Vérification par model-checking de programmes concurrents paramétrés sur des modèles mémoires faibles

David Declerck

université PARIS-SACLAY

NNT : 2018SACLS336

Thèse de doctorat

UNIVERSITÉ PARIS SUD

Vérification par Model-Checking de programmes concurrents paramétrés sur des modèles de mémoire faibles

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

École doctorale n°580
Sciences et Technologies de l'Information et de la Communication

Spécialité : Informatique

Thèse présentée et soutenue à Gif-sur-Yvette le 24 septembre 2018, par

M. DAVID DECLERCK

Composition du jury

M. Philippe DAGUE Professeur des Universités, LRI, Université Paris-Sud	Président
M. **Ahmed BOUAJJANI** Professeur des Universités, IRIF, Université Paris Diderot	Rapporteur
M. Dominique MÉRY Professeur des Universités, Loria, Université de Lorraine	Rapporteur
M. Philippe QUÉINNEC Professeur des Universités, IRIT, Toulouse INP-ENSEEIHT	Examinateur
M. Luc MARANGET Chargé de Recherche, INRIA Paris	Examinateur
Mme. Fatiha ZAÏDI Maître de conférences HDR, LRI, Université Paris-Sud	Directrice de thèse
M. Sylvain CONCHON Professeur des Universités, LRI, Université Paris-Sud	Co-encadrant de thèse

Résumé

Les multiprocesseurs et microprocesseurs multicœurs modernes mettent en oeuvre des modèles mémoires dits *faibles* ou *relâchés*, dans lesquels l'ordre apparent des opérations mémoires ne suit pas la *cohérence séquentielle* (SC) proposée par Leslie Lamport. Tout programme concurrent s'exécutant sur une telle architecture et conçu avec un modèle SC en tête risque de montrer à l'exécution de nouveaux comportements, dont certains sont potentiellement des comportements incorrects. Par exemple, un algorithme d'exclusion mutuelle correct avec une sémantique par entrelacement pourrait ne plus garantir l'exclusion mutuelle lorsqu'il est mis en oeuvre sur une architecture plus relâchée.

Raisonner sur la sémantique de tels programmes s'avère très difficile. Par ailleurs, bon nombre d'algorithmes concurrents sont conçus pour fonctionner indépendamment du nombre de processus mis en oeuvre. On voudrait donc pouvoir s'assurer de la correction d'algorithmes concurrents, quel que soit le nombre de processus impliqués. Pour ce faire, on s'appuie sur le cadre du Model Checking Modulo Theories (MCMT), développé par *Ghilardi et Ranise*, qui permet la vérification de propriétés de sûreté de programmes concurrents *paramétrés*, c'est-à-dire mettant en oeuvre un nombre arbitraire de processus. On étend cette technologie avec une théorie permettant de raisonner sur des modèles mémoires faibles. Le résultat ce ces travaux est une extension du model checker Cubicle, appelée Cubicle-$\mathcal{W}$, permettant de vérifier des propriétés de systèmes de transitions paramétrés s'exécutant sur un modèle mémoire faible similaire à TSO.

Par ailleurs, un certain nombre de protocoles étudiés étant écrits en langage d'assemblage x86, il est nécessaire de les traduire manuellement vers les systèmes de transitions de Cubicle-$\mathcal{W}$ pour pouvoir les analyser. On souhaite automatiser cette tâche afin de la rendre plus rapide et éviter le risque d'erreur qu'implique une traduction manuelle. On propose donc un schéma de traduction d'un sous-ensemble du langage d'assemblage x86 vers les systèmes de transitions de Cubicle-$\mathcal{W}$, dont la correction est prouvée par simulation. Ce schéma de traduction a été implémenté dans un outil et a permis la traduction et la vérification de plusieurs algorithmes d'exclusion mutuelle écrits en assembleur x86.

Table des matières

Table des figures

Liste des tableaux

1

Introduction

Les problématiques liées à la sémantique des programmes concurrents sur des architectures parallèles à mémoire partagée constituent un problème ancien. En 1979, Leslie Lamport propose que ces architectures se comportent de manière séquentiellement cohérente (sequential consistency, SC) : “le résultat de toute exécution est le même que si les opérations de tous les processus étaient exécutées dans un ordre séquentiel quelconque, tel que les opérations de chaque processus apparaissent dans cette séquence dans l'ordre spécifié par leur programme” [102].

