UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS
LABORATOIRE DE SÛRETÉ ET SÉCURITÉ DES LOGICIELS, CEA LIST

THÈSE présentée par :

Allan BLANCHARD

soutenue le : 6 Décembre 2016
pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline : **Informatique**

Aide à la vérification de programmes concurrents par transformation de code et de spécifications

THÈSE DIRIGÉE PAR :

Frédéric LOULERGUE	Professeur, Université d'Orléans

RAPPORTEURS :

Catherine DUBOIS	Professeur, ENSIIE
Sylvain CONCHON	Professeur, Université Paris-Sud XI

JURY :

Sébastien LIMET	Professeur, Université d'Orléans, Président
Nikolai KOSMATOV	Ingénieur-Chercheur, CEA List, Encadrant
Stephan MERZ	Directeur de Recherche, INRIA, Examinateur
Jan-Georg SMAUS	Professeur, Université de Toulouse, Examinateur
Louis RILLING	Ingénieur, DGA, Examinateur

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS
LABORATOIRE DE SÛRETÉ ET SÉCURITÉ DES LOGICIELS, CEA LIST

THÈSE présentée par :

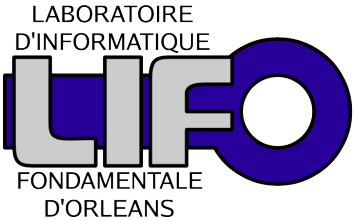
Allan BLANCHARD

soutenue le : 6 Décembre 2016

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline : **Informatique**

Aide à la vérification de programmes concurrents par transformation de code et de spécifications



Remerciements

Je remercie tout d'abord Catherine Dubois et Sylvain Conchon pour avoir accepté de rapporter ma thèse, ainsi que Stephan Merz, Jan-Georg Smaus et Louis Rilling pour avoir accepté d'examiner mes travaux. Merci également à Sébastien Limet pour avoir présidé mon jury de thèse. Un grand merci à Nikolai Kosmatov et Frédéric Loulergue pour leur disponibilité et leurs conseils, ainsi que pour avoir témoigné autant d'intérêt pour mes travaux tout au long de ces 3 années. Je remercie tous les membres du jury pour leurs commentaires et questions qui m'ont permis d'améliorer le présent manuscrit et de préparer le futur de ces travaux.

Avant d'entamer une thèse, il y a les études et on y fait des rencontres. On rencontre notamment des enseignants qui donnent envie d'aller plus loin, d'en apprendre plus. Un grand merci donc aux enseignants que j'ai pu avoir pendant mes cinq années d'étude à l'IUT puis à la Faculté et qui ont su transmettre cette envie. Merci notamment à Sylvain Jubertie, sans qui je ne me serai probablement pas engagé sur ce chemin.

Un grand merci à tous les membres du LSL, pour leur accueil, leur envie de partager (et confronter !) les connaissances. Merci à David Bühler, avec qui nous avons partagé un bureau pendant ces trois années de thèse, dont quelques mois de course à la rédaction, le soutien mutuel c'est une bonne aide pour avancer. Merci aux membres de l'équipes Frama-C pour la précieuse aide qu'ils m'ont fournie dans l'utilisation, parfois complexe admettons-le, de leur outil. N'en déplaise à certaines mauvaises langues : « *ça marche, et ça continuera de marcher* ».

Merci à ma famille pour m'avoir toujours soutenu et encouragé dans mes choix et pour leur précieuse présence pendant ces trois années. Merci également à tous mes amis avec qui il est toujours bon, le temps d'un week-end, de se détendre ensemble et d'oublier tout le reste.

Comment terminer, finalement, sans remercier Mélina, qui partage ma vie depuis plus de 10 ans maintenant et qui a su continuer à me supporter pendant ces trois années, *y compris* pendant les dernières semaines de rédaction. Je pense que je n'aurai jamais ni le temps ni les mots pour lui dire assez merci.

Table des matières

Liste des figures

Introduction

1

Sommaire

« Il n'y a pas de programme sans *bugs* ».

Cette phrase est souvent utilisée en commentaire d'actualité du monde informatique. Chaque jour, dans les logiciels que nous utilisons, de nouveaux *bugs* sont découverts, ces erreurs de programmation qui rendent parfois nos logiciels agaçants, en les fermant subitement à des moments inopportuns ou en les rendant inutilisables, ou même inquiétants, en y exposant des failles de sécurité. Cela fournit une vision assez claire de ce que pensent les utilisateurs ou même les développeurs du monde de l'informatique. Malgré tous les efforts déployés pour faire des programmes corrects, il semble que cette tâche soit insurmontable, que nous soyons condamnés à avoir des programmes qui font des erreurs.

Pourtant nous connaissons beaucoup de programmes sans *bugs*. Cependant, acquérir la certitude qu'un programme est exempt de *bug* est difficile. Une des causes de cette difficulté est la taille des codes sources de nos logiciels. Aujourd'hui, le noyau de système d'exploitation Linux a dépassé les seize *millions* de lignes de code, une distribution complète reposant sur ce noyau est de l'ordre de 200 à 300 *millions* de lignes. Même un « simple » navigateur web est composé d'au moins une dizaine de *millions* de lignes. Dans de telles quantités de code, il est difficile de s'assurer qu'aucune fonction n'est mal utilisée ou qu'aucune faute de frappe n'a changé subtilement le comportement d'une ligne.

De plus en plus de techniques existent pour faciliter le développement et la validation de systèmes informatiques. Généralement en validation, plus la confiance apportée par une technique est élevée, plus il est difficile de l'appliquer

sur de grandes quantités de code. Par exemple, tester un programme donne une assurance plutôt faible mais est facile à appliquer, même sur de grandes quantités de code. Prouver un programme est difficile sur de grandes quantités de code, mais donne une grande confiance.

Ces dernières années, un nouveau facteur est venu s'ajouter aux difficultés que l'on rencontre lorsque l'on doit écrire du logiciel à grande échelle. Nos processeurs sont devenus « multi-cœurs ». Un processeur est maintenant composé de plusieurs unités de calcul qui peuvent, dans une certaine mesure, calculer indépendamment les unes des autres, tout en étant capables d'accéder à la même mémoire. Les développeurs ont donc commencé à écrire des programmes tirant parti de tels processeurs.

Cependant, nos processeurs peuvent accéder à la même mémoire, et de fait, plusieurs parties du programme s'exécutant en même temps peuvent accéder aux mêmes informations. Il en résulte que, si prises une à une ces parties peuvent sembler tout à fait correctes, leur interaction pourrait, elle, provoquer une erreur. Par exemple, si deux parties modifient des informations de manière incohérente l'une par rapport à l'autre.

Ce problème existait déjà avant la naissance des processeurs multi-cœurs, qui l'ont rendu plus répandu. Un processeur mono-cœur suffit pour exécuter plusieurs programmes accédant à la même mémoire. Par exemple, nos systèmes d'exploitation sont depuis longtemps capables de faire fonctionner de multiples logiciels en leur donnant accès à un processeur chacun à leur tour, mais suffisamment vite, pour que l'on ait l'impression que tous ces programmes agissent en même temps.

Nous appelons « programmation concurrente », le paradigme de programmation dans lequel nous tenons compte du fait que dans un même programme, il existe plusieurs fils d'exécution, ou plusieurs processus, qui possèdent chacun leur propre mémoire, mais peuvent également accéder à une mémoire commune, par exemple pour communiquer. Du fait des difficultés de prendre en compte le déroulement et les effets de plusieurs processus exécutés en même temps, il est communément admis qu'il est plus difficile d'écrire, ou d'assurer la correction des programmes concurrents que des programmes séquentiels (ne comprenant qu'un seul programme dans son propre environnement). De ce fait, les techniques qui permettent le développement et la validation de programmes concurrents sont généralement plus difficiles à mettre en œuvre que celles dédiées aux programmes séquentiels.

Dans cette thèse, nous visons à fournir un moyen d'utiliser les techniques

de vérification dédiées aux programmes séquentiels, notamment la preuve de programmes, pour vérifier des programmes concurrents. Nous explorons ici la possibilité de transformer les programmes concurrents en programmes séquentiels équivalents, pour tirer parti des outils utilisés pour assurer le fonctionnement correct des programmes séquentiels. Nous présenterons plus en détails notre proposition en fin de chapitre. Dans un premier temps, nous allons rappeler la problématique de la validation et l'intérêt de la preuve pour l'effectuer. Cela nous permettra d'introduire quelques notations utilisées dans la suite de ce manuscrit.

1.1 Correction des programmes

Nous voulons nous assurer que nos logiciels font ce que nous voulons d'eux. Donc dans un premier temps, nous devons être capables d'exprimer ce besoin.

Dans le développement de logiciels, cela se traduit généralement par la définition d'un cahier des charges. Celui-ci décrit, aussi clairement que possible, les besoins auxquels le logiciel va devoir répondre ainsi que les contraintes sous lesquelles il devra le faire. Ce sont, le plus souvent, des documents textuels, rédigés en langue naturelle, qui sont d'autant plus vastes et complexes que le système décrit l'est lui-même. Implémenter ou vérifier la correction d'un programme nécessite de comprendre correctement ces documents. Le fait que ces documents soient en langue naturelle introduit un premier risque, autant pour l'implémentation que pour la validation : les phrases peuvent être sujettes à l'interprétation et les personnes qui vont se succéder dans la création, la maintenance et la vérification de la correction d'une fonctionnalité n'auront peut être pas la même interprétation.

Une fois la fonctionnalité implémentée, vient la deuxième étape qui est sa validation. Généralement, cette validation est effectuée par l'exécution de tests. Ils vont exécuter la fonctionnalité que l'on souhaite valider dans divers scénarios dont on connaît le résultat attendu. La fonctionnalité produira ou non ce résultat, validant ou non le test correspondant.

Par exemple, nous pouvons prendre un petit programme qui calcule la valeur absolue d'un entier :

```
1 int abs(int value){
2   return (value < 0) ? -value : value ;
3 }
```

La spécification d'une telle fonctionnalité pourrait se résumer à : « *La fonction* `abs` *calcule la valeur absolue de la valeur qu'elle reçoit en paramètre. La valeur* `INT_MIN` *est exclue de son champ d'application car sa valeur absolue n'est pas définie pour un entier de 32 bits* ».

Une suite de tests possible pour cette fonction pourrait simplement être :

entrée	Sortie
`INT_MIN+1`	`INT_MAX`
-42	42
-1	1
0	0
1	1
42	42
`INT_MAX`	`INT_MAX`

où nous testons les valeurs calculées pour les cas aux limites : les bornes minimale et maximale de l'entier reçu, des valeurs proches de 0 (marquant les valeurs pour lesquelles la fonction doit changer de comportements), mais aussi pour des valeurs supplémentaires entre ces limites.

Cependant, une telle suite ne couvre pas tous les scénarios d'utilisation possibles. Nous pouvons régler le problème en testant exhaustivement la fonction. C'est-à-dire en essayant toutes les entrées comprises entre `INT_MIN+1` et `INT_MAX`.

Mais nous ne pouvons pas toujours être exhaustif. Par exemple, si nous voulons tester une fonction effectuant le tri d'un tableau d'entiers, il faudrait tester la fonction pour toutes les tailles de tableau entre 0 et $2^{64} - 1$ et pour toutes les valeurs d'entiers pour chaque case de chaque tableau.

Généralement, l'ensemble des scénarios d'utilisation défini par le cahier des charges sera très grand, voire infini. Nous ne pouvons donc pas tous les tester. Notre campagne de tests ne pourra donc être qu'une validation partielle de ce qui est spécifié par le cahier des charges.

C'est là que les méthodes formelles entrent en jeu, nous donnant la possibilité de spécifier mathématiquement tous les comportements acceptables de notre programme, et nous donnant, la possibilité de vérifier que le code de notre implémentation assure effectivement que ces comportements sont respectés.

1.2 Preuve de programmes

Comme pour d'autres méthodes de validation et vérification, l'objectif de la preuve de programmes est de déterminer si un programme répond à sa spécification. Seulement, comme il n'est généralement pas possible d'effectuer toutes les exécutions d'un programme pour examiner leur résultat, la preuve prend le parti de faire cette vérification sans exécuter le programme. En preuve, nous raisonnons à propos des propriétés des valeurs des variables du programme, en prenant en compte les effets que vont avoir les instructions sur elles.

Par exemple :

```
1 int a = rand()%10 ;
2 a += 1;
```

Après la première instruction, nous savons qu'une propriété de `a` est $0 \leq$ `a` < 10. L'instruction suivante ajoute 1 à `a`, donc nous pouvons en déduire qu'après la deuxième instruction $1 \leq$ `a` < 11, et ceci sans exécuter le programme pour toutes les valeurs possibles de `a`.

Pour chaque instruction possible dans un langage, nous pouvons définir comment elle fait évoluer la mémoire pendant l'exécution, et donc comment les propriétés respectées par les éléments de cette mémoire évoluent. Nous pouvons, de ce fait, créer des outils qui sont capables d'effectuer ce type de raisonnement. Nous y reviendrons plus précisément dans le chapitre 2.

Le greffon Wp est l'un des analyseurs de la plateforme d'analyse de programmes C Frama-C, que nous présenterons dans le chapitre 2. Il implémente un calcul de « plus faibles pré-conditions « (que nous présenterons également ultérieurement). Ce calcul fonctionne dans le sens inverse de l'exemple précédent. À savoir que nous allons, à partir de la *post-condition* voulue du programme (« après mon programme, la valeur retournée doit être la valeur absolue du nombre en entrée »), calculer les conditions en début de programme qui garantissent cette post-condition. Ce calcul est fait en remontant les instructions une à une dans le sens inverse du programme.

Par exemple, dans le programme précédent, si nous voulons qu'il termine avec la propriété $1 \leq$ `a` < 11, que doit être la propriété de `a` avant l'exécution de `a += 1` ? Il faut que `a` respecte $0 \leq$ `a` < 10.

Si la pré-condition donnée par l'utilisateur est au moins aussi forte que la condition nécessaire calculée, le programme est vérifié.

Pour vérifier notre fonction de calcul de la valeur absolue avec Frama-C et Wp, nous devons d'abord spécifier ce que l'on attend d'elle. Cela se fait par

l'ajout d'annotations en ACSL. Nous présenterons plus précisément ce langage de spécification dans le chapitre 2, nous précisons simplement que le mot clé `requires` introduit une pré-condition, et `ensures` une post-condition. Le mot clé `\result` correspond à la valeur de retour de la fonction, et l'opérateur `==>` à l'implication logique.

```
1 /*@
2   requires INT_MIN < value;
3   ensures  value <  0 ==> \result == -value;
4   ensures  value >= 0 ==> \result == value;
5 */
6 int abs(int value){
7   return (value < 0) ? -value : value;
8 }
```

Ensuite, nous pouvons demander à FRAMA-C et WP d'effectuer la vérification, comprenant la recherche d'erreurs d'exécution potentielles, et les propriétés en question seront bien validées. La capture d'écran suivante illustre cette validation, les pastilles vertes montrent les propriétés assurées par la pré-condition. À côté de la pré-condition, nous pouvons voir un cercle bleu qui signifie qu'aucune vérification n'a été tentée. En effet, ces propriétés sont vérifiées par l'outil aux points d'appel de la fonction.

```
/*@ requires -2147483647-1 < value;
    ensures \old(value) < 0 ⇒ \result ≡ -\old(value);
    ensures \old(value) ≥ 0 ⇒ \result ≡ \old(value);
 */
int abs(int value)
{
  int tmp;
  if (value < 0) {
    /*@ assert rte: signed_overflow: -2147483647 ≤ value; */
    tmp = - value;
  }
  else {
    tmp = value;
  }
  return tmp;
}
```

Les techniques de preuve permettent d'assurer que pour tout scénario d'exécution, le programme est conforme à sa spécification. Cependant comme nous n'exécutons pas le programme, nous construisons une abstraction des états du programme et celle-ci peut s'avérer imprécise. Cela peut rendre d'une part la

preuve difficile et d'autre part rendre l'identification de *bugs* complexe. Si la preuve ne réussit pas, le programme est-il faux ? La spécification est-elle insuffisante ? Ou mon outil est-il trop limité ?

Par exemple, si nous ne donnons pas la pré-condition indiquant que la valeur reçue par `abs` doit être supérieure à `INT_MIN`, l'assertion affirmant que l'opération `-value` ne déborde pas n'est pas validée. Au premier abord, il n'est pas facile de déterminer pourquoi, car la seule information obtenue est « la preuve ne passe pas ». Dans ce genre de cas, tester devient intéressant car cela permettra de trouver des cas qui sont effectivement des erreurs.

Dans la suite de cette thèse, nous nous intéresserons en priorité aux techniques de preuve de programmes puisque c'est le premier objectif que nous avons visé lorsque nous avons développé notre méthode d'analyse.

1.3 Concurrence

Nous désirons effectuer la preuve de programmes concurrents. Par exemple, si nous avons un programme parallèle comme celui présenté ci-dessous, nous voulons lui appliquer des techniques de preuve. La syntaxe de la forme `{{{ i1 ||| i2 }}}` exprime (dans ce programme en pseudo-C) « exécuter parallèlement `i1` et `i2` ».

```
1 int x = 0;
2 {{{ /*t1*/ x++;  ||| /*t2*/ x++; }}}
```

Dans un tel programme, une propriété que l'on pourrait attendre est que la valeur `x` après l'exécution est 2 : on a deux fils d'exécution, ajoutant chacun 1 à `x` qui vaut initialement 0.

En fait, si l'on suit la sémantique de ce programme, il expose un comportement indéterminé sous la forme d'un *data-race*. Commençons par réécrire le programme avec un grain plus fin. Lorsque l'on écrit `x++`, nous avons trois opérations : `x` est lue en mémoire, incrémentée, puis réécrite en mémoire.

```
1 int x = 0;
2 {{{   // t1
3       int y1 = x;
4       x = y1+1;
5 |||   // t2
6       int y2 = x;
```

```
7         x = y2+1;
8 }}}
```

Dans un tel programme, rien ne nous garantit par exemple que si la lecture de `x` dans `y1` est 0, la lecture de `x` dans `y2` est 1. Les deux programmes peuvent très bien lire 0 en même temps puis écrire 1 dans la mémoire chacun à leur tour, car rien n'ordonne ces instructions les unes par rapport aux autres.

De manière plus générale, un *data-race* apparaît lorsque deux instructions qui ne sont pas ordonnées l'une par rapport à l'autre, dont l'une au moins est une écriture, peuvent accéder à la même donnée en mémoire et produire ainsi une incertitude sur les valeurs manipulées. Cela rend le résultat des opérations imprévisible et on considère qu'un tel programme comprend une erreur d'exécution.

Il existe divers moyens d'assurer l'absence de *data-race* dans un programme ou de vérifier cette absence. Nous ne considérons pas ce problème dans cette thèse. Ici par exemple, un moyen simple de supprimer la situation de *data-race* est d'utiliser une fonction d'incrémentation atomique. En langage C, cette fonction s'appelle `atomic_fetch_add`, et garantit que les opérations de lecture, incrément, puis écriture sont effectuées dans un pas d'exécution indivisible (atomique). Donc l'une des opérations aura lieu avant l'autre, nous ne savons pas laquelle, mais la situation de *data-race* expliquée précédemment ne peut plus arriver. Pour simplifier la lecture, nous conservons l'écriture `x++`.

Pour prouver ce programme, la difficulté principale est d'amener d'un fil d'exécution à l'autre, l'information sur le travail effectué. Par exemple ici, il nous faut un moyen pour signifier à `t2` que `t1` peut avoir déjà réalisé son incrément de `x`, et inversement.

Dans la méthode d'Owicki-Gries [77] (que nous présenterons plus en détails dans le chapitre 2), cela passe par l'ajout de variables auxiliaires qui vont permettre d'exprimer comment l'état évolue dans un fil, en fonction de l'état des autres fils.

Dans cette nouvelle version du code, on ajoute des variables auxiliaires qui indiquent pour un fil donné s'il a effectué son instruction d'incrémentation. On considère que dans un fil, l'exécution de `x++` et la modification de la variable auxiliaire correspondante sont effectuées dans le même pas d'exécution (les variables auxiliaires ne sont ici présentes que d'un point de vue logique, elles n'existent pas réellement dans le programme).

```
1 int x = 0;
2 /* done_1 = 0 ; done_2 = 0 ; */
3 {{{ /*t1*/
4       x++; /*done_1 = 1*/
5 ||| /*t2*/
6       x++; /*done_2 = 1*/
7 }}}
```

Pour prouver ce programme, l'idée de la méthode d'Owicki-Gries est alors d'exprimer la pré-condition de chaque fil en fonction de l'état des variables auxiliaires. Par exemple, pour `t1` :

$$\{\texttt{done\_1} = 0 \wedge (\texttt{done\_2} = 0 \Rightarrow \texttt{x} = 0) \wedge (\texttt{done\_2} = 1 \Rightarrow \texttt{x} = 1)\}$$

Nous pouvons alors calculer qu'après l'exécution de son incrément, la propriété est :

$$\{\texttt{done\_1} = 1 \wedge (\texttt{done\_2} = 0 \Rightarrow \texttt{x} = 1) \wedge (\texttt{done\_2} = 1 \Rightarrow \texttt{x} = 2)\}$$

Nous pouvons appliquer un raisonnement similaire pour le fil d'exécution `t2` et par la connaissance générée qu'à la fois `done_1` et `done_2` valent 1, déduire que `x` vaut 2.

La méthode d'Owicki-Gries fournit aussi des règles supplémentaires pour assurer que les variables auxiliaires ne modifient pas le comportement du programme. Nous n'expliquons pas ce point ici.

Il existe diverses méthodes pour raisonner à propos des programmes concurrents. Celles-ci ont généralement été conçues avec cette optique particulière et y sont donc dédiées. Il est par conséquent difficile d'adapter des outils développés pour les programmes séquentiels en se basant sur ces méthodes. Nous pouvons donc nous demander comment tirer parti des outils conçus pour l'analyse de programmes séquentiels afin d'analyser des programmes concurrents.

Dans cette thèse, nous nous focalisons sur les programmes concurrents exécutés selon un modèle particulier d'accès à la mémoire : le modèle *séquentiellement consistant*. Dans ce modèle, nous considérons que les exécutions d'un programme peuvent être vues comme les entrelacements des instructions de ses différents fils d'exécution.

Pour notre programme d'exemple, nous aurions :

— t1:x++ puis t2:x++ ;

— t2:x++ puis t1:x++.

Dans un cadre plus général, celui des modèles de nos architectures matérielles par exemple, les entrelacements ne sont pas les seules exécutions autorisées. Leurs modèles mémoire, dits « faibles », permettent d'autres exécutions qui ne correspondent pas à des entrelacements. Cependant, une propriété importante que la majorité d'entre eux respecte, est que si les exécutions d'un programme selon le modèle mémoire séquentiellement consistant ne contiennent pas de *data-race*, alors l'ensemble de ses exécutions a un comportement séquentiellement consistant. Une manière possible de s'assurer que l'on peut traiter un programme selon cette hypothèse est donc de vérifier l'absence de *data-race*.

Dans le cadre du modèle mémoire séquentiellement consistant, nous pouvons voir les exécutions d'un programme concurrent comme un ensemble d'exécutions séquentielles où les instructions de chaque fil d'exécution sont entrelacées. Une approche est donc de construire ces exécutions afin de simuler le fonctionnement du programme concurrent. Il convient en revanche de ne pas construire explicitement chacune de ces exécutions (une nouvelle fois, nous ne pouvons nous permettre d'être exhaustif), mais de produire un programme dont on a l'assurance que toutes ses exécutions sont équivalentes à celles du programme concurrent simulé.

1.4 Proposition

Dans cette thèse, nous proposons de traiter les programmes concurrents en produisant une simulation de leurs exécutions concurrentes. Le code de cette simulation pourra être analysé statiquement ou dynamiquement.

Le principe de cette simulation consiste à faire l'hypothèse d'un nombre de fils d'exécution constant mais arbitrairement grand possédant chacun un identifiant. Le contexte d'exécution de chaque fil est modélisé de telle manière que chaque variable locale est simulée par l'usage d'un tableau associant à chaque identifiant de fil la valeur de la variable. Une autre zone mémoire permet de sauver pour chaque fil d'exécution sa position actuelle dans l'exécution du programme, c'est-à-dire son compteur de point de programme. Chaque action atomique réalisée par le programme est ensuite simulée par une fonction qui effectue la même opération mais où chaque accès est effectué dans les tableaux simulants et à l'indice correspondant au fil d'exécution actuellement exécuté. L'ensemble des fonctions simulantes est ensuite exécuté par une boucle les entrelaçant de manière non déterministe. Nous supposons donc que les programmes

ont une sémantique d'entrelacements et qu'ils sont exécutés selon le modèle mémoire séquentiellement consistant.

Cette méthode est implémentée par un greffon de la plateforme Frama-C appelé Conc2Seq. Cela a deux implications pour la méthode. D'une part, le programme d'origine est spécifié par l'utilisateur. Comme nous le modifions, les spécifications en question sont adaptées par le greffon, conformément à la transformation effectuée sur le code, afin de pouvoir les prouver sur le programme simulant. D'un point de vue technique, nous fournissons également de nouvelles primitives dans le langage acsl et certaines facilités pour pouvoir parler plus aisément des propriétés de programmes concurrents.

L'autre implication de l'application de la méthode au langage C est due à la norme du langage. En effet, celle-ci impose, pour qu'un programme soit correct, qu'il soit dépourvu de *data-race* [53, Sec. 5.1.2.4]. Comme expliqué dans la section 2.4, une telle propriété nous garantit pour les modèles mémoires courants que le programme n'exhibe que des comportements séquentiellement consistants, satisfaisant notre hypothèse. L'absence de *data-race* reste encore à prouver mais nous pouvons déléguer cette partie du travail au greffon MThread de Frama-C (que nous décrivons succinctement dans le chapitre 2), et voulons fournir un moyen de prouver des propriétés fonctionnelles à propos des programmes concurrents.

Nous nous intéressons néanmoins au fonctionnement des programmes selon les modèles mémoire faibles à travers le prototypage d'un outil d'analyse des programmes sous un modèle mémoire faible. Partant du constat que les modèles mémoires sont généralement exprimés axiomatiquement, et contraignent les comportements autorisés par le modèle, nous utilisons les langages Prolog et CHR pour modéliser et résoudre ces contraintes sur un langage basique et déterminer les exécutions autorisées par un modèle mémoire donné. Cela nous permet entre autres de bénéficier de l'usage d'un moteur de résolution tel que ceux utilisés pour Prolog par rapport à une implémentation directe d'un moteur de résolution dédié aux modèles mémoire faibles.

La suite de ce manuscrit s'articule comme suit. Nous présentons dans le chapitre 2, une vue d'ensemble des méthodes et outils existants pour l'analyse de programmes. Dans le chapitre 3 nous présentons l'étude de cas pour laquelle nous avons appliqué notre méthode d'analyse par simulation et les résultats de l'étude effectuée, notamment les besoins identifiés pour son automatisation. La méthode automatisée est présentée dans le chapitre 4, où nous énonçons les fonctionnalités du greffon écrit pour Frama-C et la manière dont elles sont

implémentées. Ensuite, dans le chapitre 5 nous montrons la correction de notre méthode, en particulier l'équivalence sémantique entre le programme concurrent en entrée et le programme séquentiel simulant en sortie. Le chapitre 6 présente notre prototype de solveur pour les modèles mémoire faibles, ses fonctionnalités, sa correction et son évaluation. Finalement, le chapitre 7 conclut sur notre travail et présente les perspectives envisagées.

Des versions préliminaires des résultats présentés dans ce mémoire ont été présentés et publiés dans les colloques suivants :

— A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. *A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C*. FMICS, Formal Methods for Industrial Critical Software, Springer 2015 [18] ;

— A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. Conc2Seq*: A* Frama-C *Plugin for Verification of Parallel Compositions of C Programs*. SCAM, Source Code Analysis and Manipulation, 2016 [19] ;

— A. Blanchard, N. Kosmatov, and F. Loulergue. *A CHR-Based Solver for Weak Memory Behaviors*. CSTVA, Constraint Solvers in Testing, Verification, and Analysis, co-located with ISSTA, 2016 [20].

État de l'art

2

Sommaire

Dans ce chapitre, nous présentons un ensemble de travaux de la littérature à propos de l'analyse de programmes. Certains points déjà évoqués dans le chapitre 1 sont de nouveaux abordés pour les replacer par rapport aux autres méthodes et logiciels mentionnés ici. Nous donnons également un peu plus de détails à leur sujet.

Dans un premier temps, nous nous intéressons aux programmes séquentiels (dans la section 2.1), pour lesquels nous présentons diverses méthodes d'analyse et logiques de programmes. Pour ces dernières, nous présentons, dans la section 2.2, les logiques qui en ont été dérivées pour le traitement de programmes concurrents. Dans la section 2.3, nous présentons plusieurs outils d'analyse de programmes écrits en langage C, dont Frama-C, la plateforme pour laquelle nous avons implémenté notre méthode. Finalement, la section 2.4 répertorie divers travaux à propos de la prise en compte des modèles mémoire des processeurs lors de l'analyse de programmes concurrents.

2.1 Méthodes de vérification

2.1.1 Test

Pour vérifier le bon fonctionnement des programmes, la méthode la plus employée en industrie est le test. C'est une analyse *dynamique*, elle nécessite d'exécuter le programme sur un ensemble d'entrées afin d'assurer que les résultats produits sont conformes. La spécification du programme en vue de sa validation prend alors la forme d'une association des entrées testées vers les sorties attendues du programme. Ces tests peuvent être réalisés sur l'ensemble du programme (souvent dénommé « test d'intégration et de validation ») ou sur des fonctionnalités isolées (test unitaire).

L'avantage du test est principalement sa précision. Le résultat que nous obtenons est très précisément celui produit par le programme puisque nous l'avons obtenu en l'exécutant. Donc si notre analyse nous indique qu'un résultat n'est pas conforme à l'attente, le verdict est sûr. En revanche, le test n'est pas complet, nous ne pouvons essayer qu'un nombre limité d'entrées pour un programme donné, et les entrées réelles peuvent être en nombre très grand, ou infini. Il en résulte que si nos tests ne trouvent pas de bugs, à moins que nous ayons testé exhaustivement toutes les entrées possibles du programme, nous n'avons pas de certitude qu'il ne contient aucun bug.

Il existe diverses métriques pour déterminer la qualité de la couverture d'un programme par les tests [9]. Celles-ci peuvent être exploitées, en combinaison avec des analyses statiques, pour générer automatiquement un grand nombre de tests susceptibles d'améliorer la qualité de la couverture et donc la confiance que l'on peut avoir dans le programme. Par opposition aux analyses dynamiques,

les analyses statiques sont effectuées sans exécuter le code, nous en présentons plusieurs par la suite.

Tester permet d'éliminer, avec des coûts raisonnables, une bonne quantité de bugs. En l'occurrence, cela permet de montrer, sur l'ensemble des tests exécutés, l'absence d'un certain nombre de comportements identifiés comme problématiques, ou la présence de comportements identifiés comme voulus, mais pas pour l'ensemble des entrées et des comportements définis par la spécification générale. Pour cela, il est nécessaire de se tourner vers les méthodes formelles.

2.1.2 Interprétation abstraite

L'interprétation abstraite [36] est une technique d'analyse *statique* [72]. Contrairement au test, le programme n'est pas exécuté pour déterminer sa correction. On raisonne sur l'ensemble des comportements qui pourraient apparaître pendant l'exécution d'un programme afin de déterminer si ceux-ci sont effectivement contenus dans les comportements acceptables. Ces analyses sont en général indécidables [62] et reposent sur une approximation des états possibles d'un programme. Ces techniques peuvent donc être complètes (à condition que les approximations le permettent) mais sont imprécises, du fait des approximations.

Dans le cas de l'interprétation abstraite, l'approximation repose sur la théorie des treillis et le calcul de point fixe. D'une certaine manière, on analyse le comportement sur toutes les entrées possibles sans exécuter les instructions pour chacune d'entre elles. Le principe est de résoudre un ensemble d'équations déterminées à partir du programme et représentant sa sémantique. Les limites de cette méthode vont donc se trouver dans la capacité de l'outil à résoudre ces équations. S'il n'existe pas de procédure de décision automatique pour la résolution, les résultats restent très imprécis.

2.1.3 Model-Checking

La vérification de modèle (*model checking*) [28, 87] n'est pas à proprement parler une technique de vérification de code mais plutôt de systèmes (informatiques ou électroniques), représentés par un modèle. Cela peut par exemple concerner des protocoles ou des circuits de transistors.

Les propriétés du système sont généralement exprimées en utilisant des logiques temporelles. Le système est modélisé sous la forme d'un graphe états/-transitions à partir duquel on vérifie les propriétés voulues de manière automa-

tique. Concrètement, s'il existe un chemin dans le graphe qui ne respecte pas la propriété énoncée, alors le système contient une erreur. Il faut ensuite analyser cette erreur afin de déterminer si elle est due à la représentation du système par le modèle, ou si le système est effectivement fautif.

La méthode étant automatique, elle est facile à mettre en place pour un utilisateur (modulo l'expertise nécessaire pour la modélisation et l'analyse des erreurs). En revanche, les états atteignables d'un programme peuvent être très nombreux suivant le système vérifié. Il est possible de limiter l'explosion combinatoire en représentant les états de manière symbolique et non concrète. Dans le cas des programmes, le nombre d'états pouvant être vraiment colossal, ces représentations peuvent ne pas être suffisantes pour permettre la vérification. C'est notamment le cas quand le nombre d'états est infini, dans cette situation, des méthodes d'abstractions sont utilisées pour construire un modèle approximatif du système vérifié n'utilisant qu'un nombre fini d'états.

Ce type d'analyse est par exemple implémenté par les *model-checkers* Spin [51] ou encore Cubicle [34].

2.1.4 Logiques de Hoare

Les logiques de Hoare [49] sont des logiques qui permettent de raisonner sur les programmes et leurs propriétés. Les triplets de Hoare, de la forme $\{P\}\ c\ \{Q\}$ où P et Q sont des assertions logiques et c une séquence d'instructions, nous indiquent que depuis un état où le programme respecte la propriété P, la séquence d'instructions c nous amène dans un nouvel état où Q est respectée.

En logique de Hoare, il est possible de raisonner en terme de correction totale ou partielle. En correction totale, le triplet signifie que depuis un état qui respecte la pré-condition P, la séquence d'instructions c termine et atteint un nouvel état où Q, la post-condition, est respectée. En correction partielle nous ne considérons pas la preuve de terminaison, la signification est alors que depuis un état respectant P, si la séquence d'instruction c termine, alors l'état du système après exécution respecte Q.

Les logiques de Hoare reposent sur des systèmes d'inférence sur les triplets précédemment cités. Chaque instruction du programme, et les compositions successives de ces instructions, donnent lieu à un triplet dans l'arbre de dérivation. Il est donc souhaitable que cette inférence soit automatisée au maximum. C'est notamment l'objectif du calcul de plus faible pré-conditions [41], qui définit comment, à partir d'une post-condition donnée et d'un programme, on peut calculer

la pré-condition la plus faible nécessaire pour avoir un triplet valide. Ce calcul se fait par transformations successives du prédicat d'origine. Pour certaines instructions, ce calcul n'est cependant pas automatisable. C'est notamment le cas des boucles pour lesquelles il faudra renseigner les invariants que celles-ci doivent respecter.

Les outils automatiques utilisant cette méthode combinent généralement deux phases : un premier outil, le générateur d'obligations de preuve (*verification condition generator*, VCGen) reçoit en entrée le programme et sa spécification, à partir de ces éléments, il produit l'arbre de dérivation correspondant au plus faible pré-condition, avec aux feuilles les formules devant être vérifiées pour garantir la validité du programme : les obligations de preuves. Ces obligations de preuves sont ensuite déchargées par l'usage de prouveurs automatiques (solveurs SMT par exemple) ou interactifs (comme les assistants de preuve Coq [91] ou Isabelle/HOL [74]) indépendants du langage d'entrée.

Les outils basés sur ce type de fonctionnement se montrent très efficaces lorsque les obligations de preuve peuvent être déchargés par les solveurs automatiques. Lorsque ce n'est pas le cas, il est nécessaire d'intervenir manuellement. Dans le cas des solveurs automatiques, il est possible de les guider par l'ajout d'assertions intermédiaires dans le code source ou des lemmes supplémentaires dans la base de connaissances, mais cela nécessite une certaine expertise des outils pour déterminer les raisons qui font que la preuve ne passe pas. Il est également possible de réaliser la preuve par l'usage de prouveurs interactifs. Le problème est alors que les obligations de preuve que produisent les générateurs sont souvent complexes à lire, en raison de trop fortes ou trop faibles simplifications effectuées par les heuristiques du générateur d'obligations, ainsi qu'à l'encodage des sémantiques pour les prouveurs cibles.

L'outil et le langage Why3 [43] permettent par exemple de produire ce calcul de plus faible pré-conditions à partir d'un programme. Les obligations générées peuvent ensuite être déchargées par exemple par Alt-Ergo [33], CVC3 [13] ou encore CVC4 [14], que nous utilisons, dans le présent travail, pour décharger les obligations générées pour les programmes que nous analysons.

2.1.5 Logique de séparation

Les logiques de Hoare originelles permettent de raisonner sur les programmes mais sont rapidement limitées lorsque l'on souhaite raisonner sur des programmes qui manipulent des pointeurs. Dans ce type de cas, il est nécessaire

d'ajouter une couche supplémentaire de raisonnement pour expliciter le comportement de la mémoire, notamment en ce qui concerne les notions de partage de données et leur séparation spatiale. Or, une grande quantité de langages de programmation réels ont besoin des ressources dans le tas.

La logique de séparation [83] définit un moyen de raisonner à propos des programmes agissant sur le tas (elle permet notamment l'arithmétique de pointeurs). Le langage de base considéré par Hoare est enrichi avec des primitives concernant l'allocation, la désallocation et l'accès à des ressources du tas. Les triplets correspondant à cette logique sont définis pour chaque commande pour le raisonnement avant et arrière.

Le principal intérêt de la logique de séparation repose dans l'ajout de deux opérateurs à la logique du premier ordre usuel, à savoir la « *separating conjunction* » ($P * Q$) et la « *separating implication* » ($P -\!\!* Q$). La première signifie que P et Q sont des propriétés qui concernent des parties disjointes du tas ; la seconde indique que si l'on étend le tas avec une partie disjointe satisfaisant P, le tas résultant satisfait Q.

Selon Reynolds, l'ensemble des règles en raisonnement arrière fournit un calcul de plus faible pré-conditions complet [54]. Du fait de la présence des assertions à propos du tas, certaines règles d'inférence de la logique de Hoare sont invalides par défaut et ne peuvent donc être utilisées qu'en prouvant la pureté des assertions, c'est-à-dire en prouvant l'absence de propriétés à propos du tas.

Si les règles d'inférence de la logique de séparation peuvent effectivement être appliquées de manière automatique, la satisfiabilité de ces formules est un sujet encore complexe. Les techniques existantes traitent des sous-ensembles de la logique, soit en les transformant pour les rendre compréhensibles par des solveurs SMT standards [79, 82], soit par des algorithmes dédiés [24]. La logique de séparation est donc encore peu utilisée dans les outils d'analyse de programmes réalistes ou alors nécessite une instrumentation assez forte du code.

2.2 Programmes concurrents

Les programmes concurrents sont des programmes qui sont composés de plusieurs programmes s'exécutant simultanément (fils d'exécution ou *threads*) et pouvant accéder à des ressources partagées entre eux.

De tels programmes sont plus difficiles à analyser que les programmes séquentiels. Dans un programme séquentiel, une des difficultés les plus importantes apparaît lorsque nous sommes en présence d'*aliasing*, c'est-à-dire dans une situation où plusieurs variables d'un programme nous permettent un accès potentiel aux mêmes ressources. Il est alors plus difficile de déterminer comment évolue la ressource en question et donc ce que l'on peut lire depuis les différents points d'accès. Traiter de tels programme est l'une des motivations de la logique de séparation.

Dans un programme concurrent, nous faisons face au même problème en pire : non seulement nous sommes en présence d'*aliasing*, mais en plus les accès aux ressources partagées peuvent être effectués depuis plusieurs fils simultanément. En réalité, un ensemble d'opérations en mémoire (par exemple une lecture et une écriture) ne peuvent pas être effectuées simultanément pour une position en mémoire donnée au niveau matériel. Les requêtes de lecture ou d'écriture seront donc effectuées dans un certain ordre que nous sommes généralement incapables de déterminer de manière sure. Cette situation est appelée *data-race*. Si deux actions dont au moins une écriture sont effectuées « en même temps » à une position mémoire donnée, nous ne pouvons pas prévoir le résultat des lectures pour une écriture et des lectures (on ne sait pas si l'écriture a effectivement été faite avant une lecture), ou la valeur désormais présente en mémoire si plusieurs écritures sont mises en jeu car on ne sait pas dans quel ordre elles seront exécutées.

Même en l'absence de *data-race*, assurer et vérifier la correction de propriétés fonctionnelles est une tâche complexe, l'ensemble des comportements possibles du programme explosant rapidement du fait des exécutions simultanées des différents fils d'exécution. Dans cette section, nous nous intéressons aux méthodes existantes pour prouver la correction de tels programmes.

Les techniques que nous présentons dans cette section supposent un modèle mémoire séquentiellement consistant tel que présenté dans [61], c'est-à-dire un modèle où les exécutions d'un programme concurrent peuvent se résumer aux entrelacements des instructions de chacun des fils d'exécution. Dans la section 2.4, nous nous pencherons sur les limites de cette hypothèse.

2.2.1 Méthode de Hoare

Un des travaux précurseurs en preuve de programmes parallèles est décrit par Hoare dans [48, 50]. Il s'agit une nouvelle fois d'une définition axiomatique

des comportements des programmes, mais concurrents cette fois. Les idées principales mises en avant sont : le besoin d'un langage de haut niveau pour la description abstraite de programmes parallèles et celui de s'abstraire des questions de temporalité pour assurer le bon fonctionnement de tels programmes. Si la première idée n'élimine pas le besoin d'un moyen de raisonner à plus bas niveau, ne serait-ce que pour implémenter les mécanismes proposés par un langage de plus haut niveau, la seconde est fondamentale car la diversité du matériel et des situations qui peuvent survenir pendant l'exécution du programme parallèle rendent la prédiction de l'ordre exact des événements impossible dans le cas général. Ce problème est donc abstrait dans toutes les logiques que nous présentons dans cette partie.

La logique définie par Hoare pour les programmes concurrents suit l'idée de [40], à savoir que les différents fils d'exécution d'un programme concurrent devraient s'exécuter au maximum indépendamment excepté en de rares points de synchronisation. Elle définit par conséquent des règles permettant de séparer les raisonnements sur les fils en leur attribuant chacun les ressources dont ils ont l'accès exclusif (à noter que l'on hérite quand même des problèmes déjà présents en logique de Hoare standard lorsque nous sommes en présence d'*aliasing*, voir section 2.2.4), et des règles permettant d'exprimer les moments de synchronisation, desquelles on évacue les problèmes de temporalité par la notion de ressources à accès exclusif. La syntaxe fournie est de la forme : `with resource r do c`. `c` pouvant être une suite d'instructions et `r` ne pouvant être acquise que par un seul fil d'exécution à la fois, nous sommes sûrs que deux fils ne peuvent pas exécuter la séquence d'instruction `c` en même temps. Une telle séquence est appelée *section critique*.

2.2.2 Méthode d'Owicki-Gries

La méthode d'Owicki-Gries [77] complète la logique de Hoare concurrente. Elle ajoute la notion d'invariant de ressource qui va permettre de vérifier des informations globales à propos du système lors de l'exécution de sections critiques. Ces invariants ne peuvent être (temporairement) violés que pendant l'exécution d'une section critique, il faut donc vérifier qu'aucune autre opération n'en a le pouvoir et que les sections restituent bien ces invariants (à noter qu'en dehors des sections, on ne peut pas supposer ces invariants puisqu'un autre fil peut être en train de le manipuler). Elle définit également la notion de variable auxiliaire qui pourrait aujourd'hui être vue comme la notion de variable « fantôme »,

c'est-à-dire des variables qui peuvent être assignées mais jamais lues par le programme, seulement du côté de la logique.

Par la combinaison de ces deux notions, il est possible de prouver des propriétés plus complexes à propos des programmes qu'en logique de Hoare concurrente, notamment lorsque celles-ci mettent en jeu la combinaison des opérations réalisées par chacun des fils d'exécution. En revanche, vérifier que les interférences entre ceux-ci n'entrent pas en conflit est coûteux et peu modulaire car cela nécessite de comparer les arbres de preuve des différents fils après la preuve de chacun d'eux, ce qui peut invalider la totalité du travail de vérification en fin de processus de preuve. Un exemple d'utilisation de cette méthode peut être trouvé dans le chapitre 1.

Cette méthode est par exemple implémentée en Isabelle/HOL [73]. Et elle a été utilisée avec succès sur des problèmes de la vie réelle comme par exemple de la vérification de système d'exploitation [10].

2.2.3 Rely-Guarantee

Comme mentionné précédemment, la méthode d'Owicki-Gries possède le défaut de ne pas être suffisamment modulaire, notamment en ce qui concerne les propriétés de non-interférence. Pour pallier à ces difficultés, dans [56], Jones présente la *Rely-Guarantee*, une méthode permettant de définir précisément les interférences autorisées pour les fils d'exécutions. Elle est par exemple implémentée en Isabelle/HOL [81].

Les « Rely »-conditions nous définissent comment l'environnement est susceptible de changer entre deux états du programme. Inversement, les « Guarantee »-conditions énoncent les interférences qu'un fil d'exécution est autorisé à générer entre deux états successifs du programme. Ces conditions sont donc des ensembles d'actions $P \rightsquigarrow Q$ où P est une assertion à propos de l'état de départ et Q à propos de l'état d'arrivée. Les « Rely/Guarantee »-conditions sont définies au niveau global, elles doivent être réflexives (pour permettre à l'environnement de ne pas évoluer) et transitives (pour autoriser une série de changements) [56, Sec 3.1].

Nous pouvons par exemple exprimer qu'une variable croît de manière monotone d'un état à l'autre par la « Guarantee » condition : $\forall x.t = x \rightsquigarrow x \leq t'$. Où t' est la valeur de la variable t dans l'état qui suit. Cette relation entre les états est bien réflexive et transitive.

Une faiblesse de cette approche est que ces interférences sont définies globa-

lement. Elles doivent donc être vérifiées pour chaque modification de l'état du programme même si l'on sait trivialement que l'opération ne peut pas générer d'interférences dans un autre fil d'exécution.

2.2.4 Logique de séparation concurrente

Nous avons mentionné dans la section 2.2.1 que la logique décrite hérite des mêmes problèmes que la logique de Hoare en ce qui concerne la présence d'alias, ou encore d'exclusion non-structurée (dans le cas général) comme les sémaphores, la méthode Owicki-Gries héritant des mêmes limitations. Le but de la logique de séparation concurrente [75] est d'adapter la méthode d'Owicki-Gries en utilisant la logique de séparation.

L'isolation des ressources et l'absence de partage (quand c'est nécessaire) sont assurées par l'usage de la conjonction de séparation. Par exemple, nous pouvons imaginer un programme qui commence par allouer un tas tel que sur deux parties disjointes p et q, les propriétés P et Q sont respectées. Nous avons alors $P * Q$. Ensuite, ce programme lance un fil d'exécution en lui transmettant un accès à p et va, de son côté, manipuler le tas q tout en assurant qu'il ne modifiera pas p. Après le lancement du fil, le programme principal n'est plus associé qu'à la propriété Q et le fil d'exécution à la propriété P.

De cette manière, en tout point de programme, on peut connaître quelle partie du tas appartient à quel fil d'exécution (et par conséquent ce qu'il a le droit de modifier). Les invariants de ressources doivent néanmoins être des prédicats précis, à savoir qu'il ne peut exister qu'au plus un sous-tas du tas global tel que l'invariant formulé est vrai. Sinon, le transfert de ressource en entrée de bloc n'est plus déterministe et peut mener à des incohérences.

Contrairement à la *rely-guarantee*, cette méthode ne définit pas les interférences sur des états successifs. Elle exploite le même concept de section critique avec ressource que dans la logique de Hoare et son extension avec l'invariant de ressource apporté par la méthode d'Owicki-Gries. Comme dans cette dernière, exprimer les propriétés, souvent relationnelles, qu'impliquent les interférences nécessite de nombreuses variables auxiliaires qui peuvent complexifier la preuve.

2.2.5 Combinaison de Rely-Guarantee et logique de séparation

Le travail présenté par [94] vise à concilier les forces de la *rely-guarantee* et de la logique de séparation concurrente, à savoir conserver la modularité de la

preuve et la séparation des ressources tout en autorisant une expression précise des interférences dans les programmes concurrents.

Cette technique repose sur la différenciation au niveau du tas des ressources partagées et locales. Les propriétés concernant les ressources partagées sont traitées par l'usage de la *rely-guarantee*. Les propriétés locales sont gérées par l'usage de la logique de séparation, le transfert de ressources étant défini par la définition des interférences.

2.3 Outils d'analyse de code C

2.3.1 Frama-C

Frama-C [57] est une plateforme d'analyse de programme écrits en langage C développée au CEA LIST. Elle est conçue de manière modulaire, au sens où elle propose différentes analyses sous la forme de greffons exploitant un noyau qui fournit les fonctionnalités nécessaires pour extraire les informations de base des programmes, en permettant par exemple d'accéder à son arbre de syntaxe abstraite ou aux annotations fournies par l'utilisateur.

Ces fonctionnalités de base reposent sur l'usage de la bibliothèque CIL [71] qui normalise les programmes C (par exemple : transformation de toutes les boucles en `while`, instruction `return` unique pour chaque fonction). Frama-C l'étend pour supporter les annotations en ACSL (présenté plus tard dans cette section).

Les analyses fournies peuvent être statiques, dynamiques ou la combinaison des deux. Nous pouvons par exemple citer :

— l'interprétation abstraite (section 2.1.2) avec Value [25] ;
— le calcul de plus faible pré-conditions (section 2.1.4) avec Wp ;
— la génération automatique de tests (section 2.1.1) avec PathCrawler [96] ;
— la combinaison de tests et de preuve avec StaDy [78].

