
7      B := ∅;
8      while BWD(Pre ∘ Approx, S, Θ) = (unsafe, t̄) do
9          if la trace d'erreur t̄ est due à l'approximation F then
10             B := B ∪ {From(F)}
11         else (* Θ est atteignable par la trace t̄ *)
12             return unsafe
           (* Retour en arrière *)
13     return safe
```

approximation trop grossière, le candidat invariant qu'on a inséré dans BWD n'est pas un vrai invariant. On se souvient du mauvais choix d'approximation dans $\mathcal{B}$ et on relance l'analyse BWD.

On utilise Pre $\circ$ `Approx`, comme calcul de pré-image dans BWD car on veut calculer des pré-images d'approximations. `Approx`$(\varphi)$ retourne soit une sur-approximation de $\varphi$ soit la formule $\varphi$ elle-même si elle n'a pu trouver de « bonne « approximation. Étant donné un ensemble de candidats potentiels pour approximer $\varphi$, la fonction `Approx` fait appel à un oracle qui dit si une formule parmi cet ensemble est un invariant potentiel du système. En fait il n'existe aucune contrainte sur l'oracle, il peut répondre « oui » ou « non » sans influencer la correction de l'algorithme. Cependant, plus il est précis plus l'algorithme sera

efficace.

Pour que BRAB soit correct, on exige très peu de choses de la part des fonctions utilisées par l'algorithme 6. On demande seulement que :

1. BWD soit correcte, *i.e.* $\text{BWD}(\textsc{Pre}, \mathcal{S}, \Theta) = \mathit{safe} \implies \neg \mathit{Reachable}(\mathcal{S}, \Theta)$
2. $\forall \psi.\ \psi \in \texttt{candidates}(\varphi) \implies \varphi \models \psi$

Les conditions supplémentaires suffisantes pour que BRAB termine sont les suivantes :

1. BWD termine
2. Quelque soit $\varphi$, $\texttt{candidates}(\varphi)$ est un ensemble fini d'approximations strictes de $\varphi$, *i.e.* $\forall \psi.\ \psi \in \texttt{candidates}(\varphi) \implies \psi \not\models \varphi$
3. La fonction `Oracle` termine

Sous les conditions de terminaison précédentes, on obtient également que si l'algorithme BWD est complet (*i.e.* $\text{BWD}(\textsc{Pre}, \mathcal{S}, \Theta) = (\mathit{unsafe}, _) \implies \mathit{Reachable}(\mathcal{S}, \Theta)$) alors BRAB est lui aussi complet.

Dans la section suivante, on donne une implémentation concrète de BRAB dans le cadre théorique des systèmes à tableaux. On donne également la preuve de correction de l'algorithme obtenu sous les conditions énoncées précédemment.

### 5.1.3 Algorithme complet pour les systèmes à tableaux

Notre version de BRAB pour le cadre des systèmes à tableaux est donnée par l'algorithme 7. Cette implémentation utilise une exploration avant finie comme oracle (algorithme 8).

BRAB prend en entrée un système paramétré $\mathcal{S} = (Q, I, \tau)$ et une formule dangereuse $\Theta$. En plus de l'ensemble des nœuds visités $\mathcal{V}$ et de la file de travail, notre implémentation a besoin de deux variables $\mathcal{B}$ et $\mathcal{F}$, ainsi que de deux dictionnaires `Kind` et `From`. Ces variables sont utilisées comme suit :

— $\mathcal{B}$ est un ensemble de mauvaises (trop grossières) sur-approximations ;

— $\mathcal{F}$ contient le dernier nœud visité qui n'a pas réussi le test de sûreté (*i.e.* $\mathcal{F} \wedge I$ est satisfiable) pendant l'analyse d'atteignabilité arrière ;

— `Kind` est un dictionnaire à valeurs dans $\{\textsf{Orig}, \textsf{Appr}\}$. Il sert à identifier les nœuds qui ont été obtenus par calcul de pré-image successifs à partir de la formule dangereuse originale (Orig) ou à partir d'une approximation (Appr) ;

— `From` associe à un nœud la formule ancêtre (originale ou approximation) à partir de laquelle il est dérivé

Les dictionnaires `From` et `Kind` permettent d'annoter les formules afin de savoir si la trace d'erreur renvoyée par la boucle d'atteignabilité est une fausse alarme due à l'utilisation d'approximations.

Comme dans la section précédente, le point d'entrée de l'algorithme est la fonction `BRAB`. Elle commence par initialiser l'ensemble $\mathcal{B}$ à vide, annote la formule $\Theta$ par Orig dans `Kind`, et défini son ancêtre comme étant elle même dans `From`. BRAB entre ensuite dans sa boucle de vérification (ligne 32). Il fait tout d'abord appel à `BWDA` qui vérifie l'atteignabilité de $\Theta$ dans le système $\mathcal{S}$ par un calcul de pré-image approximé. Si `BWDA` renvoie *safe*, alors `BRAB` fait de même. Sinon, $\mathcal{F}$ contient la dernière formule qui a échoué le test de sûreté. Si $\mathcal{F}$ est une vraie pré-image de $\Theta$, `BRAB` renvoie *unsafe*. Si $\mathcal{F}$ a été obtenue par le calcule de pré-images successives d'approximation, alors `BRAB` ignore la fausse alarme, enregistre l'ancêtre de $\mathcal{F}$ dans $\mathcal{B}$ pour éviter de reproduire la même trace, et recommence la boucle de vérification.

`BWDA` implémente une boucle d'atteignabilité arrière très similaire à celle de l'algorithme 4. Elle diffère en seulement deux endroits. Premièrement, elle enregistre la formule courante dans $\mathcal{F}$ avant de retourner *unsafe* (ligne 23). Ensuite, elle appelle la fonction `Pre_Approx` au lieu de la fonction $\textsc{Pre}_\tau$ (ligne 27) pour faire des sur-approximations et calculer les pré-images. Le seul travail de `Pre_Approx` est de propager les annotations contenues dans `Kind` et `From` après calcul de la pré-image.

La fonction `Approx` cherche un candidat intéressant grâce à l'oracle. Bien sûr, ce candidat ne doit pas être connu comme étant mauvais. Afin de garder les bonnes propriétés du cadre des systèmes à tableaux, la fonction `candidates` renvoie ici un ensemble fini de *cubes*. Pour obtenir ces cubes, on considère seulement dans notre implémentation les sous-formules syntaxiques de $\varphi$ (voir détails en section 5.2.2). Si `Approx` trouve une nouvelle approximation, elle est annotée par Appr dans `Kind` et et on l'enregistre comme étant à l'origine de cette approximation dans `From`.

Pour concevoir un oracle, on a une liberté totale. BRAB ne demande pas même qu'il soit correct, juste qu'il termine. Toutefois on souhaite un oracle rapide et assez précis. Une des forces de BRAB réside dans le choix de cet oracle. Dans notre implémentation l'oracle construit un modèle $\mathcal{M}_k$ du système de départ en fixant le paramètre $k$, le nombre de processus. Ce choix est tout particulièrement pertinent car bien souvent les instances même petites du système permettent de voir beaucoup de ses comportements intéressants. L'oracle constitue donc une source extérieure de connaissance sur le système. Celui qu'on a choisi ici tire cette connaissance d'une seule exploration du système. Il exhibe les bonnes propriétés car il est rapide et ne se trompe que très rarement (*cf.* l'évaluation expérimentale de la section 5.3).

Cet oracle est donné par l'algorithme 8. Il garde un état interne $\mathcal{M}_k$ qui est initialisé par une exploration avant d'une instance du système $\mathcal{S}$ avec $k$ processus. La première boucle maintient une file de travail initialement remplie avec les états initiaux possibles pour chacun des $k$ processus. Ensuite à chaque tour de boucle on vérifie que l'état n'a pas déjà été considéré puis on calcule ses post-images par rapport à la relation de transition. Ici $\tau(k)$ représente l'instance de la relation de transition pour $k$ processus. On enregistre au passage la profondeur de chaque état dans le graphe d'exploration. De cette façon on peut

**Algorithme 7 :** Analyse d'atteignabilité avec approximations et retour en arrière (BRAB)

**Entrées** : un système paramétré $\mathcal{S} = (Q, I, \tau)$ et une formule dangereuse $\Theta$
**Variables** :
$\mathcal{V}$ : nœuds visités
$\mathcal{Q}$ : file de travail
$\mathcal{B}$ : comptabilisation des mauvaises approximations
$\mathcal{F}$ : dernier nœud visité lorsqu'une trace d'erreur est découverte
`Kind` : dictionnaire de formules $\longmapsto$ {Orig, Appr}
`From` : dictionnaire des formules (nœuds) $\longmapsto$formules (nœuds)

```
1  function Approx(φ) : begin
2      foreach ψ in candidates(φ) do
3          if ψ ∉ B ∧ Oracle(ψ) = Good then
4              Kind(ψ) := Appr;
5              if Kind(φ) = Orig then From(ψ) := ψ;
6              else From(ψ) := From(φ);
7              return ψ
8      return φ
9
10 function Pre_Approx(φ) : begin
11     let P = Pre_τ(Approx(φ)) in
12     foreach ψ ∈ P do
13         Kind(ψ) := Kind(φ);
14         From(ψ) := From(φ)
15     return P
16
17 function BWDA(S, Θ) : begin
18     V := ∅;
19     push(Q, Θ);
20     while not_empty(Q) do
21         φ := pop(Q);
22         if I ∧ φ sat then
23             F := φ;
24             return unsafe
25         else if φ ⊭ V then
26             V := V ∪ {φ};
27             push(Q, Pre_Approx(φ))
28     return safe
29
30 function BRAB(S, Θ) : begin
31     B := ∅ ; Kind(Θ) := Orig ; From(Θ) := Θ;
32     while BWDA(S, Θ) = unsafe do
33         if Kind(F) = Orig then return unsafe;
34         B := B ∪ {From(F)};
35     return safe
```

**Algorithme 8 :** Oracle : Exploration avant limitée en profondeur

**Entrées** : un système paramétré $\mathcal{S} = (Q, I, \tau)$, la profondeur maximale $d_{\max}$ pour l'exploration avant et le nombre $k$ de processus à considérer pour l'instance finie du système

**Variables** :

$\mathcal{M}_k$ : l'ensemble des états du modèle construit

$\mathcal{Q}$ : file de travail

```
1  begin
       (* Initialisation de l'oracle : exploration avant jusqu'à profondeur d_max *)
2      M_k := ∅;
3      push(Q, (0, I(#1) ∧ ··· ∧ I(#k)));
4      while not_empty(Q) do
5          let (d, φ) = pop(Q) in
6          if φ ∉ M_k ∧ d ≤ d_max then
7              M_k := M_k ∪ {φ};
8              let Ψ = {(d + 1, ψ) | ψ ∈ Post_τ(k)(φ)} in
9              push (Q, Ψ)
10
11 function Oracle(ψ) : begin
12     if M_k ⊭ ψ then return Good;
13     else return Bad;
```

s'arrêter quand on atteint la profondeur limite $d_{\max}$ (si $d_{\max} \neq \infty$).

Quand BRAB interroge l'oracle (fonction `Oracle` à la ligne 3 de l'algorithme 7), celui-ci a maintenant seulement besoin de vérifier que le candidat n'est pas atteignable dans le modèle construit préalablement, *i.e.* $\mathcal{M}_k \not\models \psi$, soit une opération linéaire en la taille de $\mathcal{M}_k$.

### Correction

On suppose dans le reste de cette section qu'on est dans le cas favorable où l'algorithme 4 termine. La correction de BRAB pour les systèmes à tableaux repose sur les invariants de la fonction `BWDA` suivants :

1. $\mathcal{V}$ ne contient pas de cube *directement* atteignable, *i.e.*

$$\mathcal{V} \models \neg I \tag{5.1}$$

2. $\textsc{Pre}^*_\tau(\Theta)$ est calculé *incrémentalement* à l'aide de $\mathcal{V}$ et $\mathcal{Q}$, *i.e.*

$$\textsc{Pre}^*_\tau(\Theta) \models \mathcal{V} \vee \textsc{Pre}^*_\tau(\mathcal{Q} \backslash \mathcal{V}) \tag{5.2}$$

3. Soit $\varphi$ un cube,

$$\texttt{Kind}(\varphi) = \mathsf{Orig} \implies \varphi \models \textsc{Pre}^*_\tau(\Theta) \tag{5.3}$$

**Proposition 5.1.1.** *Soit $\mathcal{S}$ un système à tableaux et $\Theta$ une formule dangereuse. Si* $\texttt{BRAB}(\mathcal{S}, \Theta)$ = ***safe*** *alors $\mathcal{S}$ est sûr vis-à-vis de $\Theta$.*

*Démonstration.* On sait que BRAB renvoie *safe* si et seulement si une exécution de BWDA renvoie *safe*. BWDA calcule une sur-approximation de $\textsc{Pre}^*_\tau(\Theta)$ donc si $\texttt{BWDA}(\mathcal{S}, \Theta)$ = *safe* alors on est sûr que $\textsc{Pre}^*_\tau(\Theta) \wedge I$ est insatisfiable. □

**Proposition 5.1.2.** *Soit $\mathcal{S}$ un système à tableaux et $\Theta$ une formule dangereuse. Si l'algorithme 4 est complet, alors BRAB est complet, i.e. si* $\texttt{BRAB}(\mathcal{S}, \Theta)$ = unsafe *alors $\Theta$ est atteignable dans $\mathcal{S}$.*

*Démonstration.* Si $\texttt{BRAB}(\mathcal{S}, \Theta)$ = *unsafe*, alors il existe un cube $\mathcal{F}$ tel que $\mathcal{F} \wedge I$ est satisfiable et $\texttt{Kind}(\mathcal{F})$ = Orig. Grâce à l'invariant (5.3), on conclut que $\textsc{Pre}^*_\tau(\Theta) \wedge I$ est satisfiable. □

**Proposition 5.1.3.** *Sous les conditions énoncées précédemment, BRAB termine.*

*Démonstration.* Supposons que l'algorithme 7 ne termine pas. On se trouve donc dans l'un de ces deux cas :

1. BWDA ne termine pas, ou
2. La boucle de la fonction BRAB ne termine pas

(1) Étant donné que BWDA diffère de la fonction BWD de l'algorithme 4 seulement par la fonction Pre_Approx, sa terminaison est assurée par le fait que candidates renvoie un ensemble fini de formules, que la fonction Oracle termine et que BWD termine. (2) Le domaine (l'ensemble des clefs) du dictionnaire From est un sous-ensemble de $\mathcal{A} = \bigcup_{\varphi \in \mathcal{V}_f} \texttt{candidates}(\varphi)$ (garanti par la ligne 5), où $\mathcal{V}_f$ est l'ensemble final obtenu par $\texttt{BWDA}(\mathcal{S}, \Theta)$. Comme $\mathcal{V}_f$ est fini (BWDA termine), alors $\mathcal{A}$ est aussi un ensemble fini. La condition $\psi \notin \mathcal{B}$ de la ligne 3 garantit que les approximations ajoutées à $\mathcal{B}$ ligne 34 sont toujours distinctes les unes des autres et sont dans $\mathcal{A}$. En d'autres termes, $\mathcal{B}$ ne peut pas croître indéfiniment, donc la boucle de BRAB termine. □

## 5.2 Heuristiques et détails d'implémentation

### 5.2.1 Oracle : exploration avant bornée

L'implémentation de l'algorithme 8 repose principalement sur l'implémentation concrète de la fonction $\textsc{Post}_{\tau(k)}$. Selon la représentation choisie des états de $\mathcal{M}_k$, le calcul de la

post-image peut se faire de manière symbolique. Par exemple dans notre cas, la forme des transitions autorisées dans la théorie des tableaux nous permet de calculer effectivement les post-images lorsqu'on a fixé la cardinalité du domaine `proc` au préalable. Une approche symbolique, qui manipulerait des formules, pourrait présenter un avantage pour certains problèmes mais nous avons remarqué qu'une exploration *énumérative* était la plus efficace sur les problèmes qui nous intéressaient. Cette remarque a déjà été faite par le passé concernant les protocoles asynchrones à échanges de messages [95] où le model checking énumératif est même la méthode standard utilisée dans l'industrie aujourd'hui pour vérifier les protocoles de cohérence de cache [33, 56].

Les techniques sont parfois mélangées. Dans ces approches, une partie de l'état peut être représentée de manière symbolique tandis que le reste des valeurs est énuméré. On adopte une approche similaire pour l'implémentation concrète de notre oracle. Afin d'assurer que l'exploration soit la plus rapide possible, on utilise des structures de données efficaces en interne (par exemple des tableaux d'entiers) pour représenter les états. D'un autre côté, on s'autorise à désigner des variables d'état comme *non initialisées* en leur affectant une valeur spéciale. Cela permet de factoriser certaines parties de l'exploration, notamment celles qui ne dépendent pas des valeurs de ces variables. En outre, la formule initiale $I$ du système à tableaux décrit parfois des relations entre les variables et certaines restent non spécifiées (voir haut de la figure 5.2 par exemple). En utilisant des valeurs spéciales à ces endroits on explore un sur-ensemble des états atteignables.

**Exemple.** Par exemple si l'état initial est $I \equiv \mathtt{X} = \mathtt{Y}$, on considérera que les variables `X` et `Y` ne sont pas initialisées et on oubliera la relation d'égalité qui lie leurs valeurs.

Sur la figure 5.3 on donne un exemple de transition pour un état dans lequel la valeur de la variable booléenne `X` n'est pas initialisée. On note cette valeur « ? ». Comme la garde demande que `X` soit égal à `Z` (qui vaut `False`), on sait que `X` vaut `False` après exécution de la transition.

*Remarque.* On a choisi d'implémenter nous même une boucle de model checking énumératif naïve car c'est une tache assez simple et le fait que l'oracle soit directement accessible est une facilité pour le prototypage. Il est tout à fait envisageable d'utiliser un outil externe comme les model checkers Spin [93] ou Mur$\varphi$ [55] pour profiter des années de travail qui ont abouti au succès de ces logiciels. Cependant un point crucial de l'oracle est d'être capable de répondre aux requêtes rapidement donc on ne veut pas relancer le model checker à chaque approximation. Il faut donc pouvoir sauvegarder les états (plutôt que de simples valeurs de *hashs*) et être capable de réaliser les tests d'invariants (linéaires en la taille de l'espace d'états atteignables) *a posteriori*.

Ici, on fixe la valeur de $k$ et on ne fait qu'une seule exploration pour construire $\mathcal{M}_k$. Il est aussi possible d'adapter l'oracle pour construire plusieurs modèles, par exemples pour des valeurs du paramètre allant de 0 jusqu'à $k$. Il suffit ensuite de remplacer le test en ligne 12

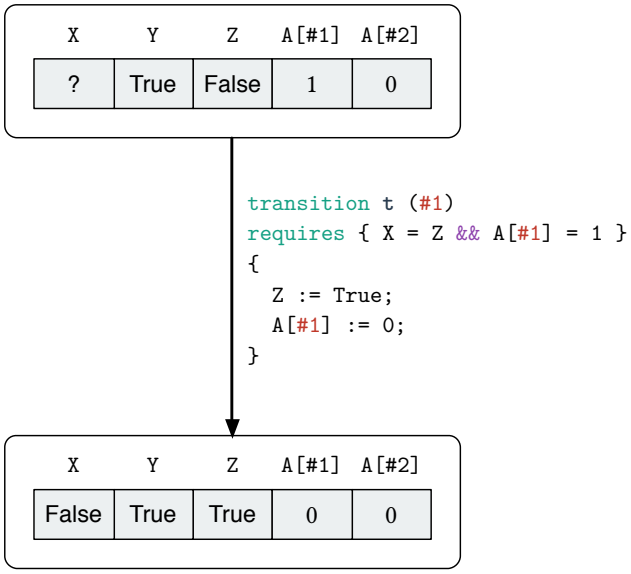



FIGURE 5.3 – Transition sur un état avec variable non-initialisée

de l'algorithme 8 par $\forall i \in [\![0,k]\!].\ \mathcal{M}_i \not\models \psi$. Cette modification est utile lorsque certaines transitions ne sont pas déclenchables sur un domaine de taille $k$. Par exemple la transition suivante n'est possible que lorsque la cardinalité de `proc` est exactement 1 :