Si cette notion de cohérence séquentielle paraît naturelle et permet de raisonner aisément sur des programmes concurrents, il s'agit toutefois d'une notion trop forte, qui limite fortement les possibilités d'optimisation et les performances potentielles des programmes concurrents. Ainsi la recherche va plutôt s'orienter vers des modèles dans lesquels l'ordre des instructions ne suit pas un ordre séquentiel - on parle alors de modèle mémoire faible, ou relâché. En 1986, Dubois *et al.* s'intéressent aux multiprocesseurs ayant recours à des tampons de lecture et d'écriture afin de limiter les latences dues aux accès mémoires, et proposent le modèle de cohérence faible (*weak ordering*) [71]. En 1990, Adve et Hill [16] introduisent un modèle de synchronisation sans situation de compétition (*data-race-free* model, DRF). Ce modèle logiciel garantit à tout programme qui le respecte un comportement *en apparence* séquentiellement cohérent, même si l'architecture matérielle sous-jacente expose un modèle de cohérence faible. Par la suite, de nombreux modèles verront le jour (cohérence PRAM (PRAM consistency) [107], cohérence cache (cache consistency) [82], cohérence processeur (processor consistency) [82, 18], cohérence lente (slow consistency) [92], cohérence au relâchement (release consistency) [78], cohérence causale (causal consistency) [17], ...), et plusieurs architectures matérielles implémenteront de tels modèles au niveau du processeur.

Dans le cadre de cette thèse, on s'intéresse aux modèles mémoires effectivement mis en œuvre dans les multiprocesseurs et processeurs multicœur modernes. Ces modèles mémoires peuvent être exprimés en termes de relâchements sur l'ordre apparent des instructions d'un même processus, par rapport à l'ordre séquentiellement cohérent. Les relâchements possible sont les suivants :

— W → R : les écritures peuvent être réordonnées après les lectures suivantes
— W → W : les écritures peuvent être réordonnées après les écritures suivantes
— R → R : les lectures peuvent être réordonnées après les lectures suivantes
— R → W : les lectures peuvent être réordonnées après les écritures suivantes
— lecture précoce intra-processus : les lectures peuvent lire depuis une écriture précédente du même processus avant que cette écriture ne devienne globalement visible à tous les processus
— lecture précoce inter-processus : les lectures peuvent lire depuis une écriture précédente d'un autre processus avant que cette écriture ne devienne globalement visible à tous les processus

Le Tableau 1.1, basé sur celui donné en [15], présente les modèles les plus communs et leurs relâchements.

Modèle	W → R	W → W	R → RW	lecture précoce intra-processus	lecture précoce inter-processus
SC					
x86/AMD64	√			√	
SPARC TSO	√			√	
SPARC PSO	√	√		√	
SPARC RMO	√	√	√	√	
PA-RISC	√	√	√	√	√
IA64	√	√	√	√	√
POWER	√	√	√	√	√
ARM	√	√	√	√	√
Alpha	√	√	√	√	

TABLEAU 1.1 – Modèles mémoires courants et leurs relâchements

On illustre quelques-uns de ces relâchements dans les exemples présentés en Figure 1.1. Dans ces exemples, X et Y sont des variables partagées, tandis que R1, R2 et R3 sont des registres spécifiques à chaque processus.

L'exemple en Figure 1.1a illustre le relâchement W → R. Partant de l'état $X = Y = 0$, on ne peut atteindre l'état $R1 = 0 \wedge R2 = 0$ en SC. Toutefois cet état est atteignable avec le relâchement W → R (TSO, PSO, etc) : si les écritures ont effectivement lieu après les lectures, alors les lectures peuvent toutes deux lire la valeur 0 depuis les variables X et Y. On explique généralement ce comportement à l'aide de tampons d'écriture : si l'on considère que chaque processus dispose d'un tampon, alors les écritures $X \leftarrow 1$ et $Y \leftarrow 1$ transitent par ce tampon, et elle peuvent ne pas encore avoir été propagées en mémoire lorsque les lectures de X et Y sont effectuées.

L'exemple en Figure 1.1b est une variante du précédent, qui illustre la lecture précoce intra-processus. Ce comportement n'a du sens que pour les modèles qui relâchent l'ordre W → R (TSO, PSO, etc...). Partant de l'état $X = Y = 0$, on ne peut atteindre l'état $R1 = 1 \wedge R2 = 0 \wedge R3 = 1 \wedge R4 = 0$, à moins que les processus ne puissent lire la valeur qu'ils ont écrite avant que celle-ci ne devienne visible aux autres processus. En termes de tampons d'écriture, cela revient à dire que lorsqu'un processus effectue la lecture d'une