Actuellement, le seul greffon de Frama-C en mesure de traiter du code concurrent est MThread, qui ne permet pas de prouver des propriétés fonctionnelles et qui fonctionne nécessairement en *whole-program*, c'est-à-dire en analysant la totalité du programme qui contient du code concurrent. Notre but est d'ouvrir la plateforme à l'analyse de programmes concurrents avec différents greffons en se focalisant dans un premier temps sur Wp pour vérifier des propriétés fonctionnelles (voir section 1.4).

```
1 /*@
2   requires \valid(a) && \valid(b);
3   requires \separated(a,b);
4   assigns *a, *b;
5   ensures *a == \old(*b) && *b == \old(*a);
6 */
7 void swap(int* a, int* b);
```

FIGURE 2.1 – *Fonction* `swap` *spécifiée en* ACSL

Notions d'ACSL

Tout au long de ce manuscrit, nous utiliserons ACSL [17] (*ANSI C Specification Language*). C'est un langage formel de spécification comportementale [46] inspiré du langage JML [63], qui permet d'exprimer des spécifications fonctionnelles sur les programmes C. Il permet par exemple d'écrire des pré-conditions, des post-conditions, des invariants ou encore des assertions. Les annotations sont écrites en logique du premier ordre. Il est également possible de définir ses propres fonctions, prédicats et types logiques.

Les contrats de fonctions sont composés d'une pré-condition (introduite par des clauses `requires`), la propriété attendue du système avant l'appel de la fonction, et d'une post-condition (clauses `ensures`), la propriété que la fonction doit assurer pour le système après son exécution. Une classe particulière de post-condition est définie par la clause `assigns` qui définit les éléments de l'état global du programme qui peuvent être modifiés par la fonction pendant son exécution (et par conséquent, ceux qui ne le seront pas).

La figure 2.1 illustre l'usage de ces fonctionnalités sur une simple fonction d'échange des valeurs de deux variables entières. La première clause `requires` (ligne 2) demande que `a` et `b` soient des pointeurs vers des zones de mémoire « valides », c'est-à-dire qui peuvent être accédées indifféremment en lecture et écriture sans erreur. La seconde clause `requires` (ligne 3) exprime la séparation des adresses pointées par `a` et `b`, donc d'une part que ces adresses sont différentes mais également que les zones mémoires pointées ne se "chevauchent" pas. La clause `assigns` (ligne 4) indique que seules les locations pointées par `a` et `b` peuvent être modifiées (et pas celles qui pourraient faire partie de la même plage d'adresse par exemple). Finalement la clause `ensures` (ligne 5) indique qu'après l'exécution de la fonction la valeur pointée par `a` sera celle anciennement pointée par `b` et inversement.

Le langage offre aussi la possibilité d'ajouter des annotations dans le code,

par exemple des assertions ponctuelles introduites par la clause `assert`, ou encore des invariants de boucle, introduits par la clause `loop invariant`.

Wp

Wp repose sur le calcul de plus faibles pré-conditions. À partir d'un programme C annoté en ACSL, Wp effectue la génération des obligations de preuves nécessaires à vérifier que le programme correspond à la spécification fournie par l'utilisateur. Une phase de simplification des obligations de preuve est effectuée par un module appelé Qed [35], ces simplifications peuvent déjà décharger les obligations les plus simples, évitant l'appel aux prouveurs externes. Il dispose d'une sortie vers divers prouveurs automatiques et interactifs. La traduction est effectuée nativement vers Alt-Ergo et Coq. Pour d'autres cibles, une traduction est d'abord effectuée vers Why3 qui se charge ensuite de transmettre les obligations de preuve vers les autres prouveurs.

2.3.2 VCC

VCC [89, 30] est un analyseur pour les programmes concurrents (ou non) écrits en langage C. Les spécifications sont intégrées dans le code source à l'aide de macros, avec un vocabulaire semblable à celui que l'on peut trouver en ACSL. Les preuves sont effectuées par calcul de plus faible pré-condition à l'aide de l'outil Boogie [12] qui transmet ensuite les obligations de preuve générées au solveur SMT Z3 [38].

Contrairement à Frama-C, VCC est intrinsèquement conçu pour vérifier des programmes C concurrents. Il possède donc des notions supplémentaires pour la spécification par rapport à ACSL. Il repose notamment sur la notion de possession de ressources [39] (*Object Ownership*) initialement conçue pour raisonner à propos des programmes écrits dans un style orienté objet, qu'il hérite de Spec# [11]. Celle-ci énonce notamment que l'on ne peut modifier un objet que si, au point de programme considéré, le point d'accès à l'objet en a effectivement la possession. Pour permettre de modifier cet état de possession, le langage de spécification fournit des notations qui s'accompagnent de règles définissant sous quelles conditions la prise de possession est autorisée.

L'autre différence majeure de VCC et ACSL est la notion d'invariant sur deux états (semblable à ce que l'on peut trouver chez Owicki-Gries) et d'invariant vérifié localement [31] (*LCI : Locally Checked Invariant*). Chaque objet du programme, incluant les fils d'exécution eux-mêmes, se voit attribuer un invariant qui pour

toute paire d'états consécutifs dans les exécutions doit être vérifié. Cela inclut la notion d'invariant « stable », maintenu par une action sur l'objet concerné, et « réflexif », maintenu en cas d'inaction. Nous pourrions par exemple reprendre la « Guarantee » condition présentée comme exemple dans la section 2.2.3 mais où la variable t serait une variable membre d'un objet concurrent. Un tel invariant ne doit être vérifié que lors d'un accès à un objet concurrent.

D'autre part un ensemble d'obligations de preuve est généré pour assurer que la preuve d'un invariant au niveau local est suffisante pour prouver que les invariants de plus haut niveau sont eux-mêmes prouvés. Par exemple, avec un ensemble d'objet concurrent répondant à l'invariant local précédent, nous pourrions garantir que la somme des t de chacun de ces objets ne peut elle-même que croître.

Notions de VCC dans Jessie

Jessie [70] est un greffon de Frama-C qui, avant Wp, était le greffon de vérification déductive par calcul de plus faibles pré-conditions. Les principales différences résident dans le fait que Jessie, contrairement à Wp, n'implémente qu'un seul modèle mémoire (basé sur la séparation de régions) et sur le fait que le C spécifié en acsl est d'abord traduit vers un langage orienté objet intermédiaire (aussi appelé Jessie).

Le projet Astraver [68] a donné lieu à des expérimentation avec Jessie où le but était justement de profiter de cette dernière particularité (le langage intermédiaire orienté objet) pour implémenter à travers Frama-C et Jessie des notions que nous avons mentionnées à propos de VCC: la possession et l'invariant à deux états, afin de vérifier des programmes concurrents.

2.3.3 VeriFast

VeriFast [55] est un outil de preuve de programmes écrits en langage C ou Java qui implémente la logique de séparation. Il implémente également des éléments de la méthode d'Owicki-Gries (invariants de ressources) et le comptage de permissions [22]. Cet analyseur est donc également intrinsèquement concurrent.

La vérification est effectuée en partie de manière interactive. L'utilisateur n'écrit pas, au sens propre, la preuve. Mais il doit annoter le code de façon à déplier les formules en logique de séparation qui énoncent des propriétés à propos des structures de données, au travers d'annotations au fil du code. La preuve

de validité de ces formules intermédiaires est déléguée à un solveur SMT embarqué ou encore à Z3.

2.4 Modèles mémoire faibles

Les modèles mémoire définissent comment les actions d'un programme font évoluer l'état de la mémoire, en particulier quand plusieurs fils d'exécution interagissent avec celle-ci.

Par exemple, le modèle mémoire séquentiellement consistant définit que lorsqu'une opération d'écriture en mémoire est effectuée, la valeur écrite est immédiatement prise en compte au niveau de la mémoire, et qu'elle est par conséquent visible par tous les autres fils d'exécution.

Nos processeurs ne peuvent pas respecter une telle propriété car un tel niveau de synchronisation est coûteux en performances. Ils implémentent des modèles mémoire dits « faibles ». Dans de tels modèles, les écritures peuvent par exemple être placées dans un tampon avant d'être finalement envoyées vers la mémoire. Une écriture effectuée par un fil d'exécution, pourrait alors ne pas être directement visible par les autres fils d'exécution. Ces modèles vont également pouvoir autoriser le réordonnancement des instructions, ou encore à spéculer sur les valeurs lues.

Dans un tel contexte, raisonner sur les programmes concurrents devient encore plus complexe car l'exécution d'un programme ne peut plus y être vue comme un entrelacement de ses instructions.

2.4.1 Formalisations

Il existe différentes formalisations des modèles mémoire faibles. Celles-ci tombent généralement dans deux catégories [5] : formalisations opérationnelles et formalisations axiomatiques. Les formalisations opérationnelles modélisent le comportement des programmes selon les modèles en construisant une abstraction des machines réelles comprenant le fonctionnement (idéalisé) des files et des tampons d'écriture qui peuvent composer le matériel. Les formalisations axiomatiques ont la forme d'un ensemble de règles à propos des comportements qui vont contraindre les exécutions jusqu'à éventuelle terminaison des propagations de contraintes sans détecter d'erreur (exécution autorisée) ou détection d'une incohérence (exécution interdite). L'outil que nous proposons, présenté dans le

chapitre 6 sert à appliquer les règles énoncées par des formalisations axiomatiques sur des programmes.

Les formalisations de modèles mémoire faibles peuvent porter sur les langages de programmation comme C++ [21] ou Java [69], mais également sur les architectures processeurs elles-mêmes : Total Store Order (TSO) [90], Power [86], ARM [2].

Ces formalisations sont nécessaires car ce sont elles qui vont nous permettre d'assurer qu'une logique ou une méthode de preuve est correcte par rapport à un ou plusieurs modèles mémoire cibles. De plus, c'est un moyen de vérifier que le matériel ou le compilateur d'un langage ne produit bien que des comportements conformes au modèle qu'il est sensé implémenter. Elles permettent également d'identifier des défauts dans les modèles (comportements mal définis ou non souhaitables) en vue de les corriger. Parfois, le défaut peut être dû à une mauvaise traduction du modèle matériel vers sa représentation. Dans ce cas, une preuve qui reposerait dessus serait fausse, mais l'on peut corriger le problème. Une situation plus problématique apparaît lorsque le matériel implémente un comportement que ses concepteurs voulaient en réalité interdire. Sarkar et al. [86] ont identifié de tels comportements dans le modèle Power par exemple.

2.4.2 Identification d'exécutions

Parmi les outils pour l'analyse de programmes sous des modèles mémoire faibles, certains sont dédiés à l'identification des exécutions réalisables par un programme compte tenu du modèle. Par exemple, les outils PPCMem [86] et CPPMem [16] sont dédiés à l'identification des exécutions de programmes sous les modèles respectifs Power et C++11.

Le solveur Herd [5] est quant à lui conçu comme un analyseur générique sur le modèle mémoire. Il implémente un modèle axiomatique générique extrêmement faible qui autorise un grand nombre d'exécutions. Les modèles mémoire sont ensuite fournis par l'utilisateur à travers un langage de description qui va définir les relations respectées par le modèle comme des combinaisons des relations de base du modèle générique, ainsi que les propriétés que doivent respecter ces relations. Nous reviendrons sur le formalisme associé dans le chapitre 6 car c'est celui que nous utilisons.

Les *constraint-based concurrent memory machines* [84] sont un formalisme pour la définition des modèles mémoire faibles sous la forme de contraintes sur

l'ordre des événements et sur les valeurs mises en jeu par les exécutions. Selon l'auteur, ce formalisme est adaptable à tout modèle. Dans [84], c'est le modèle mémoire de Java qui est formalisé. Une implémentation en Prolog et CHR est proposée par Schrijvers [88]. Comme dans ce travail, nous utilisons également CHR avec un moteur de résolution existant : celui du langage Prolog. Cela nous permet d'éviter d'implémenter le mécanisme de résolution, comme c'est le cas dans [5].

2.4.3 Logiques

Une possibilité attrayante pour la vérification de programmes concurrents en prenant en compte les modèles mémoire faibles est l'intégration du modèle directement au niveau de la logique de programme utilisée. C'est par exemple le cas de la logique de séparation relaxée [93]. Basée sur le modèle mémoire C++11, celle-ci étend la logique de séparation concurrente avec les types d'accès du modèle (*sequentially consistent*, *release*, *acquire*, *relaxed*) et les associe avec des transferts de propriété des ressources manipulées.

La logique GPS [92] (*ghosts, protocols and separation*) généralise la logique de séparation relaxée en y réintégrant les notions de *rely-guarantee* par la définition de protocoles. L'idée est de lier à chaque position mémoire atomique un invariant définissant les transferts de propriétés qui vont avoir lieu lors de la lecture ou l'écriture d'une valeur, d'abord pour la sémantique *release-acquire*, puis (dans l'article [47]) avec les opérations relaxées. Ces dernières permettant d'acquérir une connaissance sur laquelle on ne peut se reposer pour la preuve tant qu'aucune barrière mémoire n'a validé cette connaissance.

2.4.4 Retrouver le modèle séquentiellement consistant

Une autre manière commune d'analyser les programmes concurrents tout en prenant en compte les modèles mémoire faibles est d'assurer dans un premier temps que les programmes ont un comportement séquentiellement consistant, puis de les analyser en utilisant le modèle séquentiellement consistant. En effet, une propriété fondamentale attendue d'un modèle mémoire, même faible, est que si ses exécutions séquentiellement consistantes ne contiennent pas de *data-race* alors toutes ses exécutions peuvent être considérées séquentiellement consistantes. Cette propriété est énoncée par [85], et est vérifiée sur les modèles mémoire faibles courants [23].

Assurer qu'un programme est exempt de *data-race* peut être effectué par une

analyse dédiée, par analyse statique [37, 80], par exemple Frama-C propose le greffon MThread [57, Section 9], qui peut effectuer ce type de détection. Néanmoins la détection de *data-race* est un problème difficile qui nécessite dans le cas général de faire des analyses sur le programme entier.

Connaissant un modèle mémoire particulier, il est possible de réduire la difficulté de cette détection. Par exemple, pour le modèle TSO, Owens [76] propose la notion de programme *triangular race-free*. Un *triangular race* apparaît entre une lecture et une écriture de deux fils d'exécution différents lorsque le fil d'exécution lecteur a effectué une écriture à la même adresse précédemment et que rien ne garantit que son tampon d'écriture a été vidé (entraînant une incertitude sur la valeur lue). Nous pouvons obtenir une telle garantie par l'ajout d'une barrière mémoire. Les barrières mémoire sont des opérations spécifiques qui vont demander au processeur d'assurer :

— que toutes les opérations mémoire qui la précèdent sont effectuées ;
— que toutes les opérations qui suivent ne sont pas encore exécutées ;
— que son tampon d'écriture est vide.

Par l'ajout d'une barrière mémoire avant lecture, on garantit l'absence de *triangular-race* et l'absence de *data-race* dans le programme. En effet, une barrière mémoire est une instruction spécifique qui va permettre de forcer le processeur à assurer que les opérations qui la précède sont bien réalisées avant de commencer à exécuter les instructions qui la suivent. Cohen [29] pointe qu'en pratique, cette méthode peut se révéler coûteuse et propose de combiner une analyse équivalente avec la notion d'*ownership* de VCC pour limiter le nombre de barrières mémoire (et donc le coût à l'exécution).

Il est possible d'aller plus loin dans l'automatisation avec par exemple des outils d'insertion automatique de barrières qui permettent, étant donné un programme et un modèle mémoire, d'ajouter les barrières qui vont ramener les exécutions à des exécutions séquentiellement consistantes en déterminant les synchronisations les moins coûteuses. C'est par exemple le cas de [4].

Finalement, une manière de raisonner à propos des comportements implicites des programmes sous les modèles mémoire faibles est de rendre ces comportements explicites en incorporant la sémantique opérationnelle des modèles directement au niveau du code source, simulant donc l'effet des composants physiques au niveau du code [3].

Étude de cas : module d'adressage 3

Sommaire

3.1 Introduction

La méthode de vérification que nous proposons a d'abord été expérimentée à travers un cas d'étude [18]. Celui-ci concerne la vérification d'une partie d'un module de micro-noyau de système d'exploitation sécurisé. Le but de ce chapitre est de présenter ce cas d'étude reposant sur notre méthode de vérification pour donner une estimation de son applicabilité sur un problème réel.

Ce micro-noyau, Anaxagoros [64], est capable de virtualiser des systèmes d'exploitation existants, par exemple des machines virtuelles Linux. Il peut donc être utilisé dans un environnement *cloud*, comme hyperviseur. Une distinction

importante entre Anaxagoros et la majorité des hyperviseurs est sa capacité à exécuter, de manière sécurisée, des tâches ou des systèmes d'exploitation temps réel dur, par exemple le système temps réel PharOS [66], simultanément avec des tâches non temps réel, sur des processeurs multi-cœurs ou non. Ce besoin de sécurité est une cible intéressante de vérification, notamment parce qu'elle implique de l'exécution concurrente. Notre étude concerne l'utilisation de Frama-C pour vérifier une partie du module d'adressage virtuel (appelé aussi module de mémoire virtuelle) qui est l'un des plus critiques.

Dans la suite de ce chapitre, nous présentons cette étude de cas. En supposant un modèle mémoire séquentiellement consistant, nous avons effectué la vérification des versions séquentielle et concurrente d'une partie clé du module de mémoire virtuelle, dont le but est la manipulation des références vers les pages de mémoire. Nous y expérimentons la méthode proposée pour être capable de prendre en compte la notion d'exécution concurrente avec Frama-C et Wp qui ne la supportent pas nativement. L'avantage principal est de pouvoir profiter de la capacité de Wp à dialoguer avec les solveurs automatiques de formules SMT sans avoir à effectuer de développement à l'intérieur de ce greffon pour lui permettre de traiter un programme concurrent. Nous avons dû prouver quelques lemmes manuellement, mais ceux-ci ne sont pas liés aux problèmes de concurrence. Wp nous permet d'extraire automatiquement les objectifs de preuve qu'il génère vers l'assistant de preuve Coq, et donc de compléter la preuve.

Dans cette étude, la vérification peut être considérée complètement formelle selon l'hypothèse qu'aucune autre fonction n'interfère avec les mêmes variables que celles manipulées par la fonction que nous vérifions. Cette hypothèse est réaliste car les références aux pages de mémoire ne peuvent être modifiées que par une seule autre fonction qui a un comportement similaire à celle que nous vérifions. D'un autre côté, même en voyant cette vérification comme partielle, une étude de ce type sur un module critique, en isolation du reste du système, reste efficace pour éviter les problèmes de sécurité : si l'on vérifie que les invariants fondamentaux de la structure de la mémoire ne peuvent pas être violés lors de l'usage des fonctionnalités fournies pour sa modification, nous obtenons déjà des garanties de fiabilité.

La suite du chapitre est organisée comme suit. La section 3.2 présente l'hyperviseur Anaxagoros et le module d'adressage virtuel. La vérification de ce module est présentée à la section 3.3, où l'on détaille l'usage de la méthode pour la simulation (section 3.3.1), les moyens employés pour spécifier des propriétés de comptage dans les pages (section 3.3.2), la preuve automatique de

la spécification avec Frama-C (section 3.3.3) et de plusieurs lemmes avec Coq (section 3.3.4). La section 3.4 discute la correction de notre analyse globale, ainsi que les bénéfices et limitations que nous avons constatés dans son usage. La section 3.5 présente un ensemble de travaux de la littérature liés à la preuve de systèmes d'exploitation.

3.2 Module d'adressage virtuel d'Anaxagoros

Anaxagoros [64, 65] est un micro-noyau sécurisé qui peut virtualiser des systèmes d'exploitation existants, par exemple des machines virtuelles Linux. L'une des préoccupations majeures de ce système est la sécurité et notamment celle des ressources. Il fournit donc des garanties sur la qualité de service et un compte précis des ressources données aux machines, qu'elles soient du temps processeur ou de la mémoire.

Un composant critique pour assurer la sécurité dans un micro-noyau comme Anaxagoros est son module d'adressage virtuel. Les architectures x86 (comme beaucoup d'autres architectures) fournissent un mécanisme pour la traduction d'adresse virtuelle, qui permet de transformer une adresse manipulée au sein d'un programme en adresse physique réelle. Un des buts de ce mécanisme est d'aider à organiser l'espace d'adressage, par exemple en permettant à un programme d'allouer de grandes zones de mémoire contiguë. L'autre but est de contrôler la mémoire accessible par un programme. La mémoire physique est divisée en blocs de taille égale que l'on appelle *pages*. Ces pages peuvent avoir différentes utilisations : pages de données, tables de pages, répertoires de pages. Elles sont organisées sous la forme d'une hiérarchie : les répertoires de pages contiennent des références vers des tables de pages, qui à leur tour contiennent des références vers les pages de données.

Anaxagoros ne contrôle pas totalement ce qui est écrit dans les pages. Il autorise les tâches invitées à effectuer n'importe quelle opération dans les pages, à condition que cela n'affecte pas la sécurité du micro-noyau lui-même, ni celle des autres tâches invitées. Pour faire cela, il faut assurer deux propriétés. La première est que le programme ne peut changer que des pages qu'il possède, la seconde qu'il ne peut effectuer des modifications que si celles-ci sont compatibles avec l'utilisation indiquée par la structure qui la décrit (page de données, table de pages, etc).

En effet, le matériel n'empêche pas une table de page ou un répertoire de pages d'être aussi utilisés comme des pages de données. Donc si on n'ajoute pas de mécanisme de protection, une tâche malintentionnée pourrait changer les références dans une table ou un répertoire, et après une certaine séquence d'opérations de modifications, elle pourrait finalement avoir accès à n'importe quelle page du système, y compris celles qui ne lui appartiennent pas.

Le rôle du module d'adressage virtuel est justement de prévenir ce type de modifications non-autorisées. Cela s'appuie sur l'enregistrement du type de chaque page et le maintien d'un compteur de références vers chaque page (c'est-à-dire le nombre de fois où une page est référencée par les pages de plus haut niveau dans la hiérarchie). Le module s'assure ensuite que l'appel d'une fonction pour effectuer la modification d'une page est conforme à son type. De plus, pour permettre la réutilisation dynamique de la mémoire, le module doit permettre de changer le type d'une page. Pour éviter les attaques, changer le type demande quelques propriétés supplémentaires. Par exemple le contenu des pages doit être nettoyé avant de changer le type ; après une interruption du nettoyage celui-ci doit reprendre correctement ; les compteurs de références doivent être maintenus correctement ; les pages en cours de nettoyage ne doivent plus être référencées ; *etc*.

La fonction étudiée à travers cette étude est présentée en figure 3.1. Elle est chargée de créer une référence vers une page dans une table, et doit donc mettre à jour les compteurs de référence en conséquence pour prendre en compte la référence créée et une éventuelle référence supprimée. Les compteurs sont maintenus dans des tableaux qui informent, pour chaque page, du nombre de fois où elle est référencée. Le but est d'assurer que pour toute page, le nombre réel de références à cette page est au plus égal à la valeur indiquée par son compteur. De cette manière, si nous savons que le compteur est à zéro, nous savons également que la page n'est plus référencée. Elle pourrait donc être nettoyée et son type pourrait être changé.

Techniquement, l'algorithme doit aussi prendre garde au cache de l'unité de gestion de la mémoire (*memory management unit*, MMU), le *translation lookaside buffer* (TLB) qui permet d'accélérer les traductions. Celui-ci doit être vidé avant qu'une page ne puisse effectivement être réutilisée si son adresse est présente dans le cache. En effet, si cette adresse est conservée dans le cache, cela autoriserait un programme à changer une page après son nettoyage par le micro-noyau. Comme le vidage du TLB est coûteux, l'algorithme doit malgré tout éviter de l'effectuer quand c'est possible, par exemple lorsque l'on est capable d'assurer

qu'il n'y a pas d'adresse non voulue dans le TLB. Ces parties du système ne sont pas considérée dans la présente étude.

Nous nous focalisons sur une version simplifiée du module d'adressage virtuel qui comprend les aspects clés de celui-ci, à savoir les pages de données et les tables de pages utilisées conformément à leur type, l'ajout de référence vers des pages de données, le maintien correct des compteurs de références, ainsi que les exécutions concurrentes de la fonction de modification des références. Les simplifications effectuées concernent le remplacement des champs de bits normalement utilisés dans les descriptions des pages par un ensemble de tableaux et le fait que nous ne considérons qu'un niveau de hiérarchie dans les tables de pages. Une autre caractéristique de notre version est que certaines fonctions qui étaient normalement des blocs atomiques « longs » ont été séparés en multiples instructions atomiques plus courtes, permettant d'aboutir à une concurrence à un grain plus fin que celle, non vérifiée, présente dans Anaxagoros.

3.3 Vérification formelle

Cette étude a motivé le développement du greffon que nous présenterons dans le chapitre 4. En effet, Anaxagoros est un micronoyau préemptif, il est donc intrinsèquement concurrent. Par conséquent, nous devons prendre en compte cette concurrence. Frama-C ne traite actuellement pas les programmes concurrents et il n'y a pas de primitives de concurrence pour la version C99 de C, qu'il considère. Enfin, nos architectures récentes et leurs modèles mémoire faibles accroissent encore la difficulté d'analyse des programmes concurrents. Nous supposerons un modèle séquentiellement consistant dans l'analyse, et argumenterons de la validité de cette hypothèse quant aux modèles faibles en section 3.4.

Comme nous n'avons pas de primitives de concurrence, nous considérons deux classes de fonctions. La première contient les fonctions bas-niveau qui sont effectivement atomiques et que nous vérifions donc comme du code séquentiel. Nous avons spécifié toutes les fonctions bas-niveau du module d'adressage virtuel en acsl (15 fonctions, $\approx$ 500 lignes de code annoté). Ces fonctions ont été prouvées avec Frama-C et Wp, en utilisant les solveurs SMT Z3, CVC3 et CVC4. Cette preuve est automatique et prend environ 90 secondes. Cette phase de l'étude étant assez standard pour le domaine, nous ne la présentons pas en détails.

```
1  #define NOF  2048 //nb of frames
2  #define MAX   256 //max nb of mappings
3  #define SIZE 1024 //size of a page
4
5  page_t get_frame(uint fn);
6
7  uint mappings[NOF];
8
9  int set_entry(uint fn, uint idx, uint new){
10   // Step 1 -> read_map_new
11   uint c_n = mappings[new];
12   // Step 2 -> test_map_new
13   if(c_n >= MAX) return 1;
14   // Step 3 -> CAS_map_new
15   if(!compare_and_swap(&mappings[new], c_n, c_n+1))
16     return 1;
17   // Step 4 -> EXCH_entry
18   page_t p = get_frame(fn);
19   uint old = atomic_exchange(&p[idx], new);
20   // Step 5 -> test_map_old
21   if(!old) return 0;
22   // Step 6 -> FAS_map_old
23   fetch_and_sub(&mappings[old], 1);
24   return 0;
25 }
26
```

FIGURE 3.1 – *La fonction* `set_entry` *écrit la référence* `new` *dans la page* `fn` *à l'indice* `idx`

La seconde classe de fonctions considérée est celle des fonctions de plus haut niveau qui ne sont pas atomiques. Nous les décomposons comme des séquences d'instructions atomiques afin de *simuler* la concurrence par entrelacement des instructions, conformément à notre proposition. Nous nous concentrons sur la fonction du module en charge de modifier les références de pages. Le reste de cette section présente l'usage manuel de la méthode présentée dans le chapitre 1 pour simuler le parallélisme, l'introduction des propriétés principales que nous vérifions et leur preuve.

3.3.1 Simulation des exécutions parallèles

La figure 3.1 présente la fonction que nous voulons vérifier. Dans ce code,

— `NOF` (ligne 1) correspond au nombre de pages existantes en mémoire ;

— `MAX` (ligne 2) au nombre maximum de références que l'on peut poser sur une page ;
— `SIZE` (ligne 3) à la taille de chaque page en nombre de références (une référence nécessite 4 octets donc nous avons des pages de 4Ko).

La fonction `get_frame` déclarée ligne 5 permet d'obtenir un point d'accès à la page identifiée avec la valeur transmise par le paramètre `fn`. Le tableau `mappings` (ligne 7) indique, pour chaque identifiant de page, combien de fois elle est actuellement référencée par des tables de pages.

La fonction `set_entry` en figure 3.1 permet de placer une référence vers une page de données identifiée par `new` dans une table de pages `fn` à l'indice `idx`. Ce qui correspond à écrire la valeur `new` à cet indice dans la table de pages. Dans le même temps, les compteurs de références doivent être mis à jour : la page `new` est maintenant référencée une fois supplémentaire, et l'éventuelle référence précédemment présente ayant été effacée, la page mentionnée est maintenant référencée une fois de moins.

À l'étape 1 (lignes 10–11 de la figure 3.1), le nombre de références à la page `new` est lu en mémoire et stocké dans la variable locale `c_n`. Cette valeur doit être inférieure au nombre de références maximal autorisé, ici 256, pour éviter un dépassement de capacité (nous rappelons que cette valeur est normalement stockée dans un champ de bits), sinon l'opération est annulée (étape 2, lignes 12–13). À l'étape 3 (lignes 14–16), le compteur est incrémenté mais seulement si la valeur présente à ce moment en mémoire est égale à celle précédemment lue. Nous assurons l'atomicité de cette opération par l'usage d'une opération `compare_and_swap`. À noter que la valeur lue en mémoire peut avoir été modifiée plusieurs fois depuis notre lecture, l'important est que la valeur lue soit la même que celle déjà lue à l'étape 1. L'étape 4 (lignes 17–19) récupère un pointeur vers la table de page `fn` puis utilise une opération `atomic_exchange` pour échanger atomiquement la valeur à l'indice `idx` avec la valeur `new`, l'ancienne valeur étant renvoyée par cette fonction, et récupérée dans la variable `old`. L'étape 5 (lignes 20–21) contrôle cette valeur pour déterminer si elle correspond à une référence, à savoir que c'est un identifiant non-nul. Si c'est le cas, l'étape 6 (lignes 22–23) décrémente atomiquement le nombre de références à la page `old` étant donné que la référence en question n'existe plus. Cette opération est réalisée à l'aide d'une opération `fetch_and_sub` qui va atomiquement, lire une valeur en mémoire, calculer la soustraction voulue et écrire le résultat en mémoire. Nous pouvons noter que si `new` et `old` sont équivalentes, le même

compteur est d'abord incrémenté puis décrémenté, le nombre de références et la référence elle-même restent donc inchangés.

Le code de simulation obtenu est présenté en figures 3.2 et 3.3. Dans la première (figure 3.2), chaque étape mentionnée dans le code d'origine est maintenant simulée par une fonction prenant en paramètre le numéro du fil d'exécution exécuté. Elle effectue l'opération de l'étape en question pour ce fil et écrit la prochaine opération à exécuter. L'étape 0 génère des valeurs d'entrée pour les arguments passés à la fonction `set_entry`. Quand la dernière fonction de simulation est réalisée, nous déterminons que la fonction retourne à l'étape 0 et peut recommencer avec d'autres arguments. Les cas d'erreur sont traités de la même manière. Les instructions `//@ghost` seront détaillées dans la section 3.3.2. Le parallélisme est modélisé par une boucle d'entrelacements (figure 3.3) qui, à chaque itération, choisit un fil d'exécution au hasard et lui fait exécuter un pas de calcul.

Les valeurs des entrées et des variables locales pour les différents fils d'exécution sont gardées dans des tableaux (`fn`, `idx`, `new`, `c_n`, `old`) qui associent, à chaque numéro de fil d'exécution, la valeur de la variable correspondante pour ce fil. Le tableau `pct` stocke l'étape actuelle de chaque fil d'exécution (compteur de point de programme). Les instructions atomiques comme `compare_and_swap`, `atomic_exchange` et `fetch_and_sub` sont simulées par du code C standard dans les fonctions de simulation (puisque que chacune de ces fonctions de simulation est supposée être une étape atomique dans notre approche).

3.3.2 Compteurs de références et invariants globaux

Une des propriétés clé assurée par Anaxagoros est que le nombre de références indiqué par le compteur de références (`mappings[p]`) pour une page valide p est toujours supérieur ou égal au nombre de références existant effectivement en mémoire. En ajoutant à cela que la valeur de ce compteur est inférieure à une certaine limite, nous savons également que le nombre exact de références est aussi en dessous de cette limite. Nous ne comptons pas le nombre de références à la page 0 car cette valeur est considérée comme étant une absence de référence.

Notons Occ_a^v le nombre d'occurrences de la valeur v dans un tableau a (qui peut être une page), et Occ^v le nombre d'occurrences de v dans toutes les tables de pages de la mémoire. L'invariant global du module d'adressage pour le comp-

```
#define NOF  2048 //nb of frames
#define THD  1024 //max nb of threads
#define MAX   256 //max nb of mappings
#define SIZE 1024 //size of a page
uint mappings[NOF];
uint new[THD], idx[THD], fn[THD];
uint old[THD], c_n[THD];
uint pct[THD];
//@ghost uint ref[THD];

page_t get_frame(uint fn);
void gen_args(uint th){     // Step 0
  /* generate function args */
  pct[th] = 1;
}
void read_map_new(uint th){ // Step 1
  c_n[th] = mappings[new[th]];
  pct[th] = 2;
}
void test_map_new(uint th){ // Step 2
  pct[th] = (c_n[th] < MAX)? 3 : 0;
}
void CAS_map_new(uint th){  // Step 3
  if(mappings[new[th]] == c_n[th]){
    mappings[new[th]] = c_n[th]+1;
    //@ghost ref[th] = new[th];
    pct[th] = 4;
  }
  else pct[th] = 0;
}
void EXCH_entry(uint th){    // Step 4
  page_t p = get_frame(fn[th]);
  old[th] = p[idx[th]];
  p[idx[th]] = new[th];
  //@ghost ref[th] = old[th];
  pct[th] = 5;
}
void test_map_old(uint th){  // Step 5
  pct[th] = (!old[th])? 0 : 6;
}
void FAS_map_old(uint th){   // Step 6
  mappings[old[th]]--;
  //@ghost ref[th] = 0;
  pct[th] = 0;
}
```

FIGURE 3.2 – *Simulation des exécutions concurrentes de* `set_entry` *de la figure 3.1, opérations atomiques*

```
void interleave(){
  while(true){
    int th = choose_a_thread();

    switch(pct[th]){
    case 0 : gen_args(th);      break;
    case 1 : read_map_new(th); break;
    case 2 : test_map_new(th); break;
    case 3 : CAS_map_new(th);  break;
    case 4 : EXCH_entry(th);   break;
    case 5 : test_map_old(th); break;
    case 6 : FAS_map_old(th);  break;
    }
  }
}
```

FIGURE 3.3 – *Simulation des exécutions concurrentes de* `set_entry` *de la figure 3.1, entrelacements*

tage de références peut être formalisé sous la forme :

$$\forall e, \texttt{validpage}(e) \Rightarrow \text{Occ}^e \leq \texttt{mappings}[e] \leq \texttt{MAX}.$$

où $\texttt{validpage}(e)$ signifie simplement que le numéro de la page est entre 1 et `NOF`. Mais, s'il est possible de montrer que cette propriété est maintenue par la fonction `set_entry` analysée comme une fonction séquentielle non préemptible, elle n'est pas assez précise pour être prouvée dans un contexte *multi-thread*. En effet, nous ne possédons pas assez d'information pour assurer qu'avant de décrémenter le compteur (cf. étape 6 dans la figure 3.1) celui-ci est nécessairement supérieur strictement à 0.

Pour tracer plus précisément les valeurs, nous utilisons un invariant de cette forme :

$$\forall e, \texttt{validpage}(e) \Rightarrow \exists k, 0 \leq k \wedge \text{Occ}^e + k = \texttt{mappings}[e] \leq \texttt{MAX},$$

où k peut être défini comme étant l'écart entre le nombre réel de références à e dans les tables de pages (c'est-à-dire le nombre d'occurrences de ce e dans les tables) et la valeur indiquée par son compteur. Cet écart provient des références déjà comptées mais pas encore inscrites dans les tables de pages (entre les étapes 3 et 4 de la figure 3.1) et des références déjà supprimées des tables mais dont le compteur n'a pas encore été décrémenté (entre les étapes 4 et 6 de la figure 3.1). Autrement dit, un fil qui exécuterait `set_entry` crée un écart de 1 pour les références à la page `new` à l'étape 3, cet écart est résorbé à l'étape 4 mais en crée un nouveau pour le compteur de références à `old` (si `old` était une référence

vers une page valide, c'est-à-dire différente de 0), finalement, l'étape 6 supprime ce dernier écart (si la référence était 0, l'étape 5 arrête l'exécution avant cette dernière étape). Donc un fil d'exécution peut créer un écart d'au plus 1 pour au plus une référence à un instant donné.

Pour modéliser cet écart, nous ajoutons un tableau fantôme. Les variables fantômes introduites en ACSL avec le mot clé `ghost` sont des variables qui peuvent être écrites mais pas lues par un programme. Elles vont généralement servir du côté de la spécification à rendre explicite un état logique implicite. Ici, le tableau fantôme `ref` associe à chaque fil d'exécution le numéro de page pour laquelle il a créé un écart non résorbé et 0 sinon. Ce tableau est mis à jour par des instructions fantômes aux lignes 26, 35 et 43 de la figure 3.2 qui nous permet d'assurer la propriété énoncée pour `ref` formalisée par le prédicat ACSL de la figure 3.7.

La définition précise de k est donc $Occ^e_{\texttt{ref}}$, et l'invariant global final est donc :

$$\mathcal{I} : \ \forall e, \texttt{validpage}(e) \Rightarrow \ \mathrm{Occ}^e + \mathrm{Occ}^e_{\texttt{ref}} = \texttt{mappings}[e] \leq \texttt{MAX}.$$

Pour exprimer et prouver des propriétés à propos du nombre d'occurrences d'une valeur `e` dans les pages de mémoire, nous définissons en ACSL deux fonctions logiques abstraites accompagnées d'axiomes définissant leur comportement. La première produit le comptage pour le nombre d'occurrences dans un tableau (nous l'utiliserons pour compter dans des pages) dans une plage d'indices `[from,to[` (Fig. 3.4). La seconde (Fig. 3.5) permet ensuite de considérer un ensemble de pages, indicées `[from, to[`, et de ne compter les occurrences des références que dans celles qui sont des tables de pages et non de simples pages de données. Le label `L` définit le point de programme où l'on considère les valeurs. Par exemple, la valeur de Occ^e au label `L` s'écrit `occ_m{L}(e,0,NOF)` où la valeur `NOF` correspond au nombre de pages.

Les axiomes de la figure 3.4 définissent les cas suivants :

— la plage `[from,to[` est vide, donc il n'y a aucune occurrence de `e` (axiome `end_occ_a`);

— la plage `[from,to[` n'est pas vide et l'élément le plus à droite est la valeur `e` alors le résultat est 1, plus le nombre d'occurrences dans la plage `[from,to-1[` (axiome `iter_occ_a_true`);

— la plage `[from,to[` n'est pas vide et l'élément le plus à droite n'est pas la valeur `e` alors le résultat est le nombre d'occurrences dans la plage `[from,to-1[` (axiome `iter_occ_a_false`).

De la même manière, nous définissons les axiomes de `occ_m` tel que présenté en figure 3.5, l'idée étant alors de sommer le comptage d'occurrences dans les

```
/*@ axiomatic OccArray{
logic ℤ occ_a{L}(ℤ e, uint* t, ℤ from, ℤ to)
    reads *(t+(from .. to - 1));

axiom end_occ_a{L}:
∀ ℤ e, uint* t, ℤ from, to;
  from >= to ==> occ_a{L}(e,t, from, to) == 0;
axiom iter_occ_a_true{L}:
∀ ℤ e, uint* t, ℤ from, to;
  (from < to && t[to-1] == e) ==>
    occ_a{L}(e,t,from,to) == occ_a{L}(e,t,from,to-1) + 1;
axiom iter_occ_a_false{L}:
∀ ℤ e, uint* t, ℤ from, to;
  (from < to && t[to-1] != e) ==>
    occ_a{L}(e,t,from,to) == occ_a{L}(e,t,from,to-1);
}*/
```

FIGURE 3.4 – *Fonction logique* `occ_a` *pour le comptage d'occurrences dans les tableaux et axiomes*

pages (dont le pointeur de début est obtenu à l'aide de la fonction `frame`) si la page concernée est bien une table de pages (ce qui est indiqué par le tableau `pagetable`).

3.3.3 Preuve avec le greffon Wp de Frama-C

Nous avons utilisé WP pour prouver la simulation écrite, paramétré avec le modèle mémoire par défaut de WP à savoir son modèle typé. Dans celui-ci, les valeurs placées dans le tas sont stockées dans des tableaux logiques différents en fonction de leur type (un par type fondamental : entier, flottant et pointeur), ce qui interdit par exemple certaines conversions de type.

Nous ajoutons les contrats de chacune des fonctions simulantes ainsi que quelques lemmes, que nous détaillons dans la section 3.3.4, pour aider les prouveurs automatiques. Dans le code de simulation présenté par les figures 3.2 et 3.3, le but est d'assurer, pour chacune des fonctions simulantes, que si l'invariant global $\mathcal{I}$ est vrai en entrée, il est bien maintenu par la fonction. Cet invariant $\mathcal{I}$ est donc placé à la fois en précondition et postcondition dans le contrat. La figure 3.6 présente cet invariant en ACSL. Comme la fonction `occ_a` reçoit un pointeur en entrée, nous récupérons l'adresse de début de `ref` par la syntaxe `&ref[0]`, la conversion de tableau vers pointeur n'étant pas implicite en ACSL.

Les autres clauses fournies pour les fonctions simulantes énoncent des propriétés comme la définition de bornes pour les variables locales, les relations entre les variables locales, notamment avec le *ghost* `ref`, présenté par la figure 3.7 et les points de programme en entrée et sortie de chaque fonction.

```
/*@ axiomatic OccMemory{
logic ℤ occ_m{L}(ℤ e,ℤ from,ℤ to)
     reads *(memory+(from..to*SIZE-1)), *(pagetable+(0.. NOF-1));

axiom end_occ_m{L}:
∀ ℤ e, ℤ from, to;
  from >= to ==> occ_m{L}(e, from, to) == 0;
axiom iter_occ_m_true{L}:
∀ ℤ e, ℤ from, to;
  from < to && pagetable[to-1] == true ==>
    occ_m{L}(e,from,to) == occ_a{L}(e,frame(to-1),0,SIZE)
                         + occ_m{L}(e,from,to-1);
axiom iter_occ_m_false{L}:
∀ ℤ e, ℤ from, to;
  from < to && pagetable[to-1] != true ==>
    occ_m{L}(e,from,to) ==  occ_m{L}(e,from,to-1);
}*/
```

FIGURE 3.5 – *Fonction logique* `occ_m` *pour compter les occurrences dans une plage de pages et axiomes*

```
/*@ predicate counter_relation =
  mappings[0] == 0 &&
  (∀ ℤ k; 0 < k < NOF ==> 0 <= mappings[k] <= MAX) &&
  (∀ ℤ k; 0 < k < NOF ==> 0 <= occ_a(k,&ref[0],0,THD) <= MAX) &&
  (∀ ℤ k; 0 < k < NOF ==>
           mappings[k] == occ_m(k,0,NOF)+occ_a(k,&ref[0],0,THD));
*/
```

FIGURE 3.6 – *Invariant du compteur*

```
/*@ predicate pct_imply_for_thread(ℤ th) =
  (pct[th] <= 3 ==> ref[th] == 0      ) &&
  (pct[th] == 4 ==> ref[th] == new[th]) &&
  (pct[th] == 5 ==> ref[th] == old[th]) &&
  (pct[th] == 6 ==> ref[th] == old[th] && old[th] != 0);
*/
```

FIGURE 3.7 – *Prédicat définissant le lien entre le compteur de programme et le tableau* `ref`

```
/*@ lemma occ_a_separable{L}:
∀ ℤ e, uint* t, ℤ from, cut, to;
  from <= cut <= to ==>
    occ_a{L}(e,t,from,to) ==
      occ_a{L}(e,t,from,cut)+occ_a{L}(e,t,cut,to);
*/
```

FIGURE 3.8 – *Exemple de lemme* ACSL *pour compter dans deux sous-plages*

Le code simulant de la fonction `set_entry` est finalement composé de 610 lignes de code, comprenant 530 lignes d'annotations ACSL. 140 lignes sont nécessaires pour les axiomes et les lemmes à propos des compteurs d'occurrences. Nous définissons également quelques prédicats pour exprimer les bornes des variables simulantes (environ 50 lignes). Les lignes restantes sont les contrats de fonctions et quelques assertions ajoutées pour guider les prouveurs. Dans les contrats de fonction, 200 lignes sont en fait l'invariant global (10 lignes) qui est dupliqué pour chaque fonction.

WP génère environ 320 obligations de preuves, 190 d'entre elles étant dédiées à la boucle d'entrelacement (ces preuves sont en fait triviales puisqu'elles sont directement impliquées par les pré et post-conditions de chaque appel). Mis à part les lemmes, toutes les preuves sont déchargées par Z3 (v.4.3.1) ou CVC4 (v.1.3) en environ 65 secs sur QuadCore Intel Core i7-4800QM @2.7GHz.

Nous avons également testé différentes configurations pour les constantes à propos de la taille d'une page (`SIZE`), le nombre de pages (`NOF`) et du nombre maximal de fils d'exécutions (`THD`), afin de déterminer leur impact sur le temps de preuve. Il s'avère qu'elles n'ont pas d'impact. Poser une définition axiomatique des fonctions logiques de comptage, plutôt qu'écrire des fonctions récursives en ACSL, empêche les prouveurs automatique de tenter de dérouler la récursion pour calculer (ce pour quoi ils ne sont pas conçus).

3.3.4 Preuve de lemmes avec Coq

Pour faciliter la preuve de formules utilisant les fonctions logiques des figures 3.4 et 3.5, nous ajoutons des lemmes ACSL qui expriment des propriétés utiles à leur propos. Pour chaque fonction, nous avons 3 lemmes qui expriment les mêmes idées mais au niveau correspondant : soit tables de pages, soit la mémoire complète. La preuve de ces lemmes nécessite un raisonnement par induction où l'on doit faire attention à utiliser les bonnes hypothèses et les bons axiomes, ce que ne peuvent pas faire Z3 et CVC4. WP nous permet de termi-

ner la preuve en produisant une extraction des obligations correspondantes vers l'assistant de preuve Coq, où l'on peut effectuer interactivement la preuve.

Un bon exemple de lemme à propos du comptage d'occurrences dans un tableau est la propriété illustrée par la figure 3.8. Celle-ci énonce que nous pouvons séparer une plage d'indices d'éléments `[from,to[` dans laquelle nous voulons compter en deux sous-plages `[from,cut[` et `[cut,to[`, compter séparément dans les deux, et additionner les deux résultats pour obtenir le nombre d'occurrences dans la plage complète. Cette propriété nous est très utile puisqu'elle nous permet de partitionner des plages d'indices, de façon à ne garder que celles qui ont changé entre deux points de programmes, et d'assurer que tous les autres éléments sont restés inchangés, de même donc que les nombres d'occurrences dans ces plages.

La preuve de ce lemme repose sur une induction sur la distance entre les valeurs `to` et `from` et une analyse de cas sur `cut`. Dans le cas de base, la distance entre `to` et `from` est 0, auquel cas, la plage dans laquelle nous comptons est nulle et nous l'évacuons avec l'axiome `no_elem`. L'induction présente trois cas :

— `from` $=$ `cut`;
— `from` $<$ `cut` $<$ `to`;
— `cut` $=$ `to`.

Dans le premier cas, la sous-plage `[from, cut[` est vide et donc le nombre d'occurrences dans celle-ci est 0 par l'axiome `no_elem`, et la sous-plage restante est exactement la plage restante, d'où l'égalité. Le dernier cas est son symétrique. Finalement, la preuve du cas restant consiste à séparer en deux sous cas selon la valeur de la case du tableau à l'indice `to` et à choisir convenablement les deux axiomes `iter_occ_m_false` ou `iter_occ_m_true`.

Le second lemme que nous ajoutons énonce que si dans une plage d'indices, aucun élément n'a changé entre deux points de programme, alors pour chaque valeur, son nombre d'occurrences dans la plage en question n'a pas changé. Cette preuve est réalisée par une simple induction reliant l'égalité des valeurs à l'égalité du nombre d'occurrences.

Le dernier lemme énonce que si un unique élément d'un tableau a changé entre deux points de programme, alors le nombre d'occurrences diminue de 1 pour l'ancienne valeur, augmente de 1 pour la nouvelle, et que toutes les autres ont conservé le même nombre d'occurrences. La preuve de ce lemme utilise les deux précédents : nous utilisons le lemme de séparation pour partitionner les parties changées et inchangées. Pour ces dernières nous utilisons le lemme d'égalité des tableaux pour assurer que les nombres d'occurrences dans ces cas

n'ont pas changé. Finalement, une analyse par cas termine la preuve pour l'élément changé.

Ces trois lemmes sont également exprimés pour la fonction `occ_m`. La preuve compte environ 300 lignes de code Coq.

3.4 Discussion

3.4.1 Correction des hypothèses

Dans notre analyse, nous faisons deux hypothèses importantes : le programme analysé s'exécute selon le modèle mémoire séquentiellement consistant et la fonctionnalité vérifiée en isolation est la seule à pouvoir modifier les tables de pages. Dans cette première section, nous justifions que ces hypothèses sont réalistes.

Les accès à la mémoire partagée étant tous réalisés par l'intermédiaire de primitives atomiques qui vident les caches d'écriture, aucun des accès ne produit de *data-race* : `compare_and_swap` pour incrémenter le compteur de la nouvelle page, `atomic_exchange` pour changer la référence et `fetch_and_sub` pour décrémenter le compteur de l'ancienne page. Nos instructions n'écrivent pas les adresses des variables locales dans l'espace partagé donc les accès vers celles-ci sont sûrs.

Comme nous l'avons mentionné dans le chapitre 2, une propriété fondamentale attendue d'un modèle mémoire, même faible, est que si ses exécutions séquentiellement consistantes ne contiennent pas de *data-race*, alors toutes ses exécutions peuvent être considérées séquentiellement consistantes, une telle propriété étant effectivement respectée par les modèles mémoires courants. Par conséquent, dans cette étude, l'approche par simulation est correcte et le reste en présence de modèles mémoire faibles. Ici, la vérification d'absence de *data-race* est informelle car nous ne cherchons par à considérer le problème dans cette thèse, un outil automatique serait d'une grande aide pour le cas général.

Nous pouvons également justifier que l'ordre des instructions est (presque) total, empêchant ainsi les exécutions *out-of-order*. L'appel de la fonction `set_entry` ne peut pas arriver après les premières instructions de la fonction. Les trois premiers pas (lecture, test, et CAS) sont ordonnés par leurs dépendances de données et de contrôle. Le CAS et l'instruction `atomic_exchange` qui la suit sont strictement ordonnées par dépendance de contrôle. Le test de

la valeur de `old` est en dépendance de donnée avec l'échange atomique et la décrémentation de son compteur est en dépendance de contrôle avec ce test.