```
transition only_one (i)
requires { forall_other j. False = True }
{ ... }
```

### 5.2.2 Extraction des candidats invariants

La deuxième heuristique essentielle de notre algorithme est le choix des candidats invariants. En effet, l'oracle a beau répondre de manière exacte aux requêtes qui lui sont faites, son aide sera inutile si on lui pose les mauvaises questions. Dans notre cas ces questions sont formulées par un ensemble de candidats invariants parmi lesquels il faut extraire un élément qui a de grandes chances d'être un vrai invariant du système.

Dans la plupart des approches pour inférer des invariants, les candidats sont générés à partir d'informations extérieures [130, 158], d'une abstraction du système [82, 126, 138], ou à partir d'une analyse sémantique de la description du système. Le nombre d'invariants possibles pour un système est souvent infini, c'est pourquoi certaines techniques emploient un schéma général avec patrons pour générer des candidats en nombre raisonnable [65, 97]. Peu de ces techniques sont guidées par le but, dans le sens où les candidats cherchés ne

dépendent pas de la propriété à vérifier. Dans BRAB on est guidé par les nœuds produits par l'algorithme d'atteignabilité pour trouver ces candidats. C'est aussi le cas de l'approche adoptée dans [75, 77].

Les nœuds du graphe construits par la fonction `BWD` de l'algorithme 4 sont des cubes. Pour rester dans le même fragment les candidats recherchés sont aussi des cubes. La fonction `candidates` doit respecter sa spécification, à savoir produire des formules qui subsument (strictement) son argument. Une façon simple de produire de telles formules est de considérer toutes les sous-formules syntaxiques d'un cube.

Soit un cube $\varphi = \exists \bar{i}.\ \Delta(\bar{i}) \ \wedge\ l_1 \wedge \ldots \wedge l_n$ où $L_\varphi = \{l_1, \ldots, l_n\}$ est l'ensemble des littéraux de $\varphi$.

1. On appelle un *sous-cube* de $\varphi$ tout cube $\psi = \exists \bar{j}.\Delta(\bar{j}) \ \wedge\ k_1 \wedge \ldots k_m$ tel que $\{\bar{j}\} \subseteq \{\bar{i}\}$ et $L_\psi = \{k_1, \ldots, k_m\} \subseteq L_\varphi$.

2. Si en plus $L_\psi \subsetneq L_\varphi$ alors on dit que $\psi$ est un *sous-cube strict* de $\varphi$.

**Proposition 5.2.1.** *Soit $\varphi$ un cube et $\psi$ un sous-cube strict de $\varphi$. Alors $\varphi \models \psi$ et $\psi \not\models \varphi$.*

*Démonstration.* Corollaire de la proposition 4.2.1 du chapitre 4. □

Dans l'implémentation de BRAB au sein de Cubicle, on prend donc $\texttt{candidates}(\varphi) = \{\psi \mid \psi \text{ sous-cube strict de } \varphi\}$. Cet ensemble est fini car l'ensemble des littéraux d'un cube $\varphi$, $L_\varphi$ est lui aussi fini. En pratique on s'autorise à prendre n'importe quel sous-ensemble de $\{\psi \mid \psi \text{ sous-cube strict de } \varphi\}$ pour limiter le nombre de candidats produits à chaque approximation. De plus, cet ensemble est donné à l'oracle sous la forme d'une liste triée (ou séquence) pour privilégier les candidats invariants les plus généraux possibles. On définit l'ordre d'apparition dans cette liste $<_c$, de telle façon que si $\psi_1, \psi_2 \in \texttt{candidates}(\varphi)$ et $\psi_1 <_c \psi_2$ alors $\psi_1 \not\models \psi_2$. La fonction `Approx` de l'algorithme 7 choisira donc l'élément le plus petit pour cet ordre que l'oracle considère comme un bon candidat invariant.

**Exemple.** Pour le cube $\exists i, j.\ i \neq j \wedge \texttt{Exg} = \mathsf{true} \wedge \texttt{Cmd} = \mathsf{rs} \wedge \texttt{Cache}[i] = \mathsf{E} \wedge \texttt{Shr}[j] = \mathsf{true}$ de la figure 5.2, la fonction `candidates` renvoie l'ensemble ordonné suivant dans lequel

on a surligné la première approximation acceptée par l'oracle :

$$
\begin{aligned}
\{\ & \texttt{Exg} = \mathsf{true}, \\
& \texttt{Cmd} = \mathsf{rs}, \\
& \texttt{Exg} = \mathsf{true} \wedge \texttt{Cmd} = \mathsf{rs}, \\
& \exists i.\ \texttt{Shr}[i] = \mathsf{true} \wedge \texttt{Cache}[i] = \mathsf{E}, \\
& \exists i.\ \texttt{Cmd} = \mathsf{rs} \wedge \texttt{Shr}[i] = \mathsf{true}, \\
& \exists i.\ \texttt{Cmd} = \mathsf{rs} \wedge \texttt{Cache}[i] = \mathsf{E}, \\
& \exists i.\ \texttt{Exg} = \mathsf{true} \wedge \texttt{Shr}[i] = \mathsf{true}, \\
& \exists i.\ \texttt{Exg} = \mathsf{true} \wedge \texttt{Cache}[i] = \mathsf{E}, \\
& \exists i.\ \texttt{Exg} = \mathsf{true} \wedge \texttt{Cmd} = \mathsf{rs} \wedge \texttt{Shr}[i] = \mathsf{true}, \\
& \exists i.\ \texttt{Exg} = \mathsf{true} \wedge \texttt{Cmd} = \mathsf{rs} \wedge \texttt{Cache}[i] = \mathsf{E}, \\
& \exists i, j.\ i \neq j \wedge \texttt{Cache}[i] = \mathsf{E} \wedge \texttt{Shr}[j] = \mathsf{true}, \\
& \exists i, j.\ i \neq j \wedge \texttt{Cmd} = \mathsf{rs} \wedge \texttt{Cache}[i] = \mathsf{E} \wedge \texttt{Shr}[j] = \mathsf{true}, \\
& \exists i, j.\ i \neq j \wedge \texttt{Exg} = \mathsf{true} \wedge \texttt{Cache}[i] = \mathsf{E} \wedge \texttt{Shr}[j] = \mathsf{true} \quad \}
\end{aligned}
$$

### 5.2.3 Retour en arrière

Dans la boucle de la fonction BRAB (Algorithmes 6 et 7) on redémarre de zéro à chaque retour en arrière. Bien que ces redémarrages surviennent rarement en pratique, il sont tout de même coûteux. En réalité, on peut faire mieux que de repartir complètement de zéro pour éviter de répéter des raisonnements identiques.

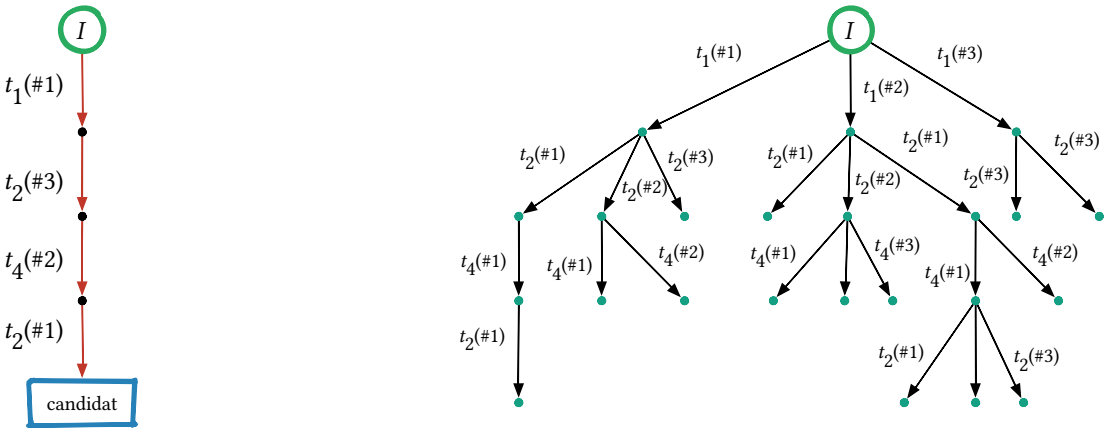
Une première possibilité évidente pour accélérer les exécutions prochaines de la fonction BWDA est de se rappeler des candidats invariants utilisés lors de l'exploration précédente. En effet, les approximations qui n'ont pas participé à la trace d'erreur seront reproduites. En outre, il est toujours autorisé d'ajouter n'importe quel candidat invariant au système sans perdre aucune des propriétés de BRAB.

Ensuite il est aussi possible d'intégrer à l'algorithme un mécanisme d'apprentissage similaire dans l'esprit aux techniques d'*apprentissage de clauses conflit* (aussi appelé *clause learning*) des solveur SAT modernes [154, 165] et aux procédures de raffinement CEGAR (pour *counter-example guided abstraction refinement*) [7, 13, 35, 147]. Pour ce faire il faudrait inférer à partir de la trace d'erreur une classe plus générale de mauvaises approximations plutôt que seulement enregistrer le coupable dans $\mathcal{B}$. Dans Cubicle on utilise plusieurs heuristiques pour apprendre de l'erreur. Par exemple, il peut arriver qu'une trace fasse intervenir une coopération entre trois processus distincts afin d'infirmer le candidat invariant. Si le modèle utilisé par l'oracle n'a été construit que pour deux processus ($\mathcal{M}_2$), il est toujours possible d'augmenter ce dernier avec une exploration additionnelle de $\mathcal{M}_3$. Pour des raisons similaires un redémarrage peut aussi survenir lorsqu'on a exploré l'instance finie avec une profondeur limitée.

Plus simplement, la trace d'erreur peut-être rejouée pour découvrir tous les états qui

sont accessibles dans un modèle adéquat. Refaire une exploration d'une instance avec un processus supplémentaire peut être exponentiellement plus long, donc en pratique on adopte une approche se situant à mi-chemin. La trace d'erreur est abstraite en enlevant les constantes de processus, puis est rejouée en faisant toutes les instances pour une même transition.

**Exemple.** La trace d'erreur $t_1(\#1), t_2(\#3), t_4(\#2), t_2(\#1)$ est abstraite en $t_1, t_2, t_4, t_2$ et on explore tous les états accessibles par n'importe quelle instance de cette trace avec trois processus (voir Figure 5.4).




Trace d'erreur

Exploration de la trace abstraite

FIGURE 5.4 – Apprentissage à partir d'une exploration supplémentaire avant redémarrage

On limite aussi l'effet négatif des retours en arrière en essayant de les déclencher le plus tôt possible, ce qui revient à trouver les mauvaises approximations aussi rapidement que possible. Une heuristique acceptable pour ce faire est de donner la priorité aux approximations dans la file $\mathcal{Q}$ de la fonction `BWDA` (Algorithme 7) de façon à vérifier les candidats invariants avant d'avoir trop déroulé la relation de transition sur la formule dangereuse originale ($\Theta$).

Au lieu de redémarrer de zéro, une amélioration possible (mais non implémentée) de ce schéma consisterait à garder les informations de l'exécution précédente qui restent pertinentes. Il est possible, mais toutefois assez coûteux, de maintenir des informations de dépendance et d'historique à l'exécution, puis de s'en servir pour conserver les nœuds présents dans $\mathcal{V}$ et $\mathcal{Q}$ qui n'ont pas étés affectés par le mauvais choix d'approximation.

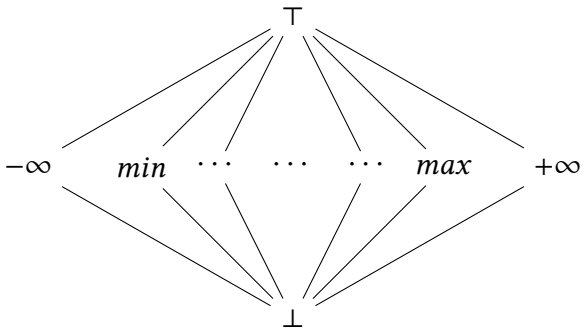
### 5.2.4 Invariants numériques

Les systèmes à tableaux sont des systèmes infinis en premier lieu car ils sont paramétrés, mais aussi car ils peuvent manipuler des variables dont les types sont infinis. En particulier dans Cubicle, on peut déclarer des variables et tableaux contenant des entiers mathématiques, des réels ou encore des valeurs d'un type abstrait.

L'oracle qu'on a construit dans la section 5.2.1 permet de s'affranchir du côté paramétré mais son aspect énumératif nous autorise seulement à considérer des instances avec un nombre d'états *fini*. (Par exemple, une transition qui ferait `X := X + 1` générerait un nombre infini d'états.) Une solution simple à ce problème consiste à oublier toutes les variables (et données) à domaine non borné. On est alors certain de considérer un surensemble des états atteignables du système mais cette abstraction nous empêcherait de découvrir des invariants *numériques*.

Dans les programmes qui manipulent des entiers ou des réels, ces invariants sont d'une importance capitale. L'inférence d'invariants numériques est un problème compliqué et constitue un domaine de recherche très actif. Les méthodes les plus prospères se reposent toutes sur des techniques d'interprétation abstraite [80, 124] car elles sont particulièrement adaptées à l'analyse de programmes dépendant fortement des données. Elles sont cependant moins adaptées à l'analyse de propriétés de contrôle [84] pour lesquelles le model checking est plus efficace.

Plutôt que d'abstraire toutes les variables numériques comme suggéré ci-dessus, on définit une famille de domaines abstraits très simples qui nous permettent de conserver une partie des informations dans notre explorations énumérative. On n'utilise cependant aucune des techniques classiques d'interprétation abstraite. La seule modification qu'on apporte à notre oracle est une interprétation du système de transition et des états sur un domaine construit à partir de deux valeurs $min$ et $max$ et définit par le treillis suivant :




Après avoir défini une borne inférieure $min$ et une borne supérieure $max$, on interprète les valeurs entières par les entiers dans l'intervalle $[\![min, max]\!]$, ou par le symbole $-\infty$ (*resp.* $+\infty$) si la valeur est inférieure à $min$ (*resp.* supérieure à $max$). Pour les réels, le treillis est identique au détail près qu'on remplace ses éléments par des intervalles $[min, min +$

$1[, \cdots [max-1, max[, [max, max]$. Les variables d'un type abstrait restent quant à elles abstraites. Comme l'ensemble des éléments d'un tel domaine est fini, on peut utiliser l'exploration énumérative efficace décrite précédemment.

Bien sûr cette solution n'est pas complètement satisfaisante car elle demande tout d'abord à l'utilisateur de fournir les valeurs *min* et *max*. Ensuite, le nombre d'états à explorer peut potentiellement devenir $m \times (max - min + 2)$ fois plus grand, où $m$ est le nombre de variables du système instancié. Enfin, bien qu'un oracle reposant sur cette approche permette de filtrer des mauvaises approximations, il devient nécessaire de générer des candidats invariants pertinents. On modifie donc légèrement notre fonction `candidates` pour qu'elle généralise les formules arithmétiques par des approximations sur des relations entre variables ou entre variables et constantes. Cette méthode simpliste est toutefois suffisante pour traiter des exemples avec horloges (comme l'algorithme d'exclusion mutuelle distribué de Ricart et Agrawala [145]) ou compteurs (*cf.* Section 5.3).

Des travaux récents s'intéressent à générer automatiquement des interpréteurs abstraits pour des systèmes de transitions exprimables dans des fragments décidables de la logique [160]. Comme notre oracle utilise des instances finies du système de transition (*i.e.* sans quantificateurs), ces techniques — en particulier l'approche itérative de [72] — pourraient être combinées et adaptées à notre contexte pour améliorer la génération d'invariants numériques dans Cubicle.

### 5.2.5 Implémentation dans Cubicle

Nous avons implémenté la version de BRAB pour système à tableaux de la section 5.1.3 dans notre model checker Cubicle. L'architecture de la nouvelle version de Cubicle reste tout aussi modulaire (Figure 5.5), la seule différence fondamentale est que le module BRAB est maintenant le point d'entrée du programme.

Ce dernier implémente la boucle de retour en arrière en appelant la fonction search de BWD (voir ci-contre). Celle-ci prend comme paramètres optionnels une liste d'invariants du système et une liste de candidats invariants. Les invariants sont supposés vrais et donc ajoutés directement à l'ensemble des nœuds visités $\mathcal{V}$ alors que les candidats doivent êtres prouvés invariants. Ils sont donc ajoutés à la file de travail $\mathcal{Q}$. Comme le résultat renvoyé par la fonction `search` contient la liste des approximations découvertes, on peut, lors d'un retour en arrière, transmettre cette liste au prochain appel de `search` au moyen de l'argument nommé `~candidates`.

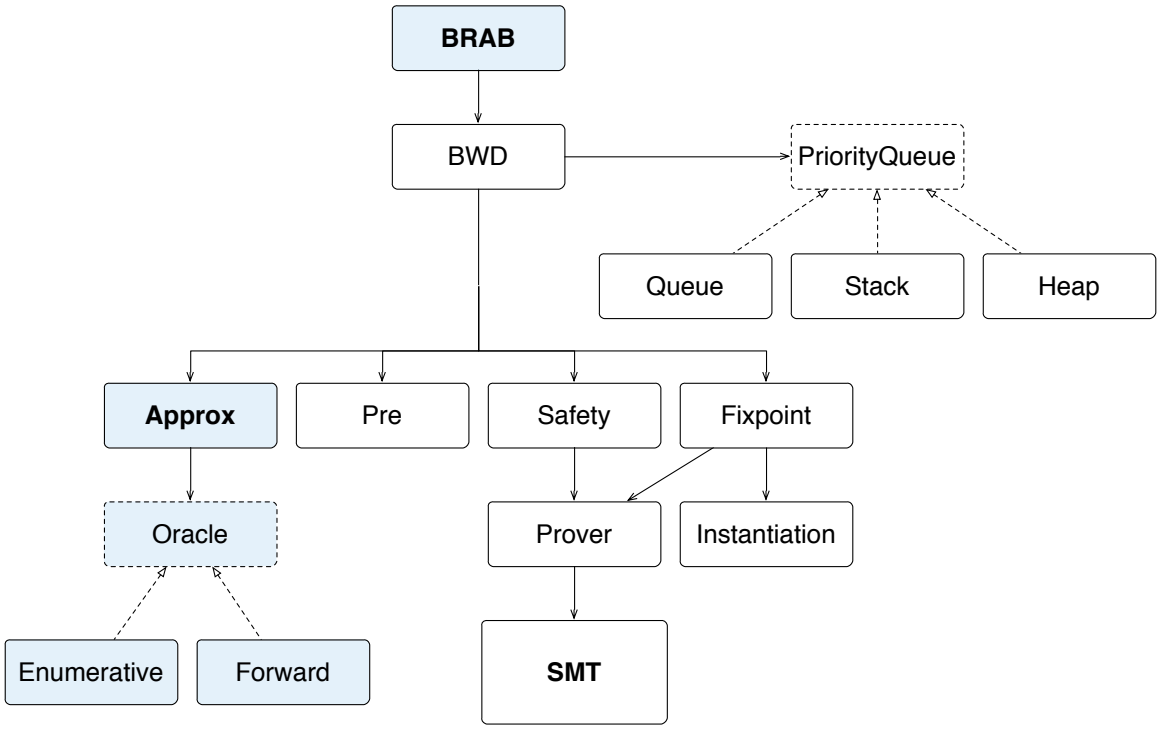



FIGURE 5.5 – Architecture de Cubicle avec BRAB

***brab.ml***

```
module BWD = Bwd.Selected
module Oracle = Approx.SelectedOracle

let rec search_and_backtrack candidates system =
  let res = BWD.search ~candidates system in
  match res with
  | Bwd.Safe _ -> res
  | Bwd.Unsafe (faulty, candidates) ->
     let o = Node.origin faulty in
     if o.kind = Orig then res
     else  (* Redémarrage *)
       let candidates = Approx.learn_bad system faulty candidates in
       search_and_backtrack candidates system

let brab system =
  Oracle.init system;
  search_and_backtrack [] system
```

La fonction `Approx.learn_bad` fonctionne par effets de bord. Elle implémente l'heuristique de la section 5.2.3 pour enrichir l'ensemble de mauvaises approximations $\mathcal{B}$, et accroître la connaissance de l'oracle. Elle supprime au passage certaines approximations de la liste `candidates`. Le module Approx de la figure 5.5 fournit un foncteur (`Make`) paramétré par un module Oracle. Le module implémente une fonction `good` qui prend en argument un nœud (*i.e.* un cube riche) et retourne une « bonne » approximation si elle existe.

***approx.mli***

```
val learn_bad : t_system -> Node.t -> Node.t list -> Node.t list

module type S = sig
    val good : Node.t -> Node.t option
end

module Make ( O : Oracle.S ) : S
```

Cette architecture permet d'utiliser différents types d'oracles du moment qu'ils implémentent la signature suivante :

***oracle.mli***