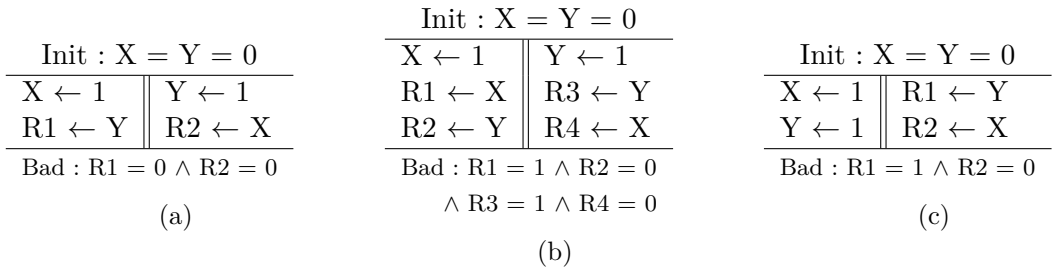


FIGURE 1.1 – Exemples illustrant certains relâchements

variable, il récupère en priorité la valeur la plus récente qu'il possède pour cette variable dans son tampon d'écriture, et ne récupère la valeur depuis la mémoire que si son tampon ne contient aucune écriture sur cette variable. Ce comportement est *en général* observé par les architectures relâchant l'ordre W → R, car il permet de préserver la sémantique des programmes séquentiels.

L'exemple en Figure 1.1c illustre quant à lui les relâchements W → W (PSO, RMO...) et R → R (RMO, Power...). Partant de l'état X = Y = 0, on ne peut atteindre l'état R1 = 1 ∧ R2 = 0 en SC. Toutefois, si l'on relâche l'ordre W → W (PSO), cet état est atteignable : les écritures n'étant plus ordonnées, X ← 1 peut avoir lieu après Y ← 1, de ce fait même si la lecture R1 ← Y récupère la valeur 1, R2 ← X peut encore récupérer la valeur 0. Il en va de même si l'on relâche l'ordre R → R : même si les écritures restent ordonnées, la lecture R2 ← X peut avoir lieu dès le début et récupérer la valeur 0, tandis que la lecture R1 ← Y aura lieu en dernier et récupérera la valeur 1 fournie par l'écriture Y ← 1.

Les principes énoncés ci-dessus restent toutefois généraux, chacune des architectures mentionnées dans le Tableau 1.1 étant susceptible de présenter des variations substantielles dans ses comportements. Ces particularités dépassent toutefois le cadre de cette thèse.

Par ailleurs, les problématiques liées aux modèles mémoires faibles concernent également les langages de programmation mettant en œuvre des processus légers ou *threads* : les instructions d'un thread peuvent être réordonnées, non seulement par le processeur, mais aussi par le compilateur lui-même à des fins d'optimisation. Pour qu'un programmeur puisse avoir des garanties fortes sur le code concurrent qu'il écrit, il est nécessaire que ces langages donnent une sémantique précise aux programmes concurrents et fournissent des primitives de synchronisation adaptées. On retiendra le cas des langages C et C++, qui avant l'arrivée des standards C11 [96] et C++11 [97] en 2011, ne donnaient aucune sémantique aux threads et ne fournissaient aucun mécanisme de synchronisation particulier. Les programmeurs avaient alors recours à des extensions telles que les threads POSIX [94], qui imposent l'absence de situation de compétition mais ne proposent aucune sémantique ou modèle mémoire.

Avec l'explosion du nombre de modèles mémoires faibles existant et la démocratisation de la programmation concurrente, il était nécessaire de définir formellement la sémantique de ces modèles mémoires, d'autant plus que de nombreux constructeurs se contentent de

décrire leurs modèles de façon informelle. Depuis les années 70, et encore plus ces dix dernières années, ces travaux ont conduit à de nombreuses formalisations, dont quelques-unes sont recensées dans le Tableau 1.2.

Modèle	Auteurs	Références
TSO	Sarkar, Sewell, Nardelli, Owens, Ridge, Braibant, Myreen, Alglave, Boudol, Petri	[121, 116, 48, 126]
PSO	Boudol, Petri	[48]
RMO	Park, Hill	[117]
Alpha	Attiya, Friedman	[36]
Power	Adir, Attiya, Shurek, Alglave, Fox, Ishtiaq, Myreen, Sarkar, Sewell, Nardelli, Maranget, Williams, Mador-Haim, Memarian, Owens, Alur, Martin, Gray, Kerneis, Mulligan, Pulte	[14, 30, 122, 109, 86]
ARM	Chong, Alglave, Fox, Ishtiaq, Myreen, Sarkar, Sewell, Nardelli, Gray, Sezgin, Maranget, Pulte, Flur, Deacon, French	[58, 30, 75, 119]
IA-64	Chatterjee, Gopalakrishnan, Higham, Jackson, Kawash	[56, 1, 89]
C11/C++11	Boehm, Adve, Batty, Owens, Sarkar, Sewell, Weber, Vafeiadis, Narayan, Nienhuis, Memarian	[43, 40, 134, 132, 114]
Java	Manson, Pugh, Adve, Cenciarelli, Knapp, Sibilio	[110, 55]