L'instruction `page_t p = get_frame(fn)` est la seule qui soit susceptible d'être réordonnée à n'importe quelle étape entre les étapes 0 et 4. En fait, la valeur lue par cette fonction est uniquement dépendante de la valeur d'entrée `fn` et de la valeur du pointeur de début de la mémoire qui n'est pas mutable. Aucun réordonnancement de cette instruction ne pourrait donc avoir un impact sur l'exécution du programme, donc nous avons décidé de la placer proche du point d'usage de la valeur reçue, à savoir l'étape 4. Cela nous évite d'ajouter une information sur la valeur du pointeur dans l'invariant.

Concernant la correction, il est important que la fonction vérifiée soit la seule autorisée à modifier les tables de pages et leurs propriétés. En fait, il existe une autre fonction dans le micro-noyau qui en a le pouvoir. Celle-ci est dédiée au nettoyage des pages avant leur restitution au système. Cette fonction pourrait être ajoutée à la vérification et l'algorithme qu'elle met en jeu est le suivant : « pour chaque référence dans la page, remplacer par 0, puis décrémenter le compteur de la page précédemment présente ». La manière de faire la preuve serait donc très similaire puisqu'elle a le même comportement que la suppression de référence dans la fonction d'échange.

3.4.2 Bénéfices et limitations constatés

Cette étude de cas confirme que la vérification déductive basée sur des prouveurs automatiques, combinée avec la possibilité de preuve interactive pour compléter la vérification est une approche efficace. La plupart des propriétés spécifiées sont prouvées à l'aide des solveurs SMT, permettant, lors de la vérification, de se concentrer sur les propriétés qui présentent une réelle difficulté. Le temps nécessaire constaté pour terminer les preuves interactives de ce programme est finalement beaucoup plus court que le travail global de spécification en lui-même.

La méthode de simulation utilisée nous permet bien de traiter du code C concurrent avec Frama-C/Wp qui n'offre pas cette possibilité à l'origine. De plus, l'effort nécessaire pour construire le code concurrent restait, dans cette étude, raisonnable comparativement à l'effort de spécification. Pour être complète, elle nécessite néanmoins d'analyser l'ensemble des fonctions susceptibles d'avoir un impact sur l'invariant des données accédées. Celles-ci étant alors isolées du reste du programme, cela limite finalement la quantité de code à traiter,

facilitant la vérification. Un point qui reste à assurer est une propriété d'encapsulation, assurant que cet ensemble de fonctions ne permet pas de fournir à l'appelant un accès aux ressources partagées qui ne soit pas exclusif ou permette la rupture de l'invariant.

Une limitation de cette approche manuelle est l'utilisation de tableaux dont la taille est connue et fixée. Fixer le nombre de fils d'exécution ne présente pas de problème particulier puisque nous ne souhaitons pas traiter les questions de démarrage et arrêt de ceux-ci. En revanche, donner une valeur connue à ce nombre peut écarter des comportements, par exemple un invariant peut être vérifié si le nombre de fils est inférieur à 4000 mais devenir faux au delà. Si le nombre de fils a été fixé à 2000 pendant l'analyse, cela la rend incorrecte. Dans notre analyse, le nombre de fils d'exécution intervient dans le calcul de Occ^{e}_{ref}. Cette valeur est bornée par `MAX_MAPPINGS` et par le fait que les nombres d'occurrences sont supérieurs à 0. En fixant la valeur du nombre de fils d'exécution au delà de cette borne, nous n'écartons donc pas de comportements. Une meilleure manière de procéder serait de rendre le nombre de fils d'exécution abstrait (fixe mais arbitrairement grand).

Écrire la simulation et sa spécification manuellement est bien entendu fastidieux et expose au risque d'erreur, nous voulons donc avoir accès à un outil permettant leur génération automatique à partir du programme d'origine. Cela nécessite en revanche d'étendre le langage ACSL pour pouvoir exprimer plus précisément les propriétés faisant intervenir la concurrence. Par exemple, le tableau fantôme `ref` n'existe pas dans le code d'origine et chaque valeur qu'il contient exprime l'état d'un fil d'exécution particulier. La propriété d'intérêt à son sujet, les occurrences dans ce tableau, nécessite une connaissance de l'ensemble des fils. Sans moyen d'exprimer cette propriété dans le code original, l'invariant ne peut pas non plus y être exprimé.

L'approche globale, même si elle est correcte, semble avoir un passage à l'échelle limité pour des programmes vraiment conséquents. En effet, traiter un grand nombre de fonctions demande de tracer correctement un grand nombre de variables locales de manière globale. Cela peut rendre la preuve automatique plus difficile. En effet, si les accès à des éléments locaux ne demandent pas d'appels au modèle mémoire, les variables globales le nécessitent, compliquant les objectifs de preuve. Néanmoins, les API concurrentes représentent généralement un volume de code limité répondant au besoin de limiter les phases de synchronisation, coûteuses en développement et en performance. Nous pensons donc que pour ces algorithmes, généralement critiques, conduire une preuve en pro-

fondeur sur les propriétés d'intérêt en utilisant cette méthode de simulation reste une approche pratique pour identifier des problèmes ou en prouver la validité.

3.5 À propos de la preuve d'OS

Cette partie décrit une sélection de travaux consacrés à la vérification de systèmes d'exploitation. Ceux-ci ne sont pas directement liés aux objectifs de cette thèse [1], mais il nous semble nécessaire de placer cette étude par rapport à la littérature de ce domaine.

3.5.1 Preuve interactive

Klein et al. [59] présentent la vérification formelle de seL4, un micro-noyau permettant aux systèmes l'exécutant d'atteindre le critère commun EAL7. Une autre vérification de micro-noyau est présentée dans [7]. Ces deux projets prennent en compte la concurrence entre le processeur et les composants autres (représentés par leurs pilotes), alors que notre but était de traiter la concurrence multi-processeur pour une fonction particulière. Leurs vérifications ont été effectuées par preuve interactive avec le prouveur de théorème Isabelle/HOL.

Un autre travail récent à propos de la vérification d'un module d'adressage virtuel [95] s'appuie sur le fait que ces modules sont généralement construits sous la forme de couches successives d'abstractions. Cette connaissance est utilisée pour structurer la preuve par raffinements successifs selon le même découpage, facilitant la création de la preuve et sa maintenance. Un cadre est fourni pour alléger le travail demandé pour les raffinements et la définition des couches d'abstraction. La preuve est réalisée interactivement avec l'assistant de preuve Coq.

Barthe et al. [15] présente la vérification d'un modèle de virtualisation. À la fois l'implémentation et la vérification sont effectuées en utilisant Coq, l'éloignant donc d'une implémentation réelle. Ce modèle permet de raisonner à propos de l'isolation entre les tâches invitées sur une base axiomatique qui modélise le comportement de l'hyperviseur, incluant les caches et le TLB. Notre travail s'intéresse à du code réel en langage C.

1. C'est pourquoi nous ne les avons pas intégrés au chapitre 2

Xu et al. [97] s'intéresse à la vérification d'un micro-noyau temps réel commercial, celui-ci est écrit en langage C et n'a pas été à l'origine écrit dans l'optique d'être vérifié. Ils se basent sur la logique de séparation concurrente. La notion de préemption est précisément modélisée par des règles d'inférence dédiées puisque c'est une notion clé pour prouver des propriétés sur les systèmes temps réel (notamment pour prouver que l'on assure effectivement le temps réel). La preuve est effectuée à l'aide de l'assistant de preuve Coq.

Les techniques de vérification que nous avons mentionnées dans cette section sont interactives. Les automatisations passent généralement par l'utilisation d'un langage de tactiques. Notre but est de maximiser la quantité de preuve automatique par l'usage de solveurs SMT. De tels solveurs augmentent la taille de la base de confiance mais la quantité d'automatisation est supérieure à celle que l'on peut obtenir par l'usage de tactiques dans les prouveurs interactifs lorsque l'on veut prouver des propriétés, même non triviales. Nous restons limités par le pouvoir de preuve de ces solveurs mais l'extraction depuis Wp vers Coq nous permet, au besoin, de terminer interactivement les preuves lorsque c'est nécessaire.

3.5.2 Preuve automatique

La vérification formelle d'un hyperviseur proposée dans [6] s'appuie sur VCC, un outil de vérification basé sur la logique du premier ordre. L'architecture interne est modélisée précisément dans VCC, où le système est ensuite prouvé correct. À la différence des travaux mentionnés dans la section précédente, la technique utilisée est basée sur de la preuve automatique avec Boogie et Z3. La vérification consiste à assurer que l'invariant du système est respecté par une succession infinie d'étapes.

Dans [8], Alkassar et al. présentent la vérification de la virtualisation d'un *translation lookaside buffer*, un composant crucial des hyperviseurs modernes. Puisque certains périphériques, comme les unités de gestion de la mémoire (MMUs), fonctionnent en parallèle avec le logiciel exécuté par le processeur, leur analyse nécessite de raisonner en terme de programmes concurrents même sur simple cœur. Leur travail donne une méthodologie pour vérifier les implémentations de composants virtuels, et l'utilise pour la vérification de la virtualisation d'un TLB avec VCC.

Cohen et al. [29] présente une discipline de programmation pour écrire des programmes concurrents séquentiellement consistants par construction.

Chen [27] indique que cette méthode n'est pas suffisante pour analyser des programmes capables de modifier leurs propres tables de pages et propose une extension pour compléter la discipline de programmation en question. Au lieu de considérer un modèle précis de la MMU x86, il propose un modèle abstrait de MMU qui permet de vérifier que la MMU associée à un fil d'exécution n'accède pas aux tables de pages d'un autre fil d'exécution. Comme nous l'avons mentionné dans la section 3.2, notre analyse ne prend pas en compte la MMU, ni le TLB. Il serait intéressant de déterminer comment ces résultats peuvent être intégrés dans notre approche.

Comme dans ces travaux, notre méthode essaie de tirer au maximum parti des possibilités offertes par les solveurs SMT. Une différence importante est que ces outils prennent intrinsèquement en compte la concurrence alors que Frama-C ne le fait pas par défaut, de même que Wp. Notre but est d'amener cette possibilité en minimisant les modifications à effectuer sur les outils. Nous pensons que la simulation d'exécution est le moyen le plus rapide d'étudier les difficultés à surmonter pour une intégration plus profonde.

3.5.3 Anaxagoros

La présente étude continue les efforts de vérification de modules critiques d'OS avec Frama-C. Pour minimiser les coûts, nous nous appuyons au maximum sur les techniques de preuve automatique, complétée avec de la preuve interactive lorsque c'est nécessaire.

Le seul travail existant [60] sur la vérification d'Anaxagoros présente une vérification formelle partielle du module d'adressage virtuel, en version séquentielle uniquement, incluant la fonction de nettoyage des pages. Cette vérification est complétée par de la génération de tests pour les fonctions non prouvées. Une fois les spécifications posées, la vérification est conduite par une succession de tentatives de preuve et l'ajout de nouvelles annotations pour fournir plus d'informations aux prouveurs SMT. Le processus est itéré jusqu'à obtenir la preuve ou identifier un point bloquant. Dans ce cas, la fonction est séparée en fonctions plus courtes ne comprenant que les instructions problématiques pour simplifier la preuve. Si certaines fonctions ne peuvent pas être prouvées, elles sont vérifiées par génération automatique de tests selon le critère *tous les chemins* de PathCrawler.

Notre travail permet d'ajouter la notion de concurrence absente dans le précédent travail. Si la phase d'ajout de précisions dans les annotations reste complè-

tement manuelle, le découpage en blocs courts est intrinsèque à notre méthode et permet d'avoir des objectifs de preuve qui sont, pour la majorité, relativement simples. Avec l'automatisation que nous présentons dans le chapitre 4, le processus de vérification du module se trouve fortement simplifié. Finalement, nous n'avons pas eu besoin de recourir aux tests pour terminer la preuve, nous évitant de devoir limiter les domaines de valeurs de nos variables pour achever la vérification, ce qui expose au risque que l'approximation se trouve être trop agressive.

La vérification formelle reste coûteuse à mettre en place sur du vrai code. Klein et al. [58] estime que la vérification de seL4 a consommé environ 25 personne-années, et a requis des experts du domaine. seL4 n'est composé que de 10,000 lignes de code C séquentiel, et la vérification a coûté environ $700 par ligne de code. Ces chiffres doivent être modérés par le fait que cette étude avait aussi un but de compréhension et que certaines phases auraient pu être automatisées plus fortement, mais les coûts restent quand même très élevés.

3.6 Conclusion

Dans ce chapitre, nous avons présenté la vérification d'une partie critique du micro-noyau Anaxagoros, à savoir son module d'adressage virtuel. Cette vérification comprend deux parties principales : la vérification des fonctions bas-niveau qui sont atomiques et celles de plus haut niveau qui ne le sont pas. Les premières sont vérifiées comme des fonctions séquentielles et les spécifications ACSL sont prouvées automatiquement avec Frama-C en utilisant le greffon de calcul de plus faibles préconditions Wp et les solveurs automatiques Z3, CVC3 et CVC4.

La fonction de changement de référence dans une page est de plus haut niveau et n'est donc pas atomique. Nous la prouvons en utilisant la méthode définie dans le chapitre 2. Nous simulons le parallélisme en modélisant le contexte d'exécution et produisons un programme effectuant les entrelacements des instructions des fils d'exécution. Les spécifications sont écrites en ACSL et les preuves sont déchargées en utilisant Z3 et CVC4. Les lemmes introduits pour la preuve des propriétés de comptage sont utilisés par ces solveurs, permettant la preuve. En revanche, les lemmes sont prouvés interactivement en utilisant l'assistant de preuve Coq.

Cette étude de cas illustre la capacité de notre méthode à traiter des programmes concurrents à l'aide de Frama-C et du greffon Wp, qui ne permet pas, à l'origine, de traiter de tels programmes. Les limitations identifiées par cette étude sont : un passage à l'échelle qui semble limité, le fait que l'écriture manuel de la simulation soit fastidieuse et propice aux erreurs, de même que la traduction des spécifications. Nous proposons de traiter les deux derniers axes par la création d'un greffon à Frama-C qui effectue cette transformation automatiquement à partir du programme d'origine. Nous présentons ce greffon dans le chapitre 4.

4 Transformation automatique de code

Sommaire

4.1 Introduction

Dans le chapitre 3, nous avons montré comment la méthode proposée dans le chapitre 1 nous permet d'effectuer la preuve d'un ensemble de fonctions accédant à une structure de données de manière concurrente. Bien que permettant la preuve, cette méthode reste lourde à mettre en place, notamment parce qu'elle

nécessite une transformation complète du programme d'origine. Une telle traduction est fastidieuse à réaliser manuellement et expose au risque d'erreur. De plus, un changement dans le code d'origine doit être manuellement porté dans le code transformé.

Dans ce chapitre, nous proposons un greffon à Frama-C appelé Conc2Seq [19], permettant de transformer automatiquement le code d'une API d'accès à une structure de données partagée concurrente en un code séquentiel équivalent qui la simule par entrelacements. Le premier greffon dont nous visons le support est Wp, mais l'objectif derrière le fait de procéder par transformation est de permettre à d'autres greffons de pouvoir analyser le code généré. Dans ce support, nous incluons notamment des constructions propres au monde concurrent, à savoir la notion de variable globale locale à un fil d'exécution et les blocs atomiques.

Comme nous transformons le code d'origine, les spécifications fournies par l'utilisateur doivent également être traduites et ajoutées au code généré. Cette tâche est aussi réalisée automatiquement par le greffon. Pour faciliter ces spécifications, nous supportons diverses constructions permettant d'exprimer plus aisément des comportements concurrents. Nous fournissons ainsi des spécifications pour des fonctions du fichier d'en-tête "atomic.h", ainsi qu'une opération de réduction sur les variables de différents fils d'exécution (fonction logique en langage acsl), qui facilite l'expression des spécifications.

Ce chapitre est organisé comme suit. Dans la section 4.2, nous présentons un tour d'horizon de Conc2Seq, ainsi qu'un exemple d'illustration. Nous présentons notamment les fonctionnalités offertes pour la spécification de programmes concurrents et les notions du C supportées. Ensuite, dans la section 4.3, nous présentons le fonctionnement de Conc2Seq, en particulier la manière dont le code et les spécifications sont transformés par le greffon. La section 4.4 présente la manière dont nous générons le code acsl nécessaire pour la fonction de réduction sur les variables des fils d'exécution. La section 4.5 discute les avantages et inconvénients de notre implémentation et la section 4.6 conclut sur ce chapitre.

4.2 Fonctionnalités du greffon Conc2Seq

Dans cette section nous présentons un exemple qui nous servira à illustrer le fonctionnement de Conc2Seq ainsi que l'ensemble des fonctionnalités fournies à

l'utilisateur pour le développement et la spécification, nous indiquons également les restrictions imposées pour l'usage de celles-ci.

Comme dans la partie précédente, les spécifications sont fournies par l'utilisateur par l'usage d'annotations en langage ACSL, introduites dans le code par la syntaxe `//@ <annotation>` ou `/*@ <annotation> */`. Ces annotations sont lues par Frama-C et peuvent être traitées par les greffons.

4.2.1 Exemple : simple écrivain, multiples lecteurs

Nous illustrons le fonctionnement du greffon Conc2Seq à l'aide d'un exemple de politique « simple écrivain multiples lecteurs », présentée dans la figure 4.1, dont le but est de donner la garantie que l'on accède à une donnée partagée avec un écrivain unique à un instant donné (tout en interdisant la lecture) mais que l'on n'empêche pas la lecture par de multiples lecteurs quand personne n'écrit. Plusieurs fils d'exécution peuvent exécuter de manière concurrente les fonctions `read` et `write` pour accéder à la variable partagée `d`.

Le paramètre `l` de la fonction `read` est l'emplacement mémoire auquel la valeur partagée sera écrite, cet emplacement est supposé, par pré-condition, *valide* au sens ACSL du terme, à savoir que c'est une position mémoire qui peut être lue et écrite de manière sure. Nous imposons également que le pointeur en question ne pointe ni la donnée partagée `d`, ni la variable `acc` (ligne 29).

La variable `acc` est associée à la variable `d` et permet d'encoder les droits d'accès à celle-ci. Si sa valeur est 0, aucun fil d'exécution n'accède actuellement à `d`. Si, par l'intermédiaire de la fonction `write`, un fil d'exécution passe la valeur de 0 à -1, il obtient un accès exclusif à la ressource, lui permettant de changer sa valeur. L'accès en lecture, quant à lui, se fait par l'incrémentation de la valeur d'`acc`, à condition que celle-ci soit supérieure ou égale à 0. Ainsi, de multiple fils d'exécution peuvent acquérir ce type d'accès. Finalement, pour restituer le droit d'accès obtenu, un fil d'exécution doit incrémenter (resp. décrémenter) la variable `acc` pour restituer son accès en écriture (resp. lecture). Nous supposons le nombre de fils d'exécution borné, ne nous préoccupant donc pas des dépassements de capacité sur la variable `acc`.

Le but de notre vérification est d'assurer que le contrôle de l'exclusion mutuelle est correctement réalisé par ce code.

```
1  int d, acc;
2  //@ghost int rd __attribute__((thread_local));
3  //@ghost int wr __attribute__((thread_local));
4
5  /*@ logic ℤ sum(ℤ a, ℤ b) = a+b ; */
6
7  /*@ predicate inv =
8        (acc >= -1) && 0 <= rd <= 1 && 0 <= wr <= 1
9      && (acc == -1 <==> (1   == thread_reduction(sum, wr, 0))
10                     && (0   == thread_reduction(sum, rd, 0)))
11     && (acc >=  0 <==> (acc == thread_reduction(sum, rd, 0))
12                     && (0   == thread_reduction(sum, wr, 0)));
13     global invariant sw_mr: inv;                    */
14
15 int write(int value){
16   int r, exp = 0;            // previous value must be 0
17   ATOMIC( r = atomic_compare_exchange(int, &acc, &exp, -1);
18           /*@ ghost wr = (r == 1) ;*/ );
19   if(! r )              // ensure write access was granted
20     return 0;
21
22   d = value;                                    // writing d
23
24   ATOMIC( atomic_fetch_add(int, &acc, 1); // increments acc
25           /*@ ghost wr = 0;*/ );
26   return 1;
27 }
28
29 /*@ requires \valid(l) && \separated(l, &acc, &d); */
30 int read(int* l){
31   int r, a = acc;
32   if(a >= 0){     // previous value must be non-negative
33     ATOMIC( r = atomic_compare_exchange(int, &acc, &a, a+1);
34             /*@ ghost rd = (r == 1) ;*/ );
35   }
36   else
37     return 0;
38   if(! r )              // ensure read access was granted
39     return 0;
40
41   *l = d;                                       // reading d
42
43   ATOMIC( atomic_fetch_sub(int, &acc, 1); // decrements acc
44           /*@ ghost rd = 0;*/ );
45   return 1;
46 }
```

FIGURE 4.1 – *Exemple de politique multiples lecteurs/simple écrivain*

4.2.2 Variables globales locales au fil d'exécution

Comme dans le chapitre précédent, nous allons utiliser du code fantôme pour expliciter des états implicites du programme.

Ici, nous les utilisons pour définir pour chaque fil d'exécution si c'est un lecteur ou un écrivain. Nous ajoutons deux variables *thread-local* fantômes `rd` et `wr` (cf. lignes 2–3) qui vont explicitement mentionner si le fil d'exécution est un écrivain (`wr`= 1, `rd`= 0), un lecteur (`wr`= 0, `rd`= 1), ou aucun des deux : (`wr`= 0, `rd`= 0).

Une variable *thread-local* est une variable globale dont chaque fil d'exécution a son propre exemplaire et qui n'est visible que par lui (à moins bien sûr qu'il transmette son adresse à un autre fil d'exécution). Cette fonctionnalité a été standardisée en C par la norme C11 [53], son support restant cependant optionnel par les compilateurs. Néanmoins, quand le support de cette fonctionnalité particulière n'est pas présent, la vaste majorité des compilateurs propose une solution alternative dont le comportement est équivalent.

Dans notre greffon, la spécification de cette notion se fait par l'usage d'un attribut :

```
1 type variable __attribute__((thread_local));
```

Un cas typique d'utilisation d'une variable locale à un fil d'exécution est la variable `errno`, permettant de collecter les erreurs en C. Si une telle variable était complètement globale, il serait impossible pour un fil d'exécution de contrôler correctement les erreurs apparaissant pendant son exécution car elle pourrait à tout moment être écrasée par un autre fil.

Dans le cadre de la spécification, sous la forme de code fantôme, elles nous permettent d'encoder l'état global d'un fil d'exécution très simplement et ainsi de poser des relations entre les états des différents fils d'exécution.

4.2.3 Primitives atomiques

Lors de l'écriture de fonctions concurrentes, il est souvent nécessaire de recourir à des primitives d'accès atomique à la mémoire. Ces primitives sont disponibles dans le standard C depuis C11 par l'intermédiaire du fichier d'en-tête `stdatomic.h`. Ces fonctions sont :

— `atomic_store`
— `atomic_load`
— `atomic_exchange`

— atomic_fetch_add
— atomic_fetch_sub
— atomic_fetch_and
— atomic_fetch_or
— atomic_fetch_xor
— atomic_compare_exchange

Chacune de ces fonctions possède deux versions, la version de base où le modèle est implicitement le modèle séquentiellement consistant et une version `func_explicit` où l'utilisateur peut explicitement indiquer le type d'ordre mémoire qu'il veut respecter pour cette opération. Nous ne faisons pas cette distinction étant donné que nous supposons travailler avec le modèle séquentiellement consistant.

`atomic_load` et `atomic_store` permettent respectivement de lire et écrire atomiquement à une position mémoire donnée. La fonction `atomic_exchange` combine les deux opérations précédentes en écrivant atomiquement une valeur à une position mémoire donnée à la place de l'ancienne valeur, qu'elle récupère et renvoie.

Les fonctions `atomic_fetch_<operation>(ptr_var, n)` permettent d'aller remplacer la valeur v présente à une position mémoire `ptr_var` par le résultat de l'opération indiquée entre v et `n`. Dans notre exemple, nous utilisons `atomic_fetch_add` et `atomic_fetch_sub` pour respectivement incrémenter et décrémenter atomiquement la variable `acc`.

L'opération `atomic_compare_exchange(ptr_var, ptr_exp, des)` peut être vue comme la version atomique du calcul :

```
int result;
if(*ptr_var == *ptr_exp){
  *ptr_var = des;
  result = 1;
} else {
  *ptr_exp = *ptr_var;
  result = 0;
}
```

Chaque version (explicite et implicite) possède deux variantes : faible et forte. La différence réside dans le fait que la version faible peut se comporter comme si la valeur `*ptr_var` était différente de la valeur `*ptr_exp` même si elles sont effectivement égales. Cela permet sur certaines d'architectures d'obtenir de meilleures performances. Il n'est donc pas nécessaire de prendre en compte cette

```
#define need_atomic_compare_exchange(TYPE)
/*@
  requires \valid(loc) && \valid(exp);
  assigns  *loc, *exp;
  atomic   \true ;
  ensures  \result == 1 || \result == 0;
  behavior succeed:
    assumes *exp == *loc;
    ensures \result == 1;
    ensures *loc == des ;
    ensures *exp == \old(*exp) ;
  behavior failed:
    assumes *exp != *loc;
    ensures \result == 0;
    ensures *loc == \old(*loc) ;
    ensures *exp == \old(*loc) ;
  disjoint behaviors ;
  complete behaviors ;
*/
int atomic_compare_exchange_strong_##TYPE(
  TYPE volatile* loc,
  TYPE* exp,
  TYPE des
)

#define atomic_compare_exchange(TYPE, loc, exp, des)
  atomic_compare_exchange_strong_##TYPE(loc, exp, des)

```

Figure 4.2 – *Spécification de* `atomic_compare_exchange`

différenciation : si nous faisons un accès de ce type c'est que nous ne sommes pas sûr de ce que nous allons lire, ces deux variantes ne changent finalement que la probabilité d'échec.

Nous avons spécifié toutes ces fonctions en utilisant le langage ACSL afin de permettre leur traitement par les greffons de Frama-C. Par exemple pour la fonction `atomic_compare_exchange`, le code est présenté en figure 4.2.

Une subtilité technique de ces fonctions est qu'elles sont en réalité des macros. On ne peut donc pas directement les spécifier en ACSL. Pour régler ce problème, nous implémentons un ensemble de macros qui permettent, pour un type donné, de créer les prototypes spécifiés des fonctions atomiques dont on a besoin. La seule restriction à l'usage est que nous demandons le passage d'un paramètre supplémentaire qui est le type cible. Par exemple, dans notre

code d'exemple, au lieu d'écrire `fetch_add(&acc,1)`, nous devons écrire `fetch_add(int, &acc, 1)`.

Un autre choix qui aurait pu être fait aurait été de laisser le préprocesseur faire la transformation de code correspondante avec des macros équivalentes où nous aurions spécifié les blocs générés atomiques (nous expliquerons cette fonctionnalité dans la section 4.2.4). Cependant, cela ne ferait qu'alourdir le traitement du code généré. Par exemple, dans le cas du calcul de plus faible précondition, chaque usage de fonction nécessiterait d'effectuer à nouveau la même passe de calcul et alourdirait les objectifs de preuve, alors qu'une spécification de fonction sans corps associé est considérée comme valide par hypothèse.

4.2.4 Fonctions et blocs atomiques

Nous donnons à l'utilisateur la possibilité de spécifier qu'une fonction, ou un bloc de code est effectué de manière atomique. Une telle information est alors considérée comme une hypothèse de l'analyse.

Actuellement, Conc2Seq n'accepte pas, au sein d'une fonction analysée, d'avoir des appels à des fonctions non atomiques car nous ne modélisons pas la pile d'exécution dans le code simulant, seulement un contexte par fonction. Spécifier des fonctions comme atomiques nous permet notamment de pouvoir appeler les primitives atomiques (que nous avons modélisées par des fonctions), mais également de pouvoir appeler des fonctions pures, n'accédant pas à la mémoire globale ou encore accédant à des valeurs statiquement déterminées. Comme aucun accès d'une telle fonction n'est soumis à des problèmes de concurrence, nous pouvons simplement considérer qu'elle agit de manière atomique.

Cette spécification est effectuée en ajoutant une clause ACSL `atomic \true;` à la spécification d'une fonction donnée :

```
/*@ atomic \true; */
void foo(){}
```

De la même manière, nous utilisons les contrats de bloc pour spécifier qu'un ensemble d'instructions est effectué de manière atomique :

```
void foo(){
  /*@ atomic \true; */ {
     i1 ;
     i2 ;
  }
}
```

Cette construction est particulièrement utile pour assurer que l'état que nous modélisons explicitement avec du code fantôme est cohérent avec l'état des autres variables du programme.

Par exemple dans notre code, l'état des variables `rd` et `wr` doit être cohérent avec la valeur de `acc` à tout moment. En effet, le but des deux variables fantômes est de rendre explicite les propriétés « ce fil est un lecteur », ou « ce fil est un écrivain ». Nous faisons cela en ajoutant des instructions explicites dans le programme, mais ces instructions ne correspondent pas à des opérations atomiques effectivement réalisées. Elles donnent juste un sens plus précis à l'instruction d'écriture effectuée sur `acc`. Pour maintenir la cohérence de l'état logique qu'elles représentent, les mises à jour de `wr` et `rd` doivent donc être réalisées dans la même étape atomique que la mise à jour de la variable d'accès. C'est pourquoi, lorsque l'on acquiert un droit d'accès en écriture (resp. en lecture), nous devons mettre à jour `wr` (resp. `rd`) avec la valeur 1 (et 0 lorsque l'on relâche cet accès) dans le même pas atomique que la mise à jour de `acc`.

Les blocs atomiques peuvent également être déclarés à l'aide de la macro `ATOMIC`. C'est cette dernière que nous utilisons dans le code d'illustration.

La seule restriction à propos de ces blocs atomiques est qu'il est interdit de sauter (via un `goto` par exemple) depuis un bloc atomique vers l'extérieur, ou inversement depuis l'extérieur vers l'intérieur. Cela briserait la sémantique de l'atomicité du bloc. La vérification que l'on ne brise pas cette atomicité n'est pas encore implémentée par le greffon. Cela sera ajouté dans les versions futures, cela ne présente en effet pas de difficulté particulière d'implémentation grâce à la connaissance (fournie par le graphe de flot de contrôle de Frama-C) pour chaque instruction, de ses successeurs et prédécesseurs, ainsi que de ses blocs englobants : s'il existe un saut entrant, une des instructions aura un prédécesseur

hors du bloc atomique ; s'il existe un saut sortant une instruction de saut aura un successeur hors du bloc.

Par cohérence avec la sémantique choisie pour les blocs atomiques dans notre formalisation (présentée dans le chapitre 5), les blocs atomiques imbriqués doivent être refusés par le greffon. Actuellement, l'erreur n'est pas clairement indiquée à l'utilisateur, ce traitement d'erreur doit être implémenté.

4.2.5 Réduction sur les fils d'exécution

Nous sommes capables, par l'intermédiaire des variables *thread-local*, de modéliser l'état d'un fil d'exécution donné. Pour pouvoir définir un état global du programme parallèle, il nous faut un moyen de combiner les propriétés de chacun d'eux.

Par exemple dans notre programme d'illustration, nous voulons être capables de garantir l'exclusion. Nous avons deux situations possibles : soit il existe une variable `wr` d'un fil d'exécution dont la valeur est 1, et auquel cas, ce doit être la seule et toutes les variables `rd` doivent être à 0. Soit toutes les variables `wr` sont à 0 et dans ce cas, il peut y avoir un nombre positif de variables `rd` à 1. Nous avons donc besoin de compter combien de variables ont une valeur donnée parmi tous nos fils d'exécution, et à tout moment de l'exécution.

Pour faciliter la spécification d'une telle propriété, nous étendons la bibliothèque d'ACSL avec une fonction logique *built-in* (intégrée directement au niveau de FRAMA-C), nommée `thread_reduction`. Cette fonction permet, à la manière d'un `fold` dans les langages de programmation fonctionnelle, d'effectuer une opération accumulant (selon une définition de l'accumulation fournie par une fonction en paramètre) tous les éléments d'un ensemble de valeurs. Cette fonction prend trois paramètres : la fonction d'accumulation à appliquer à chaque élément avec le précédent résultat calculé, la variable dont on veut calculer l'accumulation sur tous les fils d'exécution, et la valeur initiale depuis laquelle on commence le calcul.

Ici, nous l'utilisons pour exprimer, du côté de la logique, le nombre de fils d'exécution qui ont des accès en lecture et écriture. Par exemple, l'expression `thread_reduction(sum, wr, 0)` calcule le nombre d'écrivains comme la somme des `wr` de chaque fil d'exécution. Concrètement, le calcul est $sum(wr_0, sum(wr_1, ...sum(wr_{max}, 0)...))$, si on a max fils d'exécution.

4.2.6 Invariant global

La dernière notion supportée par le greffon Conc2Seq est la notion d'invariant global. Une propriété spécifiée comme telle sera une pré-condition de toute action atomique et devra être maintenue par cette action atomique. Elles sont introduites par la clause `global invariant`.

Ici, nous définissons que l'état de notre programme doit à tout moment respecter le prédicat `inv` qui définit dans un premier temps que la valeur de `acc` est supérieure ou égale à -1. Les autres propriétés définissent l'exclusion mutuelle. Si la valeur de `acc` est -1 alors, la première équivalence nous indique que le nombre d'écrivains est exactement 1 et que le nombre de lecteurs est exactement 0 (ce qui est compatible avec la seconde équivalence qui nous indique alors que soit le nombre de lecteurs est différent de `acc`, ce qui est vrai, soit que le nombre d'écrivains est différent de 0, ce qui est également vrai). Si la valeur de `acc` est supérieure ou égale à 0, sa valeur est exactement le nombre de lecteurs, et le nombre de d'écrivains est 0 (c'est également compatible avec la première équivalence qui nous indique alors que soit le nombre d'écrivains est différent de 1, ce qui est vrai, ou que le nombre de lecteurs est différent de 0, ce qui est vrai si on a au moins un lecteur). `acc` ne pouvant être à la fois supérieur et inférieur à 0, on a bien une exclusion entre lecteurs et écrivain.

Cela repose bien sûr sur le fait que notre modélisation des fils d'exécution écrivains et lecteurs par les variables fantômes est correcte. En effet, en écrivant par exemple `wr = 0`, nous spécifions d'une certaine manière que nous sommes certains, à cette étape, que le programme ne peut plus écrire.

Une manière de renforcer la preuve pourrait être de lier logiquement la valeur du compteur de programme à la valeur des variables `wr` et `rd`. Par exemple, nous pourrions ajouter une assertion exprimant l'exclusion sur la ligne de l'écriture (et de même sur la lecture). Les assertions ne sont pas encore supportées par la transformation, mais nous discutons cette possibilité dans la section 4.5. Une autre possibilité serait de pouvoir définir des propriétés comme vraies entre deux étiquettes particulières du programme, ces étiquettes seraient, à la traduction, transformées en identifiants de point de programme pour lesquels on indiquerait une propriété comme devant être vérifiée.

4.3 Production du code simulant spécifié

Cette section présente la technique de transformation que nous utilisons pour construire un code simulant séquentiel depuis un programme concurrent donné.

```
1  int *pct;
2  int *tl_rd, *tl_wr;
3  int *read_r, *read_a, **read_l;
4  int *write_r, *write_exp, *write_value;
5
6  /*@ axiomatic Validity_of_simulating_vars {
7    predicate simulation{L} reads <all simulation pointers> ;
8
9    axiom all_simulations_separated{L}:
10   simulation ==>
11     \separated( <all simulating memory blocks/globals> );
12
13   axiom pct_is_valid{L}:
14     simulation ==>
15       (∀ ℤ j; valid_th(j) ==> \valid(\at(pct,L)+j));
16   //same kind of axioms for each simulating pointer
17   //...
18 } */
```

FIGURE 4.3 – *Simulation du contexte d'exécution*

Il décrit également comment est construit l'outil qui effectue cette transformation automatiquement.

4.3.1 Contexte d'exécution

Le contexte d'exécution que nous considérons inclut les variables globales locales aux fils d'exécution, les variables locales et le compteur de programme.

Dans le code simulant, les variables globales sont conservées telles qu'elles sont. Les autres variables (*thread-local*, locales, formelles - paramètres de fonctions) sont transformées en tableaux qui associent, à chaque identifiant de fil d'exécution, la valeur de la variable correspondante pour ce fil. Nous ajoutons également un tableau pour les compteurs de programmes (nommé `pct` pour *program-counter*). Celui-ci nous indique, pour chaque fil d'exécution, l'identifiant de la prochaine étape atomique qu'il doit exécuter (ou celle qu'il est en train d'exécuter).

Dans notre modélisation, les variables sont simulées comme l'illustre la figure 4.3 avec le compteur de point de programme en ligne 1, les variables *thread-local* (ligne 2), les variables locales de la fonction `read` (ligne 3) et celle de la fonction `write` (ligne 4). Par exemple, les expressions `tl_rd[2]` et `pct[2]` représentent la valeur de la variable fantôme `rd` et le point de programme atteint

pour le fil d'exécution numéro 2. L'expression `read_a[2]` représente la valeur de la variable `a` de la fonction `read` pendant que le fil d'exécution 2 l'exécute (sinon c'est une valeur indéfinie).

Notre simulation ne permet pas la création dynamique de fils d'exécutions, nous posons comme hypothèse que ce nombre est borné et nous modélisons la borne à l'aide d'une valeur logique `MAX_THREADS` dont on pose seulement comme hypothèse qu'elle est supérieure à 0. Les tableaux utilisés pour la simulation ne sont donc pas des tableaux statiques mais des pointeurs dont nous définissons axiomatiquement qu'ils pointent vers une zone de mémoire valide pour tout indice de fil d'exécution (lignes 15–16), c'est-à-dire valides depuis l'indice 0 jusqu'à l'indice `MAX_THREADS-1`. Nous posons également l'hypothèse que tous les blocs de mémoire simulants sont séparés (ne sont pas en *alias*) les uns des autres ainsi que des variables globales (ligne 11).

Un point technique notable de cette partie concerne la forme des axiomes. Nous ne pouvons pas simplement poser des axiomes de la forme :

```
1 /*@ axiom all_simulations_separated{L}:
2       \separated( <all simulating memory blocks/globals> );
3     axiom pct_is_valid{L}:
4       (∀ ℤ j; valid_th(j) ==> \valid(\at(pct,L)+j)); */
```

Nous obtenons ici, une preuve de FAUX, car ils ne peuvent pas être vrais pour toute valeur que peuvent prendre les pointeurs considérés. Rien ne nous indique qu'ils ne sont pas modifiables et donc que l'on ne peut pas, par exemple, écrire `pct = &d`, ce qui rendrait la séparation fausse. Rien ne nous indique non plus que dans l'état initial, les pointeurs mènent vers des zones mémoires séparées.

Nous décidons donc de supposer que notre état initial respecte une certaine propriété `simulation`. Et nous indiquons, axiomatiquement, que tant que l'état du programme respecte cette propriété, la séparation et la validité des zones mémoire pointées sont vérifiées. La séparation et la validité ne sont donc plus vraies pour tout état, mais seulement pour ceux qui respectent `simulation`. Nous ne définissons pas explicitement cette propriété, la laissant abstraite. Nous indiquons simplement qu'elle dépend de la valeur des pointeurs et donc qu'une modification de ces pointeurs l'invaliderait.

Le prédicat `simulation` devient alors un invariant de notre simulation qui doit être maintenu par toute action pour garantir que nos axiomes font sens.

Pour résumer, chaque variable locale définie dans `orig` ou *thread-local* (dans ce cas `orig` = `tl`) de la forme :

```
type var;
```

est simulée par :

```
type * orig_var;

/*@ axiomatic Validity_of_simulating_vars {
  predicate simulation{L} reads ..., orig__var, ... ;

  axiom all_simulations_separated{L}:
  simulation ==>
    \separated( ..., orig__var + (0 .. MAX_THREADS-1), ...,
                globals ...);

  axiom origin_var_is_valid{L}:
    simulation ==>
      (∀ ℤ j; valid_th(j) ==> \valid(\at(orig_var,L)+j));

  ...
}*/
```

4.3.2 Actions atomiques, entrée de fonctions, entrelacements

Frama-C repose sur la bibliothèque CIL [71], qu'il utilise pour créer et normaliser l'arbre de syntaxe abstraite (AST) et construire le graphe de flot de contrôle (CFG) du programme. En particulier, les effets de bord des expressions sont sortis dans des instructions séparées, les instructions conditionnelles qui ont des conditions composées sont découpées en plusieurs conditionnelles où les conditions ne sont plus composées, toutes les boucles sont transformées en boucles `while(1)` équivalentes contenant des instructions conditionnelles supplémentaires provoquant les instructions `break` permettant de sortir du corps de la boucle.

La normalisation crée une instruction `return` qui est unique dans chaque fonction et qui renvoie la variable `__retres`, elle aussi créée pendant la normalisation (si le type de retour n'est pas `void`). Les instructions `return` d'origine sont remplacées par une affectation de la variable `__retres` suivie d'un saut vers l'instruction `return`.

L'AST associe des identifiants uniques à chaque instruction et structure de contrôle, que nous pouvons utiliser pour modéliser facilement notre compteur

```
1  /*@ requires valid_th(th) && *(pct+th) == 22 ;
2      requires simulation && inv ;
3
4      ensures *(pct+th) == 24;
5      ensures  simulation && inv ; */
6  void write_Instr_22(unsigned th){
7    d = *(write_value + th);
8    *(pct + th) = 24;
9    return;
10 }
11
12 /*@ requires valid_th(th) && *(pct+th) == 32 ;
13     requires simulation && inv ;
14
15     ensures *(pct+th) == 33 || *(pct+th == 37);
16     ensures  simulation && inv ; */
17 void read_If_32(unsigned th){
18   if (*(read_a + th) >= 0)   *(pct + th) = 33;
19   else                       *(pct + th) = 37;
20   return;
21 }
```

FIGURE 4.4 – *Fonctions de simulation des étapes atomiques en lignes 22 et 32 de la figure 4.1*

de point de programme. Par la suite, nous avons simplifié les exemples en utilisant des numéros de ligne.

Action atomique

Une fois l'AST normalisé, nous considérons que chaque instruction est une étape atomique, de même que tout bloc spécifié atomique par l'utilisateur. Nous modélisons chacune des étapes atomiques par une fonction qui prend comme paramètre le numéro du fil d'exécution dont on simule un pas de fonctionnement. Pour modéliser une étape atomique, l'idée est d'effectuer exactement la même opération, mis à part le fait que chaque accès à une variable *thread-local* ou locale est remplacé par un accès au tableau simulant correspondant à cette variable et pour le fil d'exécution en paramètre. Lorsque l'action est effectuée, il ne reste qu'à mettre à jour le compteur de point de programme avec l'identifiant de la prochaine action à effectuer.

La figure 4.4 illustre le résultat de la transformation pour 2 instructions de la figure 4.1 :

— l'affectation `d = value` à la ligne 22 de la figure 4.1 ;

— la conditionnelle à la ligne 32 de la figure 4.1.

Comme `d` est globale, nous conservons l'accès. A l'inverse, les accès à `a` et `value` sont remplacés par des accès aux tableaux simulants à l'indice du fil d'exécution `th`. Pour chaque fonction, `pct` est mis à jour conformément aux prochaines étapes à exécuter, donc pour la conditionnelle, nous avons bien deux choix qui dépendent de l'évaluation de la condition.

La figure 4.5 donne, pour chaque type d'instruction supportée par le greffon, le code de simulation correspondant. Actuellement, le greffon supporte :

— les affectations (Fig 4.5 : 1) ;
— les appels de fonctions atomiques (Fig 4.5 : 2) ;
— les retours de fonctions (Fig 4.5 : 3) ;
— les conditionnelles `if/else` (Fig 4.5 : 4) ;
— les conditionnelles `switch` (Fig 4.5 : 5) ;
— les boucles (Fig 4.5 : 6) ;
— les blocs atomiques (Fig 4.5 : 7).

Dans la figure 4.5, la notation `e' [th]` correspond à l'expression (ou l'instruction, dans le cas du bloc atomique) `e` où tous les accès à des variables non globales ont été remplacés par un accès à la variable qui la simule avec comme indice le numéro de fil d'exécution `th`. Cette expression est obtenue en effectuant une visite par copie, où seuls les nœuds de l'AST correspondant à un accès à une variable sont modifiés. Le `_i` correspond à l'identifiant de l'instruction dans l'AST.

Les instructions de saut inconditionnel, à savoir les `break`, `continue`, `goto`, et les entrées dans des blocs non atomiques, sont simplement remplacés par une redirection au sens où, lorsqu'une instruction mène à un saut de ce type nous allons parcourir récursivement le CFG jusqu'à rencontrer une instruction qui n'est pas de ce type et rediriger le contrôle vers elle. Ce comportement est produit par la fonction `next` qui est utilisée dans la figure 4.5. Celle-ci est évaluée à travers le CFG, pour aller chercher directement la prochaine instruction d'intérêt à exécuter. Pour les instructions de saut inconditionnel, nous ne générons donc pas de fonction de simulation puisqu'elles n'auraient aucun impact sur l'état du programme que ce soit au niveau local ou global.

Les retours de fonction n'ont pas de traitement particulier d'un point de vue exécution. En revanche, leurs fonctions de simulation peuvent permettre de vérifier la post-condition de la fonction simulée par l'ajout d'assertions.

Les instructions `if/else` et `switch` nous permettent de disposer de la liste des prochaines instructions à exécuter en fonction du cas. Cette fonctionnalité est

	Instruction d'origine	Fonction simulante

1	`v = exp ;`	`void f_Instr_i(unsigned th){`
		`  v'[th] = exp'[th] ;`
		`  pct[th] = next(i) ;`
		`}`

2	`g(e1, ..., en);`	`void f_Call_i(unsigned th){`
		`  g(e1'[th], ..., en'[th]) ;`
		`  pct[th] = next(i) ;`
		`}`

3	`return exp ;`	`void f_Return_i(unsigned th){`
		`  pct[th] = 0 ;`
		`}`

4	`if(exp){`
	`  //...`
	`} else {`
	`  //...`
	`}`
		`void f_If_i(unsigned th){`
		`  if(exp'[th])`
		`  pct[th] = next(if_true(i)) ;`
		`  else`
		`  pct[th] = next(if_false(i)) ;`
		`}`

5	`switch(exp){`
	`case C1: //...`
	`...`
	`case CN: //...`
	`default: //...`
	`}`
		`void f_Switch_i(unsigned th){`
		`  switch(exp'){`
		`  case C1:`
		`  pct[th] = next(sw(i,C1)) ;`
		`  //...`
		`  case CN:`
		`  pct[th] = next(sw(i,CN)) ;`
		`  default:`
		`  pct[th] = next(sw(i,def)) ;`
		`  }`
		`}`

6	`while(1){`
	
	`}`
		`void f_Loop_i(unsigned th){`
		`  pct[th] = next(fst_loop(i)) ;`
		`}`

7	`/*@ atomic \true */{`
	`  s1;`
	`  //...`
	`  sn;`
	`}`
		`void f_Block_i(unsigned th){`
		`  s1';`
		`  //...`
		`  sn';`
		`  pct[th] = next(n) ;`
		`}`

FIGURE 4.5 – *Table des transformations*

```
/*@ requires valid_th(th) && *(pct+th) == -30;
    requires simulation && inv ;

    ensures *(pct+th) == 31;
    ensures simulation && inv ;

    ensures \valid(*(read_l+th))
         && \separated(*(read_l+th), &acc, &d);*/
void init_read(unsigned int th);
```

FIGURE 4.6 – *Initialisation de l'appel à `read` dans la simulation*

déjà proposée par le CFG de FRAMA-C. C'est également le cas pour la boucle normalisée. Si celle-ci est vide, la prochaine instruction à exécuter est l'instruction de la boucle elle-même. Les sauts terminant les blocs atteints après la conditionnelle sont eux aussi calculés par FRAMA-C. D'un point de vue syntaxique, si l'instruction traduite comprenait un *label*, celui-ci est supprimé. Cela inclut notamment les labels `case` des `switch`, car hors d'un tel bloc ils n'ont pas de sens.

Les séquences d'instructions C dont l'ordre n'est pas spécifié sont détectées par FRAMA-C qui leur donne un sens particulier dans l'AST. Par exemple, considérons l'instruction suivante :

```
x = y++ + ++y;
```

L'ordre d'exécution des effets de bords sur `y` n'est pas spécifié. Dans ce cas, FRAMA-C produit, grâce à CIL une séquence d'instructions possible à partir de cette instruction, par exemple :

```
{ /* sequence */
    tmp = y;
    y ++;
    y ++;
    x = tmp + y;
}
```

Dans le greffon CONC2SEQ nous évacuons de tels programmes comme des erreurs.

Initialisation du contexte d'une fonction

Pour chaque fonction `f` du code vérifié pouvant être exécutée de manière concurrente par les fils d'exécution, une étape particulière simule le début de l'exécution de `f`.

Nous générons une fonction `init_f` qui n'est pas implémentée, nous lui spécifions seulement un contrat écrit en ACSL. Pour assurer que la pré-condition de `f` est respectée dans le code simulant, nous l'ajoutons comme post-condition de la fonction `init_f`. D'une certaine manière, cela modélise l'initialisation du contexte d'exécution de la fonction. En faisant cela, la simulation de l'entrée dans la fonction `f` consiste simplement à positionner les paramètres formels de `f` à des valeurs qui respectent sa pré-condition.

Nous identifions les étapes d'initialisation des fonctions avec des valeurs négatives correspondant à leur identifiant de fonction.

Nous illustrons cette transformation par la figure 4.6 qui montre l'étape qui simule l'entrée dans la fonction `read`, où la post-condition aux lignes 9–10 est exactement la pré-condition de `read` (figure 4.1, ligne 29).

Avec pour cible le support du greffon WP en sortie, spécifier uniquement la fonction sans l'implémenter est suffisant pour vérifier le code appelant. Pour certains greffons, cela ne pourrait pas être suffisant. Une alternative à une telle spécification serait par exemple de générer du code exécutable depuis la spécification, ce qui fait partie des fonctionnalités en cours de développement dans FRAMA-C.

Entrelacements

Une fois que toutes les fonctions simulantes des actions atomiques et des entrées de fonction ont été créées, nous modélisons les entrelacements par une boucle infinie qui, à chaque tour, choisit un indice de fil d'exécution au hasard et lui fait effectuer la prochaine instruction qu'il doit exécuter.