```
module type S = sig  (** Interface for oracles ... *)
    val init : t_system -> unit
    val first_good_candidate : Node.t list -> Node.t option
end
```

Dans Cubicle, on fournit deux oracles : une exploration de l'espace d'état énumérative (module Enumerative) et une exploration symbolique (module Forward) qui manipule des formules et fait appel au solveur SMT. Par défaut c'est l'exploration énumérative qui est utilisée car elle est plus efficace même s'il arrive qu'elle soit moins précise. La fonction `init` peut n'avoir aucun effet si l'oracle n'a pas d'état interne. Ensuite un oracle doit fournir une fonction de filtrage `first_good_candidate` qui, étant donné une liste de candidats, renvoie le premier ce ceux-ci qui lui paraît bon. L'oracle Enumerative parcourt cette liste et pour chacun de ces éléments, elle cherche dans $\mathcal{M}_k$ (le modèle construit avec la fonction `init`) un état qui le satisfait. Si aucun n'est trouvé, le candidat est considéré comme « bon ». Sinon on élimine de la liste les candidats qui sont satisfaits par le même état et on recommence. Il arrive souvent qu'un état satisfaisant une approximation en satisfasse d'autres car ces formules sont généralement syntaxiquement très proches (certaines sont même impliquées par d'autres).

## 5.3 Évaluation expérimentale

Nous avons évalué les performances de BRAB implémenté dans le model checker Cubicle sur un ensemble de benchmarks qui présente un intérêt et un défi pour la vérification paramétrée. On a également sélectionné quelques-uns des algorithmes et protocoles vérifiés par Cubicle dans la section 4.6 pour avoir une référence de comparaison avec la version sans génération d'invariants.

On donne en figure 5.6 les résultats obtenus pour BRAB sur cette batterie d'exemples. Pour évaluer l'intérêt de notre approche, on compare ces résultats avec différents model checkers pour systèmes paramétrés. On a utilisé la version 2.5 du model checker MCMT [78] ainsi que les dernières versions disponibles de PFS [88] qui implémente le cadre de [2], et Undip [143] qui implémente les techniques développées dans [3]. On compare également ces résultats avec les temps d'exécution obtenus pour le model checker énumératif Mur$\varphi$. On utilise une distribution récente de Mur$\varphi$ 3.1 fournie par la version 5.4.9 de CMurphi [121, 136] [1]. Pour ce dernier on donne les résultats pour différentes valeurs du nombre de processus (spécifié entre parenthèses après chaque temps). La dernière colonne (sauf pour l'exemple Flash_full) correspond au nombre maximum de processus pour lequel CMurphi donne une réponse. En plus de ces outils, on effectue aussi pour référence une comparaison avec Cubicle sans l'algorithme BRAB.

Tous les résultats présentés dans le tableau de la figure 5.6 ont été obtenus sur une machine 64 bits avec un processeur quadri-cœurs Intel® Xeon® cadencé à 3,2 GHz et comportant 24 Go de mémoire. Pour chaque exécution on affecte 20 Go de mémoire à l'outil et on donne un temps imparti de 20 heures. On note T.O. lorsque l'outil ne termine pas à temps et par O.M. lorsque son utilisation mémoire dépasse la quantité autorisée. On dénote par « / » les exemples qu'on n'a pu facilement traduire à cause de restrictions syntaxiques.Par exemple PFS n'autorise pas les actions globales qui modifient les valeurs de variables dans plusieurs processus à la fois. Undip interdit de stocker des identificateurs de processus dans des variables, MCMT ne supporte pas les tableaux multidimensionnels [2], et Mur$\varphi$ énumère les états donc ne peut manipuler d'entiers non bornés (l'horloge de Ricart_Agrawala).

Les exemples utilisés sont les suivants. Szymanski_at est une version atomique de l'algorithme d'exclusion mutuelle de Szymanski [157]. Szymanski_na est une variante de cet algorithme avec une évaluation des conditions globales *non atomique*. On a utilisé pour cela l'encodage présenté en section 2.3 afin d'obtenir un système de transition dans le

1. On n'utilise pas l'optimisation de cache sur disque fourni par CMurphi mais seulement les fonctionnalités de la version originale de Mur$\varphi$.

2. Il est tout de même possible d'encoder les tableaux multidimensionnels représentant des canaux dans MCMT mais cela ne suffit pas pour représenter les compteurs locaux utilisés par Szymanski_na. Une possibilité qui n'a pas été explorée ici est d'utiliser le support plus poussé de MCMT pour les entiers afin de modéliser les compteurs comme tels.

| | *Paramétré* | | | | | *Énumératif* | | |
|---|---|---|---|---|---|---|---|---|
| | **BRAB** | **Cubicle** | **MCMT** | **PFS** | **Undip** | **CMurphi** | | |
| Szymanski_at | **0,04s**<br>I : 12<br>R : 0 | 4m41s | 0,24s | T.O. | 32,1s | 8,04s (8) | 5m12s (10) | 2h50m (12) |
| Szymanski_na | **0,05s**<br>I : 13<br>R : 0 | T.O. | / | / | / | 0,88s (4) | 8m25s (6) | 7h08m (8) |
| Ricart_Agrawala | **0,05s**<br>I : 14<br>R : 0 | 5,01s | 1m10s | / | 4,17s | / | / | / |
| German_Baukus | **0,08s**<br>I : 22<br>R : 0 | 5,06s | 33m15s | / | 9m43s | 0,74s (4) | 19m35s (8) | 4h49m (10) |
| German.CTC | **0,11s**<br>I : 23<br>R : 0 | 4,58s | T.O. | / | / | 1,83s (4) | 43m46s (8) | 12h35m (10) |
| German_pfs | **0,09s**<br>I : 37<br>R : 0 | 2m45s | 6m47s | 36m05s | T.O. | 0,99s (4) | 22m56s (8) | 5h30m (10) |
| Chandra-Toueg | **54,1s**<br>I : 4<br>R : 1 | T.O. | 4m34s | / | / | 5,68s (4) | 2m58s (5) | 1h36m (6) |
| Flash_nodata | **0,10s**<br>I : 31<br>R : 0 | O.M. | T.O. | / | / | 4,86s (3) | 3m33s (4) | 2h46m (5) |
| Flash_full | **1m30s**<br>I : 109<br>R : 0 | O.M. | T.O. | / | / | 1m27s (3) | 2h15m (4) | O.M. (5) |

FIGURE 5.6 – Résultats de BRAB sur un ensemble de benchmarks

langage de Cubicle. Les différentes versions du protocole de German sont identiques aux exemples utilisés en section 4.6. L'algorithme Chandra-Toueg est un protocole de tolérance aux pannes (transitoires) conçu par Chandra et Toueg [30]. Notre modélisation est une adaptation de la version avec pannes par omissions (c'est la version complète dans laquelle on demande aussi de prouver les lemmes intermédiaires) utilisée dans [8]. Enfin les deux dernières lignes concernent le protocole de cohérence utilisé dans l'architecture multi-processeurs FLASH [106]. La version Flash_nodata élimine du protocole la partie concernant les données, tandis que la version Flash_full constitue une modélisation fidèle du protocole original [33].

Les résultat présenté dans ce tableau correspondent aux meilleurs temps obtenus [3] pour chaque outil. On lance Cubicle avec l'option `-search bfsh` et on utilise l'option `-brab 2` dans la colonne BRAB (sauf pour Flash_nodata auquel on ajoute `-forward-depth 6` et Flash_full pour lequel on utilise les options `-brab 3 -forward-depth 14`). L'option `-brab` $n$ active l'algorithme BRAB dans Cubicle avec un oracle qui fait une exploration énumérative pour $n$ processus. L'option `-forward-depth` limite quant à elle la profondeur de cette exploration au nombre qui lui est passé en argument. Pour la colonne BRAB, on rapporte en plus le nombre d'invariants inférés (I) ainsi que le nombre de retours en arrière (R). Par exemple le protocole German.CTC est prouvé en moins d'un dixième de seconde par BRAB pour un nombre arbitraire de processus en découvrant 23 invariants. Sur le même exemple, un model checker énumératif comme Mur$\varphi$ a déjà besoin de plus d'une seconde pour quatre processus. Ce dernier n'est pas capable de vérifier German.CTC pour plus de dix processus. Notons que BRAB a besoin de faire un retour en arrière sur Chandra-Toueg à cause d'un mauvais candidat invariant réfuté sur une trace à trois processus (les autres version du protocole décrite dans [8] sont aussi prouvées par BRAB). C'est tout de même le meilleur temps qu'on obtient. En résumé, on peut conclure que Cubicle avec BRAB est plus rapide que ses concurrents de plusieurs ordres de grandeurs sur quasiment tous les exemples présentés. Notamment il est le seul capable de prouver la sûreté du protocole FLASH. La vérification de la partie contrôle (Flash_nodata) est quasiment instantanée et la preuve totale est accomplie en seulement 1m30s alors que Mur$\varphi$ ne peut vérifier ces exemples au-delà de cinq et quatre processus respectivement. Ce résultat impressionnant est possible grâce à la qualité des invariants découverts par BRAB. Ces mêmes invariants ont parfois étés ajoutés à la main dans d'autres méthodes. La section suivante détaille cet exemple en particulier et explique comment la vérification automatique est possible.

La force de notre approche réside dans deux aspects. Premièrement, le fait de considérer les approximations toutes ensembles dans la procédure de model checking permet à la preuve d'une approximation de bénéficier du travail déjà effectué pour la preuve d'autres approximations. La deuxième idée est que les instances finies du système (même petites) sont généralement de bons oracles pour guider le choix des approximations car ils peuvent

3. Il peut toutefois exister une combinaison d'options qui donne de meilleurs résultats.

être vus comme une source de connaissance concentrée du système.

## 5.4 Étude de cas : Le protocole FLASH

Comme nous l'avons présenté dans la section précédente, Cubicle (avec BRAB) permet de prouver la sûreté du protocole FLASH. C'est un exemple intéressant car ce protocole de cohérence de cache est particulièrement gros et complexe. À notre connaissance, cette preuve du FLASH est la première qui soit *entièrement automatique*. Nous revenons sur la modélisation et la preuve de ce protocole dans cette section, en particulier nous montrons la qualité des invariants découverts par Cubicle.

Le multi-processeur Stanford FLASH [106] (pour ***FL**exible **A**rchitecture for **SH**ared memory*) est une architecture modulaire conçue pour fonctionner avec plusieurs milliers (4096) de cœurs de calcul. Chaque processeur maintient un cache mémoire local dont la cohérence est assurée par un protocole de transmission de messages sur un réseau point à point de latence arbitraire.

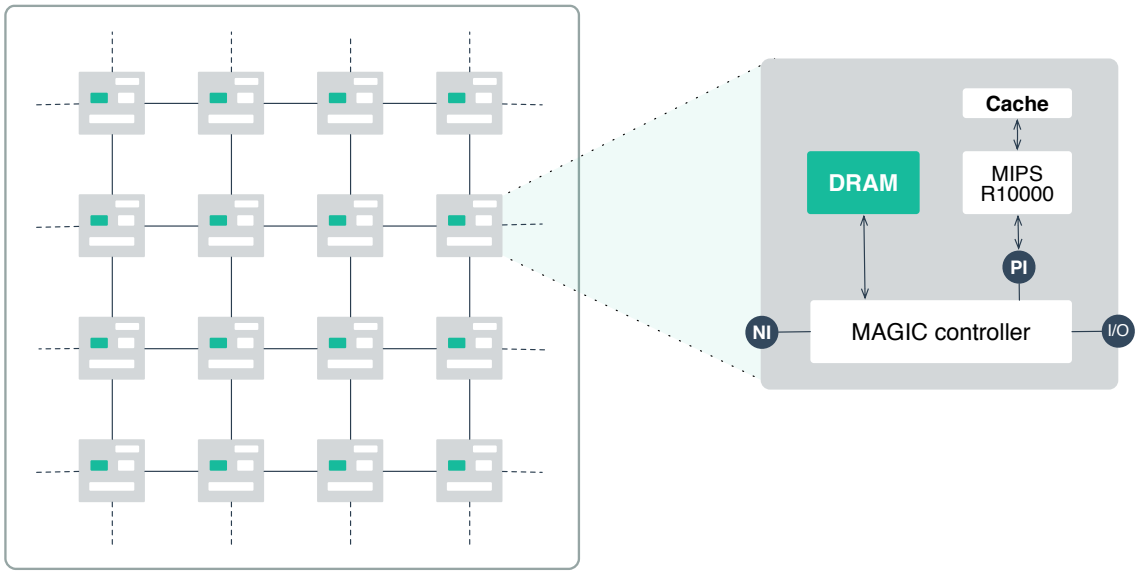



FIGURE 5.7 – Architecture FLASH d'une machine et d'un nœud [106]

Chaque nœud est composé d'un processeur MIPS R10000 avec son cache, d'une partie de la mémoire principale (physiquement distribuée) de la machine, ainsi que d'un contrôleur appelé MAGIC (pour ***M**emory **A**nd **G**eneral **I**nterconnect **C**ontroller*) [87]. Ce contrôleur est programmable, ce qui fait de FLASH une architecture flexible. C'est lui qui exécute les différents protocoles de communication, notamment le protocole de cohérence de cache.

Chaque nœud du FLASH est responsable d'une partie (ligne) de la mémoire de la machine. On appelle *Home* le nœud correspondant à l'emplacement physique d'une ligne de mémoire. Le contrôleur du responsable maintient un *répertoire* pour mémoriser des informations sur l'état du protocole. Comme pour le petit exemple en section 2.2.4, le cache d'un processeur contient une copie d'une ligne mémoire et peut être dans trois états possibles : Invalid, Shared (lisible) ou Exclusive (lisible et modifiable). Le contrôleur communique avec les autres nœuds du système par échange de messages au travers de son interface NI (pour ***N**etwork **I**nterface*, voir Figure 5.7). L'interface PI (pour ***P**rocessor **I**nterface*) permet au contrôleur MAGIC de communiquer avec le processeur R10000.

### 5.4.1 Description du protocole FLASH

On donne ici, une description succinte du protocole de cohérence de FLASH extraite de [134] et [106]. Pour chaque ligne de mémoire, un répertoire maintient un certain nombre d'informations, parmi lesquelles plusieurs variables booléennes (drapeaux) :

| Drapeau | Description |
|---|---|
| **Local** | Indique si le processeur local (*i.e.* dans le nœud *Home*) contient une copie de la ligne dans son cache. |
| **Dirty** | Indique si le *Home* a connaissance de l'existence d'une copie modifiée dans le système. |
| **Pending** | Indique si une requête pour l'accès à la mémoire est en train d'être traitée par un nœud distant. |
| **HeadVld** | Indique si la variable **HeadPtr** contient un pointeur valide vers un des nœuds du système. |

On dénombre aussi plusieurs pointeurs vers des nœuds du système. L'implémentation du protocole dépend principalement des structures de données choisies pour maintenir ces informations dans le répertoire.

| Pointeur(s) | Description |
|---|---|
| **HeadPtr** | Pointe vers le nœud qui contient la copie exclusive de la ligne s'il existe, sinon pointe vers un nœud ayant une copie partagée. |
| **ShrSet** | Un ensemble de pointeurs vers les nœuds du système qui contiennent une copie de la ligne en cache (implémenté par une liste chaînée en pratique). |
| **InvSet** | L'ensemble des pointeurs vers les nœuds du système dont les caches doivent être invalidés et qui n'ont pas encore confirmé l'invalidation (peut aussi être vu / implémenté comme un compteur). |

Le réseau de communication fait parvenir des messages entre les différents nœuds du système. Un message se compose de trois parties : une commande, l'identifiant du nœud destinataire, et éventuellement une donnée associée (*cf.* Figure 5.8). Les commandes

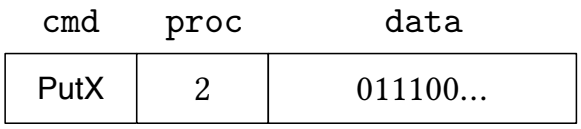



FIGURE 5.8 – Structure d'un message dans le protocole FLASH

possibles des messages sont récapitulées dans le tableau suivant.

| **Commande** | **Description** |
|---|---|
| None | Absence de message. |
| Get | Défaut de cache en lecture. |
| GetX | Défaut de cache en écriture. |
| Put | Réponse avec données pour un défaut de cache en lecture. |
| PutX | Réponse avec données pour un défaut de cache en écriture. |
| Nak | Le système n'a pas pu satisfaire la requête (***N**egative **ac**knowledgement*). |
| Inv | Invalidation du cache d'un processeur (***Inv**alidate*). |
| InvAck | Réponse pour l'invalidation (***Inv**alidate **Ack**nowledgement*) |
| Replace | Indication de réinitialisation. |
| Wb | Écriture en mémoire (***W**rite**b**ack*) |
| ShWb | Écriture en mémoire d'une ligne partagée (***Sh**ared **W**rite**b**ack*) |
| FAck | Réponse après transmission de message (***F**orward **Ack**nowledgement*) |
| Nakc | Réponse au message Nak après interruption de la commande en cours (***N**egative **ac**knowledgement **c**lear*) |

Chaque processeur gère son propre cache. Celui-ci contient potentiellement une copie de la ligne mémoire stockée dans la variable **CacheData**. Son état est représenté par la variable **CacheState** qui peut prendre trois valeurs :

— CACHE_I : Le cache est *invalide*, la copie n'est plus à jour et ne peut être ni lue, ni modifiée (***I**nvalid*).

— CACHE_S : Le cache contient une copie *partagée*, le processeur a donc un accès en lecture (***S**hared*).

— CACHE_E : Le cache contient une copie *exclusive*, le processeur peut écrire dans son cache (***E**xclusive*).

Le protocole de cohérence consiste en un ensemble de règles appelées des gestionnaires. Ces règles peuvent aussi être vue comme des transitions dont la description est donnée en figure 5.9. Le préfixe **PI** d'une règle indique que la requête est initiée sur l'interface processeur du contrôleur (voir Figure 5.7). Le préfixe **NI** dénote les règles s'appliquant aux requêtes générées sur le réseau. On identifie également la localité (du point de vue du nœud *Home*) du traitement de la requête à l'aide d'un second préfixe : **Local** lorsqu'il s'agit du *Home* et **Remote** pour un nœud distant.

Enfin pour pouvoir exprimer la propriété de cohérence des données dans le système, on ajoute plusieurs variables auxiliaires au protocole afin de garder trace de certaines actions.

| Variable auxiliaire | Description |
|---|---|
| **Collecting** | Indique si la requête d'accès par un nœud distant (telle que **Pending** = true) a envoyé des invalidations. |
| **CurrData** | Dernière donnée écrite par le système (dernière modification d'une copie exclusive). |
| **PrevData** | Avant-dernière donnée écrite par le système. |

*Remarque.* Le protocole FLASH supporte deux modes de fonctionnement selon le modèle mémoire désiré : le mode *impatient* (ou *eager*) et le mode *différé* (ou *delayed*). Le mode *impatient* est plus permissif car lors d'un défaut de cache en écriture, il autorise l'envoi du message contenant les données avant d'avoir recueilli toutes les réponses aux invalidations. Le mode *différé* est moins agressif. On peut prouver des propriétés plus fortes sur ce dernier comme la *cohérence séquentielle* [134], c'est à dire que l'observation de l'ordre des lectures et écritures en mémoire globale correspond aux observations faites par les processeurs sur leur cache local [111]. Le mode *impatient* ne garantit pas cette propriété mais assure qu'un nœud verra la dernière modification lorsque la dernière requête aura été traitée. Dans la suite de cette section on ne s'intéressera qu'au mode *impatient* du FLASH.

### 5.4.2 Vérification du FLASH : État de l'art

La première preuve de la sûreté du protocole de cohérence de FLASH à été réalisée par Park et Dill en 1996 à l'aide de l'assistant de preuve PVS [134, 135]. Cette approche nécessite de construire des invariants inductifs à la main et de détailler toutes les étapes de la preuve dans l'assistant. Ce protocole a aussi été vérifié par Das, Dill et Park en utilisant une technique d'abstraction par prédicats manuellement guidée [49].

La méthode de model checking compositionnel combinée à une approche de réduction de types de données (*data type reduction*) développée par McMillan a aussi été utilisée

| Préfixe transition | Description |
|---|---|
| **PI_Local_Get** | Action du *Home* quand le processeur local demande une copie *partagée* de la mémoire (défaut de cache en lecture). |
| **PI_Local_GetX** | Action du *Home* quand le processeur local demande une copie *exclusive* de la mémoire (défaut de cache en écriture). |
| **PI_Remote_Get** | Action d'un nœud distant quand son processeur demande une copie *partagée* (défaut de cache en lecture). |
| **PI_Remote_GetX** | Action d'un nœud distant quand son processeur demande une copie *exclusive* (défaut de cache en écriture). |
| **PI_Local_PutX** | Écriture en mémoire d'une copie exclusive en cache se trouvant dans le *Home*. |
| **PI_Remote_PutX** | Écriture en mémoire d'une copie exclusive en cache se trouvant dans un nœud distant. |
| **PI_Local_Replace** | Réinitialisation d'une copie partagée dans le *Home*. |
| **PI_Remote_Replace** | Réinitialisation d'une copie partagée par un nœud distant. |
| **NI_Nak** | Indique à un processeur que sa requête n'a pu être traitée (***N**egative **A**c**K**nowledgement*). |
| **NI_Nak_Clear** | Indique au *Home* que les réponses NAK ont bien été transmises aux nœuds concernés. |
| **NI_Local_Get** | Action du *Home* quand il reçoit une requête Get venant d'un processeur distant. |
| **NI_Local_GetX** | Action du *Home* quand il reçoit une requête GetX venant d'un processeur distant. |
| **NI_Remote_Get** | Action d'un nœud recevant une requête Get en provenance d'un autre processeur distant. |
| **NI_Remote_GetX** | Action d'un nœud recevant une requête GetX en provenance d'un autre processeur distant. |
| **NI_Local_Put** | Le *Home* traite une réponse Put qui lui est faite. |
| **NI_Local_PutX** | Le *Home* traite une réponse PutX qui lui est faite. |
| **NI_Remote_Put** | Un nœud distant reçoit une réponse Put qui l'autorise à copier le contenu de la mémoire dans son cache. |
| **NI_Remote_PutX** | Un nœud distant reçoit une réponse PutX qui lui donne une copie exclusive de la mémoire. |
| **NI_Inv** | Un nœud distant reçoit une commande INV qui le force à invalider son cache. |
| **NI_InvAck** | Action du *Home* lorsqu'un nœud distant l'informe qu'il a traité la commande d'invalidation lui étant destinée. |
| **NI_FAck** | Le *Home* reçoit le message FAck lui indiquant que la copie exclusive a bien été transmise d'un nœud distant à un autre. |
| **NI_Wb** | Le *Home* écrit la copie d'un nœud distant dans la mémoire. |
| **NI_ShWb** | Le *Home* reçoit le message ShWb lorsqu'un nœud distant avait une copie exclusive qu'il a transmis à un nœud demandant un accès en lecture. La copie est écrite dans la mémoire au passage. |
| **NI_Replace** | Le *Home* reçoit une demande de réinitialisation. |

FIGURE 5.9 – Description des transitions du protocole FLASH

avec succès dans le model checker symbolique SMV [118]. Cette approche fut ensuite adaptée et étendue par Chou, Mannava et Park dans le cadre du model checking énumératif en 2004 [33]. Cette dernière métode, appelée CMP a été formalisée l'année plus tard par Krstić [104]. Son principe fondamental consiste en une boucle de raffinement qui abstrait le protocole puis renforce les invariants et gardes du système en se reposant sur l'analyse de contre-exemples fournis par le model checker. À ce jour c'est la méthode qui passe le mieux à l'échelle mais elle demande une quantité considérable d'expertise et d'intervention manuelle pour imaginer et concevoir des *lemmes de non-interférence* à partir des contre-exemples.

En 2008, Talupur et Tuttle eurent l'idée d'utiliser les diagrammes de flux de messages fabriqués par les concepteurs de processeurs et protocoles comme une source d'invariants pour aider la méthode CMP à converger plus vite [130, 158]. Bien que leur méthode soit capable d'inférer des invariants de manière automatique à partir des diagrammes de flux de messages, il reste tout de même nécessaire d'ajouter des lemmes de non-interférence fabriqués à la main pour que CMP converge sur les protocoles German et FLASH. En particulier il n'est pas possible d'extraire des invariants mettant en relation l'état local d'un processeur avec le contenu du répertoire.

### 5.4.3 Modélisation dans Cubicle

La modélisation que nous avons faite dans le langage d'entrée de Cubicle a été adaptée des versions écrites pour le model checker Mur$\varphi$ par Ching-Tsun Chou et utilisées dans [33,158]. Notre modélisation ne prend en compte qu'une seule ligne de mémoire car les propriétés sont facilement généralisables à un nombre arbitraire d'adresses mémoire.

Dans Cubicle on ne peut pas définir de constante de processus, alors pour modéliser le nœud *Home* de la ligne mémoire en question et ne pas perdre la symétrie qui existe entre les processus, on utilise une variable `Home` de type `proc` initialisée à une valeur différente des autres processus (*i.e.* $I \implies \forall z : \texttt{proc}.\ \texttt{Home} \neq z$). On crée alors pour chaque tableau `A`, une variable globale `A_home` matérialisant la valeur de `A[Home]`.

Une deuxième différence mineure est que les structures d'enregistrements utilisées pour modéliser les messages, caches, répertoires, *etc.* sont mises à plat car Cubicle n'a pas de syntaxe prévue à cet effet. Il n'y a pas de perte d'expressivité, et on peut par exemple écrire `UniMsg_Cmd[`$src$`]` = UNI_Get pour l'accès au champ `cmd` de la structure contenue à l'indice $src$ du tableau `UniMsg`, au lieu de `UniMsg[`$src$`].cmd` = UNI_Get.

Les propriétés de sûreté qu'on souhaite vérifier sur le protocole FLASH sont de deux natures. La propriété d'exclusion mutuelle concerne la partie contrôle du protocole et dit qu'un seul processeur peut avoir une copie exclusive de la ligne mémoire dans son cache :

$$\forall x, y.\ x \neq y \wedge \texttt{CacheState}[x] = \text{CACHE_E} \implies \texttt{CacheState}[y] \neq \text{CACHE_E}$$

La propriété de cohérence concerne la partie « données » du protocole. La première stipule qu'un processeur possédant une copie exclusive voit toujours la dernière modification faite sur les données. Les deux autres propriétés assurent qu'un processeur possédant une copie partagée voit soit l'avant dernière écriture si des invalidations sont en cours, soit la dernière écriture.

$$\forall x.\ \texttt{CacheState}[x] = \texttt{CACHE_E} \implies \texttt{CacheData}[x] = \texttt{Currdata}$$

$$\forall x.\ \texttt{CacheState}[x] = \texttt{CACHE_S} \wedge \texttt{Collecting} \implies \texttt{CacheData}[x] = \texttt{PrevData}$$

$$\forall x.\ \texttt{CacheState}[x] = \texttt{CACHE_S} \wedge \neg\texttt{Collecting} \implies \texttt{CacheData}[x] = \texttt{CurrData}$$

### Encodage des données

On a modélisé les données de la mémoire comme un paramètre du système. Dans cette version on utilise un tableau `Sort` qui associe à chaque élément du type `proc` un des deux constructeurs Proc ou Data. Proc lorsque la variable représente un identificateur de processeur, et Data lorsqu'il s'agit d'une donnée. Cet encodage des types par un prédicat permet de s'affranchir du fait que Cubicle n'a qu'un seul type paramétré.