TABLEAU 1.2 – Formalisations de différents modèles mémoires faibles

Grâce à ces formalisations, différentes approches ont pu être adoptées afin de raisonner sur la correction des programmes concurrents s'exécutant sur ces modèles. En particulier, de nombreux outils d'analyse automatique ont pu être développés - on en présente un certain nombre dans le Chapitre 2. Toutefois, la quasi-totalité de ces approches n'offrent des garanties que pour un nombre fixe (et petit) de processus. On voudrait pouvoir vérifier des propriétés de sûreté de programmes concurrents s'exécutant sur des modèles mémoires faibles indépendamment du nombre de processus impliqués.

Une approche efficace permettant la vérification de propriétés de sûreté sur des systèmes concurrents est le model checking. Initialement limitée aux systèmes finis, l'approche a rapidement été étendue aux systèmes infinis, et en particulier aux systèmes paramétrés [59, 77, 9, 3], c'est-à-dire composés d'un nombre non-borné de composants. Une technique efficace employée pour vérifier de tels systèmes est le *model checking modulo theories* (MCMT), développé par *Ghilardi et Ranise* [80, 79], et mis en œuvre dans le model checker Cubicle [111].

Cubicle permet la vérification de propriétés de sûreté de programmes concurrents représentés par des systèmes de transitions manipulant des tableaux indicés par des identificateurs de processus et s'exécutant sur un modèle mémoire SC. L'objectif de cette thèse consiste alors à étendre cet outil afin que l'analyse s'effectue sur un modèle mémoire faible. On s'intéressera également à la possibilité de traduire automatiquement des programmes écrits en langage d'assemblage x86 vers le langage d'entrée de Cubicle afin de faciliter leur vérification.

Nos contributions sont donc les suivantes :

— une approche permettant la vérification de programmes concurrents sur des modèles mémoires faibles
— une extension du model checker Cubicle implémentant cette approche, nommée Cubicle-$\mathcal{W}$ et disponible à l'adresse : `http://cubicle.lri.fr/cubiclew`
— un schéma de traduction de programmes x86 vers Cubicle-$\mathcal{W}$ prouvé correct
— un outil de traduction automatique basé sur ce schéma de traduction, disponible à l'adresse : `https://www.lri.fr/~declerck/pmcx86`

2

Etat de l'art

Lorsque l'on souhaite s'assurer qu'un programme s'exécutant sur un modèle mémoire faible soit correct, on peut avoir recours à différentes approches. La première consiste à s'assurer que tout programme se comporte comme s'il était exécuté sur un modèle SC (pour une certaine notion de comportement). La seconde consiste à vérifier que le programme, exécuté sur un modèle mémoire faible donné, respecte les propriétés de sûreté que l'on attend de ce programme, même s'il montre à l'exécution plus de comportements que sur un modèle SC. En général, les outils permettant de vérifier ces propriétés permettent également d'ajouter des primitives de synchronisation afin de s'assurer que la propriété soit respectée.

La quasi-totalité de ces outils analysent des programmes pour un nombre fixe (et généralement petit) de processus. Une exception notable est l'outil Dual-TSO, qui permet la vérification de propriétés de sûreté pour une certaine forme de programmes paramétrés. Cet outil étant de par son objectif le plus proche de notre travail, nous lui consacrons une section de ce chapitre.

2.1 Restriction aux comportements SC

Une bonne façon de s'assurer qu'un programme s'exécutant sur un modèle mémoire faible soit correct consiste à restreindre ses comportements à ceux effectivement possibles sur un modèle SC. Cette approche est en général plus simple que de vérifier des propriétés de sûreté d'un programme s'exécutant directement sur un modèle mémoire faible. Grâce à cette garantie, on peut alors utiliser tout l'éventail des techniques de vérification utilisables en SC.