Si son compteur de point de programme est 0, une fonction de l'API est choisie également aléatoirement et commencera son exécution à la prochaine activation du fil d'exécution en question.

Nous illustrons ce comportement sur notre exemple dans la figure 4.7, simplifiée pour ne mentionner que les fonctions simulantes que nous avons présentées dans cette section.

```
/*@ requires simulation && inv ; */
void interleave(void)
{
  unsigned int th;
  th = some_thread();
  /*@ loop invariant simulation && inv ; */
  while (1) {
    th = some_thread();
    switch (*(pct + th)) {
      case 0   : choose_call(th);     break;
      case -15 : init_write(th);      break;
      case -30 : init_read(th);       break;
      case 32  : read_If_32(th);      break;
      case 22  : write_Instr_22(th); break;
      //... similar cases for other atomic steps
    }
  }
  return;
}
```

FIGURE 4.7 – *Simulation des exécutions concurrentes par entrelacements*

4.3.3 Spécifications

La génération automatique de spécifications dans la simulation comprend deux tâches principales : la traduction des spécifications de l'utilisateur depuis le code d'origine et l'ajout de nouvelles spécifications nécessaires pour définir le bon fonctionnement de la simulation elle-même.

Les spécifications de l'utilisateur sont de trois types :

— contrats de fonctions ;

— invariants globaux ;

— assertions ponctuelles dans le programme.

Contrats de fonction

Nous avons expliqué le support des pré-conditions dans la section 4.3.2.

Les post-conditions sont ajoutées dans la fonction de simulation de l'instruction `return` de la fonction d'origine (qui est unique grâce à la normalisation). Chaque post-condition est traduite en remplaçant les accès aux variables locales et *thread-local*, par des accès aux variables simulantes. La mention `\result` d'ACSL étant remplacée par une mention à la variable simulante de `__retres`.

Le support des assertions est discuté dans la section 4.5. Des travaux préli-

minaires ont été réalisés mais ne sont pas encore intégrés à la transformation complète.

Invariants globaux

Pour assurer la préservation des invariants globaux, nous les collectons et les insérons à la fois comme pré et post-condition de chaque fonction de simulation, ainsi que comme invariant de la boucle d'entrelacements comme l'illustrent les figures 4.4, 4.6 et 4.7. Si un invariant global comprend des variables *thread-local*, nous effectuons le même type de remplacement dans les expressions que dans la génération du code. À savoir que chaque accès à ces variables est traduit par un accès à sa version simulante. Il y a deux cas de figures pour une telle propriété : soit les variables en question sont accédées par le *builtin* `thread_reduction` et nous faisons un remplacement qui sera décrit dans la suite de la section, soit c'est un accès direct et nous ajoutons donc une quantification universelle sur tous les identifiants de fil d'exécution.

La correction du flot de contrôle est assurée dans la simulation, par le fait que nous spécifions pour chaque fonction simulante le point de départ et d'arrivée du compteur de point de programme. Nous ajoutons également un invariant global qui vérifie pour chaque identifiant de fil d'exécution que son compteur de point de programme est correct (qu'il existe effectivement un tel point de programme).

4.4 Traduction du *built-in* `thread_reduction`

4.4.1 Passage au premier ordre

La fonction `thread_reduction` est une fonction de second ordre. La traduction de ses usages comprend :

— la création d'une version de premier ordre pour chaque type et chaque fonction passée en paramètre ;
— l'utilisation de la variable simulante à la place de la variable d'origine ;
— le remplacement des usages par ces nouvelles fonctions logiques.

Nous présenterons ces phases à partir de l'exemple de code original proposé en figure 4.8.

```
// Original code :
//@ghost int wr __attribute__((thread_local));
/*@
  logic ℤ sum(ℤ a, ℤ b) = a + b ;

  predicate example =
    1 == thread_reduction(sum, wr, 0) ;
*/
```

FIGURE 4.8 – *Exemple d'usage de* `thread_reduction`

```
/*@ logic ℤ sum(ℤ a, ℤ b) = a+b; */
/*@
  axiomatic Axiomatic_red_sum_int {
    logic ℤ red_sum_int{L}
      (int *a, ℤ f, ℤ t, ℤ b)
      reads *(a+(f .. t-1));

    axiom red_sum_int_empty{L}:
      ∀ int *a, ℤ f, ℤ t, ℤ b;
        f >= t ==> red_sum_int(a, f, t, b) == b;

    axiom red_sum_int_iter{L}:
      ∀ int *a, ℤ f, ℤ t, ℤ b;
        f < t ==>  red_sum_int(a, f, t, b) ==
                     sum(*(a+f), red_sum_int(a, f+1, t, b));
  }
*/
```

FIGURE 4.9 – *Axiomatique pour* `thread_reduction`*, avec type* `int` *et fonction* `sum`

```
1 int* tl_rw; // simulation of wr
2 /*@
3   predicate example =
4     1 == red_sum_int(tl_wr, 0, MAX_THREAD, 0) ;
5 */
```

FIGURE 4.10 – *Traduction de 4.8 à partir de 4.9*

La figure 4.9 illustre la génération des axiomes correspondant à l'usage du *built-in* de la figure 4.8. Dans cette axiomatique, `a` correspond à *array*, `f` à *from*, `t` à *to* et `b` à *base*.

La définition axiomatique énonce deux cas :

— la plage de valeur est vide (`f` est supérieure ou égale à `t`), alors le résultat est la valeur de base `b` (`red_sum_int_empty`, lignes 8–10) ;

— la plage de valeur n'est pas vide (`f` inférieure à `t`), alors le résultat correspond à l'application de la fonction passée en paramètre (ici `sum`) sur le premier élément de la plage et l'application de la réduction sur les éléments restant (`red_sum_int_iter`, lignes 12–15).

Une fois la génération de l'axiomatique réalisée, nous pouvons remplacer les appels à la fonction logique originale par la fonction générée avec la variable simulante correspondante. La figure 4.10 illustre la transformation effectuée à partir du code de la figure 4.8.

4.4.2 Génération des prédicats et lemmes

Le lecteur pourra constater que ces axiomes sont proches de ceux présentés dans la section 3.3.2. De la même manière, nous définissons également des lemmes indiquant les propriétés de la valeur reçue de l'appel à la réduction en fonction des changements effectués sur les tableaux simulant les variables concernées.

Ces lemmes sont au nombre de trois, présentés dans la figure 4.11.

Premièrement, si toutes les cellules du tableau sont les mêmes (prédicat `red_sum_int_same`, lignes 2–5), le résultat est le même (`red_sum_int_same_meaning`, lignes 15–18).

Deuxièmement, si une valeur a changé à une position `i` (prédicat `red_sum_int_1mut`, lignes 8–12), on sépare le résultat de la réduction au label `L2` en trois parties (`red_sum_int_1mut_meaning`, lignes 28–33) :

```
/*@
  predicate red_sum_int_same{L1, L2}
      (int *a, ℤ f, ℤ t) =
     ∀ ℤ j; f <= j < t ==>
        \at(*(a+j),L1) == \at(*(a+j),L2);
 */
/*@
  predicate red_sum_int_1mut{L1, L2}
    (ℤ i, int *a, ℤ f, ℤ t) =
    f <= i < t && \at(*(a+i),L1) != \at(*(a+i),L2) &&
    red_sum_int_same{L1, L2}(a, f, i) &&
    red_sum_int_same{L1, L2}(a, i+1, t);
*/
/*@
  lemma red_sum_int_same_meaning{L1, L2}:
    ∀ int *a, ℤ f, ℤ t, ℤ b;
      red_sum_int_same{L1, L2}(a, f, t) ==>
        red_sum_int{L1}(a,f,t,b) == red_sum_int{L2}(a,f,t,b);
 */
/*@
  lemma red_sum_int_1mut_meaning{L1, L2}:
    ∀ ℤ b, ℤ i, int *a, ℤ f, ℤ t;
      red_sum_int_1mut{L1, L2}(i, a, f, t) ==>
        red_sum_int{L2}(a, f, t, b) ==
           red_sum_int{L1}(a, f, i,
              sum(\at(*(a+i),L2),red_sum_int{L1}(a, i+1, t, b)));
 */
/*@
  lemma red_sum_int_separable{L}:
    ∀ ℤ cut, int *a, ℤ f, ℤ t, ℤ b;
      f < cut < t ==>
        red_sum_int(a, f, t, b) ==
          red_sum_int(a, f, cut, red_sum_int(a, cut, t, b));
 */
```

FIGURE 4.11 – *Prédicats et lemmes pour* `thread_reduction`*, avec type* `int` *et fonction* `sum`

— résultat de la reduction depuis 0 jusqu'à `i` au label `L1` (avant mutation), recevant comme base, la deuxième partie, à savoir
— l'application de la fonction en paramètre de la réduction (ici `sum`), sur la cellule en position `i` au label `L2` (après mutation) et la troisième partie, qui est
— le résultat de la réduction depuis `i+1` à `MAX_THREAD` au label `L1`.

Finalement, le dernier lemme introduit est `red_sum_int_separable`, lignes 20–24, énonce que l'on peut séparer le calcul de la réduction de toute plage de valeur `[from ; to)` à n'importe quel indice intermédiaire `cut`.

La manière dont nous construisons ces lemmes à partir du *built-in* et d'une fonction ne permet pas aux solveurs SMT de prouver directement des propriétés qui nécessitent une induction précise. Par exemple, avec la fonction `sum`, nous ne pouvons pas prouver, après une mutation de la cellule `i` de n à m, que le résultat est $result_{new} = result_{old} - n + m$. En effet, nous ne pouvons pas énoncer automatiquement une telle propriété sans connaissances supplémentaires à propos de `sum`. Si l'utilisateur est capable de donner des spécifications à propos de la fonction fournie (par exemple, associative, réflexive, commutative ...), nous pourrions poser des lemmes plus précis, à nouveau générés automatiquement, ainsi que les objectifs de preuve correspondants à ces propriétés. Ceci n'est pour l'instant pas encore réalisé.

4.5 Discussion

Actuellement, quand nous générons les spécifications des fonctions simulantes, nous ne gardons pas la trace des relations qui peuvent exister entre les variables locales. Par exemple sur un programme simple comme :

```
1 int x = 1 ; int a = x;
```

Nous générons deux fonctions de simulation, mais dans la simulation de la seconde instruction, la spécification ne mentionne pas le fait que `x` vaut 1. Nous sommes donc localement incapables de prouver que `a` devient 1. L'utilisateur doit ajouter manuellement la spécification liant le point de programme avec les relations entre ses locales.

Ce type de relation pourrait être généré de manière automatique. L'idée étant de réaliser, sur la fonction d'origine, une analyse en avant. Nous devons juste prendre garde à jeter les parties de la propriété calculée qui comprennent

des mentions à des variables globales puisqu'elles peuvent être modifiées par d'autres fils d'exécution entre deux instructions. Les assertions de l'utilisateur, que nous mentionnions précédemment (section 4.3) peuvent être collectées de la même manière. L'ensemble de propriétés calculées serait ensuite transformé conformément aux variables de simulation, de la même manière que le sont les propriétés et instructions déjà traitées.

Nous avons réalisé une étude préliminaire pour cette analyse. Elle repose sur un calcul de plus forte post-condition. Pendant la collecte des instructions, qui part du début de la fonction, nous générons pour chacune d'elles :

— la propriété vraie avant son exécution, qui lui est associée ;
— un calcul de la propriété vraie après son exécution, transmise à tous ses nœuds suivants (calculés par CIL).

Si une instruction comprend des assertions, celles-ci sont collectées et ajoutées à la liste des propriétés en pré-condition de l'instruction. Ce calcul n'est pas complet, puisque nous ne calculons par exemple pas le point fixe en cas de boucle, et que nous ne traitons pas les appels de fonction atomiques. Néanmoins les propriétés générées peuvent être transformées correctement et semblent pouvoir être intégrées à la génération du contrat des simulations. Ceci n'est pas encore implémenté par le greffon Conc2Seq.

Le contexte d'exécution est modélisé en utilisant des pointeurs vers des blocs de mémoire axiomatiquement valides. Par rapport aux tableaux statiques du C, ces pointeurs ne nous offrent pas « gratuitement » la séparation entre les blocs et nous devons donc axiomatiser cette séparation, mais également le fait que la validité est dépendante de la valeur des pointeurs. Il serait plus pratique de pouvoir utiliser des tableaux statiques de taille indéfinie (`type t[]` en langage C). Cela nous permettrait de n'avoir qu'à axiomatiser la validité des blocs de mémoire (la séparation et la validité de ces « pointeurs » étant gratuites de par la sémantique de ces tableaux). Wp, qui est le premier greffon ciblé par notre travail, ne traite pas encore ce type de tableau, et c'est pourquoi nous utilisons des pointeurs. Pour cet objectif précis, ajouter le support des tableaux de taille indéfinie à Wp serait particulièrement intéressant car cela permettrait une meilleure traduction des objectifs de preuve vers les solveurs SMT. Une alternative serait d'ajouter une option au greffon, par laquelle l'utilisateur pourrait spécifier une taille fixe que nous utiliserions pour générer des tableaux de taille fixe.

Certaines API peuvent contraindre le nombre de fils d'exécution susceptibles d'utiliser les fonctionnalités concurrentes fournies. Par exemple, forcer l'utilisateur à n'avoir qu'un seul fil pour les fonctions f_1 et f_2 (et toujours le même) et

autoriser autant de fils que voulu pour les fonctions f_3, f_4 et f_5. Pour cela, nous pourrions ajouter une spécification énonçant les propriétés que doit avoir l'identifiant d'un fil pour qu'il puisse exécuter la fonction. Dans la simulation, cela influerait sur la spécification de la fonction `choose_call` qui devrait prendre en compte l'identifiant.

L'extraction de la simulation dans un fichier de sortie est pratique pour terminer la preuve. Les noms sont générés et les fonctions sont organisées de façon à faciliter la lecture et rendre plus simple l'ajout de spécifications dans le code généré. Par exemple certaines fonctions de second ordre, comme le comptage d'occurrences présenté dans la section 3.3.2, ne peuvent pas être exprimées à l'aide du *built-in* de réduction sur les fils d'exécution et nécessitent donc d'être exprimées par l'utilisateur au niveau du code simulant. De plus, certaines preuves ne sont pas déchargées automatiquement sans l'ajout d'assertions supplémentaires pour guider le processus de preuve.

Par exemple, pour compléter la preuve du code en figure 4.1 avec Wp, nous devons ajouter des assertions. Suite à cela, la plupart des objectifs de preuve sont prouvés en fournissant les relations entre les locales dans les fonctions simulantes. Sur un code légèrement différent de celui présenté dans ce chapitre, où l'on a séparé les différentes conjonctions de l'invariant pour en créer de multiples qui sont moins conséquentes, Wp génère 718 obligations de preuves, dont 441 pour la fonction entrelacement, où les preuves sont triviales (applications directes des contrats de fonction). 704 obligations sont prouvées en utilisant Frama-C Aluminium, Alt-Ergo 1.01 et Z3 4.4.2. Cela nécessite environ 260s sur un processeur QuadCore Intel Core i7-4800MQ @2.7GHz. Les 14 preuves restantes sont à propos de la somme des lecteurs et écrivains, elles sont plus difficiles car elles nécessitent d'éviter le raisonnement inductif complet par les prouveurs. Il faut donc placer les bonnes assertions en utilisant les lemmes générés à partir de la fonction *built-in* `thread_reduction`.

Les preuves des lemmes sont très semblables à celles proposées pour les lemmes de comptage du chapitre 3. Le principe est une nouvelle fois de procéder par induction sur la distance entre le début et la fin de la plage d'adresses considérée. Pour faciliter la preuve des lemmes dans de nouvelles instanciations du *builtin*, il serait intéressant d'avoir un squelette de preuve Coq générique qui puisse être appliqué pour prouver rapidement la validité des lemmes.

4.6 Conclusion

Dans ce chapitre nous avons présenté le greffon Frama-C Conc2Seq, qui implémente la méthode que nous avons proposée dans le chapitre 2. Il permet l'analyse d'API concurrentes en langage C à l'aide de greffons existants, pour le moment concentré sur Wp afin de tirer parti des solveurs SMT, avec lesquels il peut communiquer pour effectuer une vérification déductive de propriétés fonctionnelles. Cela passe notamment par l'ajout de fonctions à acsl afin de permettre la spécification de comportements concurrents.

Cette transformation est principalement syntaxique. La génération du code produit un programme qui est compréhensible, ce qui est utile pour terminer la vérification, par exemple en ajoutant des assertions dans le code généré.

Avec ce greffon, il est par exemple possible de générer le code simulant du programme que nous avons présenté dans le chapitre 3. En ce qui concerne les spécifications, certains points restent à implémenter. Il serait notamment intéressant d'avoir un moyen de spécifier la fonction logique de comptage par un *built-in*. Une autre fonctionnalité nous semble nécessaire pour dire que toute la simulation réalisée pour Anaxagoros est maintenant automatisée : la passe d'analyse collectant les relations entre les variables locales et les assertions de l'utilisateur, afin de les insérer dans les contrats des fonctions simulantes.

Actuellement, la manière dont nous effectuons cette transformation cible en priorité Wp. Certaines constructions du langage acsl que nous utilisons ne sont pas supportées par d'autres greffons importants, notamment la vérification par interprétation abstraite et le greffon de vérification à l'exécution. Il serait intéressant de combler ce manque. Il serait également intéressant d'ajouter le support des tableaux statiques de taille indéfinie à Wp.

5 Preuve de la méthode de transformation

Sommaire

5.1 Introduction

Dans le chapitre 4, nous avons présenté une implémentation de notre méthode de simulation appliquée au langage C pour l'analyse d'ensembles de fonctions pouvant être appelées de manière concurrente. Nous avons notamment présenté comment utiliser le greffon Wp de Frama-C pour prouver du

code concurrent grâce à notre greffon Conc2Seq. Pour qu'une telle preuve nous assure bien que notre code est prouvé, il faut que la transformation nous garantisse que le code résultant est bien sémantiquement équivalent au code d'origine.

C'est pourquoi nous voulons vérifier que notre transformation est formellement correcte dans le cadre de la sémantique d'entrelacements. Nous effectuons cette vérification sur un langage simplifié capturant les propriétés d'intérêt pour la validité, notamment les accès à la mémoire et les structures de données basiques ainsi que les exécutions parallèles. Nous formalisons ce langage et sa sémantique (séquentielle et parallèle) grâce à l'assistant de preuve Coq et implémentons la transformation de code pour ce nouveau langage.

Dans le langage simplifié, nous ne considérons pas toutes les structures de contrôle du langage C, nous restreignant aux conditions et aux boucles (les `goto` et les `switch` n'étant par exemple pas considérés). Pour l'affectation, nous distinguons explicitement les écritures et les lectures en mémoire globale, et celles qui ne font intervenir aucun accès à la mémoire globale. Ces instructions n'autorisent par ailleurs qu'un seul accès à la mémoire sur un pas d'exécution et la valeur chargée doit être immédiatement placée dans une variable locale. Une expression ne peut être composée que de constantes et de variables locales. Nous autorisons les appels de fonctions tant que ceux-ci ne sont pas récursifs.

La formalisation du langage et de sa sémantique opérationnelle en langage Coq sont réalisées, de même que la nouvelle fonction de transformation de code pour les instructions de base (hors bloc atomique). La sémantique du programme est à petit pas, l'évaluation des expressions est à grands pas. L'expression des équivalences d'états et de traces en Coq doivent encore être réalisées ainsi que la preuve. Dans ce chapitre, les relations et preuves correspondantes sont des preuves papier.

Dans la suite de ce chapitre, nous présentons d'abord le langage considéré et sa sémantique (section 5.2). Ensuite, nous décrivons la transformation adaptée pour ce nouveau langage (section 5.3). Nous présentons la relation d'équivalence entre programme d'origine et programme simulé, et la preuve de la simulation dans la section 5.4. Enfin, nous concluons sur cette partie de notre contribution (section 5.5).

5.2 Langage simplifié

Nous considérons un ensemble dénombrable de plages d'adresses $\mathbb{L}$. Nous ne considérons pas l'allocation dynamique de ressources, ces plages de mémoire sont donc connues tout au long de l'exécution.

L'ensemble de valeurs pouvant être prises par les variables est noté $\mathcal{V}$ et contient les adresses de base (c'est-à-dire que l'on peut stocker un l mais pas l'expression arithmétique de pointeur $l + i$), les entiers et les booléens ($\mathbb{B}$). Nous notons l'ensemble des variables locales $\mathcal{X}$. Les expressions sont définies de la manière suivante (où $\bar{e}$ dénote un tuple d'expressions) :

$$\begin{array}{lll} v & ::= & n \mid l \mid b \qquad n \in \mathbb{Z} \\ & & \qquad\qquad\quad l \in \mathbb{L} \\ & & \qquad\qquad\quad b \in \mathbb{B} \\ e & ::= & v \mid x \mid op(\bar{e}) \quad x \in \mathcal{X} \end{array}$$

$$\mathcal{V} = \mathbb{Z} \cup \mathbb{L} \cup \mathbb{B}$$

Ici, nous ne définissons pas l'ensemble des opérateurs, qui correspond à l'ensemble des opérateurs arithmétiques et booléens usuels. Il est néanmoins nécessaire de préciser que ces opérateurs n'autorisent pas l'arithmétique de pointeurs, comme mentionné au début de cette section. La seule opération sur les adresses est la comparaison. Les expressions ne produisent pas d'effets de bord.

Les programmes séquentiels sont définis comme un ensemble de méthodes et une instruction appelant la méthode principale du programme. Nous définissons une méthode par son nom, ses paramètres (variables locales) et la séquence d'instructions qui compose son corps :

$$\begin{array}{lll} \mathit{mth} & ::= & m(\mathit{list}\ x)\mathit{block} \qquad\qquad m \in \mathit{Name} \\ \mathit{main} & ::= & m(\mathit{list}\ e) \\ c & ::= & x := e \qquad\qquad \text{affectation locale} \\ & \mid & x[y] := e \qquad\qquad \text{écriture tas} \\ & \mid & x := y[e] \qquad\qquad \text{lecture tas} \\ & \mid & \textbf{while}\ e\ \textbf{do}\ \mathit{block} \\ & \mid & \textbf{if}\ e\ \textbf{then}\ \mathit{block}\ \textbf{else}\ \mathit{block} \\ & \mid & m(\mathit{list}\ e) \qquad\qquad \text{appel} \\ \mathit{block} & ::= & [] \mid c; \mathit{block} \end{array}$$

Le langage comprend les primitives habituelle d'un mini-langage impératif : séquence d'instructions, conditionnelles et boucles. L'affectation est séparée en

trois scénarios d'usage : affectation d'une locale à partir d'une expression, écriture dans le tas depuis une expression et lecture depuis le tas vers une locale. Les expressions ne peuvent pas contenir de lecture en mémoire, ces lectures doivent être effectuées vers des locales individuelles puis composées localement et finalement écrites en mémoire si nécessaire. Une méthode *mth* est définie par un identifiant m, la liste de ses paramètres et le bloc de code correspondant. Les méthodes peuvent être appelées par la syntaxe $m(l)$ où l est la liste des expressions transmises en argument, le passage de paramètres dans les appels de méthode est effectué par valeurs.

Pour les programmes parallèles, nous ajoutons une instruction supplémentaire : **atomic**(b) qui permet d'exécuter une séquence d'instructions de programme séquentiel atomiquement (mais pas de nouveau bloc atomique). Pendant une telle section de code, aucun autre fil d'exécution que celui ayant initié la séquence d'instructions correspondante ne peut s'exécuter.

Les simplifications apportées par rapport au sous-ensemble du langage C traité par notre greffon tiennent finalement dans le fait que nous ne considérons pas ici l'arithmétique de pointeurs et les expressions contenant de multiples lectures en mémoire, l'absence des structures de contrôles comme le `switch` et la simplification du typage. Les variables et la mémoire globale acceptent tout type de valeur, seules les expressions vérifient, à l'évaluation, la compatibilité des types de variables considérés.

Un programme séquentiel $prog_{seq}$ est défini par la liste des positions mémoires allouées avec leur taille, la liste des déclarations de méthodes, ainsi que le nom de sa méthode principale. Un programme parallèle $prog_{\|}$ est défini par la liste des positions mémoires allouées avec leur taille et une liste de noms de programmes principaux à exécuter.

Names est l'ensemble des noms de méthodes, qui contient deux noms réservés : *select* et "".

$$\begin{array}{lcl} memory & ::= & [(l_1, size_{l_1}) \dots (l_m, size_{l_m})] \\ prog_{seq} & ::= & \overline{mth}\ memory\ main \\ prog_{\|} & ::= & \overline{mth}\ memory\ [main_1 \dots main_n] \end{array}$$

Cela diffère de notre implémentation par le fait que dans le greffon les fils d'exécution sont supposés exécuter aléatoirement n'importe quelle suite de fonctions fournies par l'API vérifiée. Cette modélisation est plus précise, elle pourrait être implémentée dans le greffon, nous ré-aborderons ce point dans le chapitre 7.

Dans ce manuscrit, pour simplifier la présentation, nous considérons que si une méthode principale nécessite des paramètres, nous générons une nouvelle méthode auxiliaire qui ne prend pas de paramètre, et qui charge les paramètres voulus localement puis appelle la méthode principale voulue. Cela simplifie notamment l'initialisation de la fonction d'entrelacements, pour laquelle nous n'avons plus besoin de produire les environnements locaux de chaque fil d'exécution.

5.2.1 États de l'exécution et actions

L'environnement local ρ est une fonction partielle des variables locales vers les valeurs :

$$\rho \in P : \mathcal{X} \rightharpoonup \mathcal{V}$$

Le tas σ est une fonction partielle depuis les adresses, prenant un indice et renvoyant une valeur :

$$\sigma \in \Sigma : \mathbb{L} \rightharpoonup \mathbb{N} \rightharpoonup \mathcal{V}$$

Cette fonction est néanmoins, pour chaque adresse allouée, définie sur une zone contiguë d'une taille fixe et pas en dehors de cette taille :

$$\begin{array}{ll} \forall l \in \mathbb{L}. & \sigma(l) \text{ définie} \Rightarrow \\ & \exists size_l \in N. \\ & \quad \forall n < size_l. \quad \sigma(l)(n) \text{ défini} \\ & \quad \forall n >= size_l. \quad \sigma(l)(n) \text{ indéfini} \end{array}$$

Nous définissons le contexte local d'exécution comme :

$$\epsilon \in E : Names \times P \times \mathcal{C}$$

où *Names* est l'ensemble des noms de méthodes (dans ϵ, le nom *name* utilisé est celui de la méthode exécutée dans ce contexte) et $\mathcal{C}$ est l'ensemble des listes d'instructions (dans ϵ, l'élément b de cet ensemble est la liste d'instructions devant encore être exécutées, c'est donc un suffixe du corps de la méthode appelée). La pile d'appel est définie comme une pile de contextes locaux d'exécution :

$$s \in S \triangleq \bar{E}$$

Les états de programmes séquentiels et parallèles sont :

$$\gamma_{seq} \in \Gamma_{seq} : \Sigma \times S$$

$$\gamma_{\|} \in \Gamma_{\|} : \Sigma \times (\mathbb{T} \to S)$$

où $\mathbb{T}$ est un ensemble d'identifiants de fil d'exécution. Par la suite, on considère que $\mathbb{T}$ est un sous-ensemble fini de $\mathbb{Z}$, contigu et partant de 0. Un élément de $\mathbb{T}$ est donc une valeur valide pour nos langages. Nous nommons $stacks_{\gamma_{\|}}$ la fonction qui renvoie la pile d'exécution d'un fil d'exécution t de $\mathbb{T}$, et omettons $\gamma_{\|}$ lorsqu'il est évident dans le contexte.

État initial

Dans l'exécution d'un programme, la pile d'exécution initiale est de la forme :

$$[(\text{""}, \varnothing, [main()])]$$

c'est-à-dire une pile où le seul contexte d'exécution est celui d'une méthode nommée "", pour laquelle l'environnement local est vide, et dont la prochaine (et seule) action à effectuer est un appel à la méthode principale.

Si nous sommes dans un programme séquentiel, nous avons donc comme état initial :

$$(\sigma_{seq,init}, [(\text{""}, \varnothing, [main()])])$$

et dans un programme parallèle : $(\sigma_{\|,init}, stacks_{init})$ tel que :

$$\forall t \in \mathbb{T}. \quad stacks_{init}(t) = [(\text{""}, \varnothing, [main_t()])]$$

La définition d'un programme séquentiel, ou parallèle, comprend la liste *memory* des zones mémoires accessibles, avec leur taille. Pour tout couple, $(l, size_l)$ de *memory*, $\sigma_{init}(l)$ est définie et allouée avec une taille d'au moins $size_l$. Nous faisons l'hypothèse supplémentaire que le contenu de ces différentes zones de mémoire ne comprend initialement pas d'adresses.

État final et exécution sûre

Nous définissons l'état final d'un programme séquentiel par :

$$\exists \sigma. \quad \gamma_{seq,final} = (\sigma, [])$$

et l'état final d'un programme parallèle par :

$$\exists \sigma. \quad \gamma_{\|,final} = (\sigma, stacks_{final})$$

avec $\forall t \in \mathbb{T}. \quad stacks(t) = []$.

Nous définissons un état bloquant comme un état non final (atteint depuis un état initial), tel que la sémantique ne peut plus faire avancer l'exécution.

Nous appelons alors programme sûr, un programme qui depuis un état initial valide n'atteint jamais d'état bloquant. Nous pouvons noter qu'un programme qui s'exécute à l'infini est un programme sûr.

Actions

Les actions de base produites par nos programmes séquentiels sont de 5 types : les actions silencieuses τ, les lectures en mémoire `read` l n v, les écritures en mémoire `write` l n v et les appels et retours de méthode `call` m/`return` m.

$$a_{seq} \in \mathcal{A} ::= \tau \mid \texttt{call}\ m \mid \texttt{return}\ m \mid \texttt{read}\ l\ n\ v \mid \texttt{write}\ l\ n\ v$$

Pour les programmes parallèles, la notion de bloc atomique nous fait également ajouter la notion de liste d'action aux types d'actions possibles. Ainsi, dans le monde parallèle, le type action est défini comme suit :

$$a_{\|} \in \mathcal{A} ::= \tau \mid \texttt{call}\ m \mid \texttt{return}\ m \mid \texttt{read}\ l\ n\ v \mid \texttt{write}\ l\ n\ v \mid \texttt{atomic}\ (list\ a_{seq})$$

Les traces d'exécutions sont des listes d'actions pour les programmes séquentiels, et des listes de couples action/identifiant de fil d'exécution pour les programmes parallèles.

5.2.2 Sémantiques de programmes

La sémantique opérationnelle des programmes séquentiels est définie comme présenté par la figure 5.1. On peut noter que dans un programme séquentiel, nous ne donnons pas de sens aux blocs atomiques qui ne sont donc considérés que dans les programmes parallèles.

Un jugement de la sémantique séquentielle est de la forme :

$$\mathcal{M} \vdash \sigma, \bar{s} \xrightarrow{a_{seq}} \sigma', \bar{s}'$$

et exprime que l'on atteint un nouvel état $\sigma', \bar{s}'$ depuis un état $\sigma, \bar{s}$ avec une certaine action a_{seq}. La variable $\mathcal{M}$ représente les noms de méthodes du programme.

Nous utilisons la notation $l_1 +\!\!+ \; l_2$ pour signifier la concaténation de deux listes l_1 et l_2, $elem :: l$ pour placer $elem$ en tête de l et $|l|$ la longueur de la liste l. La notation $\epsilon \cdot \bar{s}$ indique que ϵ est le contexte se trouvant au sommet de la pile dont les autres éléments sont représentés par $\bar{s}$.

La notation $[\![e]\!]_\rho$ correspond à l'évaluation de l'expression e dans l'environnement local ρ. Ainsi dans une expression l'évaluation de la valeur d'une variable x, $[\![x]\!]_\rho$, est la valeur v telle que $\rho(x) = v$.

La notation $f[a \mapsto i]$ est la fonction f' telle que pour tout antécédent a' différent de a, $f'(a') = f(a')$ et $f'(a) = i$. Ainsi $\rho[x \mapsto v]$ dénote un changement dans l'environnement local tandis que $\sigma[(l, o) \mapsto v]$ dénote un changement dans le tas.

Dans la règle d'appel de méthode, la liste $\overline{arg}$ d'arguments fournis doit avoir la même longueur que la liste de paramètres $\bar{x}$. La liste de ces arguments est évaluée en une liste $\bar{v}$ dont le i^{eme} élément est la valeur du i^{eme} argument de $\overline{arg}$. Finalement, le nouveau contexte d'exécution créé reçoit un environnement local ρ_{m_2} tel qu'à chaque paramètre on associe la valeur de l'argument correspondant, noté $\bar{x} \mapsto \bar{v}$.

Nous définissons également la sémantique particulière de l'appel $select(ptid, pct, ntid)$. Contrairement aux autres règles de la sémantique des programmes séquentiels, cette règle n'est pas déterministe. Elle sélectionne une valeur t aléatoire entre 0 et $ntid$, telle que la valeur à l'adresse pct pour l'indice t est différente de 0. Cette fonctionnalité nous servira pour la définition des programmes simulant. Nous supposons qu'un programme parallèle ne peut pas appeler cette méthode.

La figure 5.2 présente la sémantique des programmes parallèles. Un fil d'exécution t est sélectionné. Si la première instruction de t n'est pas un bloc atomique, nous réduisons l'état à l'aide de la sémantique des programmes séquentiels en associant σ au contexte du fil t. L'action générée est associée à l'identifiant du fil pour créer une action de programme parallèle.

Les jugements sont de la forme :

$$\mathcal{M} \vdash \sigma_{\|}, stacks \xrightarrow{(t, a_{\|})} \sigma'_{\|}, stack'$$

reliant donc deux états parallèles pour une action d'un fil d'exécution t.

Si la première instruction est un bloc atomique, la sémantique séquentielle

$\mathcal{M} \vdash \sigma, ((m, \rho, (x := e; b)) \cdot \bar{s}) \xrightarrow{\tau} \sigma, ((m, \rho[x \mapsto v], b) \cdot \bar{s})$
[assign] si $[\![e]\!]_\rho = v$

$\mathcal{M} \vdash \sigma, ((m, \rho, (x[e_o] := e_v; b)) \cdot \bar{s}) \xrightarrow{\texttt{write } l\ o\ v} \sigma[(l, o) \mapsto v], ((m, \rho, b) \cdot \bar{s})$
[write] si $[\![e_v]\!]_\rho = v$, $[\![e_o]\!]_\rho = o$, $\rho(x) = l$, $o < size(l)$

$\mathcal{M} \vdash \sigma, ((m, \rho, (x := y[e_o]; b)) \cdot \bar{s}) \xrightarrow{\texttt{read } l\ o\ v} \sigma, ((m, \rho[x \mapsto v], b) \cdot \bar{s})$
[read] si $[\![e_o]\!]_\rho = o$, $\rho(y) = l$, $o < size(l)$, $\sigma(l, o) = v$

$\mathcal{M} \vdash \sigma, ((m, \rho, (\textbf{while } e \textbf{ do } b_l; b)) \cdot \bar{s}) \xrightarrow{\tau} \sigma, ((m, \rho, (b_l +\!\!+ \textbf{while } e \textbf{ do } b_l; b)) \cdot \bar{s})$
[while:true] si $[\![e]\!]_\rho = \texttt{true}$,

$\mathcal{M} \vdash \sigma, ((m, \rho, (\textbf{while } e \textbf{ do } b_l; b) \cdot \bar{s}) \xrightarrow{\tau} \sigma, ((m, \rho, b \cdot \bar{s})$
[while:false] si $[\![e]\!]_\rho = \texttt{false}$,

$\mathcal{M} \vdash \sigma, ((m, \rho, (\textbf{if } e \textbf{ then } b_\top \textbf{ else } b_\bot); b) \cdot \bar{s}) \xrightarrow{\tau} \sigma, ((m, \rho, (b_\top +\!\!+ b)) \cdot \bar{s})$
[if:true] si $[\![e]\!]_\rho = \texttt{true}$,

$\mathcal{M} \vdash \sigma, ((m, \rho, (\textbf{if } e \textbf{ then } b_\top \textbf{ else } b_\bot); b) \cdot \bar{s}) \xrightarrow{\tau} \sigma, ((m, \rho, (b_\bot +\!\!+ b)) \cdot \bar{s})$
[if:false] si $[\![e]\!]_\rho = \texttt{false}$,

$\mathcal{M} \vdash \sigma, ((m_1, \rho, (m_2(\overline{arg}); b_{m_1})) \cdot \bar{s}) \xrightarrow{\texttt{call } m_2} \sigma, ((m_2, \rho', b_{m_2}) \cdot (m_1, \rho, b_{m_1}) \cdot \bar{s})$
[call] si $m_2(\bar{x})b_{m_2} \in \mathcal{M}$, $|\overline{arg}| = |\bar{x}|$, $[\![\overline{arg}]\!]_\rho = \bar{v}$,
$\rho' = \bar{x} \mapsto \bar{v}$

$\mathcal{M} \vdash \sigma, ((m, \rho, []) \cdot \bar{s}) \xrightarrow{\texttt{return } m} \sigma, \bar{s}$
[return]

$\mathcal{M} \vdash \sigma, ((m, \rho, select(ptid, pct, ntid); b) \cdot \bar{s}) \xrightarrow{\texttt{write } ptid\ 0\ t} \sigma[(ptid, 0) \mapsto t], ((m, \rho, b) \cdot \bar{s})$
[select] si $0 \leq t < ntid$, $\sigma(pct, t) \neq 0$

FIGURE 5.1 – *Sémantique opérationnelle des programmes séquentiels*

$$\mathcal{M} \vdash \sigma_{\|}, stacks \xrightarrow{(t,a)} \sigma'_{\|}, stack[t \mapsto s']$$
$$\text{si } \mathcal{M} \vdash (\sigma_{\|}, stacks(t)) \xrightarrow{a} (\sigma'_{\|}, s')$$

$$\mathcal{M} \vdash \sigma_{\|}, stacks \xrightarrow{(t,\texttt{atomic}(list))} \sigma'_{\|}, stacks[t \mapsto s']$$
$$\text{avec}$$
$$stacks(t) = (m, \rho, \textbf{atomic}\ (b_{atomic}); b) \cdot \bar{\epsilon}$$
$$s' = (m, \rho', b) \cdot \bar{\epsilon}$$
$$\text{si}$$
$$\mathcal{M} \vdash (\sigma_{\|}, [(m, \rho, b_{atomic})]) \xrightarrow{list}{}^{*} (\sigma'_{\|}, [(m, \rho', [])])$$

FIGURE 5.2 – *Sémantique opérationnelle des programmes parallèles*

ne peut pas en faire la réduction (n'ayant pas de règle pour les blocs atomiques). Nous réduisons alors la totalité des instructions du bloc (une nouvelle fois en lui associant σ), ce que nous dénotons par la fermeture transitive $\xrightarrow{list}{}^{*}$ de $\xrightarrow{a_{seq}}$, où $list$ est la suite d'actions a_{seq} générée. Celle-ci est associée à t pour créer l'action de programme parallèle correspondante. Comme nous réduisons la totalité de la liste d'instruction, celle-ci est bien atomique du point de vue des actions effectuées, nous pouvons noter que nous interdisons par cette règle de créer des blocs atomiques imbriqués : un tel bloc empêcherait la réduction puisque nous ne revenons pas dans la sémantique parallèle avant d'avoir entièrement réduit la liste des instructions.

5.3 TRANSFORMATION

Nous présentons ici la traduction de la fonction de transformation pour le langage défini dans la section précédente. Il existe quelques différences notamment dues au fait que nous obligeons toute lecture en mémoire à passer par une variable locale et au fait que nous autorisons ici les appels de méthode tant qu'ils ne sont pas récursifs.

Pour chaque variable locale x possiblement allouée par une méthode dans son contexte d'exécution, nous créons une zone de mémoire simulante dont l'adresse est $\&x$, et qui est définie pour tout indice correspondant à un identifiant de fil d'exécution valide. Les adresses générées sont disjointes deux à deux et disjointes des adresses présentes dans le tas du programme parallèle d'origine. Chacune de ces zones de mémoire contient, pour chaque fil d'exécution, la valeur actuelle de la variable locale qui lui correspond du point de vue de la simulation.

Pour chaque instruction, il faut créer une méthode qui la simule. Celle-ci reçoit en paramètre l'identifiant du fil que l'on simule pour cette étape d'exécution dans une variable locale nommée `tid`. Nous définissons la fonction *load* qui, à une variable locale x du programme d'origine, associe la séquence d'instructions de simulation qui chargent la valeur correspondante depuis la zone mémoire de simulation, pour l'identifiant de fil d'exécution en entrée de méthode simulante. Dans les instructions générées, nous réutilisons le nom original de la variable locale pour créer une nouvelle variable qui sera locale à la méthode de simulation :

$load(x, \texttt{tid}) \equiv \texttt{ptr} := \&x; x := \texttt{ptr}[\texttt{tid}];$

Dans cette fonction, nous utilisons une variable locale `ptr`. Nous supposons que cette variable n'est pas déjà utilisée par la méthode de simulation. Cette variable est appelée `ptr` pour rappeler qu'elle ne sert qu'à stocker des adresses (de simulation).

La fonction *variables* nous donne l'ensemble des variables de son argument qui peut être une expression, une instruction, un bloc d'instructions ou un programme. Nous nommons *loads* la fonction qui produit la liste des instructions permettant de charger un ensemble de variables locales depuis leurs adresses simulantes, en itérant *load* sur toutes ces variables. Par exemple, pour avoir tous les chargements de variables locales correspondant à une expression e, nous écrivons :

$loads(variables(e), \texttt{tid})$

Nous définissons globalement, dans le tas du programme simulant :

— `ptid`, une adresse vers une zone mémoire de taille 1, servant à stocker l'identifiant de fil d'exécution en cours ;

— `pct`, une adresse dont la zone mémoire associe à chaque identifiant de fil d'exécution son compteur de point de programme ;

— des adresses pour chaque méthode m, dont la zone mémoire associe à chaque identifiant de fil d'exécution le point de retour d'appel depuis la méthode m.

Pour ce dernier point, nous définissons la fonction $from$ qui pour une méthode m renvoie la position mémoire en question. Par exemple, si m a été appelée par m_2, pendant l'instruction c pour le fil d'exécution t, $from(m)[t]$ contiendra l'identifiant de l'instruction qui suit c selon le graphe de flot de contrôle.

Dans la suite, nous considérons que le graphe de flot de contrôle est préalablement calculé à partir du programme d'origine et étiquette chaque instruction avec un identifiant unique $\ell \in Labels$. Chaque instruction est également étiquet-

tée avec l'identifiant de l'instruction qui la suit ℓ_{next} dans l'ordre des instructions du programme. Nous noterons par exemple que pour une liste d'instructions :

$$(\textbf{if } e \textbf{ then } b_{true} \textbf{ else } b_{false}); b$$

l'instruction qui suit le **if/else** est la première instruction de b et pas les instructions des blocs internes. Par la suite, lorsque ces informations nous sont nécessaires, nous noterons l'instruction concernée $c^{\ell}_{\ell_{next}}$.

S'il n'y a pas d'instruction suivante dans la liste considérée, soit l'instruction est dans le bloc d'un branchement conditionnel, soit c'est la dernière instruction de la méthode m où elle se trouve. Dans le premier cas, si nous sommes dans le bloc d'une boucle, ℓ_{next} correspond à l'identifiant de l'instruction de la boucle elle-même, si nous sommes dans une branche d'une instruction **if/else** cela correspond à l'instruction qui suit cette conditionnelle. Si l'instruction est la dernière de la méthode m, la valeur de ℓ_{next} que nous fournissons est un identifiant spécial $\ell_{end(m)}$ qui indique que nous devons effectuer un retour d'appel de méthode.

En effet, le graphe de flot de contrôle associe également des étiquettes à chaque méthode : l'identifiant de son instruction de début, et celui correspondant à son action de retour. Ce n'est donc pas l'identifiant d'une instruction du programme source, mais marque simplement la fin, pour laquelle un code de simulation sera associé. Lorsque nous avons besoin de ces informations à propos d'une méthode identifiée m, nous notons $m^{\ell}_{\ell_{end}}$. Si le bloc de la méthode est vide, $\ell = \ell_{end}$.

Dans la suite, nous supposons que la sémantique parallèle travaille sur des blocs d'instructions annotés par des étiquettes.

5.3.1 Affectation locale et partagée

L'affectation `x := e` identifiée ℓ est simulée par la méthode de simulation `assign_`ℓ`_simulation`, tel que présenté par la figure 5.3. Les premières instructions de la simulation sont générées à l'aide de la fonction *loads* présentée précédemment afin de charger la valeur des variables locales utilisées dans e pour le fil d'exécution simulé. On écrit ensuite la position mémoire `&x` qui simule `x`, à l'indice `tid`, avec l'expression `e` pour laquelle les variables locales ont été chargées par les instructions générées par *loads*. Finalement, on place le compteur de programme sur l'instruction ℓ_{next} suivante pour le fil d'exécution `tid`.

```
1 trans_assign(x,e,ℓ,ℓ_next) ≡
2   assign_ℓ_simulation (tid) [
3     loads(variables(e),tid) ;
4     ptr := &x ;
5     ptr[tid] := e ;
6     ptr := pct ;
7     ptr[tid] := ℓ_next ;
8   ]
```

FIGURE 5.3 – *Simulation de l'affectation*

```
1 trans_read(x,p,o,ℓ,ℓ_next) ≡
2   read_ℓ_simulation (tid) [
3     loads(variables(o),tid) ;
4     load(p,tid) ;
5     x := p[o] ;
6     ptr := &x ;
7     ptr[tid] := x ;
8     ptr := pct ;
9     ptr[tid] := ℓ_next ;
10  ]
```

FIGURE 5.4 – *Simulation de la lecture en mémoire*

La lecture en mémoire `x := p[o]` identifiée ℓ est simulée par la méthode de simulation `read_ℓ_simulation`, tel que présenté dans la figure 5.4. De la même manière que dans la transformation de l'affectation locale, nous chargeons d'abord les variables de `o`, ainsi que la variable locale correspondant au pointeur `p`. Nous reproduisons ensuite le chargement depuis la mémoire globale dans `x`. Ensuite, nous plaçons la valeur lue dans la zone mémoire `&x` de simulation pour le fil d'exécution `tid`. Finalement, on place le compteur de programme sur l'instruction suivante.

Nous simulons l'écriture en mémoire `p[o] := e` identifiée ℓ par la méthode de simulation `write_ℓ_simulation`, tel que présenté dans la figure 5.5. Nous chargeons d'abord les variables de `o` et `e`. Si des variables sont présentes dans les deux expressions, cela ne pose pas de problèmes d'un point de vue exécution : une telle variable sera simplement lue deux fois, la seconde lecture écrasant la valeur précédente avec la même valeur (celle-ci n'ayant pu changer entre les deux lectures). Nous chargeons le pointeur `p` et effectuons ensuite l'écriture en mémoire. Finalement, nous plaçons le compteur de programme à l'instruction suivante.

```
1  trans_write(p,o,e,ℓ,ℓ_next) ≡
2    write_ℓ_simulation (tid) [
3      loads(variables(e),tid) ;
4      loads(variables(o),tid) ;
5      load(p,tid) ;
6      p[o] := e ;
7      ptr := pct ;
8      ptr[tid] := ℓ_next ;
9    ]
```

FIGURE 5.5 – *Simulation de l'écriture en mémoire*

```
1  trans_cond(e,b1,b2,ℓ,ℓ_next) ≡
2    cond_ℓ_simulation (tid) [
3      loads(variables(e),tid) ;
4      ptr := pct ;
5      if e then
6        b1 = [] →
7          ptr[tid] := ℓ_next ;
8        b1 = c^{ℓ'}_{ℓ'_next} :: _ →
9          ptr[tid] := ℓ' ;
10     else
11       b2 = [] →
12         ptr[tid] := ℓ_next ;
13       b2 = c^{ℓ'}_{ℓ'_next} :: _ →
14         ptr[tid] := ℓ' ;
15   ]
```

FIGURE 5.6 – *Simulation de la conditionnelle*

5.3.2 Sauts conditionnels

On simule l'instruction conditionnelle **`if`** `e` **`then`** `b1` **`else`** `b2` par la méthode de simulation `cond_ℓ_simulation` tel que présenté dans la figure 5.6. Les variables de `e` sont chargées. Nous construisons ensuite un nouveau bloc conditionnel dont les branches sont remplacées par le placement du compteur de programme sur la prochaine instruction à exécuter selon la branche empruntée. Si celle-ci est vide, ce point est celui de l'instruction qui suit la conditionnelle (ℓ_{next}).

On simule une boucle **`while`** e **`do`** b par une méthode `loop_ℓ_simulation` tel qu'illustré par la figure 5.7. Si le bloc est vide, la boucle est nécessairement infinie si l'expression est évaluée vraie : les expressions ne peuvent pas contenir de lecture en mémoire globale, la condition sera donc toujours évaluée vraie.

```
1  trans_loop(e,b,ℓ,ℓ_next) ≡
2    loop_ℓ_simulation (tid) [
3      loads(variables(e),tid) ;
4      ptr := pct ;
5      if e then
6        b = [] → //infinite loop
7          ptr[tid] := ℓ ;
8        b = c^ℓ'_ℓ'_next :: _ →
9          ptr[tid] := ℓ' ;
10     else
11       ptr[tid] := ℓ_next ;
12 ]
```

FIGURE 5.7 – *Simulation de la boucle*

La première branche de la conditionnelle place le compteur de programme sur l'instruction de la boucle **`while`** elle-même, si le bloc est vide, sur la première instruction du bloc sinon. La branche correspondant à l'évaluation à FAUX place le compteur de programme sur l'instruction qui suit la boucle.

5.3.3 Appel de méthode et retour

Un appel de méthode `mth(l)` est simulé par le code présenté dans la figure 5.8. Nous rappelons que chaque méthode se voit attribuée une adresse dans le tas qui associe, pour chaque identifiant de fil d'exécution, le point de programme où l'exécution doit continuer quand l'exécution de la méthode termine. Cette adresse nous est retournée par la fonction $from$.