```
type sort = Proc | Data
array Sort[proc] : sort

transition store (src d)
requires { Sort[src] = Proc && Sort[d] = Data &&
           CacheState[src] = CACHE_E }
{ CacheData[src] := d;
  CurrData := d; }
```

Un autre encodage possible consiste à prendre les données dans un type *abstrait* ce qui permet d'éviter la lourdeur de la précédente approche. On est alors obligé d'utiliser une variable temporaire pour modéliser la modification arbitraire du cache :

```
type data (* type abstrait *)
var TempData : data

transition store (src)
requires { CacheState[src] = CACHE_E }
{ CacheData[src] := TempData;
  CurrData := TempData;
  TempData := ?; (* modification non déterministe *) }
```

L'avantage de ces encodages est qu'ils nous permettent de vérifier le protocole indépendamment de la représentation et du nombre des données. Dans les approches classiques (même paramétrées), on suppose que les données consistent en un seul bit, c'est-à-dire qu'une donnée peut prendre deux valeurs en tout et pour tout. En pratique, cette supposition n'est pas très dérangeante car on peut s'apercevoir, à l'aide d'un argument méta, que la validité des propriétés de ce protocole ne dépend pas du nombre de données. Notre encodage reste toutefois plus général.

Au total le fichier Cubicle correspondant comporte 1 456 lignes pour 36 ko. Il mentionne 31 variables globales et 11 tableaux de processus (unidimensionnels) et utilise 24 types de messages différents. La description de la relation de transition comporte 71 transitions.

### 5.4.4 Résultats

Pour vérifier ces propriétés de sûreté, on a utilisé Cubicle avec l'algorithme BRAB. L'oracle qu'on utilise ici est une exploration avant de l'espace d'état, limitée en profondeur. On détaille les résultats de l'exécution de Cubicle sur différentes versions en figure 5.10. Les temps donnés ont été obtenus sur la même machine 64 bits avec un processeur quadricœurs Intel® Xeon® cadencé à 3,2 GHz et comportant 24 Go de mémoire. La ligne nodata correspond à la preuve de la seule propriété de contrôle du protocole FLASH. Pour la version full, on demande à vérifier à la fois les propriétés de contrôle et de données. Pour cette dernière le type des données est aussi paramétré. La modélisation abstr est identique mais les données sont d'un type abstrait. Enfin on introduit une erreur dans le modèle de la ligne buggy. Sur cette dernière version Cubicle expose une trace d'erreur de 11 transitions.

On donne pour l'oracle, le nombre de processus ($k$) pour construire le modèle $\mathcal{M}_k$ dont on précise le nombre d'états. Pour la boucle d'atteignabilité arrière, on donne le nombre de nœuds visités par Cubicle ($|\mathcal{V}|$), le nombre d'invariants découverts (#inv), ainsi que le nombre de retours en arrière (correspondant à la taille de l'ensemble $\mathcal{B}$).

La preuve de la propriété de contrôle (nodata) est quasiment instantanée et nécessite un modèle initial pour l'oracle avec seulement deux processus. La preuve complète demande quant à elle que le modèle de l'oracle ait au moins trois processus, sinon on introduit des mauvais candidats invariants qui sont réfutés par l'analyse d'atteignabilité arrière sur des traces faisant intervenir trois processus. On peut notamment remarquer que les encodages des données dans full et abstr (voir section 5.4.3) donnent des résultats à peu près similaires, avec un léger avantage pour la version ou les données sont encodées par un type abstrait. Ceci est principalement dû à la taille plus réduite du modèle construit par l'oracle.

Signalons que le surcoût introduit par l'oracle est ici tout à fait raisonnable. Le temps nécessaire à la construction du modèle fini ajouté au temps utilisé par l'oracle (respectivement partie vert clair – *exploration avant*, et vert foncé – *filtre oracle* sur les graphiques de répartition) s'élève à moins d'un quart du temps total pour chacun de ces exemples. La majeure partie du temps est consacrée aux appels au solveur SMT (construction des

| Version | Oracle | | | | Atteignabilité | | | Temps | |
|---|---|---|---|---|---|---|---|---|---|
| | $k$ | $d_{\max}$ | $\lvert\mathcal{M}_k\rvert$ | temps | $\lvert\mathcal{V}\rvert$ | #inv | $\lvert\mathcal{B}\rvert$ | répartition | total |
| **nodata** | 2 | 6 | 439 | 0,03s | 41 | 31 | 0 | | **0,10s** |
| **buggy** | 2 | 12 | 13 389 | 0.15s | 297 | / | 0 | | **1,40s** |
| **full** | 3 | 14 | 452 523 | 7,05s | 745 | 109 | 0 | | **1m30s** |
| **abstr** | 3 | 14 | 285 045 | 4,56s | 741 | 119 | 0 | | **1m06s** |

FIGURE 5.10 – Résultats pour la vérification du protocole FLASH avec Cubicle

formules et appels SMT) dans les version complètes full et abstr.

Le procédé est entièrement automatique et permet à Cubicle de découvrir 109 invariants pour la version avec données. Comme la recherche d'invariants est guidée par le but, BRAB n'utilise que des invariants qui lui permettent de faciliter sa recherche et de progresser dans la preuve. Par exemple, la plupart des 109 invariants de la version full sont aussi vrai pour la version nodata mais Cubicle n'a besoin que de 41 d'entre eux pour la propriété de contrôle. Les invariants découverts sont loin d'être triviaux, en particulier on peut retrouver parmi ceux-ci une partie des lemmes de non-interférence conçus manuellement et utilisés dans la méthode CMP [33]. Un extrait de ces invariants est présenté ci-dessous :

- $\neg$`InvMarked_home`
- $\neg$`Dir_ShrSet_home`
- `CacheState_home` = CACHE_S $\implies$ $\neg$`Collecting`
- $\forall x$. `CacheState`$[x]$ = CACHE_E $\implies$ (`Dir_Dirty` $\wedge$ `UniMsg_Cmd`$[x]$ $\neq$ UNI_Put)

*(partie du Lemme 1 de [33])*

— $\forall x, y.\ x \neq y \ \wedge$ Home $\neq y \ \wedge$ CacheState$[x]$ = CACHE_E $\implies$ UniMsg_Cmd$[y] \neq$ UNI_PutX

*(partie du Lemme 1 de [33])*

— UniMsg_Cmd_home = UNI_Get $\vee$ UniMsg_Cmd_home = UNI_GetX $\implies$ ($\neg$Collecting $\wedge$ Dir_Pending $\wedge$ ShWbMsg_Cmd $\neq$ SHWB_FAck $\wedge$ ShWbMsg_Cmd $\neq$ SHWB_ShWb)

*(partie des Lemmes 2 et 3 de [33])*

— $\forall x.$ Home $\neq x \ \wedge$ UniMsg_Cmd_home = UNI_GetX $\implies$ InvMsg_Cmd$[x] \neq$ INV_Inv

— $\forall x.$ Home $\neq x \ \wedge$ UniMsg_Cmd$[x]$ = UNI_PutX $\implies \neg$Dir_ShrVld

— $\forall x.$ Home $\neq x \ \wedge$ InvMsg_Cmd$[x]$ = INV_InvAck $\implies$ (Dir_Pending $\wedge$ CacheState$[x]$ = CACHE_I $\wedge$ UniMsg_Cmd$[x] \neq$ UNI_PutX $\wedge$ UniMsg_Cmd_home $\neq$ UNI_Put)

*(partie du Lemme 4 de [33])*

— $\forall x, y.\ x \neq y \wedge$ Home $\neq y \ \wedge \neg$Dir_Pending $\wedge \neg$Dir_ShrVld $\wedge$ CacheState$[x]$ = CACHE_S $\implies$ CacheState$[y] \neq$ CACHE_S

## 5.5 Travaux connexes sur les invariants

### 5.5.1 Génération d'invariants

La connaissance apportée par les invariants est très précieuse. La réussite d'une analyse, que ce soit par model checking ou vérification déductive, dépend bien souvent de ces derniers. Il existe plusieurs façons d'obtenir ces informations convoitées :

— Données par l'utilisateur : lorsqu'un humain exprime une propriété ou restreint une spécification abstraite à l'aide d'un invariant, un outil de vérification peut utiliser cette information. La confiance est alors entièrement placée sur l'utilisateur donc cette solution est rarement satisfaisante.

— Suggérées par l'utilisateur : lorsqu'une propriété du système est connue par les concepteurs, il est possible de suggérer un invariant à l'analyse. Cet invariant est ensuite vérifié par le système et utilisé pour mener à bien sa tâche de vérification. C'est le cas des invariants de boucles de la vérification déductive par exemple. Un désavantage conséquent et récurrent de ces techniques est que les invariants nécessaires doivent souvent être inductifs, sont très intrusifs, et demandent généralement de l'ingéniosité pour être exprimés.

— Inférées automatiquement : c'est bien sûr le cas le plus favorable pour l'utilisateur car il n'a alors rien à faire. Les propriétés devinées rendent certains outils d'analyse très

puissants mais les techniques de génération d'invariants ne s'appliquent pas dans tous les cas. Bien souvent, les formules sont inférées à partir d'un modèle général qui n'englobe pas toutes les propriétés fonctionnelles ce qui en fait des méthodes peu prospères dans le cadre de la vérification déductive. En revanche ces inconvénients sont moins problématiques pour les approches par model checking qui n'ont pas vocation à spécifier des programmes de manière fonctionnelle.

Ghilardi et Ranise décrivent dans [77] un algorithme de génération d'invariants pour les systèmes à tableaux. Tout comme notre approche, et c'est là la plus grande similarité avec nos travaux, la recherche d'invariants est guidée par le but. Les candidats sont extraits à partir des nœuds calculés par l'algorithme d'atteignabilité arrière et filtrés sur critères syntaxiques. Cependant cet algorithme calcule toujours des formules précises (non approximées). Lorsqu'un invariant potentiel est découvert, il est vérifié par une autre instance de l'algorithme d'atteignabilité arrière pour laquelle les ressources sont limitées (nombre de nœuds, profondeur, *etc.*) Pour pouvoir être utilisé, un invariant doit être prouvé par cette analyse limitée. La suite de l'analyse principale peut ensuite s'aider de l'invariant ainsi découvert pour ses tests de point-fixes. L'avantage de cette technique est que lorsque l'analyse secondaire répond de manière positive, on est certain que l'invariant en est un. En revanche, il arrive souvent que les invariants intéressants et utiles soient aussi difficiles à vérifier que la propriété originale. C'est le point faible de cette approche.

Dans la technique dites des *invariants de réseau*, on cherche à construire un processus $I$ qui constitue une abstraction du système à $n$ processus [105, 163]. C'est le processus $I$ (ou de manière analogue, l'ensemble des états de $I$) qui constitue l'invariant de réseau. La découverte de cet invariant nécessite de nombreux raffinements successifs. Grinchtein *et al.* proposent une méthode reposant sur l'apprentissage d'automates pour inférer cet invariant automatiquement [82] mais ne démontrent pas que leur technique passe à l'échelle sur des programmes réalistes.

Les travaux de Namjoshi *et al.* [41,126] sur les *invariants partagés* (*split*) exposent les liens qui existent entre les notions sous-jacentes aux propriétés de *petit modèle*, les méthodes inductives et le raisonnement compositionnel.

### 5.5.2 Cutoffs

Une intuition générale qui est à la base de la technique dite de *cutoff* est que la preuve de la sûreté d'un système avec un petit nombre de processus est suffisante pour l'analyse du cas paramétré. Ce n'est malheureusement pas tout le temps vrai mais l'expérience montre que c'est souvent le cas. Lorsqu'un tel nombre existe, on appelle sa valeur un *cutoff*.

Plus précisément, une valeur de *cutoff* $K$ existe pour un système et un ensemble d'état donnés si, lorsqu'un état de cet ensemble est atteignable dans un système avec $n > K$ processus, alors un état de cet ensemble est aussi atteignable dans un système de taille

$n \leq K$. Traditionnellement, les valeurs de *cutoff* lorsqu'elles existent sont calculées à partir des caractéristiques du système (variables, schémas de communications, *etc.*) de manière statique et sont souvent prohibitives pour les systèmes réalistes. L'utilisation des *cutoff* a rencontré le plus de succès dans les approches de détection dynamiques de ces valeurs. La technique de [98] ne s'applique qu'aux réseaux de pétri mais les travaux plus récents de [4] montrent que cette technique est prometteuse car elle peut traiter des systèmes de topologies différentes (tableaux, anneaux, arborescentes, multi-ensembles).

Les travaux les plus proches dans l'esprit de notre technique sont sans aucun doute ceux qui traitent de la méthode des *invariants invisibles* [12, 138]. Les deux approches reposent sur l'observation fondamentale que les instances avec peu de processus capturent déjà la majorité des comportements intéressants du système. La méthode des invariants invisibles vise à construire un invariant inductif du système paramétré dont la validité est vérifiée jusqu'à une certaine borne de *cutoff* dérivée à partir de critères syntaxiques. L'invariant est inféré à partir d'une exploration avant d'une instance finie du système, laquelle est généralisée en abstrayant certains processus. Un point crucial pour la réussite de la méthode est que cet invariant doit être inductif. On utilise également une exploration avant d'une instance finie mais seulement comme un oracle pour filtrer nos candidats invariants. De plus, à la fin de notre analyse l'invariant deviné avec l'oracle n'est pas forcément inductif. Même si dans l'exemple de la section 5.1.1, la formule $P_1 \wedge P_2 \wedge P_3 \wedge \neg\Theta$ est un invariant inductif du système paramétré, ce n'est généralement pas le cas. Par ailleurs, bien que la découverte des invariants invisibles soit entièrement automatique, le succès de la méthode demande parfois de rajouter certaines variables auxiliaires (c'est le cas pour le protocole German). Des instances finies du système paramétré sont aussi utilisées en conjonction avec un mécanisme de patrons pour obtenir des formules décrivant des comportements intéressants dans [65].

### 5.5.3 Abstraction

Abdulla *et al.* proposent différentes analyses d'atteignabilité par chaînage arrière en utilisant une abstraction de la relation de transition [2, 3]. L'approche adoptée dans le cadre du *model checking régulier abstrait* par Bouajjani *et al.* est plus proche de la notre et consiste à abstraire des automates à la volée en fusionnants certains de leurs états [23].

D'autres méthodes pour la vérification paramétrée sont fondées sur des techniques d'abstraction. La méthode des *prédicats indexés* [107] infère automatiquement des prédicats quantifiés à partir desquels la technique de l'*abstraction par prédicats* est capable de construire des invariants inductifs. Cette technique est implémentée par l'outil UCLID [28] qui est capable de vérifier la sûreté du protocole German de manière automatique en quelques minutes [108, 109] mais atteint ses limites sur les programmes plus compliqués comme le FLASH. L'idée sous-jacente à la technique d'*abstraction par compteurs* est de garder trace du nombre de processus qui satisfont une certaine propriété afin d'effectuer

des réductions par symétrie [64, 139] . La méthode d'*abstraction d'environnement* combine quant à elle les techniques d'abstraction par prédicats et une généralisation de l'abstraction par compteurs [40]. Contrairement à la plupart de ces approches, on n'abstrait pas le système original. Les abstractions sont locales à certaines parties de l'analyse et sont faites à la volée pendant la phase d'exploration arrière.

## 5.6 Conclusion

On a présenté dans ce chapitre un nouvel algorithme d'atteignabilité arrière pour systèmes paramétrés appelé BRAB qui constitue une des contributions majeures de cette thèse. Cet algorithme fonctionne par approximations des pré-images calculées à la volée et s'apparente à un mécanisme de génération d'invariants. Sa force réside dans l'utilisation d'un oracle qui concentre les connaissances des instances finies du système. Il fonctionne en pratique car généralement les instances finies, même avec un petit nombre de processus, exhibent déjà la plupart des comportements intéressants du système. BRAB est capable d'inférer des invariants de qualité utiles à la preuve des propriétés de sûreté. Il a été utilisé pour vérifier les propriétés de contrôle et de cohérence du protocole FLASH, une première pour la vérification automatique.

Pour l'instant, l'oracle utilisé par Cubicle est une phase de model checking énumératif. Comme la correction de BRAB ne dépend pas de l'oracle on pourrait imaginer différentes techniques pour ce dernier. Sa qualité principale doit être de pouvoir trouver des bugs rapidement car il est utilisé pour réfuter les candidats invariants qui lui sont présentés. Une technique non complète utilisée aujourd'hui dans l'industrie est le *model checking borné* [19] particulièrement efficace sur les circuits logiques, donc les extensions avec des logiques plus riches [11] semblent tout à fait appropriées à notre cadre. Certains problèmes peuvent toutefois rester inaccessibles aux techniques de model checking de par leur taille. Dans ce cas des oracles fondés sur des techniques de test (par exemple un interpréteur du langage de Cubicle comme défini en section 2.5.3) ou de simulation pourraient donner des résultats intéressants.

Parmi les problèmes qui restent hors de portée de BRAB, ceux qui demandent de découvrir des invariants numériques de qualité sont particulièrement intéressants. Il faudrait pour cela étendre à la fois le mécanisme de génération des candidats et utiliser un oracle approprié. Les protocoles de cohérence de cache qui sont aujourd'hui utilisé dans les microprocesseurs modernes sont également des candidats compliqués pour BRAB. Leur structure *hiérarchique* fait qu'ils sont également difficilement vérifiables même en fixant un nombre de processeurs petit. L'oracle de BRAB que nous avons implémenté n'est, par exemple, pas capable de construire un modèle fini du système avec plusieurs dizaines de millions d'états en mémoire. De plus les traces qui permettent de réfuter les candidats problématiques font généralement intervenir de très nombreux messages et étapes. En effet

il est souvent nécessaire d'effectuer un aller-retour dans la structure hiérarchique pour déclencher certaines actions des différents protocoles de cohérence (voir Figure 5.11). Outre leur intérêt certain pour l'industrie, la vérification de protocoles dans des architectures hiérarchiques est un défi majeur pour la communauté du model checking paramétré. Par exemple l'architecture en figure 5.11 est paramétrée dans *deux dimensions* : par le nombre de clusters (*i.e.* de caches L2) et par le nombre de caches L1 à l'intérieur d'un même cluster.

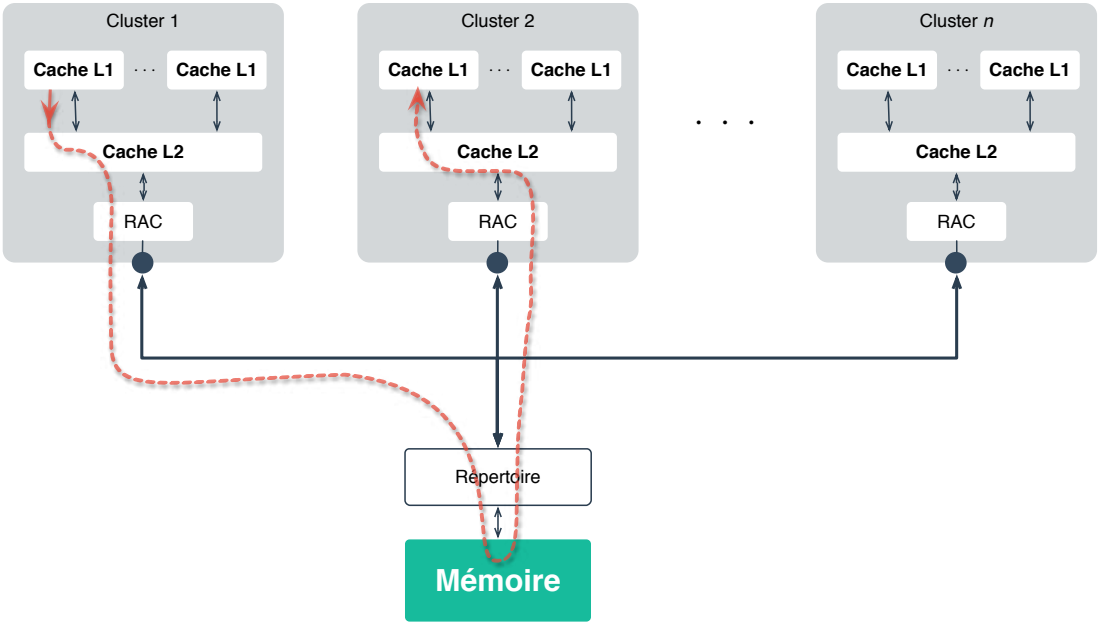



FIGURE 5.11 – Protocoles de cohérence de cache hiérarchiques [31]

Les approches qui passent à l'échelle sur ce genre d'exemple sont en grande partie manuelles et utilisent des raisonnements *compositionnels* [31]. Le challenge du model checking compositionnel est de trouver les bonnes interfaces à utiliser pour abstraire chacun des composant lorsqu'on fait la preuve d'une propriété particulière. Comme la technique mise en place par BRAB permet essentiellement de faire le lien entre les instances finies du système et la version paramétrée tout en étant guidé par la propriété à prouver, une piste d'exploration serait d'utiliser les mêmes idées pour inférer les interfaces adéquates dans une approche compositionnelle.

# 6

# Certification

## Sommaire


**6.1 Techniques de certification d'outils de vérification . . . . . 145**
**6.2 La plateforme de vérification déductive Why3 . . . . . 147**
**6.3 Production de certificats . . . . . 147**
6.3.1 Invariants inductifs pour l'atteignabilité arrière . . . . . 147
6.3.2 Invariants inductifs et BRAB . . . . . 152
**6.4 Preuve dans Why3 . . . . . 156**
**6.5 Discussion . . . . . 158**


Lorsque Cubicle retourne une trace d'erreur, il est facile de vérifier que cette trace est possible dans le système, et qu'elle conduit bien à état qui viole une des propriétés de sûreté. En revanche si Cubicle répond *safe*, l'utilisateur n'a pas d'autre choix que de faire confiance au résultat du model checker. Ce constat peut être fait pour la plupart des model checkers et comme ce sont des outils utilisés pour valider des composants de logiciels critiques, il semble logique qu'ils soient eux-mêmes vérifiés. Afin de limiter la confiance qu'on place dans le model checker, nous avons expérimenté deux approches différentes pour la certification de Cubicle, dont on décrit les résultats dans cette section. On expliquant également quels avantages sont apportés par BRAB pour cette tâche.

## 6.1 Techniques de certification d'outils de vérification

Deux axes de recherche coexistent dans la certification d'outils de vérification. La première approche consiste à faire générer des *certificats* (ou traces) qui doivent être vérifiées après coup. Une telle application de cette technique au model checking est Slab [59] qui produit des certificats sous la forme de diagrammes de vérification inductifs à destination

des solveurs SMT. Bien que peu intrusive, cette approche n'est appropriée que si les certificats ou objets de preuve sont assez simples et petits pour être vérifiés par des outils externes.

La seconde approche consiste à vérifier la correction du programme une fois pour toutes. Il existe bien sûr d'autres techniques mélangeant ces deux approches à des degrés et niveaux différents.

Dans l'esprit de cette deuxième approche, il existe différentes solutions pour vérifier qu'un programme est correct. Pour certains langages de programmation, il est possible de prouver des propriétés directement à partir du code source (*e.g.* avec ESC Java, Frama-C, VCC, F$^\star$ etc.). Ceci reste toutefois un travail laborieux car ces programmes sont souvent très complexes, les preuves deviennent rapidement alambiquées et sont rarement automatisables. En revanche les performances de tels programmes peuvent atteindre des niveaux qui sont proches de leurs homologues non certifiés et c'est là leur principal avantage. Un exemple d'une telle démarche de certification est le solveur SAT moderne `versat` qui a été développé et vérifié en utilisant le langage de programmation Guru [129]. Il n'existe à notre connaissance pas de résultat similaire pour un model checker.

Une autre possibilité est de prouver que l'algorithme est correct dans un langage descriptif (*e.g.* des assistants à la preuve interactifs comme Coq, PVS ou Isabelle) et d'obtenir un programme exécutable grâce à une (ou plusieurs) étape de raffinement ou un mécanisme d'extraction de code. Ces dernières années, les outils certifiés de ce type ont suscité l'intérêt de la communauté. Il convient de mentionner en particulier les succès qu'ont été le compilateur C CompCert [112] et le micro-noyau de système d'exploitation seL4 [102]. CompCert est écrit intégralement en Coq et utilise des oracles externes dans certaines de ses passes de compilation. Ces oracles fournissent des solutions (*e.g.* une coloration d'un graphe) facilement vérifiables par un validateur simple et certifié.

Bien que le premier effort de vérification formelle d'un model checker en Coq pour le $\mu$-calcul modal [156] remonte à 1988, c'est seulement récemment que des *outils de vérification certifiés* ont commencé à émerger. Blazy *et al.* ont vérifié un analyseur statique pour les programmes C [20] à destination du compilateur CompCert. Bien que cet analyseur statique ne puisse rivaliser avec les performances des outils commerciaux, il est largement suffisant pour permettre d'activer certaines optimisations d'un compilateur de manière sûre. Les travaux les plus proches des notre à cet égard concernent la vérification du model checker Spin avec l'assistant de preuve Isabelle [66]. Esparza *et al.* construisent un model checker correct par construction en partant d'une spécification haut niveau jusqu'à atteindre par raffinements successifs, une version fonctionnelle (en SML).

Dans ces approches, il existe traditionnellement un compromis entre un programme efficace obtenu à partir d'un algorithme précis manipulant des structures de données complexes, et un programme plus naïf obtenu à partir de spécifications haut niveau et parfois abstraites.

## 6.2 La plateforme de vérification déductive Why3

Why3 est une plateforme pour la vérification déductive de programmes [68]. Elle fournit un langage logique appelé Why pour décrire des formules dans une logique du premier ordre polymorphe avec un mécanisme de traduction vers des prouveurs automatique ou interactifs. Un des grands avantages de Why3 est qu'il permet de décrire des buts logiques dans un langage commun qu'il est possible de résoudre avec une pléthore d'outils : des solveurs SMT comme Alt-Ergo [21], CVC4 [15], Yices [61] ou Z3 [51] ; des solveurs par résolution comme iProver [103], E [150], SPASS [162], Vampire [144] ; des solveurs spécialisés pour certaines logiques comme Gappa [50] qui raisonne sur les opérations arithmétiques des nombres flottants de la norme IEEE 754 ; ou encore lorsque cela est nécessaire des assistants à la preuve comme Coq [58] ou PVS [133].

Why3 fournit également un langage de programmation appelé WhyML. C'est un langage impératif qui permet de décrire des programmes avec des spécifications logiques. Les spécifications des fonctions sont données sous la forme de pré- et post-conditions [1] et celles des boucles sous forme d'invariants. Pour vérifier qu'un programme respecte ses spécifications, Why3 génère des obligations de preuve à l'aide d'un calcul de plus faible pré-condition. Ces obligations peuvent ensuite être déchargées par les prouveurs mentionnés précédemment. Dans la suite de ce chapitre on utilise Why3 de deux manières : (1) pour vérifier des certificats produits par Cubicle, et (2) pour prouver la correction de l'algorithme BRAB du cœur de Cubicle.

## 6.3 Production de certificats

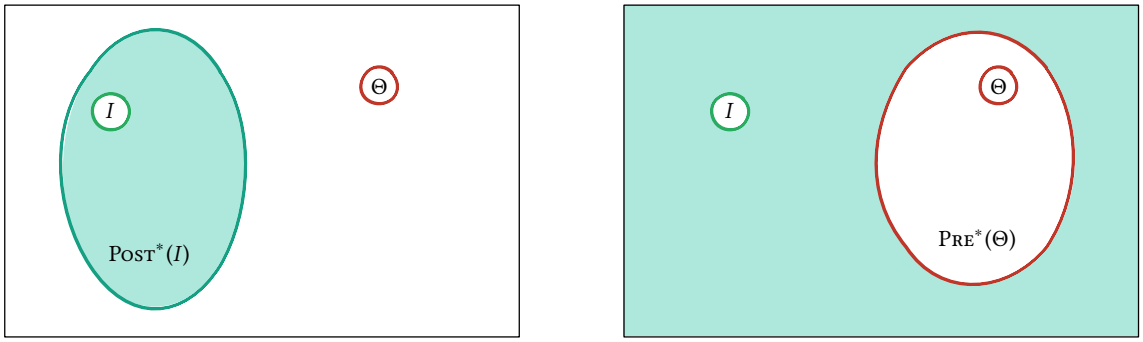
Une possibilité pour s'assurer de la véracité de la réponse de Cubicle est de vérifier son résultat. Plutôt que de répondre « safe » ou « unsafe », le model checker doit alors fournir des informations supplémentaires suffisantes pour pouvoir vérifier son raisonnement. C'est ce qu'on appelle un *certificat*.

### 6.3.1 Invariants inductifs pour l'atteignabilité arrière

Pour Cubicle, ce certificat n'a pas besoin de contenir toutes les étapes de calcul faites par le model checker comme les pré-images, les tests de point fixe ou de sûreté. La notion de sûreté d'un système est fortement liée à celle d'invariance. L'analyse de sûreté revient même à s'assurer qu'une propriété est un invariant du système. Pour un système donné, l'ensemble des états *atteignables* constitue l'*invariant inductif le plus fort* (partie verte de la figure figure 6.1(a)). De manière duale, l'ensemble des états ne pouvant atteindre un

1. Autrement dit des triplets de Hoare.

mauvais état du système constitue l'*invariant inductif le plus faible* du système (partie verte de la figure 6.1(b)).




(a) Invariant inductif le plus fort

(b) Invariant inductif le plus faible

FIGURE 6.1 – Invariants inductifs calculés par des analyses d'atteignabilité avant et arrière

L'ensemble $\mathcal{V}$ calculé par Cubicle dans l'algorithme 4 est en réalité la négation de cet invariant inductif le plus faible. Il constitue en lui même une preuve ou un certificat de la sûreté du système. En effet il est très simple de voir si une formule $\phi$ est un invariant inductif d'un système $\mathcal{S} = (Q, I, \tau)$. Il suffit pour cela qu'elle vérifie les deux conditions suivantes :

$$I(X) \models \phi(X) \qquad (6.1)$$
initialisation

$$\phi(X) \wedge \tau(X, X') \models \phi(X'). \qquad (6.2)$$
préservation

Le cas de base (6.1) dit que l'invariant $\phi$ doit être vrai dans les états initiaux du système et le cas inductif (6.2) dit que l'invariant doit être préservé par la relation de transition. Si en plus on a

$$\phi(X) \models P(X) \qquad (6.3)$$
propriété

alors la propriété $P$ est un invariant du système.

Si on prend $\phi = \neg\mathcal{V}$ et $P = \neg\Theta$, où $\mathcal{V}$ est la disjonction des éléments visités de l'algorithme 4 et $\Theta$ est la formule dangereuse du système, alors ces trois conditions sont vérifiées. En effet on a $\mathcal{V} \models \text{Pre}^*_\tau(\Theta)$, $\mathcal{V}$ est close par pré-image, *i.e.* $\mathcal{V}(X') \wedge \tau(X, X') \models \mathcal{V}(X)$ donc $\neg\mathcal{V}(X) \wedge \tau(X, X') \models \neg\mathcal{V}(X')$. Comme $\mathcal{V}$ contient $\Theta$, la condition (6.3) est aussi vérifiée.

Pour dire si le résultat de l'algorithme 4 implémenté par Cubicle est correct, il suffit alors de s'assurer que $\phi = \neg\mathcal{V}$ et $P = \neg\Theta$ satisfont les trois conditions (6.1), (6.2) et (6.3). Dans le cadre des systèmes à tableaux, $I$, $\tau$, $\mathcal{V}$ et $\Theta$ sont des formules du premier ordre. On pourrait donc prouver ces conditions avec un assistant de preuve ou à l'aide d'un démonstrateur automatique si on souhaite effectuer la certification sans intervention humaine. Dans ce dernier cas il faut faire confiance au solveur qu'on choisit. C'est pour cette raison qu'on a décidé d'utiliser Why3, on peut de cette façon accroître notre niveau de confiance lorsque différents prouveurs confirment de manière indépendante que les résultats sont corrects.

**Exemple.** Cubicle construit l'ensemble $\mathcal{V}$ suivant pour l'algorithme du mutex de la section 2.2.1 :

$$\begin{aligned}\mathcal{V} = \{\ & \exists z_1 z_2.\ z_1 \neq z_2 \wedge \texttt{State}[z_1] = \mathsf{Crit} \wedge \texttt{State}[z_2] = \mathsf{Crit},\\ & \exists z_1 z_2.\ z_1 \neq z_2 \wedge \texttt{Turn} = z_2 \wedge \texttt{State}[z_1] = \mathsf{Crit} \wedge \texttt{State}[z_2] = \mathsf{Want},\\ & \exists z_1 z_2.\ z_1 \neq z_2 \wedge \texttt{Turn} = z_2 \wedge \texttt{State}[z_1] = \mathsf{Crit} \wedge \texttt{State}[z_2] = \mathsf{Idle}\ \}\end{aligned}$$

Il génère ensuite un certificat sous la forme d'un ficher Why3 contenant quatre théories : `Mutex_defs`, `Mutex_initialisation`, `Mutex_property` et `Mutex_preservation`. La première permet de factoriser les définitions de symboles et de types.