Il existe différentes façons de caractériser les comportements d'un programme. Certaines sont basées sur l'utilisation de traces d'exécution, d'autres sur des états concrets. De même, il existe différentes façons de vérifier et d'imposer qu'un programme ne présente que des comportements SC. L'approche générale consiste à détecter les comportements potentiellement non réalisables en SC et à ajouter judicieusement des primitives de synchronisation (barrières mémoires, instructions de *lecture-modification-écriture* atomiques)

afin d'interdire ces comportements. Toutefois, il est important de ne pas utiliser trop de ces primitives, au risque d'impacter négativement la performance des programmes. La plus simple des approches consiste simplement à s'assurer que le programme ne comporte aucune situation de compétition ou *data race*. Cette approche est en général simple à mettre en œuvre mais génère un nombre important de synchronisations. Partant du constat que certaines situations de compétition sont bénignes, une approche un peu moins stricte consiste à n'interdire que les situations de compétition dites "triangulaires" [115], c'est-à-dire celles dans lesquelles la lecture est précédée d'une écriture, sans qu'aucune primitive de synchronisation ne les sépare. Là aussi, l'approche est relativement simple à mettre en œuvre, et génère moins de primitives de synchronisation que l'approche précédente. Enfin, d'autres approches plus précises étudient plus finement les comportements des programmes de façon à interdire spécifiquement certaines traces d'exécution non-SC (on parle alors de stabilité, ou robustesse, mais également de persistence). Beaucoup de ces travaux reposent sur l'utilisation des notions de trace, de cycle critique, et d'analyse de délais, telles que définies par *Shasha et Snir* dans [127].

Pensieve Java Compiler [73, 128] (introuvable) est un compilateur expérimental pour le langage Java, conçu pour analyser l'effet des modèles mémoires faibles (*i.e. x86 et Power*). Il implémente l'algorithme décrit dans [103], basé sur l'analyse de délais et de cycles critiques de *Shasha et Snir*. Différentes stratégies d'insertion de barrières mémoires sont mises en œuvre, comparées et évaluées. La connaissance des garanties offertes par le modèle sous-jacent est utilisée pour réduire le nombre d'instructions de synchronisation utilisées.

Checkfence [50] (`http://checkfence.sourceforge.net`) analyse des programmes écrits en C (CIL) et s'exécutant sur un modèle mémoire ad-hoc appelé *Relaxed* [49], qui capture l'essentiel des modèles TSO, PSO, RMO et Alpha. Il encode le programme C en un problème SAT, qu'il décharge sur un solveur SAT, qui construit un contre-exemple s'il existe une exécution qui diffère de SC. Dans ce cas, des barrières mémoires sont ajoutées afin de rendre cette exécution irréalisable.

Sober [51] (introuvable) analyse des programmes en C# et s'exécutant sur le modèle TSO. Il utilise un algorithme de moniteur pour détecter des exécutions dites "limites", c'est-à-dire des exécutions SC pouvant être prolongées en une exécution non-SC en ajoutant une seule instruction à la séquence. Cet algorithme s'appuie sur le model checker CHESS [113]. L'outil n'ajoute toutefois pas de primitives de synchronisation pour restaurer un comportement SC.

Offence [24] (`http://offence.inria.fr`) permet d'analyser des programmes écrits en langage d'assemblage x86 et Power et s'exécutant sur leurs modèles respectifs. Il est également basé sur les travaux de *Shasha et Snir*, adaptés aux modèles modernes, et permet l'ajout de barrières mémoires, d'instructions atomiques et de verrous, afin de garantir la stabilité. Différentes solutions sont générées, et leur coût est évalué, de façon à choisir la solution la plus efficace.

DFence [108] (`https://github.com/eth-srl/DFENCE`) analyse des programmes C (LLVM) sur les modèles TSO et PSO. Il utilise un ordonnanceur dit "démoniaque", qui tente de générer des exécutions non-SC. Si une telle exécution est rencontrée, des primitives

de synchronisation sont ajoutées et une nouvelle analyse est lancée.

Trencher [44] (`https://tcs.cs.tu-bs.de/trencher.html`) analyse des systèmes de transitions sur le modèle TSO. Basé sur des travaux précédents [46], il réduit le problème de robustesse au problème d'atteignabilité sur SC, en instrumentant le programme source de différentes façons afin de détecter toutes les exécutions non-SC. Il cherche pour cela des violations minimales (cycles) dans les exécutions, puis énumère toutes les solutions permettant de rétablir la robustesse. Ces solutions sont encodées sous forme de problème ILP (*Integer Linear Programming*), et un solveur se charge de déduire la moins chère.

Musketeer [26, 27] (`www.cprover.org/wmm/fence13`) analyse des programmes écrits en C et supporte les sémantiques x86-TSO, Power et ARM. Il utilise un modèle axiomatique pour exprimer la sémantique des programmes, et recherche la présence de cycles critiques (au sens de *Shasha et Snir*) pour déterminer comment placer les diverses primitives de synchronisation permettant de rétablir la "stabilité" du programme analysé.