Dans un premier temps, on charge l'ensemble des variables utilisées dans la liste de paramètres passés à la méthode `mth`. Nous définissons ensuite la fonction *combine* qui va, pour chaque variable locale de *mth*, écrire à son adresse de simulation pour le fil d'exécution en cours, l'expression de `l` qui lui correspond. Nous pouvons définir la fonction *combine* de la façon suivante :

$$
\begin{array}{ll}
combine(as, es, \texttt{tid}) \equiv & \\
as, es = [], [] & \rightarrow [] \\
as, es = a :: as', e :: es' & \rightarrow [ptr := \&a; ptr[\texttt{tid}] := e] +\!\!+ combine(as', es', \texttt{tid})
\end{array}
$$

On écrit ensuite dans `from(mth)`, à l'indice `tid`, l'identifiant de la prochaine instruction à exécuter au retour de l'appel. Finalement, on place le compteur

```
1  trans_call(mth, ℓ_mth, l, ℓ, ℓ_next) ≡
2    call_ℓ_m_simulation (tid) [
3      loads(variables(l),tid) ;
4      combine(args(mth),l,tid) ;
5
6      ptr := from(mth) ;
7      ptr[tid] := ℓ_next ;
8
9      ptr := pct;
10     ptr[tid] := ℓ_mth ;
11   ]
```

FIGURE 5.8 – *Simulation d'appel*

```
1  trans_return(m, ℓ_end) ≡
2    return_ℓ_end_m_simulation (tid) [
3      ptr := from(m);
4      aux := ptr[tid];
5      ptr := pct;
6      ptr[tid] := aux ;
7    ]
```

FIGURE 5.9 – *Simulation de retour d'appel*

de programme sur la première instruction du bloc de la méthode `mth` (nous rappelons que si ce bloc est vide, $\ell = \ell_{end}$.

Lorsque nous traitons les méthodes d'origine, nous ajoutons également une simulation pour le retour d'appel qui est identifié ℓ_{end}. La méthode simulant un retour d'appel d'une méthode m est définie telle que présenté par la figure 5.9.

Nous chargeons, depuis `from` (écrite lors de la simulation de l'appel de `m`), le numéro de l'instruction où nous devons revenir et nous plaçons ce numéro dans le compteur de programme. La variable `aux` est créée de manière à être différente de `ptr` et `tid`.

Chaque fil d'exécution reçoit sa méthode *main*. Chacune de celles-ci donne aussi lieu à la création des méthodes de simulation pour ses instructions. Un programme parallèle est donc une liste d'appels de méthode *main*. Le point de retour de chacune d'elle est identifié 0, dont nous réservons l'identifiant et qui correspond à l'action « ne pas effectuer d'action ». Il faut également s'assurer qu'initialement, chaque point de programme est bien fixé à l'identifiant de la simulation de l'appel de son *main*.

5.3.4 Section atomique

Pour produire la simulation d'un bloc atomique. Nous pouvons parcourir récursivement les instructions à exécuter. Les affectations sont remplacées par les codes de simulation précédemment définis (on peut en enlever l'écriture du compteur de programme mais la laisser ne nuit pas à la sémantique du programme). La lecture des conditions des instructions conditionnelles et des boucles est effectuée de la même manière que dans les codes simulants précédents. En revanche, les blocs contiennent récursivement les instructions de simulation des instructions qui y sont présentes et pas un changement de compteur de programme. Les variables locales intervenant dans une condition de boucle doivent être chargée à nouveau à la fin de l'exécution de son bloc.

Pour simuler les appels de méthode depuis un bloc atomique, nous pouvons simplement *inliner* la simulation de la méthode. Par contre il est bien nécessaire de construire le contexte d'exécution. Donc simuler un appel dans un bloc atomique consiste à construire le contexte (comme nous le faisons dans la simulation d'un appel), puis insérer la simulation de chacune des instructions qui compose la méthode. Nous ajoutons également la simulation du retour d'appel dans le but de simplifier l'équivalence de traces que nous présenterons dans la section 5.4. Une nouvelle fois nous n'autorisons pas les appels récursifs.

Finalement, nous pouvons placer le compteur de programme sur l'instruction qui suit le bloc atomique.

La figure 5.10 présente informellement une fonction de transformation pour les blocs atomiques. Nous supposons que nous avons des fonctions de transformation pour les affectations, lectures et écritures qui ne renvoient que le code de la méthode de simulation, sans le changement de compteur de programme et pas la méthode elle-même (nous notons cette variante `t_code` pour la transformation `t`). Les appels récursifs étant interdits, la fonction *trans_atomic_list* termine.

5.3.5 Transformation d'une instruction

Nous illustrons dans la figure 5.11, la fonction qui, à partir d'une instruction, appelle la bonne fonction de transformation conformément à l'instruction considérée.

```
1  trans_atomic_list(l,tid) ≡
2    l = [] → []
3    l = i :: l' →
4        sim :=
5                i = x := e      → trans_assign_code(x,e,tid)
6                i = x := p[o]   → trans_read_code(x,p,o,tid)
7                i = p[o] := e   → trans_write_code(p,o,e,tid)
8                i = if e then b1 else b2 →
9                  loads(variables(e),tid) ;
10                 if e then trans_atomic_list(b1,tid)
11                 else      trans_atomic_list(b2,tid)
12               i = while e do b →
13                 loads(variables(e),tid) ;
14                 while e do [
15                   trans_atomic_list(b,tid) ;
16                   loads(variables(e),tid)
17                 ]
18               i = m(ps) ℓ' / ℓ_next →
19                 [
20                   trans_call_code(m, ps, tid) ;
21                   trans_atomic_list(m.(body),tid) ;
22                   return_ℓ_end(m)_simulation(tid)
23                 ]
24        sim ++ trans_atomic_list(l',tid)
25
26 trans_atomic(l,ℓ,ℓ_next) ≡
27   atomic_ℓ_simulation (tid)
28     trans_atomic_list(l,tid)
29     ++ [
30       ptr := pct ;
31       ptr[tid] := ℓ_next
32     ]
```

FIGURE 5.10 – *Simulation d'un bloc atomique*

```
1  trans_instruction(i, ℓ, ℓ_next, tid) ≡
2   i = x := e      →  trans_assign(x, e, ℓ, ℓ_next)
3   i = x := p[o]   →  trans_read(x, p, o, ℓ, ℓ_next)
4   i = p[o] := e   →  trans_write(p, o, e, ℓ, ℓ_next)
5   i = if e then b1 else b2 →
6                      trans_cond(e, b1, b2, ℓ, ℓ_next)
7   i = while e do b →
8                      trans_loop(e, b, ℓ, ℓ_next)
9   i = atomic b    →  trans_atomic(b, ℓ, ℓ_next)
10  i = m^ℓ'(ps)    →  trans_call(m, ℓ', ps, ℓ, ℓ_next)
```

FIGURE 5.11 – *Simulation d'une instruction*

5.3.6 Entrelacements

La méthode générant les entrelacements est construite comme présentée par la figure 5.12. Dans ce code, pour faciliter la lecture, nous utilisons une instruction `switch` qui n'existe pas dans le langage. Dans la formalisation réelle, nous générons une imbrication de conditionnelles.

L'initialisation place chaque compteur de programme sur l'identifiant de son appel de méthode principale. Nous considérons, conformément à notre définition d'un état initial, que cet appel est effectué depuis une méthode `""`, dont l'appel en question est la seule instruction, et dont la valeur ℓ_{end} est 0. Nous initialisons la variable locale `terminated` qui indique s'il ne reste aucun fil d'exécution à simuler à FAUX. Nous le faisons en supposant qu'au moins un fil d'exécution a une méthode principale à exécuter. Dans le cas contraire, nous initialiserions `terminated` à VRAI. Par la suite, nous appellerons ce bloc de code b_{init}.

Ensuite, la boucle d'entrelacement choisit à chaque tour, un fil d'exécution tel que son compteur de point de programme est différent de 0. Ceci est réalisé à l'aide de la méthode `select` qui place l'identifiant de fil choisi à la position mémoire `ptid`, nous appelons cette instruction i_{select} par la suite.

La valeur à la position `ptid` est ensuite lue dans une variable locale `tid`, que l'on utilise pour récupérer le compteur de programme dans la variable `stmt`. Cet identifiant est ensuite comparé avec chaque identifiant de méthode de simulation afin de déterminer laquelle doit être appelée, et l'appelle. Nous nommons cette suite d'instructions b_{sim}.

Finalement, la condition de terminaison doit être réévaluée, ce que nous faisons avec la séquence d'instructions $b_{termination}$.

```
1  build_interleavings(M, main_calls) ≡
2  interleavings = [
3    // b_init
4    ptr := pct ;
5    ∀t ∈ 𝕋, ptr[t] := main_calls(t).ℓ ;
6
7    ptr := from("") ;
8    ∀t ∈ 𝕋, ptr[t] := 0 ;
9
10   terminated := false ;
11
12   while ¬terminated do [
13     // i_select
14     select (ptid, pct, ntid) ;
15
16     // b_sim
17     ptr  := ptid;
18     tid  := ptr[0];
19     ptr  := pct;
20     stmt := ptr[tid];
21
22     switch stmt is [
23     ∀m ∈ M, ∀c^ℓ_{ℓ_next} ∈ m,
24     ℓ : [ c_{stmt_type}_ℓ_simulation [tid] ]
25     ]
26
27     // b_termination
28     terminated := true;
29     t := 0;
30     while t < ntid do [
31       if pct[t]≠0
32       then [ terminated := false ]
33       else [ ] ;
34       t := t + 1 ;
35     ]
36   ]
37 ]
```

FIGURE 5.12 – *Boucle d'entrelacements*

5.3.7 Définition du programme simulant

Pour un programme parallèle :

$$prog_{\|} = mem_{\|}\ \overline{mth}_{\|}\ [main_1 \dots main_{\texttt{ntid}}]$$

le programme simulant généré est de la forme :

$$prog_{sim} = mem_{sim}\ \overline{mth}_{sim}\ \texttt{interleavings}$$

La liste $\overline{mth}_{sim}$ contient les méthodes simulantes de chaque instruction et retour de méthode du programme d'origine, ainsi que la méthode `interleavings`, qui est la méthode principale du programme simulant. La liste des adresses est de la forme $mem_{sim} = mem_{sim,\mathcal{X}} +\!\!+ mem_{\|}$. La liste $mem_{sim,\mathcal{X}}$ contient les couples

— $(\texttt{ptid}, 1)$ utilisé par `select` ;
— $(\texttt{pct}, \texttt{ntid})$, le compteur de programme pour chaque fil ;
— pour chaque méthode m du programme d'origine, un couple $(from(m), \texttt{ntid})$ pour enregistrer son point de retour ;
— pour chaque variable locale x du programme d'origine, un couple $(\&x, \texttt{ntid})$, modélisant sa mémoire simulante.

5.4 Équivalence des exécutions

5.4.1 Équivalence d'états et de traces

Nous appelons γ_{sim} l'état de programme séquentiel (σ_{sim}, s_{sim}) de la simulation, d'un programme parallèle sûr dans un état $\gamma_{\|}$. Dans σ_{sim}, nous distinguons deux parties disjointes $\sigma_{sim,\|}$, la réplique du tas du programme d'origine et $\sigma_{sim,\mathcal{X}}$ les adresses simulant les variables locales du programme d'origine. Dans cette dernière, nous incluons également les adresses du compteur de programme `pct`, l'adresse de sélection de fil `ptid`, et les adresses des points de retour des méthodes $from(m)$. Pour toutes ces adresses de simulation, la zone mémoire initialement allouée doit être de taille supérieure ou égale à l'identifiant maximal de fil d'exécution défini par le programme d'origine.

Pour sélectionner la partie de $\sigma_{sim,\mathcal{X}}$ qui simule le fil d'exécution t, nous utilisons la syntaxe $\sigma_{sim,\mathcal{X}}[t]$. Elle applique partiellement la fonction définie par σ_{sim} en la restreignant aux indices t. Ainsi, la fonction $\sigma_{sim,\mathcal{X}}[t](l)$ est alors $\sigma_{sim}(l, t)$.

Nous définissons l'équivalence des états comme suit :

$$
\begin{array}{ll}
(1) & \sigma_{\|} = \sigma_{sim,\|} \\
(2) & \forall t \in \mathbb{T}, \rho \in stacks(t), x \in \mathcal{X}. \quad \rho(x) = v \Rightarrow \sigma_{sim,\mathcal{X}}[t](\&x) = v \\
(3.a) & \forall t \in \mathbb{T}, \epsilon \in E. \quad stacks(t) = \epsilon \cdot \_ \Leftrightarrow \sigma_{sim,\mathcal{X}}[t](\texttt{pct}) = next(\epsilon) \\
(3.b) & \forall t \in \mathbb{T}. \quad stacks(t) = [] \Leftrightarrow \sigma_{sim,\mathcal{X}}[t](\texttt{pct}) = 0 \\
(4) & \forall t \in \mathbb{T}. \quad wf\_stack(stacks(t), \sigma_{sim,\mathcal{X}}[t]) \\
(5) & s_{sim} = (\texttt{interleavings}, \rho_{sim}, [\textbf{while } \neg\texttt{terminated } \textbf{do } b_{body}]) \cdot (\text{""}, \rho_{\varnothing}, []) \\
 & \wedge\ b_{body} = i_{select} :: b_{sim} +\!\!+ b_{termination} \\
 & \wedge\ \rho_{sim}(\texttt{terminated}) = true \Leftrightarrow \forall t \in \mathbb{T}. \quad \sigma_{sim,\mathcal{X}}[t](\texttt{pct}) = 0
\end{array}
$$

$$\gamma_{\|} \sim \gamma_{sim}$$

avec :

$$
next(\epsilon) \equiv \begin{array}{ll} \epsilon = (\_, \_, c^{\ell}_{\ell_{next}} :: \_) & \rightarrow \ell \\ \epsilon = (m^{\ell}_{\ell_{end}}, \_, []) & \rightarrow \ell_{end} \end{array}
$$

et

$$
\frac{}{wf\_stack([], \sigma_{sim,\mathcal{X}}[t])} \qquad \frac{\sigma_{sim,\mathcal{X}}[t](from(m)) = 0}{wf\_stack((m, \_, \_) \cdot [], \sigma_{sim,\mathcal{X}}[t])}
$$

$$
\frac{\sigma_{sim,\mathcal{X}}[t](from(m)) = next(\epsilon) \qquad wf\_stack(\epsilon \cdot s', \sigma_{sim,\mathcal{X}}[t])}{wf\_stack((m, \_, \_) \cdot \epsilon \cdot s', \sigma_{sim,\mathcal{X}}[t])}
$$

La condition 1 est que la partie du tas de simulation qui représente le tas d'origine soit exactement égale à celui-ci.

Ensuite (2), pour chacune des variables locales x du programme d'origine, la valeur présente à son adresse de simulation $\&x$ pour l'indice t correspondant au fil d'exécution considéré doit être égale à la valeur dans la pile d'origine. Pour simplifier l'écriture, nous notons $\rho \in stacks(t)$ pour parler des ρ de chaque ϵ de $stacks(t)$. Ce n'est qu'une simple implication car la simulation des variables n'est pas réinitialisée lors de la simulation du retour d'appel. Comme nous supposons un programme d'entrée sûr, un nouvel appel à la fonction ne fait pas d'accès en lecture à une variable locale avant de l'avoir initialisée. Donc cela ne pose pas de problème pour l'équivalence.

Nos compteurs de programmes doivent être corrects. Nous nous appuyons sur la fonction *next* qui, pour un contexte local d'exécution ϵ donné, renvoie l'identifiant de l'instruction suivante à exécuter ℓ de m, ou ℓ_{end} de m, s'il n'y a pas d'autre instruction. Dans (3.*a*) pour chaque fil, si sa pile d'exécution n'est pas vide, le compteur de programme de ce fil $\sigma_{sim,\mathcal{X}}[t](\texttt{pct})$ doit être l'identifiant

fourni par *next*. Si la pile d'exécution du fil est vide (3.*b*), le compteur doit valoir 0.

La pile d'exécution doit être correctement modélisée (4). Pour cela, nous définissons un prédicat récursif *wf_stack* qui, pour une pile d'exécution s donnée (d'un fil d'exécution t), et la simulation des données locales de t, $\sigma_{sim,\mathcal{X}}[t]$, vérifie que la modélisation offerte par $from$ est bien formée. Si la pile est vide, il n'y a pas d'informations à associer à propos de $from$ car il n'y a plus d'action de retour d'appel qui puisse être effectuée, du fait que dans ce cas 3.*b* assure que le compteur de programme est 0, qui interdit toute action pour le fil. S'il y a un unique contexte, le dernier retour d'appel doit nous permettre de ramener le compteur de point de programme à la valeur 0 pour arrêter l'exécution. Enfin, s'il y a plus d'un contexte, le contexte ϵ au sommet doit en cas de retour d'appel, revenir à la prochaine instruction de ϵ', le contexte suivant dans la pile, et le reste de la pile doit être également modélisé correctement. Une nouvelle fois, nous utilisons ici une simple implication car pour un appel à m, la valeur de $from(m)$ n'est pas réinitialisée lors du retour d'appel. Nous prouverons que cela n'impacte pas l'équivalence.

Finalement, nous définissons les états équivalents pour les états de programmes simulants tels que la prochaine action à effectuer est l'évaluation de la condition de la boucle d'entrelacements, la simulation de l'exécution d'une instruction du programme d'origine étant représentée par l'exécution complète de l'évaluation de la condition de boucle et, si besoin, du corps de celle-ci. Nous modélisons cela par la partie 5 de l'équivalence.

Pour définir l'équivalence des traces, nous filtrons une partie des événements que génère la sémantique.

Dans l'exécution du programme simulant, nous ignorons les τ-actions ainsi que toutes les actions mémoire effectuées au sein de $\sigma_{sim,\mathcal{X}}$, sauf les actions produites par l'appel à `select`, donc de la forme `write` *ptid* 0 t, qui nous permettent de conserver l'information que toutes les actions qui suivent sont réalisées par t. Nous ignorons tout appel et retour de méthode simulante, sauf les appels `call_`ℓ`_`m`_simulation` à la simulation d'un appel de m et les appels `return_`ℓ_{end}`_`m`_simulation` à la simulation d'un retour d'appel de m. Dans ce filtrage, nous devons bien entendu conserver l'ordre des événements.

Finalement, une trace $t_{\|}$ et t_{sim} sont équivalentes, une fois filtrées, si en remplaçant :

— tout $(t,$ `call` $m)$ par `write` *ptid* 0 t ; `call_`ℓ`_`m`_simulation;`

— tout $(t,$ `return` $m)$ par `write` *ptid* 0 t ; `return_`ℓ_{end}`_`m`_simulation;`

— tout $(t, \texttt{read } l\ n\ v)$ par `write` *ptid* 0 t ; `read` $l\ n\ v$;
— tout $(t, \texttt{write } l\ n\ v)$ par `write` *ptid* 0 t ; `write` $l\ n\ v$;
— tout (t, τ) par `write` *ptid* 0 t;
— tout $(t, list)$ par `write` *ptid* 0 t;$replace(list)$

dans $t_{\|}$, on obtient t_{sim} et si en faisant l'opération inverse sur t_{sim}, on obtient $t_{\|}$. L'opération $replace(list)$ correspond à effectuer le remplacement précédent dans *list*, sans ajouter l'action correspondant au `select`.

5.4.2 Correction de la simulation

Théorème 5.1 (Simulation correcte) *Soient : un programme parallèle $p_{\|}$ sûr, et p_{sim} le programme qui le simule, un état initial $\gamma_{\|,init}$ de $p_{\|}$, $\gamma_{sim,init}$ un état initial de la simulation.*

Depuis, l'état $\gamma_{sim,init}$, nous pouvons atteindre, par l'exécution de la séquence d'initialisation b_{init}, un état séquentiel $\gamma_{sim,0}$ équivalent à $\gamma_{\|,init}$ (par la définition de l'équivalence précédemment présentée).

Pour tout état $\gamma_{\|}$ atteignable depuis $\gamma_{\|,init}$, il existe un état γ_{sim} équivalent atteignable depus $\gamma_{sim,0}$ avec une trace d'exécution équivalente. (Simulation avant)

Pour tout état γ_{sim} atteignable depuis $\gamma_{sim,0}$, il existe un état $\gamma_{\|}$ équivalent atteignable depuis $\gamma_{\|,init}$ avec une trace d'exécution équivalente. (Simulation arrière)

Les différents éléments de ce théorème seront prouvés plus tard, nous donnons d'abord quelques intuitions à ce sujet. La preuve de ce théorème se base sur deux observations à propos de la sémantique parallèle et de sa traduction vers le code simulant.

La première est que le seul facteur d'indéterminisme dans la sémantique parallèle est le choix du fil d'exécution, qui n'est pas une opération du programme. Dans le programme simulant, la seule opération indéterministe est l'appel à la méthode `select` qui modélise le comportement indéterministe des règles de la sémantique parallèle.

La seconde observation est qu'une fois que la règle de la sémantique parallèle a sélectionné un fil, la réduction de l'opération à effectuer par ce fil est déléguée à la sémantique des programmes séquentiels, qui est déterministe. Le code de simulation correspondant, à savoir la résolution du compteur de programme et l'exécution de la méthode de simulation de l'instruction considérée, est également déterministe. Donc, pour chaque opération séquentielle, le code généré pour la simuler est déterministe. Or, si l'on prouve une propriété de *simulation avant* pour une transformation, et que le code résultant de la transformation

est déterministe, alors nous avons également prouvé une propriété de *simulation arrière* [67, Sec 2.1] pour cette transformation.

La preuve du théorème s'effectue par induction sur les traces. Une étape de ce raisonnement s'illustre par la figure 5.13. Pour la simulation avant, le raisonnement s'effectue par induction sur les instructions de la trace et pour la simulation arrière, sur le nombre de tours de boucle de l'exécution de la simulation.

La transition entre γ_{sim} et $\gamma_{sim:t}$ correspond à l'évaluation de la condition de la boucle, suivie de l'opération `select(ptid)` que nous nommerons i_{select}. Elle modélise le choix de fil d'exécution, à condition qu'il en existe un, effectué par la sémantique de programmes parallèles.

La transition entre $\gamma_{\|}$ et $\gamma'_{\|}$ est l'exécution de l'instruction, pour un certain fil t choisi par la règle de la sémantique des programmes parallèle, dans la sémantique des programmes séquentiels. La transition entre $\gamma_{sim:t}$ et $\gamma'_{sim:t}$ correspond à la réduction de la suite d'instructions b_{sim} :

```
1 tid  := ptid[0];
2 ptr  := pct;
3 stmt := ptr[tid];
4
5 switch stmt is [
6 ∀m ∈ M, ∀c^ℓ_{ℓ_next} ∈ m,
7 ℓ : [ c_{stmt_type}ℓ_simulation (tid) ]
8 ]
```

La transition entre $\gamma'_{sim:t}$ et γ'_{sim} correspond à la réévaluation de la conditionnelle, c'est-à-dire l'évaluation de chaque compteur de programme par la suite d'instruction $b_{termination}$.

Initialisation

Lemme 5.1 (Initialisation) *Soient : un programme parallèle $p_{\|}$ sûr, et p_{sim} le programme qui le simule, un état initial $\gamma_{\|,init}$ de $p_{\|}$, $\gamma_{sim,init}$ un état initial de la simulation.*

Depuis, l'état $\gamma_{sim,init}$, nous pouvons atteindre, par l'exécution de la séquence d'initialisation b_{init}, un état séquentiel $\gamma_{sim,0}$ équivalent à $\gamma_{\|,init}$ (par la définition de l'équivalence précédemment présentée).

Démonstration. Un état initial de la simulation est :

$$(\sigma_{sim}, (\text{""}, \varnothing, [\texttt{interleavings()}]))$$

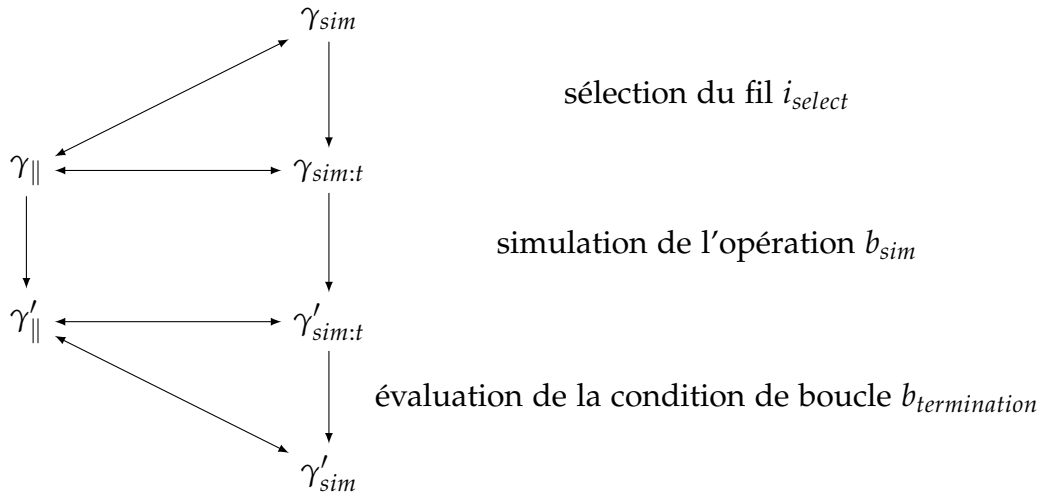


FIGURE 5.13 – *Relation de simulation*

Nous supposons également que $\sigma_{sim,\|} = \sigma_{\|}$ et que dans $\sigma_{sim,\mathcal{X}}$, les zones mémoire simulantes sont correctement allouées (cf. section 5.3.7). La partie (1) de l'équivalence est respectée.

Un état initial du programme parallèle est de la forme : $(\sigma_{\|,init}, stacks_{init})$ tel que :

$$\forall t \in \mathbb{T}. \quad stacks_{init}(t) = [(\text{""}, \varnothing, [main_t])]$$

La partie (2) de l'équivalence est donc respectée.

Par l'application de la règle [[**call**]], l'état de la simulation devient (en omettant la base de la pile pour alléger la présentation) :

$$(\sigma_{sim}, (\texttt{interleavings}, \rho_{sim}, b_{init} +\!\!+ [\textbf{while } c \textbf{ do } i_{select} :: b_{sim} +\!\!+ b_{termination}]))$$

Nous rappelons que b_{init} est de la forme :

```
1 ptr := pct ;
2 ∀t ∈ 𝕋, ptr[t] := main_calls(t).ℓ ;
3 ptr := from("") ;
4 ∀t ∈ 𝕋, ptr[t] := 0 ;
5 terminated := false ;
```

En appliquant la règle [[**assign**]] pour placer la position mémoire `pct` dans `ptr`, puis les [[**write**]] successifs pour le compteur de programme de chaque fil d'exécution. Nous assurons que pour $\sigma_{sim,\mathcal{X}}$, la partie 3.*a* est respectée, et comme les identifiants des appels aux méthodes de simulation sont différents de 0 (qui est réservé), nous avons également 3.*b*.

Ensuite, en appliquant la règle [[**assign**]] pour placer la position mémoire $from(\text{""})$ dans `ptr`, puis les [[**write**]] successifs pour le contexte d'exécution de départ de chaque fil d'exécution. Nous assurons que pour $\sigma_{sim,\mathcal{X}}$, la partie 4 est respectée.

Nous appliquons finalement [[**assign**]] pour l'affectation de `terminated`. Les compteurs de programmes sont différents de 0 et l'état résultant est $\gamma_{sim,0}$:

$$(\sigma_{sim}, (\texttt{interleavings}, \rho_{sim}, [\textbf{while } c \textbf{ do } i_{select} :: b_{sim} +\!\!+ b_{termination}]))$$

La partie (5) de l'équivalence est respectée et notre lemme est prouvé. □

Simulation avant

Lemme 5.2 (Simulation avant sur un pas d'exécution) *Soient $p_{\|}$ un programme parallèle sûr, et p_{sim} son programme simulant, $\gamma_{\|}$ et $\gamma'_{\|}$ deux états parallèles, γ_{sim} un état équivalent à $\gamma_{\|}$, (t, a) une action telle que :*

$$p_{\|} \vdash \gamma_{\|} \xrightarrow{(t,a)} \gamma'_{\|}$$

alors il existe une trace tr telle que tr est équivalente à $[(t, a)]$ et :

$$p_{sim} \vdash \gamma_{sim} \xrightarrow{tr}{}^{*} \gamma'_{sim}$$

et γ'_{sim} est équivalent à $\gamma'_{\|}$.

Démonstration. Par la relation d'équivalence, nous savons que γ_{sim} est de la forme (en omettant la base de la pile et le nom de la fonction d'entrelacements) :

$$(\sigma_{sim}, (\_, \rho_{sim}, [\textbf{while } \neg\texttt{terminated} \textbf{ do } i_{select} :: b_{sim} +\!\!+ b_{termination}]) \cdot \dots)$$

Dans la sémantique parallèle, nous effectuons un pas de réduction pour le fil d'exécution t, donc sa pile n'est pas vide, et par la partie 3.*b* de l'équivalence, nous savons que $\sigma_{sim,\mathcal{X}}[t](\texttt{pct}) \neq 0$, et par conséquent, par 5, $\rho_{sim}(\texttt{terminated}) = false$. Par la règle [[**while:true**]], nous obtenons l'état de programme simulant :

$$(\sigma_{sim}, (\_, \rho_{sim}, i_{select} :: b_{sim} +\!\!+ b_{termination} +\!\!+ [\textbf{while } \neg\texttt{terminated} \textbf{ do } b_{body}]) \cdot \dots)$$

avec $b_{body} = i_{select} :: b_{sim} +\!\!+ b_{termination}$. Nous effectuons ensuite un pas de réduction avec la règle [[**select**]]. Cela génère une action `write` *ptid* 0 t, t étant un

choix possible pour *select* car $\sigma_{sim,\mathcal{X}}[t](\texttt{pct}) \neq 0$. L'état résultant est :

$$(\sigma'_{sim}, (\_, \rho_{sim}, b_{sim} +\!\!+ b_{termination} +\!\!+ [\textbf{while } \neg\texttt{terminated } \textbf{do } b_{body}]) \cdot \dots)$$

avec $\sigma'_{sim}(ptid, 0) = t$ et tr est de la forme $(\texttt{write } ptid\ 0\ t) :: tr'$.

À ce point de la preuve, nous analysons par cas sur l'instruction à réaliser. Ces cas seront traités dans une section ultérieure. Admettons pour l'instant le lemme 5.3.

Lemme 5.3 (Simulation avant d'une instruction) *Soient i, une instruction et b_{sim} sa méthode simulante. Alors, à partir de l'état :*

$$(\sigma'_{sim}, (\_, \rho_{sim}, b_{sim} +\!\!+ b_{termination} +\!\!+ [\textbf{while } \neg\texttt{terminated } \textbf{do } b_{body}]) \cdot \dots),$$

l'exécution de b_{sim} nous permet d'aboutir à un nouvel état :

$$(\sigma''_{sim}, (\_, \rho_{sim}, b_{termination} +\!\!+ [\textbf{while } \neg\texttt{terminated } \textbf{do } b_{body}]) \cdot \dots)$$

tel que les parties 1, 2, 3 et 4 de l'équivalence avec $\gamma'_{\|}$ sont respectées. Et l'exécution de b_{sim} produit la trace tr' garantissant que $[(a, t)]$, l'action générée par i, est équivalente à $(\texttt{write } ptid\ 0\ t) :: tr'$.

L'exécution du bloc $b_{termination}$ met à jour la variable `terminated` en comparant successivement les compteurs de programme à 0. Comme nous avons maintenu 3 précédemment, nous aboutissons à un état :

$$(\sigma''_{sim}, (\_, \rho_{sim}, [\textbf{while } \neg\texttt{terminated } \textbf{do } b_{body}]) \cdot \dots)$$

tel que 5 est bien respecté. Par ailleurs, les actions générées par ce tour de boucle sont soit des lectures dans $\sigma_{sim,\mathcal{X}}$ (qui sont filtées), soit des τ-actions, également filtrées.

Nous avons donc, depuis un état γ_{sim} équivalent à $\gamma_{\|}$, construit un nouvel état γ'_{sim} ou l'équivalence avec $\gamma'_{\|}$ est vérifiée avec une trace tr équivalente à $[(a, t)]$. □

Simulation arrière

Lemme 5.4 (Simulation arrière sur un pas d'exécution) *Soient p_{sim} le programme simulant d'un programme $p_{\|}$ sûr, γ_{sim} et γ'_{sim} deux états séquentiels, $\gamma_{\|}$ un état équivalent*

à γ_{sim}, $tr = (\texttt{write}\ ptid\ 0\ t) :: tr'$ *une trace telle que :*

$$p_{sim} \vdash \gamma_{sim} \xrightarrow{tr}{}^{*} \gamma'_{sim}$$

et tr' *ne contient pas d'action* $(\texttt{write}\ ptid\ \_\ \_)$*, alors il existe une action* (t, a) *telle que* $[(t, a)]$ *est équivalente à* tr *et :*

$$p_{\|} \vdash \gamma_{\|} \xrightarrow{(t,a)} \gamma'_{\|}$$

et $\gamma'_{\|}$ *est équivalent à* γ'_{sim}*.*

Démonstration. Comme les états γ_{sim} et $\gamma_{\|}$ sont équivalents, l'état γ_{sim} est de la forme :

$$(\sigma_{sim}, (\_, \rho_{sim}, [\textbf{while}\ \neg\texttt{terminated}\ \textbf{do}\ i_{select} :: b_{sim} +\!\!+ b_{termination}]) \cdot \dots)$$

La simulation crée une trace $tr = (\texttt{write}\ ptid\ 0\ t) :: tr'$, donc la condition de la boucle s'évalue à vrai (sinon nous n'exécutons pas la boucle, et la première action de cette trace n'est pas réalisée). Par la règle [[**while:true**]], l'état résultant est :

$$\left(\sigma_{sim}, (\_, \rho_{sim}, i_{select} :: b_{sim} +\!\!+ b_{termination} +\!\!+ [\textbf{while}\ \neg\texttt{terminated}\ \textbf{do}\ b_{body}]) \cdot \dots\right)$$

et nous savons qu'il existe un t tel que $\sigma_{sim,\mathcal{X}}(pct) \neq 0$. Par l'équivalence 3, nous savons donc que dans le programme d'origine, il existe un fil identifié t tel que sa pile d'exécution n'est pas vide.

Dans le programme simulant, nous exécutons ensuite l'instruction de sélection du fil et obtenons donc un état :

$$\left(\sigma^1_{sim}, (\_, \rho_{sim}, b_{sim} +\!\!+ b_{termination} +\!\!+ [\textbf{while}\ \neg\texttt{terminated}\ \textbf{do}\ b_{body}]) \cdot \dots\right)$$

tel que $\sigma^1_{sim}(ptid, 0) = t$, de plus par l'équivalence 3.*b*, nous savons quel est l'identifiant ℓ de l'instruction à exécuter, et nous savons que dans le programme d'origine l'instruction c^ℓ est la prochaine instruction à exécuter pour t.

Comme $p_{\|}$ est sûr, il ne bloque pas, l'instruction c^ℓ de t peut être exécutée, et il existe un nouvel état parallèle $\gamma'_{\|}$ suite à cette instruction, atteint avec une action (t, a). Par le lemme 5.2, nous savons qu'il existe un état simulé $\gamma'_{sim?}$ équivalent à $\gamma'_{\|}$, atteint depuis γ_{sim} avec une trace tr_f équivalente à $[(t, a)]$. La trace tr_f commence nécessairement avec une action $\texttt{write}\ ptid\ 0\ t$ équivalente à celle

que nous avons produit pour tr et représente l'exécution de c^ℓ par t, que notre programme p_{sim} simule également. Nous pouvons en déduire que $\gamma'_{sim?} = \gamma'_{sim}$. Comme $\gamma'_{\|}$ est équivalent à $\gamma'_{sim?}$, il est également équivalent γ'_{sim}. De plus $tr = tr_f$, et tr_f est équivalente à $[(t, a)]$, donc tr est équivalente à $[(t, a)]$. □

Preuve du théorème de simulation

Théorème *Soient : un programme parallèle $p_{\|}$ sûr, et p_{sim} le programme qui le simule, un état initial $\gamma_{\|,init}$ de $p_{\|}$, $\gamma_{sim,init}$ un état initial de la simulation.*

(1) Depuis, l'état $\gamma_{sim,init}$, nous pouvons atteindre, par l'exécution de la séquence d'initialisation b_{init}, un état séquentiel $\gamma_{sim,0}$ équivalent à $\gamma_{\|,init}$ (par la définition de l'équivalence précédemment présentée).

(2) Pour tout état $\gamma_{\|}$ atteignable depuis $\gamma_{\|,init}$, il existe un état γ_{sim} équivalent atteignable depus $\gamma_{sim,0}$ avec une trace d'exécution équivalente. (Simulation avant)

(3) Pour tout état γ_{sim} atteignable depuis $\gamma_{sim,0}$, il existe un état $\gamma_{\|}$ équivalent atteignable depuis $\gamma_{\|,init}$ avec une trace d'exécution équivalente. (Simulation arrière)

Démonstration.

On prouve (1) grâce au lemme 5.1.

On prouve (2) par induction sur la liste des instructions à exécuter en utilisant le lemme 5.2.

On prouve (3) par induction sur le nombre d'itérations de la boucle d'entrelacement à l'aide du lemme 5.4. □

Terminaison

Lemme 5.5 (Terminaison (simulation avant)) *Soient : un programme parallèle $p_{\|}$ sûr, et p_{sim} le programme qui le simule, un état initial $\gamma_{\|,init}$ de $p_{\|}$, $\gamma_{sim,init}$ un état initial de la simulation.*

Depuis, l'état $\gamma_{sim,init}$, nous pouvons atteindre, par l'exécution de la séquence d'initialisation b_{init}, un état séquentiel $\gamma_{sim,0}$ équivalent à $\gamma_{\|,init}$ (par la définition de l'équivalence précédemment présentée).

Pour tout état $\gamma_{\|,final}$ atteignable depuis $\gamma_{\|,init}$, il existe un état γ_{sim} équivalent à $\gamma_{\|,final}$, atteignable depus $\gamma_{sim,0}$ tel que depuis γ_{sim}, on atteint un état final $\gamma_{sim,final}$.(Simulation avant)

Démonstration. Par le lemme 5.1, nous savons que nous pouvons atteindre $\gamma_{sim,0}$.

Par le théorème 5.1, nous savons que nous pouvons atteindre, dans la simulation l'état γ_{sim} équivalent à $\gamma_{\|,final}$. Dans ce cas, d'après la partie 3.*b* de l'équivalence :

$$\forall t \in \mathbb{T}. \quad \sigma_{sim,\mathcal{X}}[t](\texttt{pct}) = 0$$

et donc, par 5, $\rho_{sim}(\texttt{terminated}) = false$, donc l'état de la simulation :

$$(\sigma_{sim}, (\texttt{interleavings}, \rho_{sim}, [\textbf{while } c \textbf{ do } i_{select} :: b_{sim} +\!\!+ b_{termination}]) \cdot (\text{""}, \rho_{\varnothing}, []))$$

se réduit en (par [[**while:false**]]) :

$$(\sigma_{sim}, (\texttt{interleavings}, \rho_{sim}, []) \cdot (\text{""}, \rho_{\varnothing}, []))$$

puis (par [[**return**]]) :

$$(\sigma_{sim}, (\text{""}, \rho_{\varnothing}, []))$$

et finalement (par [[**return**]]) :

$$(\sigma_{sim}, [])$$

qui est bien un état final. □

5.4.3 Simulation avant des instructions

Nous prouvons ici le lemme 5.3 pour chaque type d'instruction. Nous nous replaçons donc dans le contexte de la preuve de la simulation avant, où nous voulions prouver que pour une action (t, a) d'une instruction i menant de $\gamma_{\|}$ à $\gamma'_{\|}$, nous pouvons depuis γ_{sim} équivalent à $\gamma_{\|}$, par l'exécution de b_{sim} la simulation de i, avec une trace tr équivalente à $[(t, a)]$, atteindre un nouvel état γ'_{sim} équivalent à $\gamma'_{\|}$.

Après avoir développé l'évaluation de la condition de boucle et l'exécution de `select` avec le t déterminé, nous nous sommes placés dans la situation où du côté de la simulation, nous avons atteint un état :

$$(\sigma'_{sim}, (\_, \rho_{sim}, b_{sim} +\!\!+ b_{termination} +\!\!+ [\textbf{while } \neg\texttt{terminated } \textbf{do } b_{body}]) \cdot \dots)$$

où $\sigma'_{sim}(ptid, 0) = t$, avec une trace générée de la forme $[(\texttt{write } ptid\ 0\ t)]$, et supposé que nous pouvons, par l'exécution de b_{sim}, atteindre un nouvel état avec la trace tr' :

$$(\sigma''_{sim}, (\_, \rho_{sim}, b_{termination} +\!\!+ [\textbf{while } \neg\texttt{terminated } \textbf{do } b_{body}]) \cdot \dots)$$

tel que : $tr = (\texttt{write}\ ptid\ 0\ t) :: tr'$ est équivalent à $[(t, a)]$, et cet état respecte, pour l'équivalence avec $\gamma'_{\|}$, les parties 1, 2, 3 et 4 (5 étant ensuite assurée par l'exécution de la fin du bloc de la boucle $b_{termination}$).

Nous voulons maintenant prouver que l'exécution de b_{sim} respecte effectivement cette supposition pour toute instruction atomique à exécuter.

Le bloc b_{sim} est de la forme :

```
1 tid  := ptid[0];
2 ptr  := pct;
3 stmt := ptr[tid];
4
5 switch stmt is
6 ∀m ∈ M, ∀c^ℓ_{ℓ_next} ∈ m,
7 ℓ : [ c_{stmt_type}ℓ_simulation (tid) ]
```

Quelle que soit l'instruction simulée, l'étape de récupération du numéro de fil d'exécution et la résolution du compteur de point de programme, ainsi que l'appel de la fonction simulante correspondante, procèdent de manière similaire. Nous évacuons cette phase de la preuve avant de passer à chaque instruction spécifique.

Avant d'exécuter b_{sim}, l'état est de la forme :

$$(\sigma'_{sim}, (main, \rho_{sim}, b_{sim} +\!\!+ b_{termination} +\!\!+ [\textbf{while}\ c\ \textbf{do}\ i_{select} :: b_{sim} +\!\!+ b_{termination}]) \cdot \dots)$$

Dans un premier temps, nous chargeons la valeur à la position mémoire `ptid` dans `tid` (règle [[**read**]]). Puis nous chargeons la valeur à la position mémoire `pct` pour le fil indiqué par `tid` (règle [[**read**]]). Et enfin, nous résolvons la correspondance entre ce point de programme et la méthode de simulation correspondante (nous supposons cette correspondance correcte), par une succession de conditionnelles (règle [[**if:false**]], jusqu'à atteindre la bonne valeur, règle [[**if:true**]]), plaçant alors en tête des instructions à exécuter, l'appel à la méthode simulante de l'instruction considérée. Cela nous amène donc dans un état :

$$(\sigma'_{sim}, (main, \rho'_{sim}, simulation\_id(\texttt{tid}) :: b_{termination} +\!\!+ [\textbf{while}\ c\ \textbf{do}\ b_{body}]) \cdot \dots)$$

où $\rho'_{sim} = \rho_{sim}[\texttt{tid} \mapsto t][\texttt{stmt} \mapsto id]$.

L'appel de méthode en tête produit alors, avec une action `call`$simulation\_id$, un nouveau contexte en sommet de pile (nous ignorons dans cette formulation

le reste de la pile) :

$$(\sigma'_{sim}, (simulation\_id, \rho_{\varnothing}[tid \mapsto t], stmts) \cdot \dots)$$

où *stmts* est la liste des instructions à effectuer pour simuler l'instruction du programme d'origine.

Dans cette série de réduction, les événements sont soit des τ-actions, soit des lectures dans $\sigma_{sim,\mathcal{X}}$, qui sont filtrés.

La suite de cette section montre l'équivalence pour chaque instruction, atteignant l'état :

$$(\sigma''_{sim}, (main, \rho'_{sim}, b_{termination} +\!\!+ [\textbf{while } c \textbf{ do } b_{body}]))$$

dans la simulation.

Dans la suite, nous nommerons :
— $\sigma_{\|}$, le tas du programme d'origine avant l'exécution de l'instruction ;
— $\sigma'_{\|}$, le tas du programme d'origine après l'exécution de l'instruction ;
— $\rho_{\|:t}$, l'environnement local du fil t avant l'instruction ;
— $\rho'_{\|:t}$, l'environnement local du fil t après l'instruction.

Affectation locale

Nous rappelons la règle de la sémantique pour l'affectation :

$$\mathcal{M} \vdash \sigma, ((m, \rho, (x := e; b)) \cdot \bar{s}) \quad \xrightarrow{\tau} \quad \sigma, ((m, \rho[x \mapsto v], b) \cdot \bar{s}) \quad \text{si } [\![e]\!]_{\rho} = v$$

Dans cette opération, l'expression est composée de variables locales et de constantes. On note x_1 à x_n les variables qui composent l'expression (et qui ne sont pas x elle-même, si celle-ci fait partie de l'expression), et v_1 à v_n leurs valeurs (si le pas de sémantique est effectué, e est évaluable et donc ces valeurs sont définies).

Avant de commencer l'opération de simulation, notre hypothèse est que l'état est correctement simulé. Donc pour toute variable x_i valant v_i du programme d'origine, $\sigma_{sim:t,\mathcal{X}}(\&x_i, t) = v_i$.

Nous rappelons qu'après l'appel de la méthode simulante, l'état du programme simulant est :

$$(\sigma'_{sim}, (simulation\_id, \rho_{sim}, stmts) \cdot \dots)$$

avec $\rho_{sim} = \rho_{\varnothing}[tid \mapsto t]$.

La liste *stmts* contient les instructions de la simulation, à savoir la concaténation de paires d'instructions (a) :

```
1 ptr := &x_i ;
2 x_i := ptr[tid] ;
```

pour chaque variable de e, suivie de l'écriture effective de la simulation de x (b) :

```
1 ptr := &x ;
2 ptr[tid] := e ;
```

et pour finir le changement du compteur de programme (c) :

```
1 ptr := pct ;
2 ptr[tid] := ℓ_next ;
```

Après l'exécution de (a), on a modifié ρ_{sim} tel $\forall x \in e.\rho_{sim}(x) = v_i$ car nous écrivons dans chaque x_i, la valeur v_i présente en mémoire d'après la simulation ($\sigma'_{sim,\mathcal{X}}(\&x_i, t) = v_i$).

Nous effectuons ensuite (b), l'écriture de la nouvelle valeur de x à sa position mémoire de simulation. Comme $\forall x \in e.\rho_{sim}(x) = v_i$, nous avons :

$$[\![e]\!]_{\rho_{sim}} = [\![e]\!]_{\rho_{\|:t}} = v_e$$

Donc si après l'exécution de l'instruction dans le code original, nous avons $\rho'_{\|:t}(x) = v$, nous avons dans la simulation $\sigma''_{sim,\mathcal{X}}(\&x, t) = v$, ce qui maintient la partie 2 de l'équivalence.

Finalement (c), le compteur de programme est placé sur l'identifiant de la prochaine instruction. Après l'exécution de l'affectation, l'état du programme parallèle, que l'on réduit au fil t est :

$$\sigma_{\|}, ((m, \rho[x \mapsto v], b) \cdot \bar{s})$$

avec $\sigma_{\|} = \sigma'_{\|}$.

Dans le programme simulant, le compteur de programme est placé vers la méthode de simulation de la première instruction de b, si elle existe, sinon, sur la méthode de simulation du retour m, ce qui maintient la partie 3.*a* de la relation. Comme dans le code d'origine, la pile d'exécution n'est pas vidé par l'instruction (même si c'était la dernière, il reste l'action de retour d'appel), l'identifiant reçu est différent de 0, assurant également 3.*b*.

La partie 1 de la relation est maintenue car ni l'exécution du programme d'origine, ni l'exécution du programme simulant de modifient $\sigma_{\|}$ et $\sigma'_{sim,\|}$, donc $\sigma_{\|} = \sigma'_{\|}$ et de manière équivalent $\sigma'_{sim,\|} = \sigma''_{sim,\|}$

La relation 4 est maintenue car l'exécution du programme d'origine ne modifie pas la pile d'appel et notre code de simulation, ne modifie pas les zones mémoire correspondant aux $from(m)$.

Cela nous amène dans un état :

$$(\sigma''_{sim}, (simulation\_id, \rho'_{sim}, []) \cdot \dots)$$

puis, par le retour d'appel :

$$(\sigma''_{sim}, (main, \rho'_{sim}, b_{termination} +\!\!+ [\textbf{while } c \textbf{ do } b_{body}]))$$

Dans l'exécution du programme d'origine une τ-action est générée, Donc $(t, a) = (t, \tau)$. $tr = (\texttt{write } ptid\ 0\ t) :: tr'$, et $tr' = []$ car toutes les actions effectuées par l'exécution de b_{sim} pour l'affectation sont effectuées dans $\sigma'_{sim,\mathcal{X}}$, et celles-ci sont filtrées. Ce qui assure également l'équivalence de la trace.

Lecture en mémoire

Nous rappelons la règle de la sémantique pour la lecture (nous effectuons quelques changements de noms pour faciliter la comparaison avec le code de simulation) :

$$\mathcal{M} \vdash \sigma, ((m, \rho, (p[o] := e; b)) \cdot \bar{s}) \quad \xrightarrow{\texttt{write } l\ n\ v} \quad \sigma[(l, o) \mapsto v], ((m, \rho, b) \cdot \bar{s})$$
$$\text{si } [\![e]\!]_\rho = v,\ [\![o]\!]_\rho = n,\ \rho(p) = l,\ n < size(l)$$

Nous rappelons qu'après l'appel de la méthode simulante, l'état du programme simulant est :

$$(\sigma'_{sim}, (simulation\_id, \rho_{sim}, stmts) \cdot \dots)$$

avec $\rho_{sim} = \rho_\varnothing[tid \mapsto t]$. Nous ne présentons pas la preuve du maintien de 3 et de 4 qui est équivalente à celle de l'opération d'affectation.

La liste *stmts* contient les instructions :

```
1 loads(variables(o),tid) ;
2 load(p,tid) ;
3 x := p[o] ;
4 ptr := &x ;
```

```
5 ptr[tid] := x ;
```

à partir de l'état :

L'opération de chargement des variables de `o` et de la variable `p` est équivalente à celle utilisée dans l'affectation locale. Il s'en suit qu'après cette suite d'opérations, pour chaque x_i de o, $\rho'_{sim}(x_i) = v_i$ si $\rho_{\|:t}(x_i) = v_i$, et de même pour le pointeur `p` : $\rho'_{sim}(p) = l$ si $\rho_{\|:t}(p) = l$.