```
theory Mutex_defs
  type proc
  type state = Idle | Want | Crit

  function turn : proc
  function turn' : proc
  function state proc : state
  function state' proc : state
end
```

Cette théorie déclare le type énuméré `state` et les symboles de fonction `turn` et `state` ainsi que leur version « prime ». Le type `proc` de Cubicle est interprété par les entiers mathématiques de Why3 (`int`) lorsque cela est nécessaire (en présence d'inégalités) ou par un type abstrait `proc` sinon . La théorie `Mutex_initialisation` contient les obligations de preuve correspondant à la condition (6.1).

```
theory Mutex_initialisation
  use import Mutex_defs

  axiom initial:
    forall z:proc. state z = Idle

  goal invariant_1:
    not (exists z1 z2:proc. z1 <> z2 /\
          state z1 = Crit /\ state z2 = Crit)

  goal invariant_2:
    not (exists z1 z2:proc. z1 <> z2 /\
          turn = z2 /\ state z1 = Crit /\ state z2 = Want)

  goal invariant_3:
    not (exists z1 z2:proc. z1 <> z2 /\
           turn = z2 /\ state z1 = Crit /\ state z2 = Idle)
end
```

On se place dans un état initial donc on suppose la formule initiale (axiome `initial`) et on demande à vérifier que chaque invariant est valide.

La théorie `Mutex_property` contient les obligations de preuve correspondant à la condition (6.3), c'est à dire qu'elle demande de vérifier que l'invariant inductif implique la propriété désirée.

```
theory Mutex_property
  use import Mutex_defs

  axiom invariant_1:
    not (exists z1 z2:proc. z1 <> z2 /\
          state z1 = Crit /\ state z2 = Crit)

  axiom invariant_2:
    not (exists z1 z2:proc. z1 <> z2 /\
          turn = z2 /\ state z1 = Crit /\ state z2 = Want)

  axiom invariant_3:
    not (exists z1 z2:proc. z1 <> z2 /\
           turn = z2 /\ state z1 = Crit /\ state z2 = Idle)

  goal property_1:
    not (exists z1 z2:proc. z1 <> z2 /\
          state z1 = Crit /\ state z2 = Crit)
end
```

Cette obligation de preuve est triviale car $\mathcal{V}$ contient (presque) toujours les formules

dangereuses. Ici `property_1` correspond exactement à l'axiome `invariant_1`.

Enfin la théorie Why3 `Mutex_preservation` contient les obligations de preuve à décharger pour montrer que l'invariant est inductif, c'est-à-dire qu'il est préservé par la relation de transition.

```
theory Mutex_preservation
  use import Mutex_defs

  axiom induction_hypothesis_1:
    not (exists z1 z2:proc. z1 <> z2 /\
          state z1 = Crit /\
          state z2 = Crit)

  axiom induction_hypothesis_2:
    not (exists z1 z2:proc. z1 <> z2 /\
          turn = z2 /\
          state z1 = Crit /\
          state z2 = Want)

  axiom induction_hypothesis_3:
    not (exists z1 z2:proc. z1 <> z2 /\
           turn = z2 /\
           state z1 = Crit /\
           state z2 = Idle)

axiom transition_relation:

  (* transition req *)
  (exists i:proc.
    (* requires *)
    state i = Idle /\
    (* actions *)
    turn' = turn /\
    forall _j1:proc.
      if _j1 = i then state' _j1 = Want
      else state' _j1 = state _j1)

 \/

  (* transition enter *)
  (exists i:proc.
    (* requires *)
    turn = i /\ state i = Want /\
    (* actions *)
    turn' = turn /\
    forall _j2:proc.
      if _j2 = i then state' _j2 = Crit
      else state' _j2 = state _j2)

 \/

  (* transition exit *)
  (exists i:proc.
    (* requires *)
    state i = Crit /\
    (* actions *)
    forall _j3:proc.
      if _j3 = i then state' _j3 = Idle
      else state' _j3 = state _j3)

  goal invariant_1:
    not (exists z1 z2:proc. z1 <> z2 /\
          state' z1 = Crit /\
          state' z2 = Crit)

  goal invariant_2:
    not (exists z1 z2:proc. z1 <> z2 /\
          turn' = z2 /\
          state' z1 = Crit /\
          state' z2 = Want)