Persist [2] (`https://github.com/PhongNgo/persistence`) analyse des systèmes de transitions sur le modèle TSO (en utilisant le langage d'entrée de Trencher). Le critère utilisé pour s'assurer qu'un programme n'expose que des comportements SC est celui de la *persistance*, une notion qui diffère de la robustesse (ou stabilité). Cette notion s'appuie sur des traces représentant l'ordre des opérations par processus et l'ordre dans lequel les écritures sont propagées en mémoire. Un programme sera considéré comme persistant s'il génère les mêmes traces en SC et en TSO. Dans ce cas les états atteignables seront les mêmes en SC et en TSO. Si un programme n'est pas persistant, un algorithme guidé par les contre-exemples permet d'ajouter un nombre minimal de primitives de synchronisation pour rétablir la persistance.

2.2 Vérification de propriétés de sûreté

Une autre approche permettant de s'assurer de la correction de programmes s'exécutant sur des modèles mémoires faibles consiste à vérifier directement les propriétés de sûreté de ces programmes, en prenant en compte le modèle mémoire sous-jacent. Atig *et al.* s'intéressent à la décidabilité des problèmes d'atteignabilité sur les classes de modèles TSO, PSO et RMO [34, 35]. Par réduction vers des files FIFO avec pertes, il est prouvé que le problème d'atteignabilité est décidable sur les modèles TSO et PSO [4], bien que de complexité élevée (non-primitive recursive [123]). Le problème est toutefois indécidable sur RMO.

Mmchecker [93] (`http://www.comp.nus.edu.sg/~release/mmchecker`) est un vérificateur d'invariants qui analyse des programmes C# sur le modèle mémoire .NET. L'analyse s'effectue au niveau du bytecode, par une exploration en avant et avec un algorithme DFS, en utilisant une représentation concrète des états de la machine .NET. Une première passe est effectuée pour vérifier que le programme respecte les invariants donnés sur un modèle SC, puis une seconde passe est effectuée pour détecter les exécutions non-SC ne respectant pas ces invariants. Un algorithme de flot-max/coupe-min est alors utilisé pour insérer le plus petit nombre possible de barrières mémoires permettant de rétablir les invariants.

Remmex [106, 105, 104] (`http://www.montefiore.ulg.ac.be/~linden`) analyse des systèmes décrits dans un langage Promela simplifié et s'exécutant sur les modèles TSO et PSO. L'originalité de l'approche repose sur l'utilisation d'automates finis pour représenter le contenu (potentiellement infini) des tampons d'écriture. L'analyse s'effectue en avant grâce à un algorithme DFS, et utilise une technique de réduction d'ordre partiel pour limiter le nombre d'états visités. Dès qu'une exécution viole une propriété de sûreté, une barrière mémoire est ajoutée pour rendre cette exécution infaisable, et l'analyse recommence jusqu'à ce qu'aucune violation ne soit plus détectée.

Fender [100] (introuvable) analyse des systèmes de transition finis exprimés dans un pseudo-langage d'assemblage et s'exécutant sur un modèle mémoire ad-hoc (RLX) qui englobe plusieurs modèles mémoires faibles (TSO, PSO, et partiellement RMO). Après avoir généré l'ensemble des états atteignables, il calcule pour chaque état une formule dite d'évitement, qui représente les instructions dont l'ordre doit être préservé par une primitive de synchronisation pour rendre l'état inatteignable. La formule d'évitement dépendant des états précédents dans la relation de transition, le processus est répété itérativement jusqu'à atteindre un point fixe. La formule finale est calculée en effectuant la disjonction de toutes les formules d'évitement correspondant à des états d'erreur. De cette formule finale on déduit l'ensemble de barrières minimal à insérer pour rétablir la sûreté du programme.

Blender [101] (introuvable) analyse des programmes finis s'exécutant sur les modèles TSO et PSO. Basé sur des techniques d'interprétation abstraite, il s'appuie sur une abstraction bornée du contenu des tampons d'écriture. Lorsqu'une violation d'une propriété de sûreté est détectée, des barrières mémoires sont ajoutées afin de la rétablir.

Java PathRelaxer (JPR) [98] (introuvable) est une extension de Java Pathfinder (JPF) permettant de prendre en compte le modèle mémoire de Java lors de l'analyse de programmes. Partant de l'ensemble des exécutions valides en SC, cet ensemble est itérativement enrichi de nouvelles exécutions non-SC en considérant, pour chaque lecture de chaque exécution, d'autres écritures ayant pu les satisfaire, tout en respectant un certain nombre de contraintes excluant des exécutions non réalisables. L'ensemble des exécutions obtenues est une sur-approximation des exécutions effectivement réalisables sur la JVM.