De la même manière que pour e dans l'affectation locale, nous avons bien :

$$[\![o]\!]_{\rho'_{sim}} = [\![o]\!]_{\rho_{\|:t}} = n$$

Après l'exécution de l'opération de lecture dans le code d'origine, le tas $\sigma_{\|}$ n'est pas modifié. De la même manière, l'exécution de cette méthode de simulation ne modifie pas $\sigma'_{sim,\|}$. Nous n'accédons en écriture qu'à l'adresse $\&x$ qui se trouve dans $\sigma'_{sim,\mathcal{X}}$, cette partie du tas étant disjointe de $\sigma'_{sim,\|}$. L'équivalence 1 est maintenue.

Par $[\![o]\!]_{\rho'_{sim}} = [\![o]\!]_{\rho_{\|:t}} = n$, et $\rho'_{sim}(p) = \rho_{\|:t}(p) = l$, l'opération `x := p[o]` produit une action `read` $l\ n\ v$ équivalente à celle du programme d'origine puisque par la partie 1 de la relation d'équivalence, $\sigma_{\|}(l, n) = v$ et $\sigma'_{sim}(l, n) = v$, l se trouvant dans $\sigma'_{sim,\|}$.

Par conséquent, l'écriture à la position $\&x$ pour l'indice tid de la valeur de x rétablit également la partie 2 de l'équivalence d'état car $\rho'_{\|:t}(x) = \sigma''_{sim,\mathcal{X}}[t](x)$.

Comme mentionné, la preuve du maintien de 3 par le changement du compteur de programme et de 4 est équivalente à celle de l'affectation locale.

Dans l'exécution du programme d'origine une action de lecture est générée, Donc $(t, a) = (t,$ `read` $l\ n\ v)$. $tr = ($`write` $ptid\ 0\ t) :: tr'$, et $tr' = [$`read` $l\ n\ v]$ car toutes les autres actions que la lecture en mémoire, effectuées par l'exécution de b_{sim} pour l'opération de lecture sont effectuées dans $\sigma'_{sim,\mathcal{X}}$, et celles-ci sont filtrées. Ce qui assure également l'équivalence de la trace.

Écriture en mémoire

Nous rappelons la règle de la sémantique pour l'écriture en mémoire (nous effectuons quelques changements de noms pour faciliter la comparaison avec le code de simulation) :

$$\mathcal{M} \vdash \sigma, ((m, \rho, (p[o] := e; b)) \cdot \bar{s}) \xrightarrow{\texttt{write}\ l\ n\ v} \sigma[(l, n) \mapsto v], ((m, \rho, b) \cdot \bar{s})$$
$$\text{si } [\![e]\!]_\rho = v,\ [\![o]\!]_\rho = n,\ \rho(p) = l,\ n < size(l)$$

Nous rappelons qu'après l'appel de la méthode simulante, l'état du programme simulant est :

$$(\sigma'_{sim}, (simulation\_id, \rho_{sim}, stmts) \cdot \dots)$$

avec $\rho_{sim} = \rho_{\varnothing}[tid \mapsto t]$. Nous ne présentons pas la preuve du maintien de 3 et de 4 qui est équivalente à celle de l'opération d'affectation.

```
1 loads(variables(e),tid) ;
2 loads(variables(o),tid) ;
3 load(p,tid) ;
4 p[o] := e ;
```

De la même manière que dans la simulation de la lecture, les variables locale x de e, o et p sont chargées dans les variables locales x pour la méthode de simulation. Par conséquent, après ces chargements, on a :

— $\forall x \in variables(e).\rho'_{sim}(x) = v$ avec $\rho_{\|:t}(x) = v$;

— $\forall x \in variables(o).\rho'_{sim}(x) = v$ avec $\rho_{\|:t}(x) = v$;

— $\rho'_{sim}(p) = l$ avec $\rho_{\|:t}(p) = l$

Donc, $[\![e]\!]_{\rho'_{sim}} = [\![e]\!]_{\rho_{\|:t}} = v$, $[\![o]\!]_{\rho'_{sim}} = [\![o]\!]_{\rho_{\|:t}} = n$ et $[\![p]\!]_{\rho'_{sim}} = [\![p]\!]_{\rho_{\|:t}} = l$. L'opération `p[o] := e` du code de simulation produit donc bien l'action `write` $l\ n\ v$ produite par l'exécution du code d'origine. Cette opération n'étant pas filtrée dans la programme d'origine, ni dans le programme simulant, car s'effectuant dans $\sigma'_{sim,\|}$. La trace est maintenue équivalente.

Par ailleurs, la seule modification apportée à l'état global par cette phase de simulation est dans $\sigma'_{sim,\|}$ pour le programme simulant et $\sigma_{\|}$ pour le programme d'origine. Cette modification est bien équivalente (même action sur la mémoire). On maintient donc bien 1.

L'état local de t n'est pas modifié dans l'exécution du programme d'origine. $\sigma'_{sim,\mathcal{X}}$ ne l'est pas non plus, maintenant la relation 2.

Le maintien des relations 3 et 4 est équivalent à celui de l'affectation locale.

Dans l'exécution du programme d'origine une action d'écriture est générée, Donc $(t, a) = (t, \texttt{write}\ \ l\ n\ v)$. $tr = (\texttt{write}\ ptid\ 0\ t) :: tr'$, et $tr' = [\texttt{write}\ \ l\ n\ v]$ car toutes les autres actions que l'écriture en mémoire, effectuées par l'exécution de b_{sim} pour l'opération d'écriture sont effectuées dans $\sigma'_{sim,\mathcal{X}}$, et celles-ci sont filtrées. Ce qui assure également l'équivalence de la trace.

Conditionnelle

Nous rappelons les règles de la sémantique pour la conditionnelle :

$$\mathcal{M} \vdash \sigma, ((m, \rho, (\textbf{if } e \textbf{ then } b_{true} \textbf{ else } b_{false}); b) \cdot \bar{s}) \xrightarrow{\tau} \sigma, ((m, \rho, (b_{true} +\!\!+ b)) \cdot \bar{s}) \quad \text{si } [\![e]\!]_\rho = \texttt{true},$$

$$\mathcal{M} \vdash \sigma, ((m, \rho, (\textbf{if } e \textbf{ then } b_{true} \textbf{ else } b_{false}); b) \cdot \bar{s}) \xrightarrow{\tau} \sigma, ((m, \rho, (b_{false} +\!\!+ b)) \cdot \bar{s}) \quad \text{si } [\![e]\!]_\rho = \texttt{false},$$

Nous rappelons qu'après l'appel de la méthode simulante, l'état du programme simulant est :

$$(\sigma'_{sim}, (simulation\_id, \rho_{sim}, stmts) \cdot \dots)$$

avec $\rho_{sim} = \rho_\varnothing[tid \mapsto t]$. La liste *stmts* contient (en fonction de la forme de `b1` et `b2` dans le code d'origine) les instructions

```
1  loads(variables(e),tid) ;
2  ptr := pct ;
3  if e then
4    empty(b1) →
5      ptr[tid] := ℓ_next ;
6    b1 = c^ℓ'_ℓ'_next :: _ →
7      ptr[tid] := ℓ' ;
8  else
9    empty(b2) →
10     ptr[tid] := ℓ_next ;
11   b2 = c^ℓ'_ℓ'_next :: _ →
12     ptr[tid] := ℓ' ;
```

Dans un premier temps, les variables locales de e sont chargées depuis $\sigma'_{sim,\mathcal{X}}$. De manière équivalente, aux étapes de preuve des affectations, on sait que : $[\![e]\!]_{\rho'_{sim}} = [\![e]\!]_{\rho_{\|:t}} = v$. Comme ces évaluations sont équivalentes, les exécutions du programme d'origine et de ce bloc simulant empruntent les mêmes branches de la conditionnelle pour des environnement de départ équivalents.

L'état généré par cette évaluation dans le programme d'origine est :

— $\sigma_\|, ((m, \rho, (b_{true} +\!\!+ b)) \cdot \bar{s})$ ou

— $\sigma_\|, ((m, \rho, (b_{false} +\!\!+ b)) \cdot \bar{s})$

Pour chacune des deux branches, nous avons le même schéma : soit le bloc est vide, soit il ne l'est pas. Donc soit le nouvel état contient une liste d'instruction $b_{chosen} +\!\!+ b$, soit simplement b, b pouvant elle même être vide.

Si le bloc choisi n'est pas vide le compteur de programme est placé sur la première instruction de ce bloc, sinon, il est placé sur la première instruction de

b (ceci doit être correctement calculé par le graphe de flot de contrôle). On atteint soit la prochaine instruction à exécuter, soit la fin de méthode (identifiant ℓ_{end}). Nous assurons donc 3.a, ainsi que 3.b puisque nous ne dépilons pas d'élément de la pile d'exécution.

Les tas $\sigma_{\|}$ et $\sigma'_{sim,\|}$ ne sont pas modifiés, donc 1 est maintenue. Nous ne dépilons aucun contexte d'exécution pendant l'exécution du programme d'origine, et que nous n'écrivons pas $from(\_)$, la relation 4 est maintenue. Nous ne modifions aucune variable locale dans le programme d'origine, et ne modifions pas $\sigma_{sim,\mathcal{X}}$, donc l'équivalence 2 est également maintenue.

Dans l'exécution du programme d'origine une τ-action est générée, Donc $(t, a) = (t, \tau)$. $tr = (\texttt{write}\ ptid\ 0\ t) :: tr'$, et $tr' = []$ car toutes les actions effectuées par l'exécution de b_{sim} pour l'affectation sont effectuées dans $\sigma'_{sim,\mathcal{X}}$, et celles-ci sont filtrées. Ce qui assure également l'équivalence de la trace.

Boucle

Nous rappelons les règles de la sémantique pour la boucle :

$$\mathcal{M} \vdash \sigma, ((m, \rho, (\textbf{while}\ e\ \textbf{do}\ b_{loop}; b)) \cdot \bar{s}) \xrightarrow{\tau} \sigma, ((m, \rho, (b_{loop} +\!\!+ \textbf{while}\ e\ \textbf{do}\ b_{loop}; b)) \cdot \bar{s}) \quad \text{si } [\![e]\!]_\rho = \texttt{true},$$

$$\mathcal{M} \vdash \sigma, ((m, \rho, (\textbf{while}\ e\ \textbf{do}\ b_{loop}; b) \cdot \bar{s}) \xrightarrow{\tau} \sigma, ((m, \rho, b \cdot \bar{s}) \quad \text{si } [\![e]\!]_\rho = \texttt{false},$$

Nous rappelons qu'après l'appel de la méthode simulante, l'état du programme simulant est :

$$(\sigma'_{sim}, (simulation\_id, \rho_{sim}, stmts) \cdot \dots)$$

avec $\rho_{sim} = \rho_\varnothing[tid \mapsto t]$. La liste $stmts$ contient (en fonction de la forme de `b` ans le code d'origine) les instructions

```
1 loads(variables(e),tid) ;
2 ptr := pct ;
3 if e then
4   empty(b) →
5     ptr[tid] := ℓ ;
6   b = c^{ℓ'}_{ℓ'_next} :: _ →
7     ptr[tid] := ℓ' ;
8 else
9   ptr[tid] := ℓ_next ;
```

La preuve est similaire à la preuve de la conditionnelle. Néanmoins la validité de la simulation du point de vue de l'ordre d'exécution des instructions et de l'équivalence de la trace est conditionnée par le fait que le graphe de flot de contrôle est correctement calculé et la simulation de la dernière instruction du bloc ramène à la méthode de simulation de la boucle.

Appel

Nous rappelons les règles de la sémantique pour l'appel de méthode :

$$\mathcal{M} \vdash \sigma, ((m_1, \rho_{m_1}, (m_2(\overline{arg}); b_{m_1})) \cdot \bar{s}) \xrightarrow{\texttt{call } m_2} \sigma, ((m_2, \rho_{m_2}, b_{m_2}) \cdot (m_1, \rho_{m_1}, b_{m_1}) \cdot \bar{s})$$
$$\text{si } m_2(\bar{x})b_{m_2} \in \mathcal{M},\ |\overline{arg}| = |\bar{x}|,\ [\![\overline{arg}]\!]_\rho = \bar{v},$$
$$\rho_{m_2} = \bar{x} \mapsto \bar{v}$$

Nous rappelons qu'après l'appel de la méthode simulante, l'état du programme simulant est :

$$(\sigma'_{sim}, (simulation\_id, \rho_{sim}, stmts) \cdot \dots)$$

avec $\rho_{sim} = \rho_\varnothing[tid \mapsto t]$. La liste *stmts* contient les instructions :

```
1 loads(variables(l),tid) ;
2 combine(args(m2),l) ;
3
4 ptr := from(m2) ;
5 ptr[tid] := ℓnext ;
6
7 ptr := pct;
8 ptr[tid] := ℓm2;
```

Nous ne modifions ni $\sigma_\parallel$ dans l'exécution du programme d'origine, ni $\sigma'_{sim,\parallel}$ dans l'exécution du programme simulant. Donc la relation 1 est maintenue.

De la même manière que précédemment, nous savons qu'après l'exécution des instructions de chargement des variables locales, nous avons : $\forall e \in l.[\![e]\!]_{\rho'_{sim}} = [\![e]\!]_{\rho_{\parallel:t}} = v$. Pour chaque adresse de simulation $\&v_i$ des paramètres v_i de m_2, nous y plaçons la valeur de l'évaluation de e_i. Pour cela, la fonction *combine*, produit une série d'instructions :

```
1 ptr = &args(mth)[i];
2 ptr[tid] = ei;
```

Lors de l'appel, l'exécution du programme d'origine construit un environnement local d'exécution pour lequel chaque paramètre x_i se voit associer son argument de valeur $[\![e_i]\!]_{\rho_{\|:t}}$. Les instructions générées vont assurer que pour chaque adresse de simulation $\&x_i$, pour le fil d'exécution t, on écrit $\sigma''_{sim,\mathcal{X}}(\&x_i) = [\![e_i]\!]_{\rho_{\|:t}}$. Comme, $\forall e \in l.[\![e]\!]_{\rho'_{sim}} = [\![e]\!]_{\rho_{\|:t}} = v$, nous respectons bien 2 lors de la modélisation du contexte local de m_2.

Nous fixons ensuite $from(m_2)$ pour le fil d'exécution t. Nous y plaçons l'identifiant ℓ_{next} de la prochaine instruction à effectuer. À ce point, nous savons que la pile d'exécution de t dans l'exécution du programme d'origine contient au minimum un contexte d'exécution (celui que nous exécutons). Si ce contexte est unique, par 4 nous savons que $from(m_1) = 0$ (et $m1 = ""$), il nous faut prouver que 4 pour cet appel. Or, après l'exécution de la ligne 3, $\sigma''_{sim}[t](from(m_2))$ est égal à l'identifiant de l'instruction qui suit cet appel. Sous l'hypothèse d'absence de récursion dans les appels, et à condition que les simulation des retours d'appels maintiennent également 4, par récurrence, nous modélisons correctement toute pile.

Enfin, nous plaçons le compteur de programme sur la première instruction de m_2, dont l'ensemble des instructions est empilé sur la pile d'exécution. Nous respectons bien 3.

Dans l'exécution du programme d'origine une action `call` m_2 est générée, donc $(t, a) = (t, \texttt{call } m_2)$. $tr = (\texttt{write } ptid\ 0\ t) :: tr'$, et $tr' = [\texttt{call\_}\ell\texttt{\_}m_2\texttt{\_simulation}]$ car toutes les autres actions effectuées par l'exécution de b_{sim} sont effectuées dans $\sigma'_{sim,\mathcal{X}}$, et celles-ci sont filtrées. Ce qui assure également l'équivalence de la trace.

Retour

Nous rappelons les règles de la sémantique pour le retour d'appel de méthode :

$$\mathcal{M} \vdash \sigma, ((m, \rho_m, []) \cdot \bar{s}) \quad \xrightarrow{\texttt{return } m} \quad \sigma, \bar{s}$$

Nous rappelons qu'après l'appel de la méthode simulante, l'état du programme simulant est :

$$(\sigma'_{sim}, (simulation\_id, \rho_{sim}, stmts) \cdot \dots)$$

avec $\rho_{sim} = \rho_{\varnothing}[tid \mapsto t]$. La liste $stmts$ contient les instructions :

```
1 return_ℓ_end_m_simulation (tid) [
2   ptr := from(m);
3   aux := ptr[tid];
4   ptr := pct;
5   ptr[tid] := aux ;
6 ]
```

L'exécution de ce programme ne produit pas d'opérations dans $\sigma'_{sim,\|}$, donc la relation 1 est maintenue par cette simulation car $\sigma''_{sim,\|} = \sigma'_{sim,\|}$.

Aucune variable locale n'est modifiée dans l'exécution du programme d'origine, et cette simulation n'effectue pas non plus de modification dans $\sigma'_{sim,\mathcal{X}}$ pour les adresses de simulation des locales. L'implication 2 est maintenue.

En revanche, le contexte d'exécution est dépilé, et donc les variables locales de ce contexte ne sont plus définies. À l'inverse, les valeurs placées dans $\sigma'_{sim,\|}$ pour simuler ces variables locales restent définies, c'est pourquoi nous n'avons qu'une simple implication. Pour maintenir poser une équivalence, et non une simple implication, nous aurions besoin d'une opération pour réinitialiser les simulations des variables locales à une valeur indéfinie. Comme dans cette preuve, nous considérons l'équivalence pour des programmes sûrs, nous avons la garantie que les variables locales sont, dans le programme d'origine, initialisées avant d'être lues dans une méthode. Cela nous garantit que la simulation restera correcte, car nous écrirons également $\sigma'_{sim,\|}$ lors d'une simulation subsidiaire de la méthode avant de le relire.

Lors du dépilement du contexte d'exécution, le contexte précédent est rétabli, si un tel contexte m' existe. Par 4, nous savons que $from(m)$ pour le fil en cours contient le point de programme de la prochaine instruction de m à exécuter. Après le dépilement, c'est ce contexte qui se trouve en tête de pile et donc l'écriture du point de programme à cette valeur nous garantit 3.*a*.

Si l'état local ne contenait qu'un seul contexte d'exécution, par 4, nous savons que $from(m)$ contient 0 (et $m = ""$) et le point de programme est donc placé sur cette valeur, garantissant que ce fil ne sera plus activé comme voulu par l'absence de contexte pour le fil. 3.*b* est respectée.

L'implication 4 est maintenue, du fait que nous ne faisons que dépiler le premier contexte, le reste des contextes est donc toujours bien modélisé. La simulation d'un appel subsidiaire à m maintiendrait bien 4 car il réécrirait $from(m)$.

Dans l'exécution du programme d'origine une action `return` m est générée, donc $(t, a) = (t, \texttt{return}\ m)$. $tr = (\texttt{write}\ ptid\ 0\ t) :: tr'$, et $tr' = [\texttt{return\_}\ell_{end}\texttt{\_}m\texttt{\_simulation}]$ car toutes les autres actions effectuées par l'exé-

cution de b_{sim} sont effectuées dans $\sigma'_{sim,\mathcal{X}}$, et celles-ci sont filtrées. Ce qui assure également l'équivalence de la trace.

Bloc atomique

Les blocs atomiques reprennent les idées des preuves précédentes. Au lieu d'exécuter un unique pas de simulation, on exécute les pas des simulations de toutes les instructions d'un bloc de code.

Le maintien de l'équivalence pour les affectations que nous rencontrons est identique pour les instructions d'affectation. Pour les conditionnelles, l'évaluation de la condition est équivalent à ce qui est présenté précédemment, en revanche, l'exécution de la branche empruntée repose par induction sur la bonne exécution de la traduction d'un bloc atomique. De même, pour la boucle, pour laquelle le chargement des variables locales doit de nouveau être effectué en fin d'exécution du bloc pour assurer la réévaluation de sa condition.

Les appels de méthodes comprennent d'abord la simulation de la construction du contexte d'appel, puis la simulation des instructions internes, puis la simulation du retour. Cela génère les mêmes modifications dans les états que celles mentionnées par les preuves précédentes. L'usage des méthodes de simulation pour l'appel et le retour permettent de générer les actions assurant l'équivalence au niveau de la trace.

5.5 Conclusion

Dans ce chapitre, nous avons présenté la preuve de la correction de notre méthode de transformation de code concurrent en code séquentiel dans le cadre du modèle mémoire séquentiellement consistant. Cette preuve est réalisée sous l'hypothèse que le programme d'origine ne produit pas d'appels récursifs (pour lesquels notre modélisation du contexte d'exécution n'est pas adapté).

La preuve a trois préoccupations principales :

— le tas du programme d'origine doit être correctement répliqué dans le programme simulant ;

— les environnements locaux du programme d'origine doivent être correctement modélisés par l'environnement global du programme simulant ;

— le contexte d'exécution d'origine doit être correctement modélisé par le compteur de programme pct et les adresses $from(\_)$ qui modélisent la pile.

La preuve de correction s'appuie sur le fait que d'une certaine manière, le code simulant mime la sémantique opérationnelle du programme d'origine avec ses propres instructions, dans une version simplifiée (notamment ne nécessitant pas de véritable pile d'exécution), et que pour chaque instruction séquentielle, le code résultant est déterministe.

La formalisation du langage, de sa sémantique et les fonctions de transformation de chaque instruction du monde séquentiel sont effectuées dans l'assistant de preuve Coq. La prochaine étape est d'ajouter la simulation des blocs atomiques, d'exprimer les équivalences et de réaliser la preuve mécanisée de l'ensemble.

Modèles mémoire faibles

6

Sommaire

6.1 Introduction

La technique présentée dans le chapitre 4 est valide si le programme concurrent analysé a un comportement Séquentiellement Consistant (SC). Dans [61], Lamport définit ce modèle théorique où les exécutions d'un programme concurrent correspondent à un entrelacement de ses instructions.

Dans les faits, nos processeurs ne nous assurent pas le respect d'un tel modèle car il est bien trop contraignant pour permettre d'avoir des composants performants. À la place, ils implémentent des modèles appelés modèles mémoire *faibles* ou *relaxés*, autorisant un plus grand nombre d'exécutions concurrentes que le modèle SC.

Dans de tels modèles, certaines relations d'ordre entre les instructions exécutées par le processeur peuvent ne pas être respectées. Cela se traduit notamment par l'exécution des instructions dans le désordre (*out-of-order*), la mise en mémoire tampon des écritures avant l'écriture effective en mémoire, la relaxation de l'atomicité des accès à la mémoire ou encore de la spéculation sur les lectures. Il est possible de restaurer l'ordre de certaines instructions lorsque c'est nécessaire au bon fonctionnement du programme par l'introduction de barrières mémoire.

Ces barrières ont pour effet d'empêcher certains réordonnancements des instructions à travers la barrière, ou encore de forcer le vidage des tampons d'écriture. Par exemple, sous TSO (Total Store Order) une écriture peut être réordonnée après une lecture qui la suit. Si l'on place une barrière entre les deux, ce réordonnancement ne peut plus avoir lieu.

Nous pouvons par exemple illustrer l'exécution *out-of-order* avec le programme présenté en figure 6.1, où x et y sont des variables globales.

En prenant 0 comme valeur initiale pour x et y, si l'on suit le modèle SC, ce programme n'a que trois sorties autorisées pour les registres r_0 et r_1 qui sont :

— $r0 = 0, r1 = 1$,
— $r0 = 1, r1 = 0$,
— $r0 = 1, r1 = 1$,

sb :	Thread 0	Thread 1
	(i_{00}) $\mathtt{x} = 1;$	(i_{10}) $\mathtt{y} = 1;$
	(i_{01}) $\mathtt{r_0} = \mathtt{y};$	(i_{11}) $\mathtt{r_1} = \mathtt{x};$

Figure 6.1 – *Exemple de programme concurrent. Relaxations : out-of-order, mémoire tampon*

qui correspondent à différents entrelacements. On a par exemple pour le résultat $r0 = 1$, $r1 = 1$, les entrelacements :

— $i_{00} \rightarrow i_{10} \rightarrow i_{01} \rightarrow i_{11}$,
— $i_{10} \rightarrow i_{00} \rightarrow i_{01} \rightarrow i_{11}$,
— $i_{00} \rightarrow i_{10} \rightarrow i_{11} \rightarrow i_{01}$,
— $i_{10} \rightarrow i_{00} \rightarrow i_{11} \rightarrow i_{01}$.

En revanche sur une architecture comme x86-TSO, il est autorisé d'obtenir les valeurs finales $r_0 = 0$ et $r_1 = 0$ qui ne correspondent à aucun entrelacement d'instruction. Ce comportement peut être obtenu car le modèle autorise le processeur à réordonner les écritures après les lectures lorsqu'elles concernent des positions mémoires différentes. Dès lors, sur cet exemple, le processeur est autorisé à exécuter : $i_{11} \rightarrow i_{01} \rightarrow i_{10} \rightarrow i_{00}$.

On peut également noter que même si les instructions de chaque fil d'exécution sont exécutées dans l'ordre, ce résultat peut être obtenu par l'utilisation d'une autre relaxation : l'usage des tampons d'écriture. L'exécution pourrait par exemple correspondre à l'une des exécutions précédemment mentionnées pour le résultat $r_0 = 1$ et $r_1 = 1$, sauf que les écritures en mémoire aux instructions i_{00} et i_{10} seraient placées en mémoire tampon et leur exécution effective serait alors différée. Les deux fils d'exécution liraient alors la valeur 0 dans une première position mémoire lors des lectures i_{01} et i_{11}, tout en ayant préalablement lancé une écriture à 1 pour l'autre position mémoire.

Écrire des programmes concurrents corrects et raisonner à leur sujet est difficile. Cette difficulté est encore accrue par les comportements qui apparaissent avec les modèles mémoire faibles. Pour traiter les programmes concurrents dans leur globalité, il faut malgré tout les prendre en compte. Soit en décidant d'identifier les programmes qui dans le modèle cible ont des comportements qui ne sont pas dans SC, et de les rejeter. Soit en raisonnant directement sur les programmes concurrents avec une logique comprenant les modèles mémoire faibles.

Dans le cadre de cette thèse, nous choisissons la première option, et décidons d'identifier les programmes ne pouvant être traités comme des programmes SC.

Pour cela, nous proposons un outil écrit en Prolog et CHR [20] (*Constraint Handling Rules*) qui, prenant un programme donné, est capable de générer toutes les exécutions autorisées par un modèle mémoire. Le principe est alors de générer toutes les exécutions autorisées par le modèle SC et toutes les exécutions autorisées par le modèle cible. Si toutes les exécutions autorisées par le modèle cible sont autorisées par SC, alors le programme a un comportement séquen-

tiellement consistant et peut donc être analysé par la méthode définie dans le chapitre 1.

La perspective principale de ce travail est donc de pouvoir compléter la méthode proposée, de manière à ne pas autoriser l'analyse d'un programme si celui-ci n'a pas un comportement séquentiellement consistant. Pour cela, le greffon Conc2Seq pourrait compiler le programme analysé vers le langage que peut traiter notre solveur de contraintes, afin de déterminer si l'ensemble de ses exécutions, selon le modèle mémoire qui l'exécute, est bien celui qui est également accepté par le modèle SC.

Ce chapitre s'articule en cinq parties. Dans la section 6.2, nous présentons le langage CHR et les relations de base que nous utilisons pour formaliser les modèles mémoire. Ensuite, dans la section 6.3, nous présentons la formalisation d'un modèle extrêmement faible autorisant tous les comportements faibles dans un programme et servant de base à la définition de modèles spécifiques. Dans la section 6.4, nous présentons la formalisation de trois modèles : SC, TSO et Partial Store Order (PSO). La section 6.5 justifie la terminaison de l'algorithme mis en place pour l'identification des exécutions autorisées pour un programme sur un modèle donné. Finalement, la section 6.6 donne des éléments nécessaires pour assurer la correction lors de la définition d'un modèle, une évaluation de la complexité du problème et un ensemble de tests de correction et de performances effectués sur le solveur.

6.2 Définitions

6.2.1 Prolog et CHR

Le prototype que nous avons développé repose sur la programmation logique par contraintes et les langages Prolog et CHR. Dans ces langages, le principe n'est pas de décrire comment résoudre un problème mais plutôt de décrire les propriétés que notre solution doit respecter. Nous pensons que ce type de méthode de résolution est particulièrement adapté pour modéliser des exécutions selon un modèle mémoire.

En effet, lorsque l'on décrit le comportement d'un modèle mémoire, comme par exemple dans le manuel d'une architecture, il n'est pas fait mention de la manière dont le processeur va ordonner les instructions, mais plutôt des propriétés que vont respecter les exécutions qui vont résulter. Il nous est donc plus aisé

de modéliser ces exécutions en transcrivant ces propriétés plutôt qu'en essayant de déduire l'algorithme respecté par le processeur.

Prolog

Prolog [32] est un langage de programmation logique. En Prolog, plutôt que définir comment l'on doit résoudre un certain problème, on définit les propriétés logiques que sa solution doit respecter, laissant le compilateur générer la séquence d'instructions qui va permettre son calcul. On définit une base de connaissances sous la forme de faits et de règles de raisonnement :

```
% faits :
femme(alice).        % Alice est une femme
homme(bob).          % Bob est un homme
femme(cleo).         % Cleo est une femme
parent(alice, bob). % Un parent d'Alice est Bob
parent(alice, cleo).% Un parent d'Alice est Cleo

% regles :
% Le pere de X est Y si c'est un homme et un parent de X
pere(X,Y) :- parent(X,Y), homme(Y), !.
% La mere de X est Y si c'est une femme et un parent de X
mere(X,Y) :- parent(X,Y), femme(Y), !.
```

Les problèmes sont ensuite posés sous la forme de requêtes qui seront résolues par calcul des prédicats du premier ordre. Originellement, les programmes Prolog étaient restreint aux clauses de Horn [52] (même si aujourd'hui les implémentations acceptent des propriétés plus complexes) car l'on sait efficacement les résoudre par résolution SLD.

Sur la base que nous avons définie, nous pouvons par exemple avoir la suite de requêtes et réponses suivante :

```
?- pere(alice, Qui). % qui est le pere d'Alice ?
Qui = bob.

?- mere(Qui, cleo).  % de qui Cleo est-elle la mere ?
Qui = alice.

?- mere(bob, cleo).  % Cleo est-elle la mere de Bob ?
false.
```

```
1 :- use_module(library(chr)).
2 :- chr_constraint a/1, b/2, c/3.
3
4 ?- a(3), b(5,4), b(X,4), c(X,32,42).
5 a(3)
6 b(X,4)
7 b(5,4)
8 c(X,32,42)
9 true
```

FIGURE 6.2 – *Contraintes en CHR*

```
9
10 ?- parent(Enfant, Parent). % Qui est l'enfant de qui ?
11 Enfant = alice,
12 Parent = bob ;
13 Enfant = alice,
14 Parent = cleo.
```

La version de Prolog que nous utilisons est SWI-Prolog 7.2.3.

Constraint Handling Rules

Un programme en langage CHR [44] est un ensemble de règles de réécriture qui vont agir sur un stock de contraintes, en enlevant ou ajoutant des contraintes à ce stock. Il est important de noter que la notion de contrainte en CHR n'est pas une notion calculatoire. Concrètement les contraintes en CHR ne sont rien de plus que des termes, elles ne produisent pas de calculs.

La figure 6.2 illustre la déclaration des contraintes et leur créations. La ligne 2 nous permet de déclarer que nous avons 3 sortes de contraintes différentes : `a` d'arité 1, `b` d'arité 2 et `c` d'arité 3.

Par la suite, si l'on demande à Prolog d'évaluer une suite de termes contenant des contraintes CHR, alors ces contraintes seront ajoutées au stock de contraintes du programme. Par exemple, à la ligne 4, nous trouvons une requête Prolog qui va générer un store avec 4 contraintes qui sont celles listées.

Avant de répondre à la requête fournie, Prolog va commencer par appliquer toutes les règles CHR possibles sur le stock de contraintes. Ici, comme nous n'avons défini aucune règle, il se contente de renvoyer les contraintes que nous avons fournies et répond `true`.

Dans la figure 6.3, nous illustrons les différentes formes de règles CHR. La première forme de règle (ligne 5) est la simplification. Cette règle s'active si l'on

```
1 % Head  : H1, ..., Hn : CHR constraints
2 % Guard : P1, ..., Pn : Prolog terms
3 % Body  : B1, ..., Bn : CHR constraints and/or Prolog terms
4
5 simplification @ Head ⇔ Guard | Body .
6 propagation    @ Head ⇒ Guard | Body .
7 simpagation    @ Head_k \ Head_r ⇔ Guard | Body .
```

FIGURE 6.3 – *Formes générales des règles CHR*

a dans le stock, un ensemble de contraintes qui sont compatibles avec la tête de la règle `Head`. Dans ce cas, à condition que les termes de la garde `Guard` s'évaluent à vrai, on remplace cet ensemble de contraintes par celles listées dans le corps `Body`. On peut l'illustrer sur l'exemple suivant :

```
1 :- chr_constraint a/2, b/2.
2 a(X,Y) ⇔ X < Y | b(X,Y).
3
4 %Requete :
5 ?- a(3,4), a(4,3).
6 a(4,3)
7 b(3,4)
8 true .
```

A noter que le corps peut lui-même contenir des termes Prolog, pour faire de l'évaluation par exemple. Une différence importante avec la garde est que l'échec de l'évaluation dans le corps fait échouer la requête alors que cet échec dans la garde se contente de ne pas appliquer la règle. Ainsi même si le stock contenait des contraintes générée, la requête est considérée échouée et son stock vidé.

```
1 :- chr_constraint a/2, b/2.
2 a(X,Y) ⇔ X < Y, b(X,Y).
3
4 %Requete :
5 ?- a(3,4), a(4,3).
6 false.
```

La deuxième forme de règle (Fig. 6.3, ligne 6) est la propagation. Cette règle s'active si l'on a dans le stock, un ensemble de contraintes qui sont compatibles avec la tête de la règle `Head`. Dans ce cas, à condition que les termes de la garde `Guard` s'évaluent à vrai, on ajoute les contraintes listées dans le corps `Body`.

```
1 :- chr_constraint a/2, b/2.
```

```
1 simplification   @ Head ⇔ Guard | Body .
2 simplification_s @ true \ Head ⇔ Guard | Body .
3 propagation      @ Head ⇒ Guard | Body .
4 propagation_s    @ Head \ true ⇔ Guard | Body .
```

FIGURE 6.4 – *Propagation et simplification par simpagation*

```
2 a(X,Y) ⇒ X < Y | b(X,Y).
3
4 %Requete :
5 ?- a(3,4), a(4,3).
6 a(4,3)
7 a(3,4)
8 b(3,4)
9 true .
```

La troisième et dernière forme de règle est la règle de « *simpagation* » (Fig. 6.3, ligne 7), qui est une combinaison de la simplification et de la propagation. Cette règle s'active si l'on a dans le stock, un ensemble de contraintes satisfaisant `Head_k` et `Head_r`. Dans ce cas, à condition que la garde s'évalue à vrai :

— les contraintes choisies pour `Head_k` sont conservées ;
— les contraintes choisies pour `Head_r` sont retirées ;
— les contraintes listées dans le corps sont ajoutées.

La *simpagation* peut être illustrée par l'exemple suivant :

```
1 :- chr_constraint a/2, b/2.
2 a(X,Y) \ a(Y,X) ⇔ X < Y | b(X,Y).
3
4 %Requete :
5 ?- a(3,4), a(4,3).
6 a(3,4)
7 b(3,4)
8 true .
```

On peut noter qu'il est possible d'écrire les règles de propagation et de simplification grâce à la règle de « simpagation », comme illustré en figure 6.4.

La sémantique de CHR telle que définie originellement, laisse l'ordre d'activation des règles en CHR indéterministe. Cependant dans un tel cas, il est difficile d'assurer que le programme aura toujours le même comportement sur une entrée donnée. Pour faciliter le travail du développeur, les implémentations de CHR reposent sur une variante raffinée de la sémantique de CHR [42] où l'ordre

d'activation des règles correspond à l'ordre dans lequel elles apparaissent dans le programme.

Par exemple le programme suivant et la requête associée peuvent selon la sémantique générale produire deux stocks finaux différents :

```
1 :- chr_constraint a/2, b/2, c/2.
2 a(X,Y) ⇔ b(X,Y).
3 a(X,Y) ⇔ c(X,Y).
4
5 %Requete :
6 ?- a(3,4).
7 b(3,4)
8 true .
9
10 % ou :
11 c(3,4)
12 true .
```

En effet, on peut choisir d'appliquer la première ou la deuxième règle en présence d'une contrainte `a(X,Y)`. Dans la sémantique raffinée, seul le premier résultat est possible car l'on applique les règles dans leur ordre d'apparition dans le programme. Par la suite, nous supposons l'utilisation de la sémantique raffinée.

Nous supposons également que les règles de *simpagation* essaient en premier lieu de supprimer les contraintes les plus récemment ajoutées dans le stock, tentant de conserver les plus anciennes. C'est une propriété généralement respectée par les implémentations de CHR [45, Sec.2.4.1], et qui est nécessaire pour la terminaison de notre solveur, comme nous l'expliquerons dans la section 6.5.

À l'exécution, le solveur applique de manière répétitive les règles définies par l'utilisateur sur le stock de contraintes. Lorsqu'un ensemble de contraintes faisant partie du stock a été utilisé pour activer une règle, il ne peut plus l'activer à nouveau avec les mêmes correspondances. En revanche, il peut servir pour une autre règle ou encore pour activer la même règle avec des correspondances différentes. Une contrainte ne peut pas être utilisée pour correspondre avec deux éléments de tête différents.

```
1 :- chr_constraint a/2, b/2.
2 a(X,Y), a(_,_) ⇒ b(X,Y).
3
```

```
?- a(3,4), a(4,3).
a(4,3)
a(3,4)
b(3,4)
b(4,3)
true .
```

L'ajout de nouvelles contraintes par les règles entraîne la possibilité de réactivation de règles pour lesquelles on avait précédemment terminé toutes les activations possibles. L'exécution du programme ne termine donc que si l'on arrête d'ajouter des contraintes capables de réactiver des règles. Nous pouvons noter qu'il est possible d'ajouter une contrainte stoppant toute activation de règle, la contrainte `false`.

```
:- chr_constraint a/2.
a(_,_) ⇒ false.

?- a(3,4).
false.
```

C'est pour cela que l'évaluation échouée d'un prédicat dans le corps fait échouer l'ensemble de la propagation de contraintes : elle introduit `false` dans le stock. L'introduction de `false` permettant de stopper les propagations, nous l'utilisons pour éviter de continuer à propager des contraintes lorsque nous savons déjà qu'une exécution n'est pas autorisée par un modèle, de façon à limiter les calculs.

Si plusieurs règles peuvent être activées pour un stock de contraintes donné, elles seront toutes activées dans l'ordre jusqu'à épuiser les possibilités d'activation (c'est-à-dire jusqu'à ce qu'il n'y ait plus d'autres règles à appliquer). Le solveur CHR ne reviendra jamais sur ses choix. Autrement dit : la sémantique des règles CHR n'inclut pas de mécanique de retour sur trace (*backtracking*). Ce comportement est particulièrement important pour comprendre les choix effectués pour la génération des exécutions de nos programmes. En effet comme on ne peut pas revenir sur nos choix en CHR, une étape le nécessitant sera forcément réalisée par l'intermédiaire d'un programme Prolog.

```
sb([x,y], [T0,T1]) :-
    T0 = [ (st,x,1), (ld,y,V0) ],
    T1 = [ (st,y,1), (ld,x,V1) ].
```

FIGURE 6.5 – *Traduction du programme de la figure 6.1 en Prolog*

```
lwmp_nofc([x,y], [T0,T1]) :-
    T0 = [ (st,x,42),      (st,y,x) ],
    T1 = [ (ld,y,r11:Y ), (ld, r11:Y, r12:X ) ].
```

FIGURE 6.6 – *Code avec dépendance d'adresse grâce aux registres nommés*

6.2.2 Langage considéré

Dans notre formalisation, un programme est modélisé par la liste des noms de variables déclarées[1] qu'il pourra manipuler, sous la forme d'une liste de constantes Prolog, ainsi que la liste, par fil d'exécution, des instructions qu'il va devoir exécuter. Une instruction peut être soit une opération en mémoire, soit une barrière.

Un chargement (respectivement une écriture) est un tuple `(ld, loc, v)` (respectivement `(st, loc, v)`), où `loc` est la position lue (respectivement écrite) et `v` la valeur lue (respectivement écrite). Ces deux paramètres peuvent être fournis comme des variables Prolog. Auquel cas, une des tâches du solveur sera d'effectuer l'unification de ces variables pour déterminer leurs valeurs (ou l'égalité de celles-ci avec d'autres variables). Nous fournissons également l'instruction `(rmw, loc, v1, v2)` pour *reads-modify-write* qui atomiquement, lit `v1` à la position mémoire `loc` puis écrit la valeur `v2`.

Dans ce langage, le programme de la figure 6.1 est implémenté comme présenté dans la figure 6.5.

Optionnellement, il est possible de nommer le registre servant à la lecture ou l'écriture de la valeur, ou encore le registre contenant la position mémoire à laquelle on veut accéder par la syntaxe `registre:Valeur`, où `registre` est une constante Prolog et `Valeur` peut être une variable ou une constante. Cela permet notamment de créer des dépendances de données ou d'adresse entre instructions pour les ordonner. Il est par exemple possible de lire une adresse à une certaine position mémoire l puis d'aller écrire (ou lire) une donnée à cette même position l, comme présenté en figure 6.6.

Dans ce programme, une exécution autorisée est la suivante : dans un premier

1. on parlera aussi de locations, ou positions mémoire pour éviter toute confusion avec les variables Prolog de l'implémentation

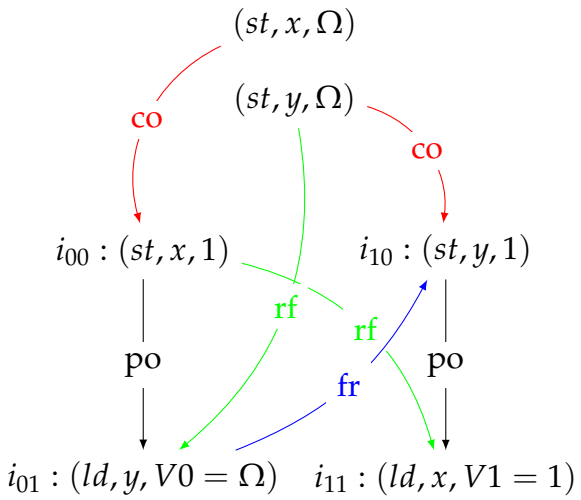


FIGURE 6.7 – *Exemple d'exécution du programme de la figure 6.1*

temps, le fil d'exécution `T0` écrit 42 à la position mémoire `x` puis écrit cette position mémoire à une autre position `y`. Le fil d'exécution `T1` lit ensuite la position mémoire `y` et place la valeur lue `x` dans le registre `r11`. Enfin, il lit la valeur présente à cette position mémoire (42) qu'il place dans le registre `r12`.

Dans notre langage, les barrières (*fences*) ont deux paramètres correspondant à des types d'opérations. Nous les écrivons : `f(t_op1, t_op2)`. Les valeurs `t_op1` et `t_op2` peuvent prendre trois valeurs possibles : `st`, `ld` ou `any`. Le principe de cette opération est d'assurer que toute opération de type `t_op1` précédant la barrière sera ordonnée avant toute opération de type `t_op2` suivant la barrière. Le mot clé `any` correspond ici à « n'importe quel type d'opération ».

Nous pouvons par exemple ajouter une barrière entre les deux écritures du fil d'exécution 1 dans le programme précédent :

```
1 lwmp_fc([x,y], [T0,T1]) :-
2     T0 = [ (st,x,42)     , f(st,st), (st,y,x) ],
3     T1 = [ (ld,y,r11:Y ),            (ld, r11:Y, r12:X ) ].
```

6.2.3 Relations de base entre instructions

Notre formalisation des relations de base caractérisant une exécution est une traduction en CHR de celle proposée par [5]. L'ensemble de relations de base caractérisant une exécution est représenté par 4 notions :

— PO : *program order*,
— CO : *coherency order*,
— RF : *reads-from*,

— FR : *from-read*.

Nous illustrerons ces relations avec un exemple d'exécution du programme de la figure 6.1 présenté en figure 6.7. Dans cette exécution, nous ajoutons deux écritures, préalables à l'exécution du programme, qui initialisent les valeurs de x et y à Ω, la valeur indéfinie. Nous considérons un scénario d'exécution où la lecture i_{01} trouve la valeur Ω à la position mémoire y, et i_{11} lit la valeur 1 depuis x.

Par la suite nous utiliserons le terme *exécution candidate* pour désigner une exécution dont la validité par rapport à un modèle mémoire donné n'est pas encore vérifiée. Le terme *exécution autorisée* par un modèle mémoire donné désigne une exécution dont on a déterminé qu'elle est acceptable sous ce modèle mémoire.

Deux opérations mémoire i_1 et i_2 sont en relation $\mathrm{PO}(i_1, i_2)$ quand i_1 apparaît avant i_2 dans l'ordre textuel du programme. Cela signifie également qu'elles font partie du même fil d'exécution. Par exemple, dans la figure 6.1, on a $\mathrm{PO}(i_{00}, i_{01})$ et $\mathrm{PO}(i_{10}, i_{11})$. Il est important de noter que les barrières n'interviennent pas dans la relation PO, elles donnent des garanties au niveau de l'exécution qui seront modélisées par d'autres relation. Donc, si une barrière séparait ces opérations, la relation PO serait quand même présente entre elles et il n'y aurait pas de relation PO concernant la barrière. C'est cette relation qui pourra principalement être relâchée par les modèles mémoire. Notons que pour les besoins de notre solveur, la relation PO peut être, dans un premier temps, définie sur des opérations successives, et ensuite entièrement calculée par la clôture transitive IPO définie dans la section 6.3.

La relation CO ordonne les écritures à une position mémoire donnée. Intuitivement cela correspond, dans l'exécution, à une histoire globale des écritures telles qu'on les verrait si l'on pouvait observer chaque écriture au niveau de la mémoire. Dans notre exemple les écritures à la position x sont ordonnées par CO tel que (st, x, Ω) a lieu avant $(st, x, 1)$. Dans ce programme, c'est le seul ordre possible car l'initialisation est toujours effectuée en premier pour une position mémoire donnée. Une nouvelle fois, nous ne définissons ici la relation CO que sur les écritures successives, la clôture transitive étant calculée par les modèles qui le nécessitent.

La relation $\mathrm{RF}(i_W, i_R)$ détermine quelle écriture i_W a placé la valeur lue par une lecture i_R à une position mémoire donnée. Elle assigne une écriture unique à chaque lecture. Dans notre exemple d'exécution, RF ordonne $(st, x, 1)$ et $(ld, x, V1 = 1)$.

```
apply_generic_model(VARS, THREADS) :-
    enrich_prog(0,THREADS,EnrichedThreads),
    compute_pos(EnrichedThreads),
    append(EnrichedThreads, PRG),
    compute_rel_all_threads(data, EnrichedThreads),
    compute_rel_all_threads(addr, EnrichedThreads),
    LOCS = [undefined | VARS],
    ops_to_each(st, LOCS, PRG, STORES),
    ops_to_each(ld, LOCS, PRG, LOADS),
    permute_stores(STORES, PSTORES),
    compute_all_cos(PSTORES),
    compute_all_rfs(LOADS, PSTORES).
```

FIGURE 6.8 – *Génération des exécutions candidates (`generic_model.pl`)*

La relation FR est dérivée de CO et RF. Elle associe à chaque lecture la liste des écritures (possiblement vide) qui vont successivement ré-écrire la position mémoire à laquelle la lecture a eu lieu. Si l'on prend la liaison RF((st, y, Ω), $(ld, y, V0 = \Omega)$), Comme CO indique que $(st, y, 1)$ a lieu après (st, y, Ω), alors que $(ld, y, V0 = \Omega)$ lit bien depuis cette dernière, nous savons que $(st, y, 1)$ a lieu après la lecture. Donc nous avons FR$((ld, y, V0 = \Omega), (st, y, 1))$.

Puisque PO est définie par le programme et FR est dérivée de CO et RF, une exécution candidate est définie par une combinaison de CO et de RF, c'est-à-dire une combinaison de permutations des écritures pour chaque position mémoire et un choix d'écriture pour chaque lecture. La figure 6.7 est donc un exemple d'exécution candidate, une autre exécution candidate serait obtenue en retirant la relation RF entre $(st, x, 1)$ et $(ld, x, V1 = 1)$ et en créant une relation RF entre (st, x, Ω) et $(ld, x, V1 = \Omega)$.

6.3 Modèle générique

Dans cette section nous présentons le modèle le plus générique de modèle mémoire faible. Ce modèle accepte toute permutation d'écriture et de lecture comme une exécution autorisée.

Le modèle générique nous permet de produire toutes les exécutions candidates d'un programme en supposant un modèle mémoire très faible qui nous donne comme seules garanties que :

— deux écritures en mémoire ne peuvent avoir lieu en même temps,

— pour toute position mémoire, sa valeur est initialement indéterminée,
— si l'on associe une lecture à une écriture par la relation RF, la valeur lue est égale à la valeur écrite.

Les exécutions générées sont ensuite déterminées comme autorisées ou interdites par les modèles mémoire spécifiques.

Le processus de génération est illustré par la figure 6.8 et peut être résumé par la séquence d'actions suivante :

— ajout d'identifiants aux instructions du programme (ligne 2),
— extraction de PO (ligne 3),
— recherche des dépendances de données et d'adresses (lignes 5–6),
— extraction pour chaque position mémoire de l'ensemble des lectures et écritures (lignes 8–9),
— génération de CO par combinaison de permutations des écritures pour chaque position mémoire (lignes 10–11),
— génération de RF par association d'une écriture possible à chaque lecture (ligne 12).

Pendant cette extraction, la relation FR, ou encore la clôture transitive de PO et CO, sont déduites à la volée à l'aide de règles CHR activées par l'ajout progressif des contraintes. Les règles indiquées par les modèles mémoire spécifiques sont également activées de cette manière. Cela permet très tôt de détecter des ensembles d'exécutions interdites par le modèle, nous évitant ainsi de les générer dans leur entièreté.

6.3.1 Extraction de PO et des dépendances

Dans un premier temps, nous enrichissons les instructions avec des informations supplémentaires : un identifiant du fil d'exécution d'origine et un identifiant d'instruction au sein de ce fil d'exécution. Ces instructions sont immédiatement ajoutées au stock de contraintes CHR.

Les opérations mémoire `(OP,Loc,Val)` deviennent des contraintes CHR `i(N_thread, N_inst, OP, Loc, Val)`, les barrières `f(OP1,OP2)` deviennent `fence(N_thread, N_inst, OP1, OP2)`. Par exemple, les instructions de la figure 6.5 sont traduites en contraintes : `i(0,0,st,x,1)`, `i(0,1,ld,y,R0)`, `i(1,0,st,y,1)` et `i(1,1,ld,x,R1)`.