  goal invariant_3:
    not (exists z1 z2:proc. z1 <> z2 /\
           turn' = z2 /\
           state' z1 = Crit /\
           state' z2 = Idle)
end
```

On suppose que l'invariant inductif est vrai initialement (par les axiomes `induction_hypothesis_`*i*) et on exprime la relation de transition sous la forme d'une formule reliant les symboles `turn` et `state` aux symboles `turn'` et `state'`. Cette formule est l'expression

directe de $\tau(X, X')$ dans le langage logique de Why3. On demande finalement à vérifier que les invariants sur les versions prime des symboles sont toujours vrais.

L'exemple jouet du mutex est trivial et tous les prouveurs automatiques sont capables de vérifier le certificat fourni par Cubicle. En revanche, si on prend l'exemple du protocole de cohérence de cache German-*esque* de la section 2.2.4, l'invariant inductif est composé de 17 clauses universellement quantifiées. On rapporte les temps de vérification obtenus pour différents démonstrateurs automatiques lancés au travers de la plateforme Why3 en figure 6.2. Chaque prouveur a été lancé avec un timeout de cinq secondes. Les temps sont donnés en secondes et les cases vertes correspondent à une réponse positive (`valid`), les cases rouges correspondent à une réponse non concluante (`unknown`). Enfin les cases oranges dénotent les exécutions qui n'ont pas abouti dans le temps imparti (T.O.).

On peut remarquer que les obligations de preuve correspondant à la condition d'initialisation (6.1) et à la propriété (6.3) sont aisément déchargées par la totalité des huit prouveurs quasiment instantanément. En revanche les obligations concernant la préservation de l'invariant sont plus difficiles. Certains prouveurs ne terminent pas à temps alors que d'autres n'arrivent pas à conclure. Ces performances plutôt décevantes peuvent être imputées aux quantificateurs (universels et existentiels) omniprésents dans ces buts et en particulier dans la représentation logique de la relation de transition. La plupart des solveurs utilisent des heuristiques sensibles pour traiter ces quantificateurs donc les performances sont parfois inégales et rarement prévisibles. Mêmes si les résultats ne sont pas idéaux, ils sont largement suffisants car toutes les obligations sont déchargées par au moins *quatre* prouveurs.

### 6.3.2 Invariants inductifs et BRAB

La taille des certificats générés par Cubicle dépend donc essentiellement du nombre de nœuds visités. BRAB peut ici nous aider à garder cette mesure la plus petite possible. Comme on l'a vu dans chapitre 5, BRAB construit une *sur-approximation* des états pouvant atteindre la formule dangereuse $\Theta$. L'invariant inductif $\neg\mathcal{V}_{\text{BRAB}}$ obtenu à la fin de l'exécution n'est donc pas le plus faible possible. Les approximations successives faites par BRAB peuvent en réalité être vues comme des renforcements de cet invariant inductif. Cette technique permet de converger plus vite, mais surtout elle construit une preuve beaucoup plus simple qu'une analyse d'atteignabilité arrière classique. L'invariant inductif se trouve donc à mi-chemin entre l'invariant inductif le plus fort et entre l'invariant inductif le plus faible (*cf.* partie verte de la figure 6.3). Le défaut de ces deux extrêmes est que leur caractérisation précise requiert souvent des formules longues et compliquées.

En revanche l'invariant calculé par BRAB est généralement bien plus simple et court (souvent de plusieurs ordres de grandeurs) ce qui en fait un certificat possédant les bonnes propriétés. Par exemple BRAB construit un invariant inductif avec seulement *quatre* clauses

| Obligations de preuve | **Alt-Ergo** (0.96) | **CVC3** (2.4.1) | **CVC4** (1.3) | **Eprover** (1.8-001) | **Spass** (3.5) | **Yices** (1.0.40) | **Z3** (4.3.2) | **iProver** (1.0) |
|---|---|---|---|---|---|---|---|---|
| **Initialisation** (6.1) | | | | | | | | |
| invariant_1 | 0,01 | 0,02 | 0,03 | 0,00 | 0,02 | 0,00 | 0,03 | 0,04 |
| invariant_2 | 0,01 | 0,00 | 0,01 | 0,01 | 0,02 | 0,00 | 0,00 | 0,03 |
| invariant_3 | 0,01 | 0,01 | 0,00 | 0,01 | 0,02 | 0,00 | 0,00 | 0,03 |
| invariant_4 | 0,01 | 0,01 | 0,00 | 0,00 | 0,02 | 0,00 | 0,00 | 0,03 |
| invariant_5 | 0,01 | 0,01 | 0,00 | 0,00 | 0,02 | 0,00 | 0,00 | 0,03 |
| invariant_6 | 0,01 | 0,00 | 0,00 | 0,00 | 0,02 | 0,00 | 0,00 | 0,02 |
| invariant_7 | 0,01 | 0,00 | 0,00 | 0,01 | 0,02 | 0,00 | 0,00 | 0,03 |
| invariant_8 | 0,01 | 0,00 | 0,01 | 0,00 | 0,02 | 0,00 | 0,00 | 0,02 |
| invariant_9 | 0,01 | 0,00 | 0,00 | 0,00 | 0,02 | 0,00 | 0,00 | 0,03 |
| invariant_10 | 0,01 | 0,00 | 0,00 | 0,00 | 0,02 | 0,00 | 0,00 | 0,02 |
| invariant_11 | 0,01 | 0,00 | 0,00 | 0,00 | 0,01 | 0,00 | 0,00 | 0,02 |
| invariant_12 | 0,01 | 0,00 | 0,00 | 0,00 | 0,02 | 0,00 | 0,01 | 0,03 |
| invariant_13 | 0,01 | 0,00 | 0,01 | 0,00 | 0,02 | 0,00 | 0,00 | 0,02 |
| invariant_14 | 0,01 | 0,00 | 0,00 | 0,00 | 0,02 | 0,00 | 0,00 | 0,02 |
| invariant_15 | 0,01 | 0,00 | 0,00 | 0,00 | 0,02 | 0,00 | 0,00 | 0,03 |
| invariant_16 | 0,01 | 0,00 | 0,00 | 0,01 | 0,02 | 0,00 | 0,00 | 0,03 |
| invariant_17 | 0,01 | 0,00 | 0,00 | 0,00 | 0,02 | 0,00 | 0,00 | 0,02 |
| **Propriété** (6.3) | | | | | | | | |
| property_1 | 0,01 | 0,01 | 0,02 | 0,01 | 0,02 | 0,00 | 0,00 | 0,04 |
| **Préservation** (6.2) | | | | | | | | |
| invariant_1 | 0,02 | 0,40 | T.O. | 0,05 | 0,07 | 0,62 | 0,02 | T.O. |
| invariant_2 | 0,01 | 0,21 | 0,34 | 0,01 | 0,07 | 0,75 | 0,01 | T.O. |
| invariant_3 | 0,02 | 0,02 | 0,16 | 0,02 | 0,14 | 0,54 | 0,01 | T.O. |
| invariant_4 | 0,01 | 0,43 | 0,07 | 0,02 | 0,03 | 5,15 | 0,01 | T.O. |
| invariant_5 | 0,02 | 0,02 | 0,07 | 0,01 | 0,03 | 0,24 | 0,01 | T.O. |
| invariant_6 | 0,03 | 0,02 | 3,43 | 0,03 | 0,27 | 0,45 | 0,01 | T.O. |
| invariant_7 | 0,03 | 0,03 | 2,71 | 0,03 | 0,30 | 0,42 | 0,01 | T.O. |
| invariant_8 | 0,02 | 0,59 | 1,32 | 0,03 | 0,05 | 0,68 | 0,01 | T.O. |
| invariant_9 | 0,01 | 0,64 | 0,93 | 0,03 | T.O. | 0,73 | 0,01 | T.O. |
| invariant_10 | 0,01 | 0,49 | 0,06 | 0,02 | 0,03 | 5,15 | 0,01 | T.O. |
| invariant_11 | 0,02 | 0,56 | 0,09 | 0,04 | 1,55 | 0,49 | 0,01 | T.O. |
| invariant_12 | 0,04 | 0,42 | 0,72 | 0,03 | 2,66 | 0,52 | 0,01 | T.O. |
| invariant_13 | 0,02 | 0,25 | 0,08 | 0,02 | 0,03 | 5,16 | 0,02 | T.O. |
| invariant_14 | 0,02 | 0,02 | 0,12 | 0,01 | 0,06 | 0,77 | 0,01 | T.O. |
| invariant_15 | 0,05 | 0,38 | 2,84 | 0,03 | 0,07 | 0,71 | 0,01 | T.O. |
| invariant_16 | 0,06 | 0,38 | 2,55 | 0,03 | 0,08 | 0,72 | 0,01 | T.O. |
| invariant_17 | 0,02 | 0,44 | 0,17 | 0,01 | 0,07 | 0,61 | 0,01 | 4,63 |

FIGURE 6.2 – Vérification du certificat Why3 de German-*esque* par différents prouveurs automatiques

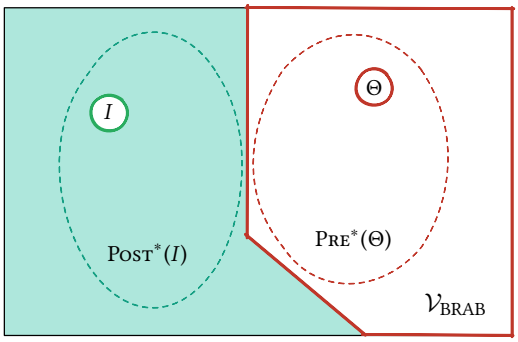



FIGURE 6.3 – Invariant inductif calculé par BRAB

quantifiées pour l'exemple German-*esque* (contre 17 pour l'analyse d'atteignabilité arrière classique). La vérification de ce certificat est par conséquent bien plus aisée et quasiment tous les prouveurs sont capables de décharger la totalité des obligations (Figure 6.4). Dorénavant pour cet exemple, chaque obligation est vérifiée par au moins *six* prouveurs de manière indépendante.

| Obligations de preuve | **Alt-Ergo** (0.96) | **CVC3** (2.4.1) | **CVC4** (1.3) | **Eprover** (1.8-001) | **Spass** (3.5) | **Yices** (1.0.40) | **Z3** (4.3.2) | **iProver** (1.0) |
|---|---|---|---|---|---|---|---|---|
| **Initialisation** (6.1) | | | | | | | | |
| invariant_1 | 0,01 | 0,01 | 0,01 | 0,01 | 0,02 | 0,00 | 0,01 | 0,03 |
| invariant_2 | 0,01 | 0,01 | 0,00 | 0,01 | 0,02 | 0,00 | 0,00 | 0,03 |
| invariant_3 | 0,01 | 0,00 | 0,01 | 0,00 | 0,02 | 0,00 | 0,00 | 0,03 |
| invariant_4 | 0,01 | 0,00 | 0,01 | 0,00 | 0,02 | 0,00 | 0,00 | 0,03 |
| **Propriété** (6.3) | | | | | | | | |
| property_1 | 0,01 | 0,00 | 0,01 | 0,01 | 0,02 | 0,00 | 0,00 | 0,04 |
| **Préservation** (6.2) | | | | | | | | |
| invariant_1 | 0,01 | 0,02 | 0,01 | 0,03 | 0,07 | 0,21 | 0,01 | T.O. |
| invariant_2 | 0,02 | 0,01 | 0,02 | 0,05 | 0,05 | 0,00 | 0,00 | 2,43 |
| invariant_3 | 0,02 | 0,02 | T.O. | 0,05 | 0,05 | 0,06 | 0,01 | T.O. |
| invariant_4 | 0,03 | 0,02 | 4,15 | 0,04 | 0,04 | 0,06 | 0,01 | T.O. |

FIGURE 6.4 – Vérification par différents prouveurs automatiques du certificat (Why3) de German-*esque* généré par BRAB

On donne les résultats obtenus pour la vérification des certificats Why3 produits par BRAB sur notre ensemble de benchmarks en figure 6.5. La colonne ∀-clauses indique le nombre de clauses quantifiées de l'invariant inductif. On précise également la taille du

fichier Why3 produit et si le certificat a été vérifié. On note par Niveau (de confiance) le nombre minimal de prouveurs différents ayant vérifié chaque obligation de preuve de manière indépendante. Enfin le temps de vérification rapporté dans la dernière colonne correspond au temps minimum nécessaire pour vérifier intégralement le certificat.

| **Benchmark** | **∀-clauses** | **Taille** | **Vérifié** | **Niveau** | **Temps** |
|---|---|---|---|---|---|
| Szymanski_at | 31 | 18 ko | Oui | 3 | 0,96s |
| Szymanski_na | 38 | 28 ko | Oui | 2 | 1,45s |
| Ricart_Agrawala | 30 | 39 ko | Oui | 2 | 1,26s |
| German_Baukus | 48 | 44 ko | Oui | 2 | 1,58s |
| German.CTC | 69 | 83 ko | Oui | 2 | 2,73s |
| German_pfs | 51 | 50 ko | Oui | 3 | 1,79s |
| Flash_nodata | 41 | 123 ko | Oui | 2 | 2,99s |
| Flash_abstr | 742 | 1,7 Mo | 1462/1475 | 0 | 35m |

Figure 6.5 – Vérification de certificats sur un ensemble de benchmarks

La plupart des certificats pour les invariants inductifs produits par BRAB sont relativement petits (moins de 150 ko) et sont vérifiés en seulement quelques secondes. Seul le certificat du protocole FLASH complet n'a pu être vérifié intégralement (1462 obligations déchargées sur les 1475 totales). En effet ce dernier fichier Why3 est très gros et certaines clauses de l'invariant inductif sont quantifiées avec quatre variables, ce qui en fait des problèmes très compliqués pour les solveurs SMT.

Pour remédier à cette difficulté on peut envisager plusieurs pistes d'amélioration des certificats. La première consiste à essayer de simplifier au maximum l'invariant inductif car son procédé de construction (par analyse d'atteignabilité arrière) fait qu'il peut contenir des redondances. La seconde possibilité est d'ajouter des lemmes intermédiaires pour aider les démonstrateurs automatiques. En particulier pour l'étape de vérification de la préservation de l'invariant, il est facile d'instrumenter le model checker afin d'extraire de l'invariant inductif complet les composantes qui sont utiles à la preuve préservation de chaque composante. En effet cette information est déjà calculée par le model checker lors de ses tests de point fixe. Il serait même possible d'ajouter seulement les instances nécessaires à la preuve de préservation de chaque invariant afin de n'envoyer aux solveurs externes que des problèmes sans quantificateurs. Dans ce cas il faut soit fournir la preuve des instances dans le certificat, soit développer un algorithme d'instantiation certifié.

Pour les problèmes d'atteignabilité les plus compliqués, les certificats construits de cette manière resteront gros car ils dépendent directement du nombre de nœuds explorés par le

model checker. Une approche différente pour la certification de Cubicle consiste à vérifier le model checker lui-même plutôt que de vérifier son résultat après chaque exécution.

## 6.4 Preuve dans Why3

Notre but ici est de prouver que l'algorithme BRAB utilisé par Cubicle est correct, c'est-à-dire qu'il répond « safe » seulement si les états représentés par la formule dangereuse $\Theta$ ne sont pas atteignables. Pour cela, on a modélisé BRAB dans le langage WhyML de Why3. Notre développement contient deux parties : une partie logique spécifiant les fonctions et relations logiques (satisfiabilité `sat` et conséquence logique $\models$) ainsi que les structures de données utilisées (formules, ensembles de nœuds, file de travail). Le reste du code est un programme impératif annoté avec des pré- et post-conditions ainsi que des invariants de boucle et des assertions logiques.

On a choisi de se placer à un niveau d'abstraction assez élevé pour faciliter notre travail et celui des démonstrateurs automatiques. La première étape de cette approche consiste donc à modéliser les notions de logique du premier ordre. Étant donné un type abstrait `formula` représentant les formules et le type des structures (*cf.* Section 2.4.2) `structure`, on suit les définitions usuelles de la satisfiabilité et on étend la notion de conséquence logique aux formules :

$$\texttt{sat}(f : \texttt{formula}) \triangleq \exists m : \texttt{structure}.\ m \vdash f$$

$$\forall f_1, f_2 : \texttt{formula}.\ f_1 \models f_2 \iff (\forall m : \texttt{structure}.\ m \vdash f_1 \implies m \vdash f_2)$$

Par extension, on dit que deux formules (de type `formula`) sont égales si et seulement si elles admettent les mêmes modèles. On définit également les connecteurs de la logique classique `&` (pour la conjonction), `++` (pour la disjonction), $\sim$ (pour la négation) et `=>` (pour l'implication). Par exemple `&` est défini par l'axiome suivant :

$$\forall m : \texttt{structure}.\ \forall f_1, f_2 : \texttt{formula}.\ m \vdash (f_1\ \texttt{\&}\ f_2) \iff m \vdash f_1 \ \wedge\ m \vdash f_2$$

Pour parler d'atteignabilité, on se donne une fonction abstraite Pre qui prend en entrée une relation de transition $\tau$, une formule $f$ et renvoie la pré-image de $f$ par $\tau$ :

$$\textsc{Pre}(\tau, f) : \texttt{formula}$$

La clôture transitive $\textsc{Pre}^*$ de Pre est axiomatisée de façon partielle comme suit :

$$\textsc{Pre}^*(\tau, f) : \texttt{formula} \begin{cases} \forall f : \texttt{formula}.\ \texttt{valid}(f\ \texttt{=>}\ \textsc{Pre}^*(\tau, f)) \\ \qquad\qquad \wedge\ \textsc{Pre}^*(\tau, \textsc{Pre}^*(\tau, f)) = \textsc{Pre}^*(\tau, f) \\ \qquad\qquad \wedge\ \textsc{Pre}^*(\tau, \textsc{Pre}(\tau, f))\ \texttt{++}\ f = \textsc{Pre}^*(\tau, f) \end{cases}$$

Bien sûr il faut faire confiance à cette axiomatisation mais sa concision permet de se convaincre de sa correction. On dit qu'un état représenté par la formule $f$ est atteignable à partir d'un état de la formule $I$ par la relation de transition $\tau$ si il existe un modèle (*i.e.* un état) qui est à la fois dans $I$ et dans la clôture transitive de la pré-image de $f$. Autrement dit :

$$\texttt{reachable}(\tau, I, f) = \texttt{sat}(\textsc{Pre}^*(\tau, f) \ \texttt{\&} \ I)$$

On représente la file utilisée par BRAB ($\mathcal{Q}$ dans l'algorithme 7) par un enregistrement à champs mutables contenant l'ensemble de ses formules et un champ *fantôme* représentant la disjonction de toutes ces formules. Le champ `f` est dit *fantôme* car il est seulement utilisé pour les besoins de la spécification. Un extrait de code en Why3 donne :

```
type t model { ghost mutable f : formula;
               mutable elts : set formula }

val push (f:formula) (q:t) : unit writes  { q }
  ensures { q.f = f ++ (old q.f) /\ q.elts = add f (old q.elts) }
```

Comme on a donné une version impérative de BRAB (Algorithme 7), coder l'algorithme dans le langage WhyML s'avère assez direct. Prouver que la post-condition suivante de la fonction `brab` est vraie assure la correction de l'algorithme. C'est-à-dire que si $\texttt{BRAB}(\tau, I, \Theta) = \textsf{Safe}$ alors la formule $\Theta$ n'est pas atteignable.

```
let brab (tau:trans_rel) (init:formula) (theta:formula) =
    ensures  { result = Safe -> not (reachable tau init theta) }
    ...
```

L'invariant qu'on écrit pour spécifier le comportement de la boucle interne de la fonction `BWDA` de l'algorithme 7 reflète ceux qui ont été utilisés pour faire la preuve de correction à la main (invariants (5.1), (5.2) et (5.3) de la section 5.1.3). Le premier dit que les nœuds visités ne contiennent pas de formule s'intersectant avec $I$.