MEMORAX [8, 6, 10, 11] (`https://github.com/memorax/memorax`) est un outil permettant l'analyse de programmes écrits dans un langage ad-hoc et s'exécutant sur les modèles TSO et PSO. Il a recours à deux abstractions pour représenter le contenu des tampons. La première abstraction, *single-buffer* (SB), consiste à encoder les tampons de tous les processus en un unique tampon contenant des images mémoire complètes. De cette façon, il existe un ordre sur les configurations, ce qui permet d'avoir recours aux systèmes de transitions bien structurés. La seconde abstraction, *predicate abstraction with buffer bounding* (PB), consiste à trouver une abstraction finie des tampons d'écriture en se limitant aux k écritures les plus anciennes qu'ils contiennent, en augmentant itérativement k si nécessaire. Quelle que soit l'abstraction utilisée, lorsqu'un contre-exemple est découvert, des primitives de synchronisation sont ajoutées afin d'exclure ce contre-exemple, et une nouvelle analyse est effectuée.

Goto-instrument [29] (`http://www.cprover.org/wmm/esop13`) transforme des programmes C afin que leur exécution SC simule des exécutions non-SC (TSO, PSO, RMO, et

un sous-ensemble de Power). Cette transformation utilise une machine abstraite comportant des tampons d'écriture et des files de lecture pour simuler les modèles plus relâchés. Cela permet ensuite d'utiliser n'importe quel outil de vérification SC sur les programmes obtenus.

CBMC [23] (`http://www.cprover.org/cbmc/`) est un model checker borné pour les programmes écrits en C, initialement conçu pour le modèle SC. De ce fait, il ne peut garantir la correction que pour des programmes finis. Lorsqu'un programme présente des boucles infinies, celles-ci peuvent être déroulées un certain nombre de fois, la correction n'est donc garantie que pour un certain ordre de déroulage de boucles. Son extension aux modèles mémoires faibles s'appuie sur l'approche axiomatique d'Alglave *et al.* [28]. Les exécutions sont représentées par des événements et des relations sur ces événements, auxquels on associe des horloges devant respecter différentes contraintes. Une exécution est considérée comme valide s'il existe une affectation de ses horloges qui respecte les contraintes énoncées.

Trencher [47] (`https://tcs.cs.tu-bs.de/trencher.html`), présenté en section précédente, a également été étendu pour permettre la vérification de propriétés de sûreté définies en terme d'atteignabilité. Il vérifie d'abord si les états dangereux sont atteignables en SC. Si ce n'est pas le cas, un oracle est utilisé pour proposer des séquences d'instructions susceptibles d'être problématiques en TSO. Le programme est alors enrichi pour simuler en SC ces exécutions non-SC, et une nouvelle analyse a lieu. Le procédé est répété itérativement. L'oracle utilise l'analyse de robustesse déjà mise en œuvre dans Trencher afin de suggérer les séquences d'instructions pour lesquelles la sémantique TSO devrait être utilisée. Cette approche permet la recherche rapide de bugs mais ne garantit pas la correction d'un programme.

Nidhugg [12] (`https://github.com/nidhugg/nidhugg`) est un model checker pour les programmes C s'exécutant sur les modèles TSO et PSO. Il analyse directement le code assembleur produit par LLVM. Dans cette approche, les exécutions sont représentées sous formes de *traces chronologiques*, qui correspondent en fait à la notion de traces de *Shasha et Snir*. Cette représentation permet d'exprimer une relation d'ordre partiel sur les événements constituant une exécution et de mettre en œuvre des techniques de réduction d'ordre partiel.

2.3 Vérification paramétrée de propriétés de sûreté : Dual-TSO

L'outil **Dual-TSO** [13, 5] (`https://github.com/PhongNgo/memorax`) est basé sur l'architecture et langage d'entrée de MEMORAX. Il permet la vérification de propriétés de sûreté de programmes paramétrés manipulant un nombre fini de variables sur des domaines finis et s'exécutant sur le modèle TSO. Sa particularité est de mettre en œuvre une approche duale aux tampons d'écriture de TSO, en les remplaçant par des tampons de lecture. Ces tampons FIFO représentent les opérations de lecture potentielles des processus.