Nous devons également remplacer les opérations *read-modify-write* par une lecture et une écriture successives. Nous ajoutons en même temps une contrainte CHR `rmw`(i_{read}, i_{write}). Cette contrainte permet, par l'utilisation d'une règle que

nous présenterons en section 6.3.3, de garantir l'atomicité des deux opérations effectuées en refusant les exécutions qui violent cette atomicité.

La création des contraintes `i`, `fence` et `rmw` et l'extraction de PO sont assurées par le code Prolog présenté en annexe dans la section A.1. L'extraction de PO consiste simplement à prendre les instructions deux à deux et les lier par une contrainte CHR `po`, dans le même temps nous créons les contraintes qui numérotent les instructions et les barrières.

En interne de chaque fil d'exécution, outre la relation PO, nous devons extraire deux types de dépendances : les dépendances de données et les dépendances d'adresse. Une dépendance existe depuis une opération i_1 vers une opération i_2 si i_1 lit une valeur v qui est ensuite utilisée par i_2. Si i_2 est une écriture dont la valeur écrite dépend de v, on a une dépendance de donnée $\texttt{data}(i_1, i_2)$, si i_2 est une opération mémoire dont l'adresse accédée dépend de v, on a une dépendance d'adresse $\texttt{addr}(i_1, i_2)$.

Dans la figure 6.6, la dépendance d'adresse présente produit donc une contrainte : `addr( (1,0,ld,y,r11:Y) , (1,1,ld,r11:Y,r12:X) )`. Si `T1` écrivait ensuite cette valeur à une position mémoire `z`, nous aurions en plus `data( (1,1,ld,r11:Y,r12:X) , (1,2,st,z,r12:X) )`.

Nous présentons ces opérations en annexe dans la section A.1. Dans les modèles mémoires que nous présentons par la suite, ces dépendances sont ignorées car elles ne sont pas nécessaires. Cependant, pour d'autres modèles, elles peuvent l'être. Par exemple, pour définir le modèle ARM (que nous ne présentons pas dans cette thèse), nous avons besoin des dépendances d'adresses et de valeurs. Nous voulons que notre solveur puisse servir pour définir tout modèle, c'est pourquoi nous générons ces contraintes.

6.3.2 Extraction de CO et RF

CO est une relation qui ordonne les écritures pour chaque position mémoire. Il nous faut donc extraire l'ensemble d'écritures du programme pour pouvoir produire la totalité des permutations candidates de cette relation.

Pour former une exécution candidate d'un programme, nous commençons par former une combinaison des permutations des écritures à chaque position mémoire. (Celles-ci seront ensuite complétées par la combinaison des associations lecture/écriture admissibles selon les adresses et valeurs mentionnées dans ces opérations.) A chaque permutation, nous ajoutons une écriture d'une valeur indéterminée `undefined` qui est placée en tête de liste et qui représente la va-

leur en mémoire avant toute autre écriture. Cette instruction est de la forme `i(-1,-1,st,LOC,undefined)`. Les contraintes CO correspondantes sont ensuite extraites. Le code Prolog de cette opération est présenté en annexe dans la section A.3.

L'extraction des contraintes RF nécessite à la fois l'ensemble des lectures et l'ensemble des écritures à chaque position mémoire. Les combinaisons de couples écriture/lecture à une position donnée vont être ajoutées et combinées avec chaque ensemble de contraintes CO généré.

Si la lecture est effectuée à une position mémoire `undefined`, l'écriture correspondante est considérée comme étant également de valeur `undefined`. Dans ce cas, une contrainte `rf(undefined, Load)` est générée. Par la suite, une règle CHR ajoutera dans ce cas une contrainte `rte`, pour *runtime-error*, qui nous signifiera que l'exécution comporte un accès indéfini (et cette exécution pourra être autorisée ou interdite par le modèle).

Sinon, on génère à chaque fois une contrainte de la forme :

```
1  rf(i(T1,N1,st,Loc,Val), i(T2,N2,ld,Loc,Val)
```

À noter que dans ce dernier cas, en vérité, Loc et Val peuvent, tout en étant égaux sur un plan valeur, différer par la présence d'une précision de registre. Nous ne donnons cette forme textuellement que pour donner l'idée générale. La forme détaillée est donnée par les prédicats Prolog produisant le calcul de RF dans la figure A.7 présentée en section A.4 de l'annexe.

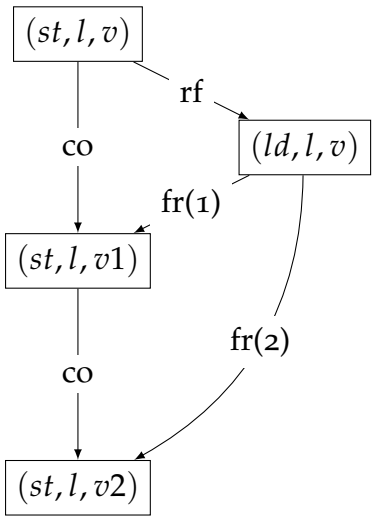
6.3.3 Production de FR et IPO, atomicité de RMW

Nous dérivons les relations FR et IPO des relations précédemment décrites.

Pour rappel, la relation FR associe à une lecture, la liste des écritures qui, d'après CO, vont successivement écraser la valeur lue par cette opération de lecture. Nous l'implémentons par les deux règles CHR mentionnées dans la partie droite de la figure 6.9.

La première règle nous indique que si une lecture `LD` lit la valeur écrite par une écriture `ST`, et que d'après `CO`, l'écriture `ST1` vient écraser la valeur lue, alors on a une relation FR entre `LD` et `ST1`. Cela correspond, dans l'exemple d'exécution en partie gauche de la figure, au repérage de la relation FR numérotée 1.

La seconde règle produit les relations FR suivantes à partir du dernier FR calculé. Si `ST` est une écriture qui écrase la valeur lue par `LD`, et que d'après CO, `ST2` écrase la valeur écrite par `ST`, alors on a également une relation FR entre




```
1 rf(ST,LD), co(ST,ST1) ⇒ fr(LD,ST1).
2 fr(LD,ST), co(ST,ST2) ⇒ fr(LD,ST2).
```

FIGURE 6.9 – *Calcul de FR (`generic_model.pl`)*

```
1 % PO transitive closure
2 ipo(I0,I1) \ ipo(I0,I1) ⇔ true.
3 po(I0,I1) ⇒ ipo(I0,I1).
4 ipo(I0,I1), ipo(I1,I2) ⇒ ipo(I0,I2).
```

FIGURE 6.10 – *Calcul de IPO (`generic_model.pl`)*

`LD` et `ST2`. Cela correspond, dans l'exemple d'exécution en partie gauche, au repérage de la relation FR numérotée 2.

On définit la relation IPO comme la clôture transitive de la relation PO. Celle-ci nous est nécessaire pour la définition de certains modèles mémoire. On sait qu'elle est acyclique par définition. On la produit par l'ajout des règles illustrées en figure 6.10. Pour chaque contrainte PO existant entre deux instructions, on ajoute une nouvelle contrainte IPO (ligne 3). On applique ensuite la transitivité avec la règle en ligne 4. On supprime les éventuels doublons par l'ajout de la règle en ligne 2.

Finalement, connaissant RF, CO et FR, nous pouvons ajouter les règles CHR permettant de garantir l'atomicité de l'opération RMW, composée d'une lecture suivie d'une écriture à une position mémoire. Pour garantir l'atomicité, nous devons simplement nous assurer qu'aucune écriture ne puisse écraser la valeur lue par la lecture de l'opération de *read-modify-write* avant que son écriture l'ait elle-même écrasée. Nous exprimons cette règle par la ligne 1 de la figure 6.11. Les comportements que nous interdisons par cette règle sont illustrés par la figure 6.12.

```
1 rmw(LD,ST), fr(LD,CST), co(CST,ST) ⇔ false.
```

FIGURE 6.11 – *Règles de cohérence de RMW (`generic_model.pl`)*

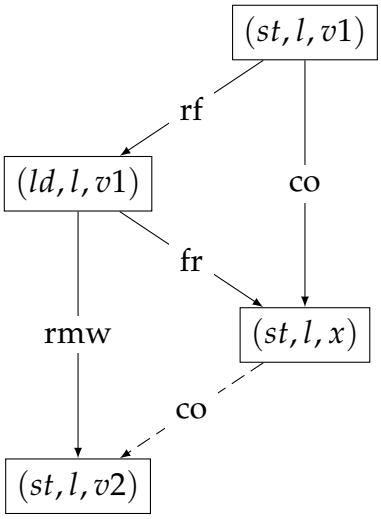


FIGURE 6.12 – *Atomicité de RMW : scénarii interdits*

6.3.4 Dérivation des barrières

Les barrières forcent l'ordre de certaines paires d'opérations mémoire au sein d'un fil d'exécution. Le développeur (ou le cas échéant, le compilateur) place donc ces barrières dans le code pour assurer que l'ordre de certaines opérations sera bien respecté. Selon l'architecture du processeur, ces barrières peuvent être plus ou moins précises, tenant compte des types d'accès à la mémoire effectués, pour déterminer si oui ou non ils peuvent être réordonnés. C'est pourquoi les barrières dans notre langage demandent deux paramètres qui sont des types d'accès à la mémoire : `f(type1, type2)`. Toutes les opérations de type `type1` précédant la barrière sont strictement ordonnées avant toutes les opérations de type `type2` suivant la barrière.

Nous encodons ce comportement par dérivation des contraintes CHR déjà ajoutées au store, à savoir les instructions enrichies avec les numéros de fil d'exécution et d'instruction, comme illustré en figure 6.13.

Les prédicats `wait_for` (resp. `block`) nous permettent d'exprimer qu'une barrière doit attendre (resp. bloquer) l'exécution d'une instruction d'un certain type, si elle se trouve avant (resp. après) la barrière dans un fil d'exécution donné.

Ensuite la règle CHR, lignes 7 à 10, détermine pour chaque paire d'opérations mémoire et chaque barrière, si les opérations mémoire en question sont effectivement ordonnées par la barrière. Il faut donc s'assurer que la première instruction est attendue par la barrière en question, qui bloque elle-même l'exé-

```
wait_for(fence(T,NF,any,_), i(T,NI,_,_,_)) :- NI =< NF.
wait_for(fence(T,NF,O,_), i(T,NI,O,_,_)) :-   NI =< NF.

block(fence(T,NF,_,any), i(T,NI,_,_,_)) :- NI >= NF.
block(fence(T,NF,_,O), i(T,NI,O,_,_)) :-   NI >= NF.

i(T,N0,O0,L0,V0), fence(T,N,OF0,OF1), i(T,N1,O1,L1,V1) ⇒
   wait_for(fence(T,N,OF0,OF1), i(T,N0,O0,L0,V0)),
   block(fence(T,N,OF0,OF1), i(T,N1,O1,L1,V1))
   | barrier(i(T,N0,O0,L0,V0), i(T,N1,O1,L1,V1)).
```

FIGURE 6.13 – *Calcul des barrières (`generic_model.pl`)*

cution de la deuxième instruction. Dans ce cas, on ajoute une contrainte CHR `barrier` dans le store.

6.4 Modèles spécifiques

Le modèle générique nous permet de produire toutes les exécutions candidates d'un programme en supposant un modèle mémoire très faible. Une exécution est définie par une combinaison de CO et RF. D'autres relations comme l'ordre du programme PO ou les dépendances d'adresses et de valeurs ne caractérisent pas une exécution candidate.

En décrivant un modèle mémoire, le principe est justement de déterminer quelles relations doivent être prises en compte pour accepter ou non une exécution candidate comme exécution autorisée.

Pour cela, nous allons définir une nouvelle relation sous la forme d'une contrainte CHR homonyme au modèle considéré (pour SC, `sc(I0,I1)`, pour TSO, `tso(I0,I1)` ...). Cette contrainte est définie comme l'union des relations que le modèle assure respecter. Pour chaque exécution candidate, nous construisons la clôture transitive de la nouvelle contrainte créée. Si celle-ci exhibe un cycle, l'exécution est interdite.

6.4.1 Détection de cycles

Pour un modèle mémoire donné, nous définissons la relation R qu'il va devoir respecter à propos des opérations mémoire dans une exécution, par l'union de sous-ensembles des relations de base du modèle générique. Par exemple, SC respecte PO, CO, RF et FR dans leur intégralité. En revanche, TSO ne respecte

```
1 :- chr_constraint rel/3, path/3, cycle/2.
2
3 rel(R, Begin, End) \ rel(R, Begin, End) ⇔ true.
4 path(R, Begin, End) \ path(R, Begin, End) ⇔ true.
5 path(R, Begin, End), rel(R, End, Begin) ⇒ cycle(R, Begin).
6
7 rel(R, Begin, End) ⇒ inf(Begin, End) | path(R, Begin, End).
8 path(R, Begin, End), rel(R, End, Next) ⇒
9                     inf(Begin, Next) | path(R, Begin, Next).
```

FIGURE 6.14 – *Détection de cycles (`cycle.pl`)*

qu'un sous-ensemble de PO. Par la relation R, on obtient un moyen de déterminer quelle instruction doit avoir lieu avant quelle autre, et ceci transitivement. Pour détecter une exécution interdite par le modèle, nous devons explorer les chaînes de contraintes entre les instructions. Si l'une de ces chaînes exhibe un cycle, cela signifie que, d'après le modèle, une action doit avoir lieu avant elle-même, ce qui est incohérent.

Cette détection de cycle est produite par l'intermédiaire de contraintes CHR paramétrées par la relation R, et d'un ensemble de règles pour les traiter. Ces règles produisent la clôture transitive de la relation R. Contrairement au calcul de IPO précédemment mentionné, la clôture peut cette fois comprendre des cycles. Ceux-ci doivent être correctement détectés pour que le calcul de la clôture termine. Les règles de détection sont illustrées par la figure 6.14.

Nous utilisons trois contraintes CHR. La contrainte `rel(R,Begin,End)` indique que les instructions `Begin` et `End` sont ordonnées par `R`. Cette représentation de R`(Begin,End)` permet de rendre la détection de cycle générique sur R. La clôture transitive de la relation `R` entre les instructions `Begin` et `End` est progressivement calculée par la contrainte `path(R,Begin,End)`. Elle signifie qu'il existe un chemin par R entre *Begin* et *End*, ces derniers devant être différents. Finalement, la contrainte `cycle(R, Begin)` indique qu'en partant de l'instruction `Begin`, on a trouvé un chemin non trivial de contraintes `R` revenant à cette instruction.

Nous définissons ensuite les règles qui vont propager les contraintes imposées par le modèle. Nous devons également ajouter quelques règles de simplification et *simpagation* qui vont limiter la quantité de traces générées en évacuant les doublons et en arrêtant les propagations sur les traces qui ont déjà exhibé un cycle.

Les deux premières règles que nous définissons permettent d'éviter l'explo-

sion du nombre de traces. La première règle de *simpagation* (ligne 3) permet d'évacuer les doublons pour la relation R.

De même, la règle en ligne 4 produit la suppression de chemins de contraintes avec les mêmes extrémités. En effet, s'il existe deux chemins, même différents, menant d'une instruction i_1 à une instruction i_2, il n'est pas utile de conserver cette distinction pour la détection de cycle. S'il existe un cycle passant par l'un d'eux, il existe trivialement un cycle passant par l'autre. C'est d'ailleurs pourquoi nous ne conservons pas le chemin complet dans la contrainte CHR `path`.

La règle de propagation ajoutée en ligne 5 permet, lorsque l'on détecte un chemin, ainsi qu'une contrainte le refermant, de créer une nouvelle contrainte indiquant un cycle. Dans la définition d'un modèle, l'ajout d'une règle CHR introduisant `false` dans le stock de contraintes si un cycle est détecté permet d'arrêter la propagation. En effet, nous cherchons à ne conserver que les exécutions autorisées. Sans cette règle, nous obtenons toutes les exécutions candidates, où celles interdites par le modèle aboutissent à un stock de contraintes contenant des cycles présents dans le graphe de l'exécution.

Les règles suivantes servent à propager les relations d'ordre entre les instructions.

La règle lignes 8 et 9 permet d'ajouter un nouveau chemin chaque fois que l'on trouve une contrainte qui peut faire grandir un chemin existant.

Pour optimiser la recherche de chemins et éviter de détecter le même cycle plusieurs fois, nous supposons une relation d'ordre arbitraire sur les instructions qui est exprimée à l'aide du prédicat Prolog `inf`, qu'on peut par exemple définir comme un ordre lexicographique sur les numéros de fil d'exécution, puis sur les numéros d'instruction. L'optimisation consiste à n'ajouter une contrainte à un chemin que si d'après l'ordre `inf`, elle est supérieure au début du chemin. En effet, nous calculons les cycles depuis toutes les instructions à la fois. Avec cette optimisation, nous ne considérons ainsi que les chemins dont l'origine est strictement inférieure aux instructions suivantes. Par ailleurs, il nous suffit de chercher des cycles sans retour, ne repassant pas plusieurs fois par le même élément. Par conséquent, il nous suffit de partir de l'instruction minimale (selon `inf`) car dans un tel cycle, il existe une instruction strictement inférieure à toutes les autres.

La création du début de chemin, ligne 7, impose la même contrainte au démarrage du chemin. Nous ne commençons donc un chemin que si l'origine est inférieure à l'instruction suivante. La correction de cette optimisation sera davantage discutée dans la section 6.5.

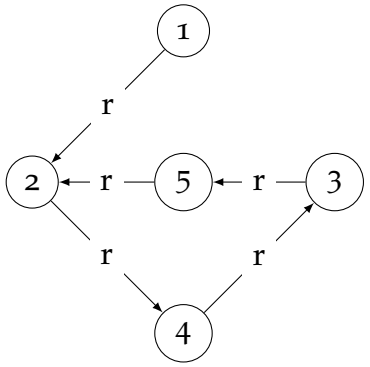


FIGURE 6.15 – *Exemple de graphe : configuration en « 6 ».*

La figure 6.15 présente un exemple de graphe où les nœuds sont simplement étiquetés par des entiers et reliés entre eux par une relation *r*. Nous pouvons dérouler la propagation des contraintes par les règles précédemment définies pour la requête suivante :

```
?- rel(r,5,2), rel(r,2,4), rel(r,4,3), rel(r,1,2), rel(r,3,5).
```

Dans un premier temps, on ajoute la contrainte `rel(r,5,2)`, aucune règle ne pouvant être activée par cette contrainte (la garde de l'ajout n'est pas validée), nous n'effectuons rien de plus pour cet ajout.

```
STORE={ rel(r,5,2) }
```

Ensuite, nous ajoutons la contrainte `rel(r,2,4)` qui active la règle d'ajout de chemin et produit donc également `path(r,2,4)`. Ce chemin ne peut pas être continué pour l'instant, nous n'effectuons rien de plus.

```
STORE={ rel(r,5,2),
        rel(r,2,4),path(r,2,4) }
```

Nous ajoutons `rel(r,4,3)`, elle ne crée pas de nouveau départ de chemin, mais peut allonger le chemin `path(r,2,4)`. On ajoute donc une contrainte `path(r,2,3)`.

```
STORE={ rel(r,5,2),
        rel(r,2,4),path(r,2,4),
        rel(r,4,3),path(r,2,3) }
```

Nous ajoutons ensuite la contrainte `rel(r,1,2)`. Cela ajoute un chemin `path(r,1,2)` qui peut être étendu avec `rel(r,2,4)`, ajoutant la contrainte `path(1,4)`, qui peut lui-même être étendu avec `rel(r,4,3)`, créant `path(r,1,3)`.

```
STORE={ rel(r,5,2),
        rel(r,2,4),path(r,2,4),
        rel(r,4,3),path(r,2,3),
        rel(r,1,2),path(r,1,2),path(r,1,4),path(r,1,3) }
```

Nous ajoutons la dernière contrainte `rel(r,3,5)`. Elle ajoute un chemin `path(r,3,5)`, nous amenant à un état où le store est le suivant :

```
STORE={ rel(r,5,2),
        rel(r,2,4),path(r,2,4),
        rel(r,4,3),path(r,2,3),
        rel(r,1,2),path(r,1,2),path(r,1,4),path(r,1,3),
        rel(r,3,5),path(r,3,5) }
```

La contrainte `rel(r,3,5)` a été utilisée pour activer la règle d'ajout de chemin. Elle peut maintenant être utilisée pour étendre un chemin existant. Dans le store, nous avons deux chemins qu'il est possible d'étendre avec cette contrainte : `path(r,2,3)` et `path(r,1,3)`. La sémantique raffinée ne donne pas de priorité à l'une des deux.

Si on choisit `path(r,2,3)`, on crée `path(r,2,5)`, qui est ensuite utilisée avec `rel(r,5,2)` pour activer la détection de cycle, donc cela termine en détectant le cycle.

Si on choisit `path(r,1,3)`, on crée `path(1,5)`, qui est également utilisée avec `rel(r,5,2)`, créant une contrainte `path(r,1,2)`, cette contrainte existe déjà, et donc la règle de suppression est activée. Cette règle de suppression donne comme garantie, dans la majorité des implémentations de CHR (dont celle que nous utilisons, SWI-Prolog) que la contrainte la plus ancienne sera conservée au détriment de la plus récente, nous reviendrons sur ce point dans la section 6.5. Il en résulte que cette nouvelle contrainte n'est pas ajoutée au stock de contrainte et donc que pour la contrainte `path(r,1,3)` la propagation s'arrête, donnant la possibilité au solveur d'explorer la propagation sur la contrainte `path(r,2,3)` précédemment citée (et détectant le cycle).

6.4.2 Le modèle SC

Le modèle séquentiellement consistant est le modèle le plus fort. Il est aussi le plus simple à modéliser avec nos contraintes CHR, puisque l'on a seulement besoin de préciser que ce modèle respecte toutes les relations PO, CO, RF et FR. Cette modélisation est illustrée en figure 6.16. Les inclusions en lignes 1 et

```
1  :- include(generic_model).
2  :- include(cycle).
3
4  % SC does not consider any fences
5  fence(_,_,_,_) ⇔ true.
6  % Nor address/data dependency
7  addr(_,_) ⇔ true.
8  data(_,_) ⇔ true.
9
10 % If we discover a cycle in SC, it is an incoherency
11 cycle(sc, L) ⇔ false.
12
13 po(I0,I1) ⇒ rel(sc, I0, I1).
14 co(I0,I1) ⇒ rel(sc, I0, I1).
15 rf(I0,I1) ⇒ rel(sc, I0, I1).
16 fr(I0,I1) ⇒ rel(sc, I0, I1).
```

FIGURE 6.16 – *Modèle SC (`sc.pl`)*

2 ajoutent respectivement la définition du modèle générique et des règles de détection de cycles.

Pour chaque contrainte PO, CO, RF ou FR rencontrée, on ajoute une contrainte `rel` correspondant à la relation SC (lignes 14–17). Les règles de détection de cycle effectuent le travail de recherche des cycles dans la clôture transitive de SC. Comme mentionné précédemment, on ajoute également une règle de simplification (ligne 12) qui arrête la propagation en introduisant `false` dans le stock de contraintes en cas de cycle et qui est utilisée pour l'élagage des branches de l'arbre de recherche (que nous détaillerons dans la section 6.6).

Nous pouvons noter que comme le modèle SC respecte la relation PO, les barrières et dépendances de données ou d'adresse qui ont été calculées n'apportent pas de nouvelles informations à propos de l'ordre d'exécution des instructions qu'elles mentionnent. C'est pourquoi nous utilisons des règles de simplification (lignes 5,7,8) pour supprimer [2] les contraintes correspondantes, permettant ainsi de ne pas encombrer le stock de contraintes avec des informations inutiles.

Par exemple, nous pouvons détecter que le comportement du programme de la figure 6.1 consistant à lire depuis les initialisations dans chacun des fils d'exécution est interdit par le modèle SC, comme l'illustre la figure 6.17. Nous

2. Ce choix de rajouter puis explicitement supprimer certaines contraintes facilite la présentation. Une implémentation optimisée évitera de générer les contraintes CHR de dépendances et de barrières pour SC

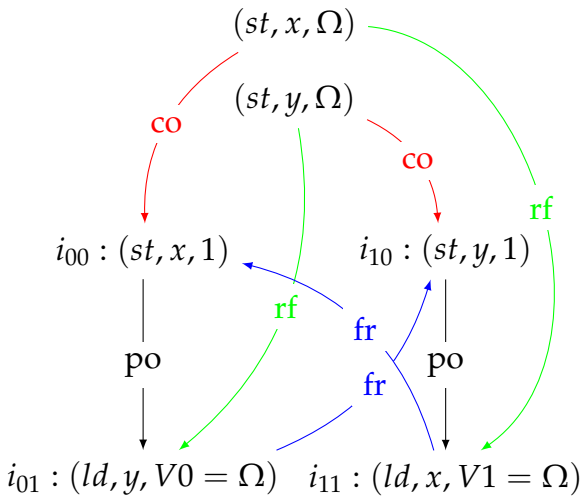


FIGURE 6.17 – *Exécution interdite par le modèle SC pour le programme de la figure 6.1*

avons bien la présence d'un cycle avec les relations PO et FR maintenues par le modèle SC.

6.4.3 Relation SC par adresse

Si l'on s'intéresse maintenant à des modèles plus faibles comme ceux des architectures communes, une première relation, présente dans la majorité des architectures [1] assure une cohérence basique sur l'exécution du code sur un seul processeur.

Elle consiste à assurer pour chaque processeur, la cohérence des accès pour chaque position mémoire (la bonne sérialisation des écritures pour une position mémoire) [26]. Si un processeur écrit v à la position l et lit ensuite v' à la position l, alors les écritures correspondantes w de v et w' de v' doivent être réalisées dans cet ordre : w' ne peut pas se trouver avant w. Suivant la formalisation définie par [5], nous nommons cette relation `sc_per_loc`. Vérifier sa cohérence consiste également à vérifier l'absence de cycle dans la relation.

Nous illustrons la définition des règles de cohérence correspondantes en figure 6.18. La relation `sc_per_loc` est définie comme étant l'union de CO, RF, FR et `po_loc` (lignes 13–16). La cohérence mono-processeur impose que cette relation soit acyclique. La relation `po_loc` (lignes 10–11) correspond à l'ordre PO mais restreint à des opérations s'effectuant à une position mémoire commune. Nous la construisons à partir de IPO (la clôture transitive de PO).

Comme dans la définition de SC, nous définissons un cycle comme étant une exécution rejetée (ligne 8).

```
1  :- use_module(library(chr)).
2  :- include(cycle).
3  :- chr_constraint po_loc/2.
4
5  % Uniproc Check : sc per location, acyclicity of
6  %                 co \cup rf \cup fr \cup po_loc
7
8  cycle(sc_per_loc, L) ⇔ false.
9
10 ipo(i(N0,Id0,O0,L,V0), i(N1,Id1,O1,L,V1)) ⇒
11   po_loc(i(N0,Id0,O0,L,V0), i(N1,Id1,O1,L,V1)).
12
13 co(I0,I1)     ⇒ rel(sc_per_loc, I0, I1).
14 rf(I0,I1)     ⇒ rel(sc_per_loc, I0, I1).
15 fr(I0,I1)     ⇒ rel(sc_per_loc, I0, I1).
16 po_loc(I0,I1) ⇒ rel(sc_per_loc, I0, I1).
```

FIGURE 6.18 – *Modèle SC par adresse (`uniproc.pl`)*

Par exemple, nous pouvons prendre un programme séquentiel qui écrit la valeur 1 à la position mémoire `x` puis lis cette position mémoire. La relation SC par adresse nous garantit que nous ne pouvons pas lire la valeur d'initialisation, comme l'illustre la figure 6.19.

Nous utiliserons cette relation comme base nécessaire pour chacun des modèles mémoire définis par la suite.

6.4.4 Les modèles TSO et PSO

Les modèles TSO et PSO relaxent l'ordre (PO) du programme pour les paires d'instructions qui commencent par une opération d'écriture. PSO est plus faible que TSO. Ce dernier ne relâche que les paires écriture-lecture tandis que le premier relâche l'ordre de toute paire d'instructions commençant par une écriture. Cette relaxation s'applique transitivement et est donc bâtie à partir de la relation IPO, comme l'illustrent les figures 6.20 (lignes 14-15) et 6.21 (lignes 14-15). L'idée ici est que toute création d'une paire *preserved program order* (`ppo`) commençant par une écriture et finissant par une lecture (respectivement, n'importe quelle instruction pour PSO), doit être immédiatement supprimée car cet ordre n'est pas respecté par le modèle.

Nous ajoutons la relation *reads-from-external*. En effet, les écritures effectuées par un processeur passent d'abord dans un tampon d'écriture interne. Si le processeur demande une lecture de la position mémoire écrite, ce tampon fait aussi

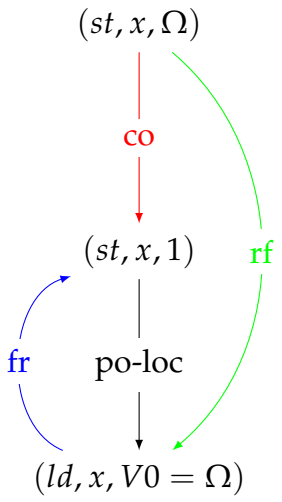


FIGURE 6.19 – *Exécution interdite par la relation* `sc_per_loc` *lors d'une lecture depuis une cellule précédemment écrite par le même fil d'exécution*

office de cache. Une telle lecture étant invisible par les autres processeurs, nous ne devons pas la prendre en compte dans l'ordre global des opérations du processeur [1]. Cette relation est créée par l'usage d'un prédicat Prolog et d'une règle CHR gardée. Deux opérations sont liées par une relation R externe, si elles sont liées par la relation R et que leur numéro de fil d'exécution est différent (ligne 8 dans les deux figures 6.20 et 6.21). Il existe une contrainte RF externe entre deux instructions si elles sont liées par RF et valident le prédicat `ext` qui vérifie que les numéros de fils sont différents. Dans ce cas, on ajoute donc une contrainte RFE (*reads-from external*).

Les deux modèles respectent la relation SC par adresse dont nous incluons les règles par l'inclusion du fichier `uniproc`.

Les barrières de ces modèles sont des barrières fortes et elles sont toutes traitées de la même manière : elles restaurent PO quand elles sont présentes. Nous assurons donc que les barrières sont toujours préservées par ces deux modèles (voir ligne 18 dans les deux figures).

Avec le modèle TSO, l'exécution du programme de la figure 6.1 interdite par le modèle SC (cf. figure 6.17) est ici autorisée car les contraintes `po` qui fermaient le cycle ne font pas partie du `ppo` que TSO considère, comme l'illustre la figure 6.22. Si nous voulons empêcher ce schéma, nous devons ajouter des barrières mémoire (comme par exemple dans le programme de la figure 6.23), ce qui nous permet d'interdite cette exécution comme le montre la figure 6.24.

```
:- include(uniproc).
:- chr_constraint rfe/2.
:- chr_constraint ppo/2.

% TSO implementation does not need addr and data deps :
addr(_,_) ⇔ true.
data(_,_) ⇔ true.

ext(i(N0,_,_,_,_), i(N1,_,_,_,_)) :- \+ N0 = N1.
rf(I0,I1) ⇒ ext(I0,I1) | rfe(I0, I1).

cycle(tso, L) ⇔ false.

% po-WR pairs are not preserved by TSO, so we remove them
ppo(i(_,_,st,_,_), i(_,_,ld,_,_)) ⇔ true.
ipo(I0,I1) ⇒ ppo(I0,I1).

barrier(I0,I1) ⇒ rel(tso, I0, I1).
ppo(I0,I1)     ⇒ rel(tso, I0, I1).
rfe(I0,I1)     ⇒ rel(tso, I0, I1).
co(I0,I1)      ⇒ rel(tso, I0, I1).
fr(I0,I1)      ⇒ rel(tso, I0, I1).
```

FIGURE 6.20 – *Modèle TSO (`tso.pl`)*

```
:- include(uniproc).
:- chr_constraint rfe/2.
:- chr_constraint ppo/2.

% PSO implementation does not need addr and data deps :
addr(_,_) ⇔ true.
data(_,_) ⇔ true.

ext(i(N0,_,_,_,_), i(N1,_,_,_,_)) :- \+ N0 = N1.
rf(I0,I1) ⇒ ext(I0,I1) | rfe(I0, I1).

cycle(pso, L) ⇔ false.

% po-(W/Any) pairs are not preserved by PSO
ppo(i(_,_,st,_,_), _) ⇔ true.
ipo(I0,I1) ⇒ ppo(I0,I1).

barrier(I0,I1) ⇒ rel(pso, I0, I1).
ppo(I0,I1)     ⇒ rel(pso, I0, I1).
rfe(I0,I1)     ⇒ rel(pso, I0, I1).
co(I0,I1)      ⇒ rel(pso, I0, I1).
fr(I0,I1)      ⇒ rel(pso, I0, I1).
```

FIGURE 6.21 – *Modèle PSO (`pso.pl`)*

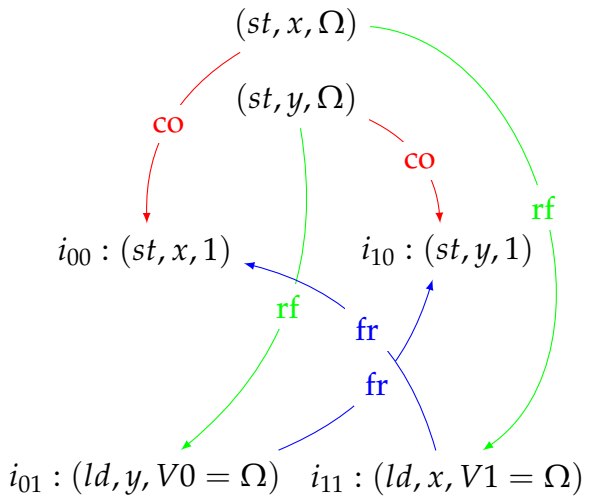


FIGURE 6.22 – *L'exécution interdite par le modèle SC pour le programme de la figure 6.1 est autorisé par TSO*

```
sb([x,y], [T0,T1]) :-
    T0 = [ (st,x,1), f(any,any), (ld,y,V0) ],
    T1 = [ (st,y,1), f(any,any), (ld,x,V1) ].
```

FIGURE 6.23 – *Ajouts de barrières au programme de la figure 6.5*

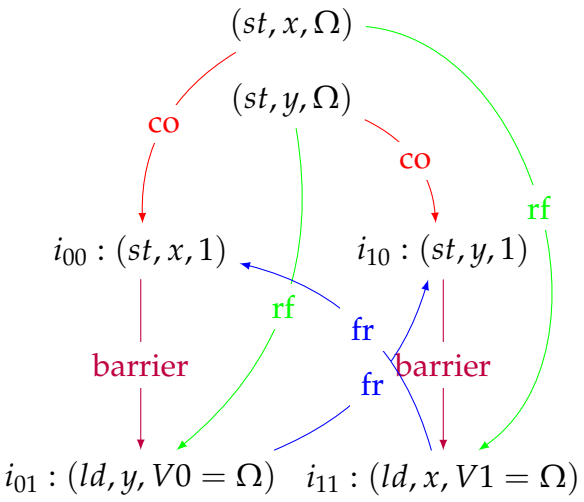


FIGURE 6.24 – *Exécution interdite par le modèle TSO pour le programme de la figure 6.23 grâce aux barrières*

6.5 JUSTIFICATION DE LA TERMINAISON

Ici, nous donnons les idées générales qui assurent la terminaison des différentes phases de l'analyse. Dans son ensemble, le solveur fonctionne de la manière suivante :

— génération des exécutions candidates ;
— déduction des relations dérivées ;
— détection des cycles dans les relations définies par les modèles.

Chacune de ces étapes doit terminer pour assurer la terminaison globale.

6.5.1 Génération des exécutions candidates

La figure 6.8 expose le prédicat qui produit la génération des exécutions candidates. Le prédicat `enrich_prog` (ligne 2), présenté dans la section A.1, consiste en un parcours des instructions pour les modifier. Le nombre de fils d'exécution étant fini, à chaque étape d'évaluation du prédicat, notre liste de fils diminue jusqu'à atteindre la liste vide qui assure la terminaison. Le nombre d'instructions étant lui aussi fini dans chaque fil, la liste d'instructions décroît à chaque étape, sauf dans le cas où l'on rencontre une instruction *read-modify-write*. Dans ce cas, l'instruction est remplacée par deux instructions de lecture et écriture, nous ramenant au cas de base, qui termine.

La justification de la terminaison pour les opérations `compute_pos` (ligne 3) et `append` (ligne 4) suit la même logique, ainsi que la terminaison de l'opération `compute_rel_all_threads` (ligne 5–6). La complexité de cette dernière est

en revanche plus élevée car on parcourt plusieurs fois chaque fil d'exécution en réduisant à chaque étape le nombre d'instructions à parcourir (complexité détaillée en section 6.6.2).

La terminaison des prédicats `ops_to_each` (lignes 8–9) pour une position est assurée par les propriétés déjà mentionnées pour les prédicats précédents. Ici, nous avons également une étape de *backtracking* (retour sur trace) si la position mémoire accédée est une variable Prolog. Ce nombre de *backtrackings* est limité par le fait que le nombre de positions mémoire (liste `VARS` avec `undefined` en plus) est fini. Donc nous ne générerons qu'un nombre fini de branches.

Les listes `STORES` et `LOADS` générées sont des listes finies de listes d'instructions, elles-mêmes finies. Cela nous assure que les prédicats suivants `permute_stores` (ligne 10) et `compute_all_rfs` (ligne 12) ne pourront générer, lors de leurs *backtrackings*, qu'un nombre fini de permutations. Le prédicat `compute_all_cos` (ligne 11) parcourt les listes de `PSTORES` qui sont des permutations de `STORES`, assurant de ce fait la terminaison.

6.5.2 Relations dérivées par des règles CHR

La construction des relations dérivées à partir des relations de base sont de deux catégories :

— renommage d'une relation existante ou union de relations ;

— composition de relations à partir de contraintes acycliques.

La première est employée pour la définition de modèles mémoires. Ce sont les règles que l'on peut par exemple rencontrer en figure 6.16, où l'on renomme `po(I0,I1)` en `rel(sc,I0,I1)`, et de même pour CO, RF et FR. Ces règles terminent trivialement car une contrainte CHR ne peut activer une règle donnée qu'une seule fois pour une position dans la tête donnée (qui est ici unique). Cela n'est bien sûr valide que si le modèle ne génère pas, par erreur, de nouvelles contraintes de base.

La seconde correspond à la dérivation de nouvelles contraintes, depuis les contraintes de base, et dont la combinaison est acyclique. Par exemple, IPO est la clôture transitive de PO (cf. figure 6.10) dont on a la garantie qu'elle est acyclique. De la même manière, la relation FR (cf. figure 6.9) est calculée par une dérivation initiale depuis les contraintes RF et CO (opération qui termine car on n'a qu'un nombre fini de combinaisons), puis dérivée itérativement sur CO pour la position mémoire considérée, ce qui ne comprend pas de cycle, et termine donc.

6.5.3 Détection de cycles

Notre ensemble de règles CHR pour produire la détection de cycles (voir figure 6.14) est une variante de celui proposé par Frühwirth dans [45] pour le calcul de la clôture transitive d'une relation dans un graphe, que nous avons optimisé pour notre problème. Dans notre cas, l'objectif est de rechercher les cycles plutôt que calculer la clôture complète : nous nous arrêtons à la découverte d'un cycle et, du fait que nous ajoutons les contraintes au fur et à mesure de la génération d'une exécution donnée, nous pouvons détecter un cycle avant d'avoir généré la totalité des contraintes représentant une exécution. Il faut malgré tout noter que si aucun cycle n'existe, la clôture est calculée en entier. De plus, nous limitons l'espace de recherche par l'optimisation avec un ordre arbitraire `inf` décrite dans la section 6.4.

La terminaison des règles proposées par Frühwirth s'appuie sur deux hypothèses à propos de l'implémentation de CHR utilisée. D'une part, elle doit reposer sur la sémantique raffinée des CHR [42], qui impose un ordre précis sur l'ordre d'application des règles (celui de leurs définitions). Et d'autre part, l'application d'une règle de simpagation doit tenter de supprimer en priorité les contraintes ajoutées le plus récemment dans le stock de contraintes. Cela évite de refaire, pour un nouveau doublon c' d'une contrainte plus ancienne c toutes les déductions qui ont déjà été faites pour c précédemment. Ces deux hypothèses sont respectées par la majorité des implémentations, dont SWI-Prolog, que nous utilisons.

Les règles proposées par Frühwirth sont les suivantes :

```
1 path(Begin,End) \ path(Begin,End) ⇔ true.
2 edge(Begin,End)                   ⇒ path(Begin,End).
3 path(Begin,End), edge(End,Next)   ⇒ path(Begin,Next).
```

Frühwirth [45, Sec.2.4.1] indique que ce programme est terminant. Intuitivement, si depuis un sommet `n`, un chemin boucle, nous allons construire un chemin `path(n,m)` jusqu'à l'un des sommets `m` du cycle. Si nous poursuivons le cycle, nous finirons par ajouter de nouveau le sommet `m`, créant ainsi une deuxième contrainte `path(n,m)`. Comme l'implémentation suit la sémantique raffinée, la règle de simpagation pour la suppression de doublons est utilisée en premier, et supprime la contrainte `path(n,m)` la plus récente. Celle-ci ne peut donc plus être utilisée pour continuer le chemin qu'on arrête donc d'allonger.

Nos optimisations consistent en :

— l'ajout de gardes sur les règles de création de chemin ;

— l'ajout d'une règle de détection de cycle.

L'ajout des gardes correspondant à notre optimisation ne perturbe pas la terminaison de l'algorithme. Elles ne font que limiter la quantité de chemins créés en début de recherche, ainsi que les possibilités d'extension de chemins existants dans le stock de contraintes. Si l'extension d'un chemin pouvant boucler n'est pas empêchée par la garde, car compatible avec elle, cela n'empêche pas la suppression du chemin créé s'il est équivalent à un chemin existant, puisque la règle de simpagation sera vérifiée sur le nouveau chemin avant de continuer son extension.

La règle de détection de cycles n'empêche pas la terminaison de l'algorithme. Elle ne fait que supprimer un chemin et la contrainte qui permettait le cycle.

Étude d'une variante des règles

Nous pouvons alors nous demander si remplacer simplement la règle de suppression des doublons par une règle de détection de cycle de la forme :

```
% Propagation => Simplification
path(R, Begin, End), rel(R, End, Begin) ⇔ cycle(R, Begin).
```

qui supprime le chemin pour lequel on trouve un cycle ainsi que la contrainte qui referme ce cycle, peut se substituer à la règle de suppression des chemins doublons pour assurer la terminaison. Ce n'est pas le cas. Nous pouvons construire un contre-exemple (pour simplifier la lecture, nous avons remplacé les instructions par de simples entiers) à partir de la configuration en « 6 », précédemment présentée (cf. figure 6.15 et section 6.4.1). Considérons la requête suivante :

```
?- rel(r,5,2), rel(r,2,4), rel(r,4,3), rel(r,1,2), rel(r,3,5).}
```

La propagation est effectuée de la même manière que dans le précédent exemple, jusqu'à atteindre le stock de contrainte:

```
STORE={ rel(r,5,2),
        rel(r,2,4),path(r,2,4),
        rel(r,4,3),path(r,2,3),
        rel(r,1,2),path(r,1,2),path(r,1,4),path(r,1,3),
        rel(r,3,5),path(r,3,5) }
```

Une nouvelle fois, la contrainte `rel(r,3,5)` a été utilisée pour activer la règle d'ajout de chemin. Elle peut maintenant être utilisée pour étendre un chemin existant. Dans le store, nous avons deux chemins qu'il est possible d'étendre

avec cette contrainte : `path(r,2,3)` et `path(r,1,3)`. La sémantique raffinée ne donne pas de priorité à l'une des deux.

Si on choisit `path(r,2,3)`, on crée `path(r,2,5)`, qui est ensuite utilisée avec `rel(r,5,2)` pour activer la détection de cycle, donc cela termine.

Si on choisit `path(r,1,3)`, on crée `path(1,5)`, qui est également utilisée avec `rel(r,5,2)` mais dans ce cas, nous ne détectons pas un cycle, nous étendons le chemin, créant ainsi une nouvelle contrainte `path(r,1,2)`, cette contrainte existe déjà, mais nous n'avons pas de règle pour la supprimer. Donc nous continuons de l'étendre avec `rel(r,2,4)`, ajoutant un doublon pour `path(r,1,4)`, puis avec `rel(r,4,3)` pour créer un nouveau doublon `path(r,1,3)`. Donc nous entrons à nouveau dans le cycle d'ajout et bouclons à l'infini.

Cela montre que la règle de suppression des doublons est essentielle pour la terminaison.

6.6 Correction et performances

Dans cette section, nous nous attachons à donner les éléments qui assurent la correction de l'algorithme (hors terminaison) et nous présentons une évaluation expérimentale en illustrant les gains obtenus.

6.6.1 Suppression de contraintes

Si une relation n'intervient que partiellement dans la définition d'un modèle mémoire, les cas qui ne sont pas respectés par le modèle ne doivent pas faire partie des contraintes conservées. Par exemple, dans les modèles que nous avons présentés, nous définissons, à partir de la relation « program-order », la relation relaxée « preserved program-order » (modélisée par la contrainte `ppo`) qui est un sous-ensemble de la première.

Nous avons deux manières de la produire, qui peuvent être chacune plus pratique selon le cas :

— ajouter seulement les contraintes pour lesquelles l'ordre est conservé ;

— tout ajouter en supprimant celles qui ne sont pas conservées.

Nous illustrons cette différence avec les deux manières d'exprimer PPO pour le modèle TSO (fig 6.25). La première version ajoute toute contrainte, en supprimant les contraintes de la forme *ppo(st,ld)*, la seconde ajoute uniquement les contraintes *ppo(st,st)* et *ppo(ld,any)*.

```
ppo(i(_,_,st,_,_), i(_,_,ld,_,_)) ⇔ true.
ipo(I0,I1) ⇒ ppo(I0,I1).
```

Suppression sélective

```
ipo(i(N,T,ld,L,V),I1) ⇒
    ppo(i(N,T,ld,L,V),I1).
ipo(i(N0,T0,st,L0,V0),i(N1,T1,st,L1,V1)) ⇒
    ppo(i(N0,T0,ld,L0,V0),i(N1,T1,st,L1,V1)).
```

Addition sélective

FIGURE 6.25 – *Deux manières de définir le sous-ensemble d'une relation*

Ce que nous devons éviter est qu'une contrainte qui doit être supprimée soit utilisée par une règle de calcul avant sa suppression. Comme nous utilisons la sémantique raffinée du calcul des CHR, les règles sont appliquées dans l'ordre de définition. En particulier lorsqu'une nouvelle contrainte est créée nous avons l'assurance que les règles qu'elle peut activer seront appliquées dans leur ordre d'écriture.

Par conséquent, pour assurer que la règle ne crée pas de comportements incorrects, nous avons juste à assurer que la règle de suppression d'un certain type de contrainte T est bien ordonnée avant toute règle qui utiliserait également le type de contrainte T. Ceci assure que la règle de suppression sera activée en premier et donc que la contrainte sera bien supprimée avant qu'elle puisse activer d'autres règles CHR. Dans les modèles que nous avons définis, nous nous sommes assurés que c'est effectivement le cas.

6.6.2 Complexité de la génération des candidats

L'identification des exécutions autorisées pour un modèle mémoire donné est clairement un problème avec une complexité élevée. Ici, nous indiquons la complexité de la génération des exécutions candidates. La figure 6.8 expose le prédicat qui produit cette génération, dans lequel :

- `enrich_prog` ajoute les identifiants aux instructions ;
- `compute_pos` crée les contraintes PO ;
- `append` produit une liste de toutes les instructions ;
- `compute_rel_all_threads` produit les contraintes de dépendances ;
- `ops_to_each(st/ld,...)` produit les listes d'écritures/lectures par position mémoire. Si pour une instruction, la position mémoire n'est pas

connue, elle est associée à chacune des positions successivement, par *backtracking* ;

— `permute_stores` génère les permutations des écritures par *backtracking* ;

— `compute_all_cos` en extrait la relation CO ;

— `compute_all_rfs` calcule et génère la relation RF par *backtracking*.

Pour la suite, nous noterons $\mathcal{N}_{op,l}$, le nombre d'opérations de type op à la position mémoire l, avec op pouvant être st pour les écritures, ld pour les lectures et op pour n'importe quelle des deux opérations. Si l n'est pas précisé, nous considérons toutes les possibilités. Nous notons également $\mathcal{N}_{locs}$ le nombre de positions mémoire déclarées dans le programme.

Les opérations d'enrichissement des informations à propos des instructions, leur extraction sous forme de contraintes PO et leur concaténation sont toutes en complexité linéaire $\mathcal{O}(\mathcal{N}_{op})$.

Les opérations d'extraction des dépendances de données et de valeur sont en complexité quadratique. Pour chaque instruction on tente de l'associer à toute instruction qui la suit, donc potentiellement tout le reste du programme, pour une complexité $\mathcal{O}(\mathcal{N}_{op}^2)$.

Ces précédentes opérations ont une complexité négligeable en comparaison du reste des opérations et donc de la complexité globale. En effet, on ne revient jamais sur les choix de ces prédicats qui restent les mêmes quelles que soient les exécutions. Donc cette complexité est additionnée (et non combinée) à celle des phases suivantes. Nous ne la mentionnons pas par la suite.

L'extraction des opérations à chaque position mémoire est la première tâche qui va potentiellement générer des opérations de *backtracking*. En effet, dans notre langage, nous autorisons l'utilisateur à écrire et lire à des adresses en mémoire lues par des instructions précédentes. Chaque instruction contenant une variable pour le paramètre correspondant à la position mémoire devra donc tour à tour être unifiée avec toutes les positions mémoire existant dans la liste `LOCS` qui contient les variables indiquées par l'utilisateur et la position mémoire `undefined`.

Pour chaque opération, nous allons parcourir la liste de paires (l, op_list) qui associe à chaque position mémoire la liste des opérations qui ont lieu à cette position. Cela rajoute un facteur $\mathcal{N}_{locs}$.

Une estimation de la complexité des appels à `ops_to_each` est donc :

$$\mathcal{O}(\mathcal{N}_{locs}^{\mathcal{N}_{op}} \times \mathcal{N}_{locs})$$

Cette complexité est souvent modérée en pratique par deux aspects :

— les instructions concernant les mêmes registres opèrent généralement avec les mêmes variables (Prolog) sur une dépendance, l'unification entraîne qu'une seule instruction produit les *backtrackings* : la première rencontrée dans la recherche des opérations ;
— la génération par *backtracking* permet de ne recalculer qu'une partie des exécutions candidates à ce point de la résolution.