```
while not (Q.is_empty q) do
 invariant { not (sat (!visited & init)) &&
             PRE* tau theta |= !visited ++
                               (PRE* tau (q.formula & ~ !visited)) &&
             (∀ phi:f. !kind[phi] = Orig -> !from[phi] = theta) }
 ...
```

On ajoute aussi quelques assertions auxiliaires à certains points du programme mais on ne les présente pas ici dans un souci de lisibilité.

| Théorie | Obligations de preuve | Temps |
|---|---|---|
| FOL | 34 | 58,56s |
| Reachability | 10 | 21,44s |
| BWD | 15 | 1m16s |
| BRAB | 39 | 50,90s |

Figure 6.6 – Informations sur le développent Why3

On récapitule dans le tableau figure 6.6 le nombre d'obligations de preuves ainsi que le temps nécessaire à leur preuve, pour chacune des théories et des modules de notre développement Why3. FOL correspond à la théorie de la logique du premier ordre et BWD contient la version de l'algorithme 4 sans approximations. Toutes les obligations de preuves ont été déchargées automatiquement (exception faite de quelques propriétés de FOL qui ont été prouvées en Coq) par au moins deux démonstrateurs automatiques.

## 6.5 Discussion

Après avoir prouvé une version abstraite de l'algorithme BRAB, on est sûr que l'algorithme tel qu'il a été conçu est correct. Bien que le même algorithme soit implémenté dans Cubicle, on en a seulement vérifié une version abstraite. Afin d'atteindre le niveau de confiance obtenu par l'utilisation de traces de la section 6.3, il faudrait vérifier l'implémentation même de Cubicle.

Un travail futur ayant pour objectif l'obtention d'une version certifiée de Cubicle consisterait à extraire du code OCaml à partir des spécifications. Ce mécanisme d'extraction est fourni nativement par Why3 mais nos spécifications sont données à un niveau d'abstraction élevé. En particulier la partie logique utilise des théories axiomatiques pour définir des opérations comme le calcul de pré-image ou les tests de satisfiabilité. Par conséquent, une extraction de notre programme `brab.mlw` résulte en un code incomplet. On peut tout de même remarquer que les opérations basiques effectuées par Why3 et Cubicle sont très proches. Une façon de compléter ce code « à trous » serait alors de tirer partit des primitives et structures de données internes de Why3 au travers de son API [2].

Cette idée, illustrée en Figure 6.7, consiste à utiliser gratuitement les fonctionnalités de calcul de WP (plus faible pré-condition) de Why3 s'apparentant au calcul de pré-image de

2. Interface de programmation ou encore *Application Programming Interface*.

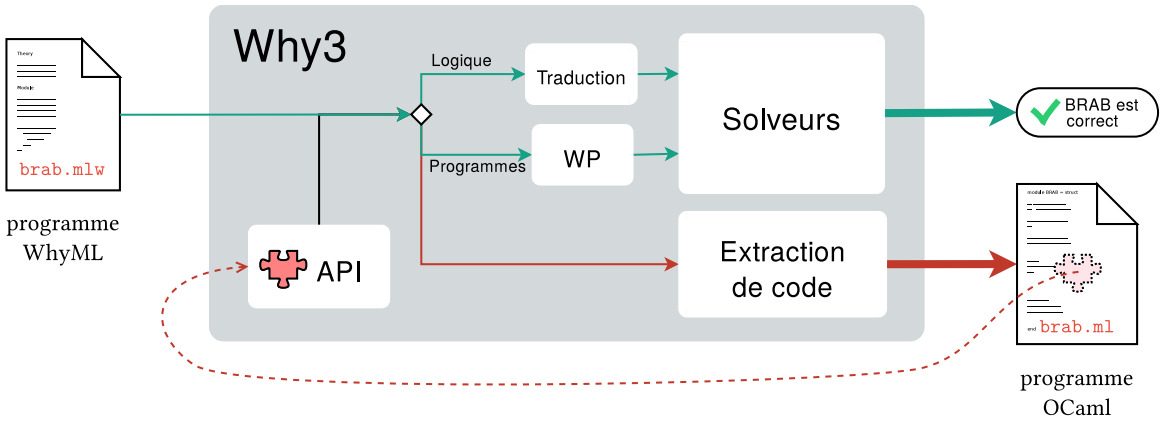



FIGURE 6.7 – Aperçu d'une technique d'extraction

Cubicle, ainsi que les facilités d'appels aux solveurs externes pour implémenter nos tests logiques. Bien sûr cela demanderait d'inclure Why3 dans notre base de confiance, mais c'est déjà le cas car on l'utilise pour faire notre preuve.

Il existe tout de même quelques obstacles qui pourraient entraver la réussite d'une telle approche. Tout d'abord le langage de Why3 est extrêmement plus riche que celui de Cubicle. Par la force des choses, le calcul de WP de Why3 est donc beaucoup plus compliqué et les calculs de pré-images successifs produisent des formules de plus en plus grosses. Le second obstacle touche aux performances d'un model checker construit par une telle technique. En effet Why3 fait appel aux solveurs au travers de fichiers temporaires ce qui n'est pas viable pour Cubicle qui nécessite parfois plusieurs millions d'appels (*cf.* Section 4.2). Remédier à ces deux problèmes demanderait donc d'avoir à la fois des fonctionnalités de simplification de formules dans Why3 et la possibilité d'appeler certains démonstrateurs automatiques incrémentalement au travers d'une interface (de programmation ou textuelle).

On pourrait aussi imaginer pousser cette approche plus loin, au point d'intégrer un model checker comme Cubicle à la plateforme Why3, d'une manière similaire à certains outils qui mélangent preuve et model checking tels que PVS ou SMV.

# 7

# Conclusion et perspectives

## 7.1 Résumé des contributions et conclusion

Dans le chapitre 2 nous avons présenté le model checker Cubicle, issu des travaux exposés dans ce document. C'est un model checker pour systèmes paramétrés (infinis) qui utilise le solveur SMT Alt-Ergo et qui repose sur le cadre théorique du model checking modulo théories du chapitre 3. On a montré qu'une implémentation naïve d'un algorithme d'atteignabilité ne fonctionne pas en pratique et nécessite un traitement particulier des quantificateurs. Pour obtenir des performances raisonnables sur l'algorithme de départ d'autres optimisations sont nécessaires comme des raisonnements syntaxiques efficaces et une analyse statique de sous-typage. Ces techniques ont permis d'aboutir à un model checker pour systèmes paramétrés compétitif. Toutefois, comme ses confrères, il demeure incapable de traiter des problèmes réels plus gros dont la complexité est beaucoup plus grande. Parmi ceux-ci on peut citer les algorithmes à évaluation de conditions non atomique (beaucoup plus proches des implémentations réelles que leurs contreparties atomiques) et les protocoles de cohérence de cache de taille industrielle.

Pour répondre au problème de cette thèse, à savoir la vérification de propriétés de sûreté de systèmes paramétrés, on a développé dans une seconde partie un algorithme d'inférence d'invariants appelé BRAB (Chapitre 5). Cet algorithme à été implémenté dans le model checker open source Cubicle. Sa force réside dans l'utilisation d'un oracle qui concentre les connaissances au travers de modèles finis du système. Il donne des résultats impressionnants en pratique car généralement les instances finies, même petites, exhibent déjà la plupart des comportements intéressants du système. Cette approche répond au problème posé car BRAB est capable d'inférer des invariants de qualité, utiles à la preuve de propriétés de sûreté de systèmes complexes. Un résultat remarquable est notamment la preuve automatique des propriétés de contrôle et de cohérence du protocole FLASH. À notre connaissance c'est la première vérification entièrement automatique de la sûreté de

ce protocole.

Afin de limiter la confiance qu'on place dans le model checker, nous avons expérimenté deux approches différentes pour la certification de Cubicle, reposant sur la plate-forme de vérification déductive Why3. Ces approches ont été présentés dans le chapitre 6. La première est une approche qui fonctionne par certificats (ou traces) et vérifie le résultat produit par le model checker. Grâce aux invariants découverts par BRAB, on a pu réduire la taille de ces certificats de manière dramatique. On a ainsi pu certifier les résultats produits par Cubicle sur la quasi totalité de nos benchmarks (excepté la version la plus difficile du FLASH). La seconde approche est une preuve de correction de l'algorithme BRAB avec Why3. Elle permet d'assurer que les algorithmes implémentés dans Cubicle sont corrects mais ne donne pas un niveau de confiance comparable à l'approche par certificats. Pour cela, il faudrait extraire du code exécutable à partir des spécifications.

## 7.2 Perspectives

**Instantiation.** On a vu qu'il était nécessaire de faire une instantiation intelligente des quantificateurs dans Cubicle en section 4.2.3 car les solveurs SMT ne sont pas capables de faire ce genre de raisonnements, sont incomplets et sont bien trop inefficaces sur les fragments de la logique avec quantificateurs. Ces techniques sont possibles dans Cubicle car on est dans un cadre bien particulier qui garantit qu'on puisse faire une instantiation exhaustive. Des approches récentes pourraient toutefois répondre en partie à ce problème. En effet Reynolds *et al.* ont développé des techniques d'instantiation utilisant à la fois un solveur pour la cardinalité des sortes et module d'instantiation sur les domaines finis [141]. Une extension de cette technique [142] qui se rapproche de nos méthodes d'instantiation présentés en section 4.2.3 parait tout à fait à propos pour Cubicle.

**BRAB.** Une étape qui semble logique pour BRAB est d'expérimenter l'approche sur des protocoles plus gros, se rapprochant des protocoles implémentés dans les architectures modernes (*e.g.* LCP chez Intel [130]). La plupart de ses architectures sont dites *hiérarchiques* car elles mettent en place une hiérarchie de protocoles de cohérence de cache différents pour les niveaux L1, L2, *etc.* Les traces qui permettent de réfuter les candidats problématiques des ces protocoles font généralement intervenir de très nombreux messages et étapes. Ceci en fait des exemples particulièrement difficiles pour notre technique. La vérification paramétrée de tels protocoles demanderait donc d'étendre notre méthode avec des approches compositionnelles par exemple. Une piste d'exploration future serait d'utiliser les mêmes idées que celles utilisées par BRAB pour inférer les interfaces adéquates aux abstractions nécessaires à ces approches. Une autre évolution possible de BRAB serait d'expérimenter différents oracles, à même de trouver des bugs plus rapidement que l'exploration énumérative implémentée aujourd'hui. Des oracles reposant sur le model checking borné ou encore

le test et la simulation pourraient donner des résultats intéressants.

**Invariants numériques.** Un point faible de BRAB est qu'il n'est aujourd'hui pas à même de découvrir des invariants numériques de qualité pour des problèmes compliqués même si des techniques ont été mises en place pour traiter les cas les plus simples (Section 5.2.4). Les structures de flot de contrôle sont souvent modélisables comme des systèmes finis sauf lorsqu'elles font intervenir la notion d'horloge comme c'est souvent le cas dans les systèmes de contrôle-commande ou les systèmes hybrides. De plus les données manipulées par ces programmes sont souvent à valeurs numériques. Appliquer BRAB à ces problèmes demande donc d'être capable d'inférer des invariants numériques (parfois même non-linéaires). Comme le domaine de prédilection de l'interprétation abstraite est la vérification de propriétés de données des programmes numériques, ces techniques pourraient être combinées à notre approche. Des travaux récents combinent déjà interprétation abstraite et model checking [72, 160], une piste considérait à adapter ces techniques à la vérification paramétrée faite par BRAB afin d'améliorer la génération d'invariants numériques de Cubicle.

**IC3.** Une technique de model checking qui a reçu un intérêt grandissant de la communauté ces dernières années est IC3 (pour ***I**ncremental **C**onstruction of **I**nductive **C**lauses for **I**ndubitable **C**orrectness*) aussi connu sous le nom d'*atteignabilité guidée par propriété* [25]. Cette technique à l'avantage principal d'être aussi bien guidée par les états initiaux du système (et par extension les états atteignables) comme les techniques d'atteignabilité avant, que par la propriété qu'on souhaite vérifier comme les techniques d'atteignabilité arrière. À l'origine prévue pour la vérification de circuits matériels, domaine dans lequel elle excelle, cette technique a récemment été adaptée au model checking de programmes grâce à l'utilisation de solveurs SMT [34]. Elle ne permettrait pas d'améliorer l'expressivité de Cubicle mais pourrait s'avérer utile comme une méthode plus efficace supplantant l'atteignabilité arrière aujourd'hui utilisée. Une étape importante de cette technique consiste à éliminer des mauvais contre-exemples en les généralisant puis en propageant l'information en arrière par calculs de pré-images (ou par interpolation). Plus la généralisation englobe de potentiels mauvais contre-exemples, plus la méthode est efficace. Similairement, plus l'information propagée en arrière est générale, plus la convergence sera rapide. Une piste d'exploration est d'utiliser le mécanisme d'approximation couplé à un oracle de BRAB pour cette phase de propagation arrière afin d'adapter la technique à la vérification paramétrée.

# A

# Syntaxe et typage des programmes Cubicle

## A.1 Syntaxe

Il est aisé de se faire une idée à partir du chapitre 2 du langage de description qu'on utilise pour les systèmes de transition paramétrés. On précise les choses dans cette annexe en commençant par définir de manière formelle la syntaxe concrète des fichiers d'entrée de Cubicle. Dans la suite, nous utilisons les notations suivantes dans les grammaires :

| Notation | Description |
|---|---|
| $\langle règle \rangle^\star$ | répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune) |
| $\langle règle \rangle_t^\star$ | répétition de la règle $\langle règle \rangle$ un nombre quelconque de fois (y compris aucune), les occurrences étant séparées par le terminal $t$ |
| $\langle règle \rangle^+$ | répétition de la règle $\langle règle \rangle$ au moins une fois |
| $\langle règle \rangle_t^+$ | répétition de la règle $\langle règle \rangle$ au moins une fois, les occurrences étant séparées par le terminal $t$ |
| $\langle règle \rangle$ ? | utilisation optionnelle de la règle $\langle règle \rangle$ (i.e. 0 ou 1 fois) |
| ( $\langle règle \rangle$ ) | parenthésage ; attention à ne pas confondre ces parenthèses avec les terminaux ( et ) |

Les mots clefs du langage sont les suivants :

```
array           candidate       case            const
forall_other    forward         init            invariant
number_procs    requires        transition      type
unsafe          var
```

Espaces, tabulations, retour-chariots, et sauts de ligne constituent les blancs. Les commentaires débutent par `(*` et s'étendent jusqu'à `*)`, et peuvent être imbriqués. Les identificateurs de variables obéissent à l'expression régulière ⟨*mident*⟩, les autres identificateurs obéissent à l'expression ⟨*lident*⟩. Les constantes entières (resp. réelles) sont reconnues par l'expression régulière ⟨*integer*⟩ (resp. ⟨*real*⟩). Les constantes représentant les identificateurs de processus commencent par le symbole #, suivi d'un nombre (expression ⟨*constproc*⟩).

⟨*integer*⟩ ::= (0–9)$^+$
⟨*real*⟩ ::= (0–9)$^+$ . (0–9)$^\star$
⟨*constproc*⟩ ::= # (1–9)$^+$(0–9)$^\star$
⟨*mident*⟩ ::= (A–Z) (a–z | A–Z | 0–9 | _ )$^\star$
⟨*lident*⟩ ::= (a–z) (a–z | A–Z | 0–9 | _ )$^\star$

La grammaire des fichiers sources de Cubicle est donnée figure A.1. Le point d'entrée est le non-terminal ⟨*system*⟩.

Les associativités et précédences des divers opérateurs et constructions sont données par la table suivante, de la plus faible à la plus forte précédence :

| **opérateur ou construction** | **associativité** |
|---|---|
| `+` `-` | gauche |
| `\|\|` | droite |
| `&&` | droite |
| `forall_other` | — |
| `\|` | gauche |

## A.2 Typage

Cette section donne les règles de typage pour le langage d'entrée de Cubicle. L'algorithme de vérification de typage se compose de cinq fonctions. La première, notée $\vdash_T$, permet de vérifier les types des termes. La deuxième, notée $\vdash_F$, est utilisée pour typer les formules. La troisième est notée $\vdash_D$ et permet de vérifier le typage des déclarations. La quatrième, notée $\vdash_A$ assure qu'une action d'une transition est bien typée. Enfin, la cinquième, notée $\vdash_{Ty}$ permet de vérifier la bonne formation des déclarations de type. Ces cinq fonctions manipulent les trois environnements de typage suivants :

⟨*system*⟩ ::= ⟨*size_proc*⟩ ? ⟨*type_def*⟩$^\star$ ⟨*decl*⟩$^\star$
⟨*init*⟩ ⟨*invariant*⟩$^\star$ ⟨*unsafe*⟩$^+$ ⟨*transition*⟩$^\star$

⟨*size_proc*⟩ ::= `number_procs` ⟨*integer*⟩

⟨*type_def*⟩ ::= `type` ⟨*lident*⟩
| `type` ⟨*lident*⟩ `=` ( `|` ) ? ⟨*mident*⟩$^+_{|}$

⟨*decl*⟩ ::= `const` ⟨*mident*⟩ `:` ⟨*lident*⟩
| `var` ⟨*mident*⟩ `:` ⟨*lident*⟩
| `array` ⟨*mident*⟩ `[` ⟨*lident*⟩$^+_{,}$ `]` `:` ⟨*lident*⟩

⟨*init*⟩ ::= `init` `(` ⟨*lident*⟩$^\star$ `)` `{` ⟨*dnf*⟩ `}`

⟨*invariant*⟩ ::= `invariant` `(` ⟨*lident*⟩$^\star$ `)` `{` ⟨*cube*⟩ `}`

⟨*unsafe*⟩ ::= `unsafe` `(` ⟨*lident*⟩$^\star$ `)` `{` ⟨*cube*⟩ `}`

⟨*transition*⟩ ::= `transition` ( ⟨*lident*⟩ | ⟨*mident*⟩ ) `(` ⟨*lident*⟩$^\star$ `)`
( `requires` `{` ( ⟨*cube*⟩ | ⟨*udnf*⟩ )$^\star_{\&\&}$ `}` )$^\star$
`{` ( ⟨*assign*⟩ | ⟨*update*⟩ )$^+_{;}$ `}`

⟨*udnf*⟩ ::= `forall_other` ⟨*lident*⟩ ⟨*literal*⟩
| `forall_other` ⟨*lident*⟩ `(` ⟨*dnf*⟩ `)`

⟨*dnf*⟩ ::= ⟨*cube*⟩
| ⟨*cube*⟩ `||` ⟨*dnf*⟩

⟨*cube*⟩ ::= ⟨*literal*⟩
| ⟨*literal*⟩ `&&` ⟨*cube*⟩

⟨*literal*⟩ ::= ⟨*term*⟩ ⟨*operator*⟩ ⟨*term*⟩

⟨*acces*⟩ ::= ⟨*mident*⟩ `[` ( ⟨*lident*⟩ | ⟨*constproc*⟩ )$^+_{,}$ `]`

⟨*varterm*⟩ ::= ⟨*mident*⟩ | ⟨*lident*⟩ | ⟨*constproc*⟩ | ⟨*acces*⟩

⟨*term*⟩ ::= ⟨*varterm*⟩
| ⟨*integer*⟩
| ⟨*real*⟩
| ⟨*varterm*⟩ ( `+` | `-` ) ( ⟨*integer*⟩ | ⟨*real*⟩ | ⟨*mident*⟩ )

⟨*operator*⟩ ::= `=` | `<>` | `<` | `<=`

⟨*assign*⟩ ::= ⟨*mident*⟩ `:=` `?`
| ⟨*mident*⟩ `:=` ⟨*term*⟩

⟨*update*⟩ ::= ⟨*acces*⟩ `:=` ⟨*term*⟩
| ⟨*acces*⟩ `:=` `case` ( ⟨*cube*⟩ `:` ⟨*term*⟩ )$^\star$ `|` `_` `:` ⟨*term*⟩

FIGURE A.1 – Grammaire des fichiers Cubicle

— l'environnement de types $\Theta$ contient les liaisons $\mathtt{t} : \{\mathtt{C}_1, \ldots, \mathtt{C}_n\}$ correspondant aux déclarations de types de la forme `type t = C1 | ... | Cn`, où $\{\mathtt{C}_1, \ldots, \mathtt{C}_n\}$ est l'ensemble des constructeurs pour un type énuméré. Lorsque la liaison $\mathtt{t} : \emptyset$ apparaît dans $\Theta$, alors le type `t` est *abstrait*.

— l'environnement $\Delta$ contient les liaisons de la forme $x : \rho$ pour les variables *globales* ou tableaux définis à l'aide des déclarations de la forme `var` $x : \rho$ ou `array` $x[\rho_1, \ldots, \rho_n] : \rho$.

— l'environnement $\Gamma$ contient les variables *locales* déclarées à l'aide des quantificateurs $\forall x : \tau$ et $\exists x : \tau$.

Les algorithmes sous-jacents à ces fonctions et procédures sont définis à l'aide de séquents de la forme suivante :

1. $\Delta, \Gamma \vdash_T t : \tau$ se lit « dans l'environnement global $\Delta$ et l'environnement local $\Gamma$, le terme $t$ est bien typé et son type est $\tau$ ».
2. $\Theta \vdash_{Ty} t$ se lit « le type `t` est bien formé dans l'environnement de types $\Theta$ ».
3. $\Delta, \Gamma \vdash_F \varphi$ qui se lit « dans l'environnement global $\Delta$ et l'environnement local $\Gamma$, la formule $\varphi$ est bien typée ». Ici, $\Delta$, $\Gamma$ et $\varphi$ sont les entrées de la fonction, qui est en fait une procédure car elle ne retourne rien.
4. $\Delta, \Gamma \vdash_A a$ qui se lit « dans l'environnement global $\Delta$ et l'environnement local $\Gamma$, l'action $a$ est bien typée ». Ici, $\vdash_A$ est également une procédure ayant pour entrées $\Delta$, $\Gamma$ et $a$.
5. $\Theta, \Delta \vdash_D d : (\Theta', \Delta')$ se lit « dans l'environnement de types $\Theta$ et l'environnement global $\Delta$, la vérification de la déclaration $d$ produit un nouvel environnement de typage $\Theta'$ et un nouvel environnement global $\Delta'$ ».

Les règles d'inférence définissant la fonction $\vdash_T$ sont présentées dans la figure A.2. Les règles pour vérifier la bonne formation des types sont données en figure A.5. La procédure $\vdash_F$ est définie à l'aide des règles de la figure A.3. La procédure pour les actions, $\vdash_A$, est définie par les règles de la figure A.4. Enfin, les règles d'inférences de la figure A.6 définissent la fonction $\vdash_D$. Ces règles utilisent les notations suivantes :

— $\Theta + \mathtt{t} : S$ est l'environnement de typage obtenu en ajoutant une liaison $\mathtt{t} : S$ ;

— « $\bar{\alpha}$ disjoints » signifie que les éléments de l'ensemble $\bar{\alpha}$ sont deux à deux disjoints ;

— $\mathtt{t} \in \Theta$ signifie « il existe une liaison pour `t` dans $\Theta$ » ;

— $\Delta + x : \rho$ (resp. $\Gamma + x : \tau$) l'ensemble obtenu en ajoutant une liaison $x : \rho$ (resp. $x : \tau$) à $\Delta$ (resp. $\Gamma$) ;

— $\Delta, \Gamma \vdash_T \bar{e} : \bar{\tau}$ est un raccourci syntaxique pour la conjonction $\Delta, \Gamma \vdash_T e_i : \tau_i$.

$$\textsc{Const}\ \frac{c \text{ est une constante de type } \tau \qquad \tau \in \{\texttt{int}, \texttt{real}\}}{\Delta, \Gamma \vdash_T c : \tau}$$

$$\textsc{Arith}\ \frac{\Delta, \Gamma \vdash_T t_1 : \tau \qquad \Delta, \Gamma \vdash_T t_2 : \tau \qquad \tau \in \{\texttt{int}, \texttt{real}\}}{\Delta, \Gamma \vdash_T t_1 + t_2 : \tau}$$

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Delta, \Gamma \vdash_T x : \tau}$$

$$\textsc{Glob}\ \frac{x \notin \Gamma \qquad x : \tau \in \Delta}{\Delta, \Gamma \vdash_T x : \tau}$$

$$\textsc{Array}\ \frac{\mathtt{A} : \bar{\tau} \to \tau \in \Delta \qquad \Delta, \Gamma \vdash_T \bar{e} : \bar{\tau}}{\Gamma \vdash_T \mathtt{A}[\bar{e}] : \tau}$$

FIGURE A.2 – Règles de typage des termes

$$\textsc{True}_F\ \frac{}{\Delta,\Gamma \vdash_F \mathsf{true}} \qquad \textsc{False}_F\ \frac{}{\Delta,\Gamma \vdash_F \mathsf{false}}$$

$$\textsc{Comp}_F\ \frac{\Delta,\Gamma \vdash_T e_1 : \tau \qquad \Delta,\Gamma \vdash_T e_2 : \tau \qquad op \in \{=, <>\}}{\Delta,\Gamma \vdash_T e_1\ op\ e_2}$$

$$\textsc{AComp}_F\ \frac{\Delta,\Gamma \vdash_T e_1 : \tau \qquad \Delta,\Gamma \vdash_T e_2 : \tau \qquad op \in \{<=, <\} \qquad \tau \in \{\mathtt{int}, \mathtt{real}\}}{\Delta,\Gamma \vdash_F e_1\ op\ e_2}$$

$$\textsc{Connectors}_F\ \frac{\Delta,\Gamma \vdash_F e_1 \qquad \Delta,\Gamma \vdash_F e_2 \qquad op \in \{\&\&, ||\}}{\Delta,\Gamma \vdash_F e_1\ op\ e_2}$$

$$\textsc{Forall}_F\ \frac{\bar{x} \notin \Gamma \qquad \bar{x}\ \text{disjoints} \qquad \Delta,\Gamma + \bar{x} : \mathtt{proc} \vdash_F e}{\Delta,\Gamma \vdash_F \forall \bar{x}.\ e}$$

$$\textsc{Exists}_F\ \frac{\bar{x} \notin \Gamma \qquad \bar{x}\ \text{disjoints} \qquad \Delta,\Gamma + \bar{x} : \mathtt{proc} \vdash_F e}{\Delta,\Gamma \vdash_F \exists \bar{x}.\ e}$$

Figure A.3 – Règles de typage des formules

$$\textsc{Assign}\ \frac{\bar{x} \in \Delta \qquad \Delta,\Gamma \vdash_T x : \tau \qquad \Delta,\Gamma \vdash_T e : \tau}{\Delta,\Gamma \vdash_A x := e}$$

$$\textsc{NonDet}\ \frac{\bar{x} \in \Delta \qquad \Delta,\Gamma \vdash_T x : \tau}{\Delta,\Gamma \vdash_A x := ?}$$

$$\textsc{Update}\ \frac{\begin{array}{c}\bar{x} \notin \Gamma \qquad \bar{x}\ \text{disjoints} \qquad \Delta,\Gamma + \bar{x} : \mathtt{proc} \vdash_T \mathtt{A}[\bar{x}] : \tau \\ \Delta,\Gamma + \bar{x} : \mathtt{proc} \vdash_F c_1, \ldots, c_n \qquad \Delta,\Gamma + \bar{x} : \mathtt{proc} \vdash_T e_1 : \tau, \ldots, e_{n+1} : \tau\end{array}}{\Delta,\Gamma \vdash_A \mathtt{A}[\bar{x}] := \mathtt{case}\ |\ c_1 : e_1\ |\ \ldots\ |\ c_n : e_n\ |\ _ : e_{n+1} : \tau}$$

Figure A.4 – Règles de typage des actions des transitions

$$\textsc{BultinType}\ \frac{\tau \in \{\texttt{int}, \texttt{real}, \texttt{bool}, \texttt{proc}\}}{\Theta \vdash_{Ty} \tau}$$

$$\textsc{EnumType}\ \frac{t : \{\texttt{C}_1, \ldots, \texttt{C}_n\} \in \Theta}{\Theta \vdash_{Ty} \texttt{t}} \qquad \textsc{AbstrType}\ \frac{t : \emptyset \in \Theta}{\Theta \vdash_{Ty} \texttt{t}}$$

FIGURE A.5 – Vérification de la bonne formation des types

$$\textsc{AbstrTypeDecl}\ \frac{\texttt{t} \notin \Theta}{\Theta, \Delta \vdash_D \texttt{type t} : (\Theta + \texttt{t} : \emptyset,\ \Delta)}$$

$$\textsc{EnumTypeDecl}\ \frac{\texttt{t} \notin \Theta \qquad \texttt{C}_1, \ldots, \texttt{C}_n \notin \Delta \qquad \texttt{C}_1, \ldots, \texttt{C}_n \text{ disjoints}}{\begin{array}{c}\Theta, \Delta \vdash_D \texttt{type t = C}_1 \texttt{ | } \ldots \texttt{ | C}_n : \\ (\Theta + \texttt{t} : \{\texttt{C}_1, \ldots, \texttt{C}_n\},\ \Delta + \texttt{C}_1 : \texttt{t} + \ldots + \texttt{C}_n : \texttt{t})\end{array}}$$

$$\textsc{VarDecl}\ \frac{x \notin \Delta \qquad \Theta \vdash_{Ty} \tau}{\Theta, \Delta \vdash_D \texttt{var}\ x\ :\ \tau : (\Theta, \Delta + x : \tau)}$$

$$\textsc{ArrayDecl}\ \frac{\texttt{A} \notin \Delta \qquad \Theta \vdash_{Ty} \bar{\tau} \qquad \Theta \vdash_{Ty} \tau}{\Theta, \Delta \vdash_D \texttt{array A}[\bar{\tau}]\ :\ \tau : (\Theta, \Delta + \texttt{A} : \bar{\tau} \rightarrow \tau)}$$

$$\textsc{Init}\ \frac{\Delta, \emptyset \vdash_F \forall \bar{x}.\ e}{\Theta, \Delta \vdash_D \texttt{init}\ (\bar{x})\ \{\ e\ \} : (\Theta, \Delta)}$$

$$\textsc{Invariant}\ \frac{\Delta, \emptyset \vdash_F \exists \bar{x}.\ e}{\Theta, \Delta \vdash_D \texttt{invariant}\ (\bar{x})\ \{\ e\ \} : (\Theta, \Delta)}$$

$$\textsc{Unsafe}\ \frac{\Delta, \emptyset \vdash_F \exists \bar{x}.\ e}{\Theta, \Delta \vdash_D \texttt{unsafe}\ (\bar{x})\ \{\ e\ \} : (\Theta, \Delta)}$$

$$\textsc{Transition}\ \frac{\bar{x} \text{ disjoints} \qquad \Delta, \{\bar{x} : \texttt{proc}\} \vdash_F g \qquad \Delta, \{\bar{x} : \texttt{proc}\} \vdash_A a}{\Theta, \Delta \vdash_D \texttt{transition}\ t\ (\bar{x})\ \texttt{requires}\ \{\ g\ \}\ \{\ a\ \} : (\Theta, \Delta)}$$

FIGURE A.6 – Règles de typage des déclarations

# Bibliographie


[1] P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parameterized system verification. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, pages 134–145, London, UK, UK, 1999. Springer-Verlag.

[2] P. A. Abdulla, G. Delzanno, N. Ben Henda, and A. Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 721–736. Springer Berlin Heidelberg, 2007.

[3] P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 145–157. Springer Berlin Heidelberg, 2007.

[4] P. A. Abdulla, F. Haziza, and L. Holík. All for the price of few. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 476–495. Springer, 2013.

[5] P. A. Abdulla, N. Henda, G. Delzanno, and A. Rezine. Handling parameterized systems with non-atomic global conditions. In F. Logozzo, D. A. Peled, and L. D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4905 of *Lecture Notes in Computer Science*, pages 22–36. Springer Berlin Heidelberg, 2008.

[6] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321. IEEE Computer Society, 1996.

[7] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy abstraction with interpolants for arrays. In N. Bjørner and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *Lecture Notes in Computer Science*, pages 46–61. Springer Berlin Heidelberg, 2012.

[8] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Automated support for the design and validation of fault tolerant parameterized systems : a case study. *ECEASST*, 35, 2010.

[9] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi. Universal guards, relativization of quantifiers, and failure models in model checking modulo theories. *JSAT*, pages 29–61, 2012.

[10] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6) :307–309, May 1986.

[11] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *Model Checking Software : 13th International SPIN Workshop, Vienna, Austria, March 30-April 1, 2006, Proceedings*, volume 3925, page 146. Springer, 2006.

[12] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In G. Berry, H. Comon, and A. Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 221–234. Springer, 2001.

[13] T. Ball and S. K. Rajamani. Boolean programs : A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, February 2000.

[14] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE : Parallel Distributed Model Checker (Tool paper). In *HiBi/PDMC*, pages 4–7, 2010.

[15] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized verification of a cache coherence protocol : Safety and liveness. In *VMCAI*, pages 317–330. Springer, 2002.

[17] S. Bevc and I. Savnik. Using tries for subset and superset queries. In V. Luzar-Stiffler, I. Jarec, and Z. Bekic, editors, *ITI*, pages 147–152. IEEE, 2009.

[18] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast : Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5) :505–525, Oct. 2007.

[19] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58 :117–148, 2003.

[20] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a c value analysis based on abstract interpretation. In F. Logozzo and M. Fähndrich, editors, *SAS*, volume 7935 of *Lecture Notes in Computer Science*, pages 324–344. Springer, 2013.

[21] F. Bobot, S. Conchon, É. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The alt-ergo automated theorem prover, 2008.

[22] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular (tree) model checking. *Int. J. Softw. Tools Technol. Transf.*, 14(2) :167–191, Apr. 2012.

[23] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Computer Aided Verification*, pages 372–386. Springer Berlin Heidelberg, 2004.

[24] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 403–418, London, UK, UK, 2000. Springer-Verlag.

[25] A. R. Bradley. Sat-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] G. Brat and W. Visser. Combining static analysis and model checking for software analysis. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 262–, Washington, DC, USA, 2001. IEEE Computer Society.

[27] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8) :677–691, Aug. 1986.

[28] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 78–92, London, UK, UK, 2002. Springer-Verlag.

[29] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking : $10^{20}$ states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, 1990. LICS'90*, pages 428–439, June 1990.

[30] T. D. Chandra and S. Toueg. Time and message efficient reliable broadcasts. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, pages 289–303, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[31] X. Chen, Y. Yang, M. Delisi, G. Gopalakrishnan, and C.-T. Chou. Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, pages 107–114, Nov. 2007.

[32] D. Chkliaev, J. Hooman, and P. van der Stok. Mechanical verification of transaction processing systems. In *Proceedings of the 3rd IEEE International Conference on Formal Engineering Methods*, ICFEM '00, pages 89–, Washington, DC, USA, 2000. IEEE Computer Society.

[33] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*, pages 382–398. Springer, 2004.

[34] A. Cimatti and A. Griggio. Software model checking via ic3. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 277–293, Berlin, Heidelberg, 2012. Springer-Verlag.

[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 2000.

[36] E. Clarke and R. Kurshan. Computer-aided verification. *Spectrum, IEEE*, 33(6) :61–67, Jun 1996.

[37] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *LICS*, pages 353–362. IEEE Press, 1989.

[38] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[39] E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '86, pages 240–248, New York, NY, USA, 1986. ACM.

[40] E. M. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2006.

[41] A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. *Form. Methods Syst. Des.*, 34(2) :104–125, Apr. 2009.

[42] S. Conchon, E. Contejean, J. Kanig, and S. Lescuyer. CC(X) : Semantic combination of congruence closure with solvable theories. *ENTCS*, 198(2) :51–69, 2008.

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

[44] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Invariants for finite instances and beyond. In *FMCAD*, pages 61–68. IEEE, 2013.

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

[46] 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.

[47] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3) :103–179, 1992.

[48] W. Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *The Journal of Symbolic Logic*, 22(03) :250–268, 1957.

[49] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV*, pages 160–171. Springer, 1999.

[50] M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Trans. Math. Softw.*, 37(1) :2 :1–2 :20, Jan. 2010.

[51] L. de Moura and N. Bjørner. Z3 : An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[52] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction : From refutation to verification. In W. A. Hunt Jr. and F. Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26. Springer Berlin Heidelberg, 2003.

[53] L. E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *American Journal of Mathematics*, 35(4) :pp. 413–422, 1913.

[54] E. W. Dijkstra. Cooperating sequential processes, technical report EWD-123. Technical report, 1965.

[55] D. L. Dill. The murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 390–393, London, UK, UK, 1996. Springer-Verlag.

[56] D. L. Dill. 25 years of model checking. chapter A Retrospective on Mur$\varphi$, pages 77–88. Springer-Verlag, Berlin, Heidelberg, 2008.

[57] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525, 1992.

[58] G. Dowek, A. Felty, H. Herbelin, G. Huet, B. Werner, C. Paulin-Mohring, et al. The coq proof assistant user's guide : Version 5.6. 1991.

[59] K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. Slab : A certifying model checker for infinite-state concurrent systems. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 271–274. Springer, 2010.

[60] C. Dross, S. Conchon, J. Kanig, and A. Paskevich. Reasoning with triggers. In P. Fontaine and A. Goel, editors, *SMT@IJCAR*, volume 20 of *EPiC Series*, pages 22–31. EasyChair, 2012.

[61] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[62] E. A. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 247–262. Springer Berlin Heidelberg, 2003.

[63] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 87–98. Springer Berlin Heidelberg, 1996.

[64] E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *LICS*, pages 70–80. IEEE Computer Society, 1998.

[65] M. Emmi, R. Majumdar, and R. Manevich. Parameterized verification of transactional memories. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 134–145, New York, NY, USA, 2010. ACM.

[66] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable ltl model checker. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 463–478. Springer, 2013.

[67] J.-C. Filliâtre and K. Kalyanasundaram. Functory : A distributed computing library for Objective Caml. In *TFP*, pages 65–81, 2011.

[68] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.

[69] A. Finkel. Reduction and covering of infinite reachability trees. *Inf. Comput.*, 89(2) :144–179, 1990.

[70] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere ! *Theor. Comput. Sci.*, 256(1–2) :63–92, 2001.

[71] R. W. Floyd. Assigning meanings to programs. In T. R. Colburn, J. H. Fetzer, and T. L. Rankin, editors, *Program Verification*, volume 14 of *Studies in Cognitive Systems*, pages 65–81. Springer Netherlands, 1993.

[72] P.-L. Garoche, T. Kahsai, and C. Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 139–154. Springer Berlin Heidelberg, 2013.

[73] Y. Ge and L. Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 306–320, Berlin, Heidelberg, 2009. Springer-Verlag.

[74] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards smt model checking of array-based systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2008.

[75] S. Ghilardi and S. Ranise. Goal-directed invariant synthesis for model checking modulo theories. In M. Giese and A. Waaler, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 5607 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin Heidelberg, 2009.

[76] S. Ghilardi and S. Ranise. Model checking modulo theory at work : the intergration of yices in MCMT. In *Proceedings of AFM 09*. ACM Digital Library, 2009.

[77] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving : Termination and invariant synthesis. *LMCS*, 6(4), 2010.

[78] S. Ghilardi and S. Ranise. MCMT : A model checker modulo theories. In *IJCAR*, pages 22–29, 2010.

[79] S. Ghilardi, S. Ranise, and T. Valsecchi. Light-weight smt-based model checking. *Electr. Notes Theor. Comput. Sci.*, 250(2) :85–102, 2009.

[80] L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In K. Yi, editor, *Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 144–160. Springer Berlin Heidelberg, 2006.

[81] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer Berlin Heidelberg, 1997.

[82] O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *IJCAR*, pages 483–497. Springer, 2006.

[83] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *CHARME*, pages 129–145, 2005.

[84] A. Gurfinkel and S. Chaki. Combining predicate and numeric abstraction for software model checking. *International Journal on Software Tools for Technology Transfer*, 12(6) :409–427, 2010.

[85] G. Hagen and C. Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08, pages 15 :1–15 :9, Piscataway, NJ, USA, 2008. IEEE Press.

[86] G. E. Hagen. *Verifying Safety Properties of Lustre Programs : An SMT-based Approach*. PhD thesis, Iowa City, IA, USA, 2008. AAI3347220.

[87] M. A. Heinrich. *The Performance and Scalability of Distributed Shared-memory Cache Coherence Protocols*. PhD thesis, Stanford, CA, USA, 1999. AAI9924431.

[88] N. B. Henda and A. Rezine. The PFS prototype model checker. `http://www.it.uu.se/research/docs/fm/apv/tools/pfs/`.

[89] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *SIGPLAN Not.*, 37(1) :58–70, Jan. 2002.

[90] G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, s3-2(1) :326–336, 1952.

[91] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, Oct. 1969.

[92] W. Hodges. *A Shorter Model Theory*. Cambridge University Press, New York, NY, USA, 1997.

[93] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5) :279–295, May 1997.

[94] R. Jhala. *Program verification by lazy abstraction*. PhD thesis, University of California, 2004.

[95] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4) :21 :1–21 :54, Oct. 2009.

[96] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 220–235. Springer Berlin Heidelberg, 2000.

[97] T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-based invariant discovery. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 192–206, Berlin, Heidelberg, 2011. Springer-Verlag.

[98] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 645–659. Springer Berlin Heidelberg, 2010.

[99] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 424–435. Springer Berlin Heidelberg, 1997.

[100] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[101] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7) :385–394, July 1976.

[102] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4 : Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[103] K. Korovin. iprover–an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning*, pages 292–298. Springer, 2008.

[104] S. Krstić. Parametrized system verification with guard strengthening and parameter abstraction. In *AVIS*, 2005.

[105] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, PODC '89, pages 239–247, New York, NY, USA, 1989. ACM.

[106] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Computer Architecture*, pages 302–313, Apr. 1994.

[107] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 267–281. Springer Berlin Heidelberg, 2004.

[108] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In R. Alur and D. A. Peled, editors, *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 135–147. Springer Berlin Heidelberg, 2004.

[109] S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Logic*, 9(1), Dec. 2007.

[110] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8) :453–455, Aug. 1974.

[111] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9) :690–691, Sept. 1979.

[112] X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4) :363–446, Dec. 2009.

[113] D. Levinthal. Performance analysis guide for Intel® Core™ i7 processor and Intel® Xeon™ 5500 processors. Technical report, 2009.

[114] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. Step : The stanford temporal prover. Technical report, Stanford University, June 1994.

[115] A. J. Martin. A new generalization of dekker's algorithm for mutual exclusion. *Inf. Process. Lett.*, 23(6) :295–297, Dec. 1986.

[116] K. L. McMillan. *Symbolic Model Checking : An Approach to the State Explosion Problem.* PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[117] K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *Proceedings of the 10th International Conference on Computer Aided Verification*, CAV '98, pages 110–121, London, UK, UK, 1998. Springer-Verlag.

[118] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *CHARME*, pages 179–195. Springer, 2001.

[119] K. L. McMillan. Lazy abstraction with interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 123–136, Berlin, Heidelberg, 2006. Springer-Verlag.

[120] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, pages 413–427. Springer, 2008.

[121] I. Melatti. CMurphi (Caching Murphi). `http://mclab.di.uniroma1.it/site/index.php/software/18-cmurphi`.

[122] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in Eddy. *STTT*, 11(1) :13–25, 2009.

[123] M. L. Minsky. *Computation : Finite and Infinite Machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[124] D. P. Monniaux. Automatic modular abstractions for linear constraints. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 140–151, New York, NY, USA, 2009. ACM.

[125] L. Moura and N. Bjørner. Efficient e-matching for SMT solvers. In *Proceedings of the 21st International Conference on Automated Deduction : Automated Deduction*, CADE-21, pages 183–198, Berlin, Heidelberg, 2007. Springer-Verlag.

[126] K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, pages 299–313, 2007.

[127] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2) :356–364, Apr. 1980.

[128] M. Nilsson. *Regular model checking*. PhD thesis, 2005.

[129] D. Oe, A. Stump, C. Oliver, and K. Clancy. versat : A verified modern sat solver. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 363–378. Springer Berlin Heidelberg, 2012.

[130] J. W. O'Leary, M. Talupur, and M. R. Tuttle. Protocol verification using flows : An industrial experience. In *FMCAD*, pages 172–179. IEEE, 2009.

[131] S. Owicki and D. Gries. Verifying properties of parallel programs : An axiomatic approach. *Commun. ACM*, 19(5) :279–285, May 1976.

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

[133] S. Owre, J. M. Rushby, and N. Shankar. Pvs : A prototype verification system. In *Automated Deduction—CADE-11*, pages 748–752. Springer, 1992.

[134] S. Park and D. L. Dill. Protocol verification by aggregation of distributed transactions. In *CAV*, pages 300–310. Springer, 1996.

[135] S. Park and D. L. Dill. Verification of flash cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, pages 288–296, New York, NY, USA, 1996. ACM.

[136] G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli. Exploiting transition locality in automatic verification of finite-state concurrent systems. *STTT*, 6(4) :320–341, 2004.

[137] A. Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the International Symposium on Semantics of Concurrent Computation*, pages 1–20, London, UK, UK, 1979. Springer-Verlag.

[138] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2001, pages 82–97, London, UK, UK, 2001. Springer-Verlag.

[139] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, $\infty$)-counter abstraction. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2002.

[140] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[141] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. Finite model finding in smt. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 640–655. Springer Berlin Heidelberg, 2013.

[142] A. Reynolds, C. Tinelli, A. Goel, S. Krstić, M. Deters, and C. Barrett. Quantifier instantiation techniques for finite model finding in smt. In M. P. Bonacina, editor, *Automated Deduction – CADE-24*, volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer Berlin Heidelberg, 2013.

[143] A. Rezine. UNDIP. `http://www.it.uu.se/research/docs/fm/apv/tools/undip`.

[144] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *CADE*, volume 1632 of *Lecture Notes in Computer Science*, pages 292–296. Springer, 1999.

[145] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1) :9–17, Jan. 1981.

[146] H. Rueß and N. Shankar. Deconstructing shostak. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, pages 19–, Washington, DC, USA, 2001. IEEE Computer Society.

[147] H. Saïdi. Model checking guided abstraction and analysis. In *Proceedings of the 7th International Symposium on Static Analysis*, SAS '00, pages 377–396, London, UK, UK, 2000. Springer-Verlag.

[148] I. Savnik. Efficient subset and superset queries. In A. Caplinskas, G. Dzemyda, A. Lupeikiene, and O. Vasilecas, editors, *DB&Local Proceedings*, volume 924 of *CEUR Workshop Proceedings*, pages 45–57. CEUR-WS.org, 2012.

[149] M. Schmidt-Schauß. *Computational Aspects of an Order-sorted Logic with Term Declarations*. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[150] S. Schulz. System description : E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 735–743. Springer, 2013.

[151] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In W. A. Hunt Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 127–144. Springer Berlin Heidelberg, 2000.

[152] J. R. Shoenfield. *Mathematical logic*, volume 21. Addison-Wesley Reading, 1967.

[153] R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1) :1–12, Jan. 1984.

[154] J. P. M. Silva and K. A. Sakallah. GRASP — a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[155] A. Sistla and M. Zhou. Combining static analysis and model checking for systems employing commutative functions. In F. Wang, editor, *Formal Techniques for Networked and Distributed Systems - FORTE 2005*, volume 3731 of *Lecture Notes in Computer Science*, pages 68–82. Springer Berlin Heidelberg, 2005.

[156] C. Sprenger. A verified model checker for the modal $\mu$-calculus in coq. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 167–183, London, UK, UK, 1998. Springer-Verlag.

[157] B. K. Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In *Proceedings of the 2nd International Conference on Supercomputing*, ICS '88, pages 621–626, New York, NY, USA, 1988. ACM.

[158] M. Talupur and M. R. Tuttle. Going with the flow : Parameterized verification using message flows. In *FMCAD*, pages 1–8. IEEE, 2008.

[159] A. S. Tanenbaum and M. Van Steen. *Distributed Systems : Principles and Paradigms ($2^{nd}$ Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[160] A. Thakur and T. Reps. A method for symbolic computation of abstract operations. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 174–192, Berlin, Heidelberg, 2012. Springer-Verlag.

[161] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.

[162] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In *Automated Deduction–CADE-22*, pages 140–145. Springer, 2009.

[163] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 68–80, London, UK, UK, 1990. Springer-Verlag.

[164] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, SFCS '83, pages 185–194, Washington, DC, USA, 1983. IEEE Computer Society.

[165] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.

# Index


**Symboles**

$S_{|\mathtt{ty}}$ . . . . . 41
. . . . . 55
$\binom{A}{k}$ . . . . . 52
$[\![.]\!]$ . . . . . 62
$\models$ . . . . . 40

**A**

API . . . . . 85, 158
arbre prefixe . . . . . 86
atteignabilité . . . . . 43

**B**

bel ordre . . . . . 60
boulangerie, algorithme de la . . . . . 28

**C**

clôture supérieure . . . . . 60
cohérence de cache . . . . . 30
cohérence séquentielle . . . . . 133
configuration . . . . . 59
conséquence logique . . . . . 40
cube . . . . . 51

**D**

deadlock *voir* iterblocage . . . . . 48
Dekker, algorithme de . . . . . 25
domaine (d'une structure) . . . . . 39

**E**

exclusion mutuelle . . . . . 23

**I**

idéal . . . . . 60
insatisfiable . . . . . 40
interblocage . . . . . 48
interprétation . . . . . 39
invariant . . . . . 43
    inductif . . . . . 148

**M**

modèles . . . . . 39

**P**

plongement . . . . . 60
post-image . . . . . 50
pré-image . . . . . 51
pré-ordre . . . . . 59
    bien fondé . . . . . 60

**S**

sûreté . . . . . 43
satisfiabilité modulo théorie . . . . . 40
satisfiable . . . . . 40
section finissante *voir* côture supérieure 60
signature . . . . . 37
    quasi-relationnelle . . . . . 38
    relationnelle . . . . . 38
Skolem
    constante de . . . . . 56
    fonction de . . . . . 56
    forme normale de . . . . . 55
Skolemisation . . . . . 56
SMT . . . . . 40
sous-cube . . . . . 120
    strict . . . . . 120
sous-structure . . . . . 40

générée . . . . . 40
structure . . . . . 39
structure de Kripke . . . . . 46
système à tableaux . . . . . 42

**T**
théorie . . . . . 39
close par sous-structure . . . . . 40
de l'arithmétique . . . . . 41
de types énumérés . . . . . 41
localement finie . . . . . 40
vide . . . . . 41
transition
paramétrée . . . . . 42
trie *voir* abre préfixe . . . . . 86
type énuméré . . . . . 41

**U**
univers . . . . . 39
unsat core . . . . . 85

**V**
vérification uniforme . . . . . 57
valide . . . . . 40

**W**
Why3 . . . . . 147
WP . . . . . 158