Viennent ensuite les deux phases les plus coûteuses de la génération des exécutions candidates : les permutations de CO et les choix d'écritures pour RF.

La génération de CO consiste à combiner les permutations d'écritures à chaque position mémoire. La complexité est donc :

$$\mathcal{O}(\prod_{l \in locs} \mathcal{N}_{st,l}!)$$

Finalement, chaque opération de lecture est associée à une opération d'écriture possible :

$$\mathcal{O}(\prod_{l \in locs} \mathcal{N}_{st,l}^{\mathcal{N}_{ld,l}})$$

Ces deux opérations représentant donc un nombre d'exécutions candidates de l'ordre de :

$$\mathcal{O}(\prod_{l \in locs} \mathcal{N}_{st,l}! \times \mathcal{N}_{st,l}^{\mathcal{N}_{ld,l}})$$

Pour finir nous pouvons estimer la complexité globale théorique (au pire cas) de la génération des exécutions candidates :

$$\mathcal{O}(\mathcal{N}_{locs}^{\mathcal{N}_{op}+1} \times \mathcal{N}_{locs} \times \prod_{l \in locs} \mathcal{N}_{st,l}! \times \mathcal{N}_{st,l}^{\mathcal{N}_{ld,l}})$$

L'efficacité des implémentations de Prolog provient de l'élagage de branches. Cela permet de ne pas visiter un sous-arbre complet de l'espace de recherche lorsque le début du calcul nous permet déjà d'affirmer qu'aucune solution n'existe dans le sous-arbre.

C'est pourquoi nous extrayons les contraintes CHR au fur et à mesure du calcul des relations de base, et que nous déclarons les règles CHR avant le déclenchement de la génération des exécutions. De cette manière, les incohérences sont détectées le plus tôt possible et des sous-arbres complets peuvent ne pas être visités, limitant ainsi le nombre d'exécutions à traiter. Cela peut notamment réduire la complexité réelle de la génération des exécutions candidates.

La complexité de cette phase du calcul dépend fortement du programme traité et peut être difficile à évaluer précisément dans le cas général. La sec-

tion suivante présente des résultats d'évaluation expérimentale sur quelques exemples.

6.6.3 Tests empiriques de correction et performances

Pour évaluer notre outil de manière expérimentale, nous avons utilisé divers exemples de programme tirés de Herd afin d'avoir un critère de comparaison des résultats. Ces exemples peuvent être trouvés sur la page de l'outil Herd à l'adresse `http://virginia.cs.ucl.ac.uk/herd/` (dans la section « armed cats »).

Sur tous les exemples testés, notre solveur produit les mêmes résultats que Herd, nous confirmant la validité du solveur pour ces exemples. Nos modèles étant basés sur ceux de Herd, nous ne cherchons pas à valider les modèles mais à valider notre implémentation du solveur.

L'exécution du solveur pour ces exemples étant très rapide, ils ne permettent pas d'évaluer précisément ses performances. Nous avons donc créé un certain nombre d'exemples impliquant plus d'instructions afin de pousser le solveur vers des résolutions plus complexes, nécessitant un plus grand nombre de calculs. Nous comparons ici notre outil avec la version 7.11 de l'outil Herd. La machine utilisée comprend un processeur Intel Core i7-4800MQ, 4 cores, 2.7Ghz, accompagné de 16Go de RAM.

La figure 6.26 présente deux exemples d'une série de passages de messages (potentiellement fausse), avec 3 et 4 fils d'exécution. Cet exemple est intéressant pour mesurer les performances sur les exemples avec une forte complexité combinatoire. En effet, il fait intervenir de multiples écritures et lectures aux mêmes positions mémoire. De cette manière nous provoquons une croissance rapide du nombre d'exécutions étant donné que nous devons générer un grand nombre de combinaisons de permutations.

La figure 6.27 présente les résultats de l'analyse pour Herd et notre solveur. Le but est de déterminer pour chaque programme quel est l'ensemble des exécutions autorisées. Pour chacun des exemples nous analysons le programme avec différents modèles mémoire. Nous indiquons le nombre d'exécutions autorisées par le modèle. Le nombre indiqué pour le modèle « Générique » indique le nombre d'exécutions candidates. Nous n'indiquons pas les temps exacts au delà de l'heure de calcul.

Notre outil est conçu pour rejeter immédiatement les exécutions incohérentes. Cela nous permet de rejeter des branches complètes de l'arbre de recherche sans

```
1 mp3(V, [T0,T1,T2]) :-
2 V  = [ x, m ] ,
3 T0 = [(st,x,1), (st,m,1), (ld,m,M0),(ld,x,X0)],
4 T1 = [(ld,m,M1),(ld,x,X1),(st,x,2) ,(st,m,2) ],
5 T2 = [(ld,m,M2),(ld,x,X2),(st,x,3) ,(st,m,3) ].
6
7 mp4(V, [T0,T1,T2,T3]) :-
8 V  = [ x, m ] ,
9 T0 = [(st,x,1), (st,m,1), (ld,m,M0),(ld,x,X0)],
10 T1 = [(ld,m,M1),(ld,x,X1),(st,x,2) ,(st,m,2) ],
11 T2 = [(ld,m,M2),(ld,x,X2),(st,x,3) ,(st,m,3) ].
12 T3 = [(ld,m,M3),(ld,x,X3),(st,x,4) ,(st,m,4) ].
13
```

FIGURE 6.26 – *Test de performances : passages de messages (code)*

Exemple	Modèle	#exécutions	Herd	Solveur CHR
MP3	Générique	147 436	1.2s	3.3s
	PSO	2 258	3.8s	6.4s
	TSO	800	4.1s	3.2s
	SC	678	5.5s	3.3s
MP4	Générique	255 000 000	1405s	$>$ 1h
	PSO	516 030	$>$ 1h	2796s
	TSO	96 498	$>$ 1h	752s
	SC	81 882	$>$ 1h	747s

FIGURE 6.27 – *Test de performances : passages de messages (résultats)*

aller explorer la totalité des exécutions qui y correspondent. Il en résulte que nous pouvons obtenir assez rapidement des résultats quand l'explosion devient particulièrement forte. Cela est nettement perceptible quand les modèles sont particulièrement forts, comme dans le cas de TSO et SC.

Herd peut générer des sous-ensembles d'exécutions candidates en se basant sur la connaissance que le modèle respecte la relation SC par adresse, ou encore en ne générant que les exécutions correspondant à une certaine valeur lue pour une opération de lecture. Néanmoins, dans le cas de Prolog, l'élagage de branches permet de rendre ce processus automatique, ne nécessitant pas d'implémentation particulière dans le solveur mis à part le fait de produire les contraintes à la volée.

Notre évaluation expérimentale montre que notre solveur produit les mêmes résultats que l'outil Herd sur des exemples de sa suite de test. Le temps de résolution de notre solveur peut être plus court qu'avec Herd pour des modèles forts, grâce à la capacité de rejeter des exécutions interdites par sous-arbres.

Cependant, Herd peut être plus rapide pour des modèles faibles, où l'élagage de branches ne permet pas d'éviter la considération d'un très grand nombre d'exécutions.

6.7 Conclusion

Dans ce chapitre, nous avons présenté un outil écrit en Prolog et CHR qui permet de déterminer, pour un programme, quelles sont les exécutions qui sont autorisées à apparaître selon un modèle mémoire donné.

Ce solveur s'appuie sur un modèle générique qui détermine un ensemble d'exécutions en supposant un modèle mémoire très faible qui ne considère, dans sa définition, que l'ordre des écritures à chaque position mémoire (CO) et les couples lecture/écriture (RF), ne prenant pas en compte l'ordre des instructions du programme. Il autorise donc toutes les relaxations possibles sur l'ordre d'exécution des instructions. Nous appelons cet ensemble les exécutions *candidates*.

Les modèles mémoires spécifiques sont ensuite définis à l'aide des relations CO et RF, mais également à l'aide de la relation qui modélise l'ordre des instructions dans le programme (PO). Le modèle mémoire peut respecter entièrement ou partiellement ces relations. Par exemple, SC respecte CO et RF (ainsi que la relation FR qui en dérive), et l'ordre PO du programme. En revanche, TSO ne

respecte pas PO pour les couples d'instructions écriture-lecture. Lorsqu'un modèle respecte une relation, il indique quelles opérations doivent, selon lui, avoir lieu avant quelles autres. Pour certaines exécutions candidates, cela peut faire apparaître des cycles dans les relations, indiquant qu'une instruction doit avoir lieu avant elle-même, ce qui est incohérent. Une telle exécution est donc refusée selon le modèle mémoire. Pour chaque exécution, nous détectons la présence de cycles, et refusons l'exécution s'il en existe. Cela nous permet de générer l'ensemble des exécutions *autorisées*.

Nous avons évalué notre solveur par rapport à un outil de la littérature : Herd. Sur les exemples de la base de tests de Herd que nous traitons, nous produisons les mêmes ensembles d'exécutions autorisées et interdites. En terme de performances, lorsque les exemples ont de très nombreuses exécutions candidates, sur les modèles forts, nous nous montrons plus rapides grâce à l'élagage de branches de Prolog.

Actuellement, les modèles implémentés sont SC, TSO et PSO, nous sommes en train d'y ajouter le modèle ARM qui est en cours d'évaluation. Nous prévoyons également d'ajouter de nouvelles instructions dans le langage, notamment les opérateurs arithmétiques usuels et les branchements conditionnels.

Conclusion et perspectives

7

Sommaire

Dans ce chapitre, nous présentons le bilan et les perspectives de cette thèse. Dans la section 7.1 nous rappelons les objectifs. Nous présentons ensuite le bilan des travaux qui ont été réalisés par rapport à ces objectifs (section 7.2). Finalement, dans la section 7.3, nous énonçons un ensemble de perspectives envisagées par rapport à ce qui est déjà réalisé.

7.1 Rappel des objectifs

Notre objectif principal était de fournir un moyen de traiter des programmes concurrents à l'aide d'outils originellement prévus pour analyser des programmes séquentiels, sans que cela nécessite de les modifier. Les techniques usuelles d'analyse de programmes concurrents, et en particulier de preuve, sont généralement explicitement dédiées à cette classe de programme, et adapter un analyseur prévu pour du code séquentiel afin d'implémenter ces techniques demande de lourdes modifications.

Une fois la méthode répondant à ce problème définie, nous voulions dans un premier temps l'expérimenter sur une étude de cas afin d'en déterminer les

bénéfices et les faiblesses. À partir de cela, nous voulions concevoir un outil automatique qui implémente cette méthode, et permette effectivement de vérifier des programmes concurrents, à l'aide d'un analyseur prévu pour les programmes séquentiels. Nous voulions de notre méthode qu'elle soit formellement correcte dans le contexte du modèle mémoire séquentiellement consistant. Finalement, nous voulions étudier les particularités des modèles mémoire faibles, en vue de comprendre comment étendre la classe des programmes concurrents que nous sommes capables de traiter.

7.2 Bilan des travaux réalisés

Pour résoudre la problématique énoncée, nous proposons de procéder par simulation. Pour traiter un programme concurrent, nous construisons un programme séquentiel qui reproduit ses comportements. Nous supposons un nombre de fils d'exécution fixé mais arbitrairement grand. Dans le programme simulant, le contexte global d'exécution du programme d'origine est conservé tel qu'il est. Chaque donnée du contexte local d'exécution se voit attribuer une zone en mémoire globale sous la forme d'un tableau. Un tel tableau associe, à chaque identifiant de fil d'exécution, la valeur de la donnée ainsi simulée. Le contexte local d'exécution contient les variables locales, le compteur de point de programme, et des informations à propos de la pile d'exécution. Chaque action atomique du programme d'origine est ensuite simulée par la création d'une fonction qui en reproduit le comportement. Ces fonctions font exactement la même opération, où les accès à des éléments locaux sont cependant remplacés par des accès aux tableaux de simulation correspondants, pour le fil d'exécution actuellement traité, dont l'identifiant est transmis comme paramètre de la fonction simulante. Finalement, le comportement global du programme est simulé par une boucle qui entrelace les instructions, en choisissant aléatoirement, à chaque tour, le fil à faire avancer et en lui faisant exécuter sa prochaine action.

Nous avons expérimenté cette méthode dans une étude de cas tirée d'un micro-noyau conçu pour tirer parti de systèmes multi-cœurs. Il permet notamment de demander la modification des tables de pages de manière concurrente et c'est cette fonctionnalité que nous avons vérifiée. L'usage de notre méthode de simulation a permis d'effectuer la vérification à l'aide Frama-C et de son greffon de preuve déductive Wp, alors que ce dernier n'est originellement pas capable de traiter des programmes concurrents.

Pour faciliter l'usage de cette méthode d'analyse de programmes concurrents, nous avons implémenté un nouveau greffon pour Frama-C : Conc2Seq. Celui-ci permet, depuis un ensemble de fonctions pouvant être appelées de manière concurrente, de générer automatiquement un code séquentiel qui le simule par notre méthode. Nous fournissons un ensemble de primitives enrichissant acsl, le langage de spécification proposé par Frama-C, afin de permettre à l'utilisateur d'exprimer plus facilement les comportements concurrents. Lors de la transformation de code, les spécifications sont également traduites conformément à la simulation du code C.

Nous formalisons la méthode de génération sur un langage simplifié, capturant le comportement d'un programme selon le modèle mémoire séquentiellement consistant, et la notion de mémoire partagée. La fonction de transformation y est définie et l'on prouve l'équivalence de comportement entre le programme parallèle d'origine et le programme séquentiel simulant généré. Actuellement, le langage est formalisé dans l'assistant de preuve Coq. La fonction de transformation est également formalisée pour toutes les instructions sauf les blocs atomiques, ce point restant à réaliser, de même que la relation d'équivalence et sa preuve dans Coq.

Pour explorer la question de l'exécution des programmes selon les modèles mémoire faibles, nous avons réalisé un prototype de solveur permettant de déterminer, pour un programme transmis par l'utilisateur, quelles sont les exécutions autorisées à apparaître d'après un modèle mémoire donné. La résolution des contraintes imposées par les modèles mémoire sur les programmes est effectuée grâce à Prolog et CHR.

7.3 Perspectives envisagées

À court terme, nous visons à effectuer la preuve de notre méthode de transformation au sein de l'assistant de preuve Coq. Actuellement, le langage et la majeure partie de la fonction de transformation y sont déjà formalisés, mais pas encore la propriété d'équivalence et sa preuve.

Nous projetons également de généraliser l'implémentation de la méthode dans le greffon Conc2Seq, notamment en ajoutant la possibilité d'effectuer des appels de fonctions tant que ceux-ci ne sont pas récursifs. Nous voudrions également fournir à l'utilisateur de plus nombreuses primitives de spécification pour la concurrence.

Nous présentons dans la suite de cette section des perspectives plus larges en rapport avec l'aide à l'analyse de programmes concurrents par transformation de code.

7.3.1 Conc2Seq et modèles mémoire faibles

Notre génération de programme simulant est valide pour les programmes qui ont un comportement séquentiellement consistant. La preuve de programme avec Conc2Seq est soumise à cette hypothèse, qui doit être respectée avant la transformation. Cependant, nous n'effectuons pas de vérification que c'est effectivement le cas.

Une piste serait que le greffon, depuis le programme d'origine, génère un programme dans le langage d'entrée de notre solveur CHR. Le solveur pourrait alors calculer l'ensemble des exécutions du programme selon le modèle mémoire séquentiellement consistant et selon le modèle mémoire cible. Si les exécutions autorisées dans le modèle mémoire cible sont incluses dans les exécutions autorisées par le modèle mémoire séquentiellement consistant, le programme a un comportement séquentiellement consistant. Le solveur pourrait alors indiquer à Conc2Seq que cette hypothèse est bien vérifiée, assurant que la transformation est valide.

Cela nécessite dans un premier temps que le langage d'entrée de CHR soit étendu, notamment avec les branchements conditionnels, pour que la traduction depuis le C soit possible.

Une autre piste est donnée par [3], comme mentionné dans notre chapitre 2, et consiste à rendre les comportements faibles explicites, en incorporant la sémantique opérationnelle des modèles mémoires dans la traduction elle-même.

7.3.2 Conc2Seq et les greffons de Frama-C

Actuellement les programmes séquentiels générés par Conc2Seq ne peuvent être traités que par le greffon Wp car c'est le premier que nous avons ciblé. De plus, comme nous l'avons évoqué dans le chapitre 4, la traduction effectuée ne place pas Wp dans des conditions optimales pour générer les obligations de preuve qui sont transmises aux prouveurs automatiques, par exemple, la séparation ne peut pas être modélisée grâce à des variables logiques différentes.

Nous voudrions ouvrir le greffon à d'autres analyses, notamment la vérification par interprétation abstraite Value et le greffon de vérification à l'exécution e-acsl. Il faudrait pour cela identifier plus précisément les types de propriétés

traités par ces greffons dans le langage C et ACSL, pour déterminer comment produire un code et une spécification qu'ils peuvent traiter. Ceci est également nécessaire pour WP si nous voulons améliorer son support, notamment en changeant notre manière de générer les zones de mémoire simulantes.

Pour pouvoir réaliser cela, il serait sûrement nécessaire de revoir la conception d'une partie du greffon CONC2SEQ afin de le rendre plus facilement extensible.

7.3.3 Expérimentation sur des applications plus complexes

Les exemples que nous avons traités jusqu'à maintenant sont soit des exemples simples afin de tester le bon fonctionnement de notre greffon, soit des exemples que nous avons traités en les modélisant manuellement.

Il serait intéressant d'expérimenter la méthode sur de plus nombreux codes réels. Nous pourrions par exemple l'utiliser pour traiter des codes de structures de données concurrentes couramment utilisées. Cela permettrait notamment d'identifier de nouvelles difficultés pour la spécification des comportements concurrents et de proposer des primitives plus adaptées pour y répondre.

7.3.4 Extension de la méthode aux fonctions récursives

Notre méthode est valide pour vérifier des programmes concurrents sous l'hypothèse que leur comportement est séquentiellement consistant, mais également que les fonctions ne sont pas appelées récursivement. Dans certains cas, cela peut représenter une forte limitation. Il serait donc utile de chercher comment lever cette limitation. Cela nécessite de modéliser plus précisément le contexte de chaque fil d'exécution.

Si d'un point de vue théorique, un certain nombre d'approches peuvent convenir (modéliser la pile explicitement par exemple), il n'est pas aussi clair qu'elles soient utilisables en pratique, par exemple dans le cadre de CONC2SEQ. En effet, de telles approches peuvent complexifier l'état du programme simulant, et les analyseurs auraient certainement un grand nombre d'information à traiter juste pour le code spécifique à la simulation, hors de la sémantique du programme initial lui-même. Cela demande donc probablement des adaptations spécifiques pour chaque analyseur.

Annexes

Génération des exécutions candidates : Code Prolog

A

Sommaire

A.1 Extraction de PO et des dépendances

Nous rappelons que dans un premier temps, nous enrichissons les instructions avec des informations supplémentaires : un identifiant du fil d'exécution d'origine et un identifiant d'instruction au sein de ce fil d'exécution. Ces instructions sont immédiatement ajoutées au stock de contraintes CHR.

Nous devons également remplacer les opérations *read-modify-write* par une lecture et écriture successives. Nous ajoutons en même temps une contrainte CHR `rmw` entre ces deux instructions.

Ces deux tâches sont assurées par le code Prolog en figure A.1. Les prédicats `enrich_inst` en lignes 4 et 6 associe l'instruction d'origine à sa version avec identifiant.

Les clauses `enrich_thread` traitent l'ensemble d'un fil d'exécution. La second est celle qui permet le traitement des instructions *read-modify-write*, La coupure placée à ce point est importante car elle empêche le retour sur trace *backtracking* de passer à la clause suivante par la suite.

Les clauses `enrich_prog` itèrent `enrich_thread` sur chaque fil d'exécution.

Nous définissons également les prédicats `remove_register` et `only_register` qui vont respectivement extraire, dans leur second paramètre, la valeur ou le registre d'une variable Prolog utilisée comme adresse

```
:- chr_constraint i/5, fence/2.

% memory operation transformation
enrich_inst(Nt, Id, (O,L,V), i(Nt,Id,O,L,V)).
% fence transformation
enrich_inst(Nt, Id, f(OP0,OP1), fence(Nt,Id,OP0,OP1)).

% enrich_thread(Nt, Id, L, NL)
% apply enrich_inst on each instruction of L, with thread
% identifier Nt incrementing Id and put all instructions
% in NL
enrich_thread(_, _, [], []).
enrich_thread(Nt, Id, [(rmw, Loc, V1, V2)|L], R) :-
    !, NL = [(ld,Loc,V1),(st,Loc,V2)|L],
    rmw(NI1,NI2), % adds a CHR constraint RMW
    enrich_thread(Nt, Id, NL, R),
    R = [NI1, NI2 | _].

enrich_thread(Nt, Id, [I|L], [NI|NL]) :-
    enrich_inst(Nt, Id, I, NI),
    NI, % adds the enriched instruction as a CHR constraint
    NextId is Id+1,
    enrich_thread(Nt, NextId, L, NL).

% enrich_prog(Nt, L, NL)
% apply enrich_thread on each thread of L and incrementing
% the thread identifier Nt and put the result in NL.
enrich_prog(_, [],[]).
enrich_prog(N, [T|OT], [ET | OET]) :-
    enrich_thread(N,0,T,ET),
    Next is N+1,
    enrich_prog(Next,OT,OET).
```

FIGURE A.1 – *Enrichissement et extraction des instructions (`generic_model.pl`)*

```
1 remove_register(RExp:Exp, Exp) :- nonvar(RExp).
2 remove_register(Exp, Exp) :- not(compound(Exp)).
3
4 only_register(RExp:_, RExp) :- nonvar(RExp).
```

FIGURE A.2 – *Enrichissement et extraction des instructions (`generic_model.pl`)*

ou valeur. Quand la variable est de la forme `rX:V`, `remove_register` rendra `V`, `only_register` rendra `rX`. Elles s'assureront en même temps que `rX` est une constante Prolog, nous n'autorisons pas ce paramètre à être variable. Si la variable n'est pas de la forme `rX:V`, l'opération `only_register` se contente d'échouer. L'opération `remove_register` vérifie que la valeur n'est pas un terme composé. Le code correspondant est présenté en figure A.2.

Pour extraire PO il suffit de prendre toutes les paires d'opérations mémoire consécutives de chaque fil d'exécution, comme illustré en figure A.3. Le traitement des barrières (que l'on ne doit pas considérer) est effectué par les clauses `C2` et `C3` de `compute_po` ou une nouvelle fois nous devons prendre garde à couper la branche en question. On rappelle que la clôture transitive est calculée ultérieurement par IPO (cf. section 6.3).

Pour réaliser l'extraction des dépendances d'adresses et de valeur, nous devons traquer l'usage de registres équivalents pour différentes opérations tout en faisant attention à repérer si les valeurs de ces registres sont écrasées. Le prédicat d'extraction prend en paramètre le type de dépendance à extraire. Pour ces dépendances, nous ajoutons également un prédicat qui définit la présence d'une relation de ce type entre deux instructions données.

Ces opérations sont illustrées en figure A.4.

Étant données deux opérations mémoire (prise dans l'ordre), les clauses de `rule` définissent ce que sont les dépendances d'adresse et de valeur. Ici, ces dépendances sont définies en terme d'égalité entre registres correspondant à deux valeurs.

Les clauses de `find_dep` permettent de parcourir, depuis une instruction i_1, toutes les instructions i_n qui la suivent, afin de déterminer s'il existe une dépendance du type transmis de i_1 vers i_n. Si le registre concerné est écrasé, la recherche s'arrête et l'on n'essaiera pas d'autre instruction suivante (ceci est modélisé par la clause en lignes 8–10), grâce à la coupure. La clause en lignes 12–16 est validée s'il existe une dépendance, dans ce cas les lignes 14 et 15 permettent respectivement la création et l'ajout de la contrainte CHR correspon-

```
:- chr_constraint po/2.

/* compute_po( LI )
  Compute the po-relations between successive instructions
  of (LI) add corresponding CHR constraints
  Fences are not included in the po-relation.
*/
compute_po([]) :- !.                    % C0
compute_po([_]) :- !.                   % C1
compute_po([fence(_,_,_,_)|Tl]) :-      % C2
    !, compute_po(Tl).
compute_po([I0,fence(_,_,_,_)|Tl]) :-   % C3
    !, compute_po([I0|Tl]).
compute_po([I0,I1|Tl]) :-               % C4
    I1 = i(_,_,_,_,_),
    po(I0,I1), % adds a PO contraint between I0 and I1
    compute_po([I1|Tl]).

/* compute_pos( LT )
  Compute po-relations of each thread in LT
*/
compute_pos([]).
compute_pos([T | Tl]) :-
    compute_po(T),
    compute_pos(Tl).
```

FIGURE A.3 – *Extraction des contraintes PO (`generic_model.pl`)*

```
1  rule_dep(addr, i(_,_,ld,_,L0), i(_,_,_,L1,_)) :-
2      only_register(L0,R), only_register(L1,R).
3  rule_dep(data, i(_,_,ld,_,V0), i(_,_,st,_,V1)) :-
4      only_register(V0,R), only_register(V1,R).
5
6  find_dep(_, _, []).
7
8  % overwrite register
9  find_dep(_, i(_,_,ld,_,V0), [i(_,_,ld,_,V1)|_]) :-
10     only_register(V0,R), only_register(V1,R), !.
11
12 find_dep(Type, I0, [I1|Other]) :- % matching instruction
13     rule_dep(Type, I0, I1), !,
14     REL =.. [Type, I0, I1],     % creates REL = Type(I0,I1)
15     REL,                        % adds it to the store
16     find_dep(Type, I0, Other).
17
18 find_dep(Type, I0, [_|Other]) :- % something else
19     find_dep(Type, I0, Other).
20
21 compute_rel(_, []).
22 compute_rel(Type, [First|Rest]) :-
23     find_dep(Type, First, Rest),
24     compute_rel(Type, Rest).
25
26 compute_rel_all_threads(_, []).
27 compute_rel_all_threads(Type, [Head|Tail]) :-
28     compute_rel(Type, Head),
29     compute_rel_all_threads(Type, Tail).
```

FIGURE A.4 – *Extraction des dépendances (`generic_model.pl`)*

dante (`REL =.. [Type, I0, I1]` unifie `REL` avec un nouveau terme formé `Type(I0,I1)`).

A.2 Prédicats préalables à CO/FR

Pour extraire les relations CO et FR, nous avons besoin de connaître, pour chaque position mémoire, l'ensemble des opérations de lectures et d'écritures qui y sont possibles. Nous définissons le prédicat `ops_to_each(OP,LOCS,PRG,L)`. Pour un opérateur `OP` donné, une liste de positions mémoire `LOCS`, et un programme (formé par la concaténation

```
1  gen_first([],[]).
2  gen_first([E|Tl],[(E,[])|GTl]) :- gen_first(Tl,GTl).
3
4  ops_to_each(_, LOCS, [], GEN) :- gen_first(LOCS,GEN).
5
6  ops_to_each(OP, LOCS, [I|Next], ResTl) :-
7      I = fence(_,_,_,_), !,
8      ops_to_each(OP, LOCS, Next, ResTl).
9
10 ops_to_each(OP, LOCS, [I|Next], ResTl) :-
11     I = i(_,_,Other_OP, _, _), Other_OP \= OP, !,
12     ops_to_each(OP, LOCS, Next, ResTl).
13
14 ops_to_each(OP, LOCS, [I|Next], ResTl) :-
15     I = i(_,_,OP,L,_),
16     remove_register(L, Loc),
17     ops_to_each(OP, LOCS, Next, Res),
18     member(Loc,LOCS),
19     add_inst_to(Loc, I, Res, ResTl).
```

FIGURE A.5 – *Extraction des opérations par position mémoire (`generic_model.pl`)*

des fils d'exécution enrichis), il produit la liste `P` des opérations de type `OP` pour chaque position mémoire appartenant à `LOCS`. `L` est une liste de paires `(loc, list)`, où `list` représente la liste (potentiellement vide) d'opérations de type `OP` à la position mémoire `loc`.

Le code correspondant est présenté en figure A.5. Une nouvelle fois, dans cette fonctionnalité, nous ne nous préoccupons pas des barrières qui sont donc ignorées (lignes 6–8), de même que les instructions ne correspondant pas à l'opération `Loc` recherchée (lignes 10–12). Si la position mémoire accédée est fixée, l'usage du prédicat `member` à la ligne 18 ne fera qu'une vérification que la position mémoire existe, si c'est une variable elle entraînera la génération d'un ensemble d'exécutions par choix dans cette liste (via *backtracking*). L'instruction est ajoutée à la liste correspondant à la position mémoire `Loc` accédée par le prédicat `add_inst_to`.

Il est important de noter que dès cette étape, nous commençons à avoir des points de *backtracking* (retour sur trace). Les opérations de lecture et d'écriture pouvant être effectuées à des positions mémoire préalablement lues par le programme, il faudra produire une exécution différente pour chaque valeur que pourra prendre la variable correspondante.

Comme nous autorisons la lecture depuis une adresse précédemment lue en

```
permute_stores([],[]).
permute_stores([(Loc,L)|Tl], Res) :-
    permutation(L,NL),
    permute_stores(Tl,NTl)
    Res = [(Loc,[i(-1,-1,st,Loc,undefined)|NL])|NTl].

compute_cos([_]).
compute_cos([I0, I1 | Tl]) :-
    co(I0,I1),
    compute_cos([I1 | Tl]).

compute_all_cos([]).
compute_all_cos([(_,PSTORE) | Tl]) :-
    compute_cos(PSTORE),
    compute_all_cos(Tl).
```

FIGURE A.6 – *Extraction des contraintes CO (`generic_model.pl`)*

mémoire, nous devons considérer le cas où cette valeur est indéfinie. la liste `LOCS` des positions mémoire considérée comprend donc une valeur `undefined` en plus des positions mémoire fournies par l'utilisateur.

A.3 Extraction de CO

Les permutations d'écritures construites pour définir la relation CO sont formées à partir des listes des écritures à chaque position mémoire que l'on extrait à l'aide du prédicat `ops_to_each` défini dans la section précédente que l'on appelle avec la valeur `st` pour la variable `OP`. Nous formons les permutations à l'aide du prédicat `permute_stores(Stores, PermStores)`. A chaque permutation, nous ajoutons une écriture qui est placée en tête de liste et qui représente la valeur en mémoire avant toute autre écriture. Cette instruction est de la forme `i(-1,-1,st,LOC,undefined)`. Les listes générées sont ensuite transmises au prédicat `compute_cos` qui extrait et ajoute les contraintes CO (pour des écritures consécutives) de la même manière que `compute_pos` procède à l'extraction des contraintes PO. Voir figure A.6.

```
choose_rf(LD, _, rf(undefined, LD)) :-
    LD = i(_, _ , ld, Loc, _),
    remove_register(Loc, undefined),
    !.

choose_rf(LD, LST, rf(ST,LD)) :-
    LD = i(_,_,ld,LocLD,ValLD),
    remove_register(LocLD, Loc),
    remove_register(ValLD, Val),
    member(ST, LST),
    ST = i(_,_,st,LocST,ValST),
    remove_register(LocST, Loc),
    remove_register(ValST, Val),
    Loc \= undefined.

assign_rf_to_each_load([],_).
assign_rf_to_each_load([LD | LDTl], LST) :-
    assign_rf_to_each_load(LDTl, LST),
    choose_rf(LD, LST, RF),
    RF.

compute_all_rfs([],[]).
compute_all_rfs(LDS, STS) :-
    LDS = [(L,LDS_L) | LDS_Oth],
    STS = [(L,STS_L) | STS_Oth],
    compute_all_rfs(LDS_Oth, STS_Oth),
    assign_rf_to_each_load(LDS_L, STS_L).
```

FIGURE A.7 – *Extraction des contraintes RF (`generic_model.pl`)*

A.4 EXTRACTION DE RF

L'extraction des relations RF consiste à parcourir la liste des lectures à chaque position mémoire et à associer à chacune d'elle une écriture à la même position. A chaque lecture, on essaiera d'associer successivement chacune des écritures à la position correspondante par *backtracking*.

Cette opération est illustrée dans la figure A.7.

Bibliographie

[1] J. Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris VII - Denis Diderot, 2010. Cité pages 152 et 154.

[2] J. Alglave, A. C. J. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The semantics of Power and ARM multiprocessor machine code. In *DAMP 2009*, 2009. doi: 10.1145/1481839.1481842. Cité page 28.

[3] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP, part of ETAPS*, 2013. doi: 10.1007/978-3-642-37036-6_28. Cité pages 30 et 172.

[4] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 508–524, 2014. doi: 10.1007/978-3-319-08867-9_33. Cité page 30.

[5] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Prog. Lang. Syst.*, 2014. doi: 10.1145/2627752. Cité pages 27, 28, 29, 138 et 152.

[6] E. Alkassar, M. A. Hillebrand, W. J. Paul, and E. Petrova. Automated verification of a small hypervisor. In *Verified software: theories, tools, experiments (VSTTE)*, pages 40–54. Springer, 2010. Cité page 50.

[7] E. Alkassar, W. Paul, A. Starostin, and A. Tsyban. Pervasive Verification of an OS Microkernel. In G. Leavens, P. O'Hearn, and S. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *LNCS*, pages 71–85. Springer, 2010. doi: 10.1007/978-3-642-15057-9_5. Cité page 49.

[8] E. Alkassar, E. Cohen, M. Kovalev, and W. J. Paul. Verification of TLB virtualization implemented in C. In *the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2012)*, 2012. Cité page 50.

[9] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2008. ISBN 978-0-521-88038-1. Cité page 14.

[10] J. Andronick, C. Lewis, and C. Morgan. Controlled Owicki-Gries Concurrency: Reasoning about the Preemptible eChronos Embedded Operating System. In *Proceedings Workshop on Models for Formal Analysis of Real Systems (MARS)*, 2015. doi: 10.4204/EPTCS.196.2. Cité page 21.

[11] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, LNCS 3362, pages 49–69. Springer, 2004. doi: 10.1007/978-3-540-30569-9_3. Cité page 25.

[12] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects (FMCO)*, pages 364–387, 2005. doi: 10.1007/11804192_17. Cité page 25.

[13] C. Barrett and C. Tinelli. CVC3. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 298–302, 2007. doi: 10.1007/978-3-540-73368-3_34. Cité page 17.

[14] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011. doi: 10.1007/978-3-642-22110-1_14. Cité page 17.

[15] G. Barthe, G. Betarte, J. D. Campo, J. M. Chimento, and C. Luna. Formally verified implementation of an idealized model of virtualization. In *19th International Conference on Types for Proofs and Programs, TYPES 2013*, 2013. doi: 10.4230/LIPIcs.TYPES.2013.45. Cité page 49.

[16] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 55–66, 2011. doi: 10.1145/1926385.1926394. Cité page 28.

[17] P. Baudin, J. C. Filliâtre, P. Cuoq, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2015. `http://frama-c.com/download.html`. Cité page 24.

[18] A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. A case study on formal verification of the Anaxagoros hypervisor paging system with Frama-C. In *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, LNCS, pages 15–30, Oslo, Norway, June 2015. Springer. doi: 10.1007/978-3-319-19458-5_2. Cité pages 12 et 31.

[19] A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs. In *16th IEEE International Conference on Source Code Analysis and Manipulation (SCAM 2016), 2-3 October 2016, Raleigh, NC USA*, 2016. Cité pages 12 et 56.

[20] A. Blanchard, N. Kosmatov, and F. Loulergue. A chr-based solver for weak memory behaviors. In *Proceedings of the 7th Workshop on Constraint Solvers in Testing, Verification, and Analysis co-located with The International Symposium on Software Testing and Analysis (ISSTA 2016), Saarbrücken, Germany, July 17th, 2016.*, pages 15–22, 2016. Cité pages 12 et 129.

[21] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI 2008*, 2008. doi: 10.1145/1375581.1375591. Cité page 28.

[22] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, 2005. doi: 10.1145/1040305.1040327. Cité page 26.

[23] G. Boudol and G. Petri. Relaxed memory models: An operational approach. In *POPL 2009*, 2009. doi: 10.1145/1480881.1480930. Cité page 29.

[24] J. Brotherston, C. Fuhs, J. A. N. Pérez, and N. Gorogiannis. A decision procedure for satisfiability in separation logic with inductive predicates. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 25:1–25:10, 2014. doi: 10.1145/2603088.2603091. Cité page 18.

[25] G. Canet, P. Cuoq, and B. Monate. A value analysis for C programs. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*, pages 123–124, 2009. doi: 10.1109/SCAM.2009.22. Cité page 23.

[26] J. F. Cantin, M. H. Lipasti, and J. E. Smith. The complexity of verifying memory coherence and consistency. *IEEE Trans. Parallel Distrib. Syst.*, 16(7): 663–671, 2005. doi: 10.1109/TPDS.2005.86. Cité page 152.

[27] G. Chen, E. Cohen, and M. Kovalev. Store buffer reduction with MMUs: Complete paper-and-pencil proof. Technical report, Saarland University, Saarbrücken, 2013. URL `http://www-wjp.cs.uni-saarland.de/publikationen/CCK13.pdf`. Cité page 51.

[28] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001. Cité page 15.

[29] E. Cohen and B. Schirmer. From total store order to sequential consistency: A practical reduction theorem. In *ITP 2010*, 2010. doi: 10.1007/978-3-642-14052-5_28. Cité pages 30 et 50.

[30] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, , and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 23–42, 2009. Cité page 25.

[31] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. volume 6174 of *LNCS*, pages 480–494. Springer, 2010. doi: 10.1007/978-3-642-14295-6_42. Cité page 25.

[32] A. Colmerauer and P. Roussel. History of programming languages—ii. chapter The Birth of Prolog, pages 331–367. ACM, New York, NY, USA, 1996. ISBN 0-201-89502-1. doi: 10.1145/234286.1057820. Cité page 131.

[33] S. Conchon. *SMT Techniques and their Applications: from Alt-Ergo to Cubicle*. Thèse d'habilitation, Université Paris-Sud, 2012. Cité page 17.

[34] S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems - tool paper. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 718–724, 2012. doi: 10.1007/978-3-642-31424-7_55. Cité page 16.

[35] L. Correnson. Qed. computing what remains to be proved. In *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, pages 215–229, 2014. doi: 10.1007/978-3-319-06200-6_17. Cité page 25.

[36] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977. doi: 10.1145/512950.512973. Cité page 15.

[37] F. Dabrowski and D. Pichardie. A Certified Data Race Analysis for a Java-like Language. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS 5674*, pages 212–227. Springer, 2009. Cité page 30.

[38] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, 2008. Cité page 25.

[39] W. Dietl and P. Müller. Object ownership in program verification. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 289–318. 2013. doi: 10.1007/978-3-642-36946-9_11. Cité page 25.

[40] E. W. Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer, 1968. Cité page 20.

[41] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975. doi: 10.1145/360933.360975. Cité page 16.

[42] G. J. Duck, P. J. Stuckey, M. J. G. de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, pages 90–104, 2004. doi: 10.1007/978-3-540-27775-0_7. Cité pages 134 et 159.

[43] J. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 125–128, 2013. doi: 10.1007/978-3-642-37036-6_8. Cité page 17.

[44] T. Fruhwirth. Theory and practice of constraint handling rules. *The Journal of Logic Programming*, 1998. doi: 10.1016/S0743-1066(98)10005-5. Cité page 132.

[45] T. Frühwirth. *Constraint Handling Rules*. Constraint Handling Rules. Cambridge University Press, 2009. ISBN 9780521877763. Cité pages 135 et 159.

[46] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187678. Cité page 24.

[47] M. He, V. Vafeiadis, S. Qin, and J. F. Ferreira. Reasoning about fences and relaxed atomics. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 520–527, 2016. doi: 10.1109/PDP.2016.103. Cité page 29.

[48] C. Hoare. Towards a theory of parallel programming. In Hoare and Perott, editors, *Operating Systems Techniques*. Academic Press, 1972. Cité page 19.

[49] C. A. R. Hoare. An axiomatic basis for computer programmation. *Commun. ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259. Cité page 16.

[50] C. A. R. Hoare. Parallel programming: An axiomatic approach. *Comput. Lang.*, 1(2):151–160, 1975. doi: 10.1016/0096-0551(75)90014-4. Cité page 19.

[51] J. Holzmann. *The Spin Model Checker*. Addison Wesley, 2004. Cité page 16.

[52] A. Horn. On Sentences Which are True of Direct Unions of Algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951. doi: 10.2307/2268661. Cité page 131.

[53] International Organization for Standardization. *ISO/IEC 9899:2011: Programming languages – C*. ISO Working Group 14, 2011. Cité pages 11 et 59.

[54] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 14–26, 2001. Cité page 18.

[55] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the verifast program verifier. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems*, APLAS'10. Springer-Verlag, 2010. ISBN 3-642-17163-X, 978-3-642-17163-5. doi: 10.1007/978-3-642-17164-2_21. Cité page 26.

[56] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans Program Lang Syst*, 5:596–619, October 1983. ISSN 0164-0925. doi: 10.1145/69575.69577. Cité page 21.

[57] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015. doi: 10.1007/s00165-014-0326-7. Cité pages 23 et 30.

[58] G. Klein. From a verified kernel towards verified systems. In *the 8th Asian Symposium on Programming Languages and Systems (APLAS 2010)*, 2010. Cité page 52.

[59] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1), 2014. doi: 10.1145/2560537. Cité page 49.

[60] N. Kosmatov, M. Lemerre, and C. Alec. A case study on verification of a cloud hypervisor by proof and structural testing. In *TAP 2014*, 2014. Cité page 51.

[61] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program. *IEEE Trans. Comput.*, 1979. doi: 10.1109/TC.1979.1675439. Cité pages 19 et 128.

[62] W. Landi. Undecidability of Static Analysis. *LOPLAS*, 1(4):323–337, 1992. doi: 10.1145/161494.161501. Cité page 15.

[63] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 2006. ISSN 0163-5948. doi: 10.1145/1127878.1127884. Cité page 24.

[64] M. Lemerre, V. David, and G. Vidal-Naquet. A communication mechanism for resource isolation. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES'09, pages 1–6. ACM, 2009. doi: 10.1145/1519130.1519131. Cité pages 31 et 33.

[65] M. Lemerre, V. David, and G. Vidal-Naquet. A dependable kernel design for resource isolation and protection. In *IIDS '10: Proceedings of the First Workshop on Isolation and Integration in Dependable Systems*, 2010. ISBN 978-1-4503-0120-6. Cité page 33.

[66] M. Lemerre, E. Ohayon, D. Chabrol, M. Jan, and M.-B. Jacques. Method and Tools for Mixed-Criticality Real-Time Applications within PharOS. In *Proceedings of AMICS 2011: 1st International Workshop on Architectures and Applications for Mixed-Criticality Systems*, 2011. Cité page 32.

[67] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. doi: 10.1007/s10817-009-9155-4. Cité page 107.

[68] M. Mandrykin and A. Khoroshilov. Towards deductive verification of concurrent linux kernel code with jessie. In *Computer Science and Information Technologies (CSIT), 2015*, 2015. doi: 10.1109/CSITechnol.2015.7358240. Cité page 26.

[69] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 378–391, New York, NY, USA, 2005. ACM. doi: 10.1145/1040305.1040336. Cité page 28.

[70] Y. Moy and C. Marché. *The Jessie plugin for Deduction Verification in Frama-C — Tutorial and Reference Manual*. INRIA & LRI, 2011. URL `http://krakatoa.lri.fr`. Cité page 26.

[71] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, pages 213–228, 2002. doi: 10.1007/3-540-45937-5_16. Cité pages 23 et 68.

[72] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005. Cité page 15.

[73] T. Nipkow and L. Prensa Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering, Second International Conference (FASE'99)*, volume LNCS 1577, pages 188–203. Springer, 1999. Cité page 21.

[74] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002. Cité page 17.

[75] P. W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375:271–307, 2007. doi: 10.1016/j.tcs.2006.12.035. Cité page 22.

[76] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010. doi: 10.1007/978-3-642-14107-2_23. Cité page 30.

[77] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976. doi: 10.1145/360051.360224. Cité pages 8 et 20.

[78] G. Petiot, N. Kosmatov, B. Botella, A. Giorgetti, and J. Julliand. Your proof fails ? testing helps to find the reason. In *Tests and Proofs - 10th International Conference, TAP 2016, Held as Part of STAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings*, pages 130–150, 2016. doi: 10.1007/978-3-319-41135-4_8. Cité page 23.

[79] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic with trees and data. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 711–728, 2014. doi: 10.1007/978-3-319-08867-9_47. Cité page 18.

[80] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: practical static race detection for C. *ACM Trans. Program. Lang. Syst.*, 33(1):3, 2011. doi: 10.1145/1889997.1890000. Cité page 30.

[81] L. Prensa Nieto. The Rely-Guarantee Method in Isabelle/HOL. In P. Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming (ESOP 2003)*, volume LNCS 2618, pages 348–362. Springer, 2003. Cité page 21.

[82] A. Reynolds, R. Iosif, and T. King. A decision procedure for separation logic in SMT. *CoRR*, abs/1603.06844, 2016. Cité page 18.

[83] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002. Cité page 18.

[84] V. A. Saraswat. Concurrent constraint-based memory machines: A framework for java memory models. In *Advances in Computer Science - ASIAN 2004, Higher-Level Decision Making, 9th Asian Computing Science Conference, Chiang Mai, Thailand, December 8-10, 2004, Proceedings*, pages 494–508, 2004. doi: 10.1007/978-3-540-30502-6_36. Cité pages 28 et 29.

[85] V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *PPoPP*, pages 161–172. ACM, 2007. doi: 10.1145/1229428.1229469. Cité page 29.

[86] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 175–186, 2011. doi: 10.1145/1993498.1993520. Cité page 28.

[87] P. Schnoebelen, editor. *Vérification de logiciels: techniques et outils du model-checking*. Vuibert, 2000. Cité page 15.

[88] T. Schrijvers. Jmmsolve: A generative java memory model implemented in prolog and CHR. In *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, pages 475–476, 2004. doi: 10.1007/978-3-540-27775-0_45. Cité page 29.

[89] W. Schulte. Vcc: Contract-based modular verification of concurrent c. In *31st International Conference on Software Engineering, ICSE 2009*. IEEE Computer Society, 2008. doi: 10.1109/ICSE-COMPANION.2009.5071046. Cité page 25.

[90] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010. doi: 10.1145/1785414.1785443. Cité page 28.

[91] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`, 1984. Cité page 17.

[92] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In A. P. Black and T. D. Millstein, editors, *OOPSLA*, pages 691–707. ACM, 2014. doi: 10.1145/2660193.2660243. Cité page 29.

[93] V. Vafeiadis and C. Narayan. Relaxed separation logic: a program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 867–884, 2013. doi: 10.1145/2509136.2509532. Cité page 29.

[94] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007. ISBN 978-3-540-74406-1. doi: 10.1007/978-3-540-74407-8_18. Cité page 22.

[95] A. Vaynberg and Z. Shao. Compositional verification of a baby virtual memory manager. In *Certified Programs and Proofs, CPP 2012*, 2012. doi: 10.1007/978-3-642-35308-6_13. Cité page 49.

[96] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing - EDCC 5: 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005. Proceedings*. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-32019-7. doi: 10.1007/11408901_21. Cité page 23.

[97] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A practical verification framework for preemptive OS kernels. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 59–79, 2016. doi: 10.1007/978-3-319-41540-6_4. Cité page 50.

Ce document a été écrit à l'aide l'éditeur de texte GNU Emacs et du logiciel de composition typographique LaTeX 2ε.

Allan BLANCHARD

Aide à la vérification de programmes concurrents par transformation de code et de spécifications

Vérifier formellement des programmes concurrents est une tâche difficile. S'il existe différentes techniques pour la réaliser, très peu sont effectivement mises en œuvre pour des programmes écrits dans des langages de programmation réalistes. En revanche, les techniques de vérification formelles de programmes séquentiels sont utilisées avec succès depuis plusieurs années déjà, et permettent d'atteindre de hauts degrés de confiance dans nos systèmes. Cette thèse propose une alternative aux méthodes d'analyses *dédiées* à la vérification de programmes concurrents consistant à transformer le programme concurrent en un programme séquentiel pour le rendre analysable par des outils dédiés aux programmes séquentiels.

Nous nous plaçons dans le contexte de FRAMA-C, une plate-forme d'analyse de code C spécifié avec le langage ACSL. Les différentes analyses de FRAMA-C sont des greffons à la plate-forme, ceux-ci sont à ce jour majoritairement dédiés aux programmes séquentiels. La méthode de vérification que nous proposons est appliquée manuellement à la vérification d'un code concurrent issu d'un hyperviseur. Nous automatisons la méthode à travers un nouveau greffon à FRAMA-C qui permet de produire automatiquement, depuis un programme concurrent spécifié, un programme séquentiel spécifié équivalent. Nous présentons les bases de sa formalisation, ayant pour but d'en prouver la validité. Cette validité n'est valable que pour la classe des programmes séquentiellement consistant. Nous proposons donc finalement un prototype de solveur de contraintes pour les modèles mémoire faibles, capable de déterminer si un programme appartient bien à cette classe en fonction du modèle mémoire cible.

Mots-clés : concurrence, vérification formelle, transformation de code, modèles mémoire faible, FRAMA-C

Assisted Concurrent Program Verification by Code and Specification Transformation

Formal verification of concurrent programs is a hard task. There exists different methods to perform such a task, but very few are applied to the verification of programs written using real life programming languages. On the other side, formal verification of sequential programs is successfully applied for many years, and allows to get high confidence in our systems. As an alternative to *dedicated* concurrent program analyses, we propose a method to transform concurrent programs into sequential ones to make them analyzable by tools dedicated to sequential programs.

This work takes place within the analysis framework FRAMA-C, dedicated to the analysis of C code specified with ACSL. The different analyses provided by FRAMA-C are plugins to the framework, which are currently mostly dedicated to sequential programs. We apply this method to the verification of a concurrent code taken from an hypervisor. We describe the automation of the method implemented by a new plugin to FRAMA-C that allow to produce, from a specified concurrent program, an equivalent specified sequential program. We present the basis of a formalization of the method with the objective to prove its validity. This validity is admissible only for the class of sequentially consistent programs. So, we finally propose a prototype of constraint solver for weak memory models, which is able to determine whether a program is in this class or not, depending on the targeted hardware.

Keywords : concurrency, formal verification, code transformation, weak memory models, FRAMA-C