Plus précisément, un tampon contient des messages de la forme (x, v) et (x, v, own), représentant la lecture de la valeur v dans la variable x depuis la mémoire ou depuis le

tampon du processus ayant réalisé l'écriture (own). Lorsqu'une écriture $x \leftarrow v$ a lieu, elle prend effet immédiatement en mémoire, mais elle est également ajoutée sous la forme (x, v, own) en queue du tampon de lecture du processus effectuant l'opération. Par ailleurs, toute variable x en mémoire et sa valeur v peuvent être propagées sous la forme (x, v) vers la queue de n'importe quel tampon de lecture, de façon non déterministe. De plus, tout message en tête d'un tampon peut être supprimé de façon non déterministe. Un processus peut effectuer une lecture $x = v$ dès lors que le message en tête de son tampon de lecture est (x, v) et qu'il ne contient aucun message de la forme (x, v', own). Si le tampon du processus contient un tel message, alors la valeur lue doit correspondre à la valeur du plus ancien message de cette forme (c'est-à-dire le message le plus proche de la tête du tampon). Les barrières mémoires et les instructions atomiques sont réalisées de la même façon qu'avec la sémantique TSO classique, c'est-à-dire en considérant que les tampons sont vides avant et après l'opération.

Cette modélisation possède les mêmes propriétés qu'une modélisation à base de tampons d'écriture en termes d'atteignabilité, mais permet une meilleure efficacité. Par ailleurs, par correspondance avec les systèmes de transitions bien structurés, il est démontré que le problème d'atteignabilité avec cette modélisation est décidable.

3

Présentation de Cubicle-$\mathcal{W}$

Sommaire

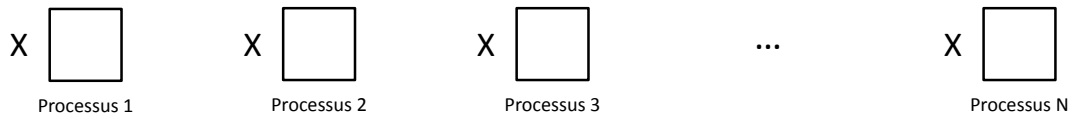
L'un des objectifs de cette thèse consistait à réaliser une extension du model checker Cubicle pour permettre la vérification de propriétés de sûreté de systèmes paramétrés s'exécutant sur des modèles mémoires faibles. Notre approche est centrée sur un modèle similaire à TSO, mais peut être généralisée à la famille des modèles exprimables avec différentes sortes de tampons d'écriture. Nous avons nommé cette extension Cubicle-$\mathcal{W}$ [1]. Dans ce chapitre, on présente cet outil du point de vue de l'utilisateur. On présente d'abord le model checker Cubicle classique (*i.e.* sans mémoire faible) et son langage d'entrée, puis on introduit notre modèle de mémoire faible, avant de donner nos extensions de syntaxe. On illustre enfin notre propos par des exemples.

1. $\mathcal{W}$ faisant référence à la première lettre du mot *weak*

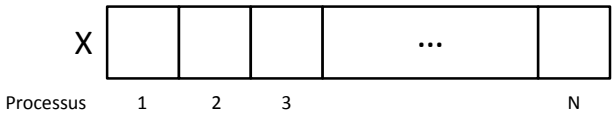
3.1 Rappels sur Cubicle

3.1.1 Des tableaux paramétrés

Dans Cubicle, on s'intéresse aux programmes paramétrés, c'est-à-dire aux programmes composés d'un nombre arbitraire de processus s'exécutant de façon concurrente. Ces programmes sont exprimés à l'aide de systèmes de transitions manipulant des variables partagées par tous les processus. Dans ce contexte, on a souvent besoin d'exprimer qu'une variable est spécifique à un certain processus : dans le schéma ci-dessous, chaque processus possède une variable `X`, on a donc autant de variables `X` que de processus.



Pour représenter ces variables, on utilise des tableaux indicés par des identificateurs de processus, comme exposé dans le schéma suivant. Le nombre de processus n'étant pas fixé, les tableaux manipulés sont donc non bornés.



Du fait de l'aspect paramétré de notre approche, on ne peut parler d'un processus possédant un identificateur spécifique. On peut en revanche utiliser les quantifications existentielles et universelles sur les processus, de façon à pouvoir parler d'une, de plusieurs, ou de toutes les cases d'un tableau. Par ailleurs, une relation d'ordre sur les identificateurs de processus permet de situer les processus les uns par rapport aux autres, on pourra donc préciser qu'une case d'un tableau est à gauche ou à droite d'une autre.

La Figure 3.1 donne quelques exemples de formules décrivant de tels tableaux. Une formule ne décrit pas nécessairement un unique tableau, mais plus souvent une famille de tableaux.

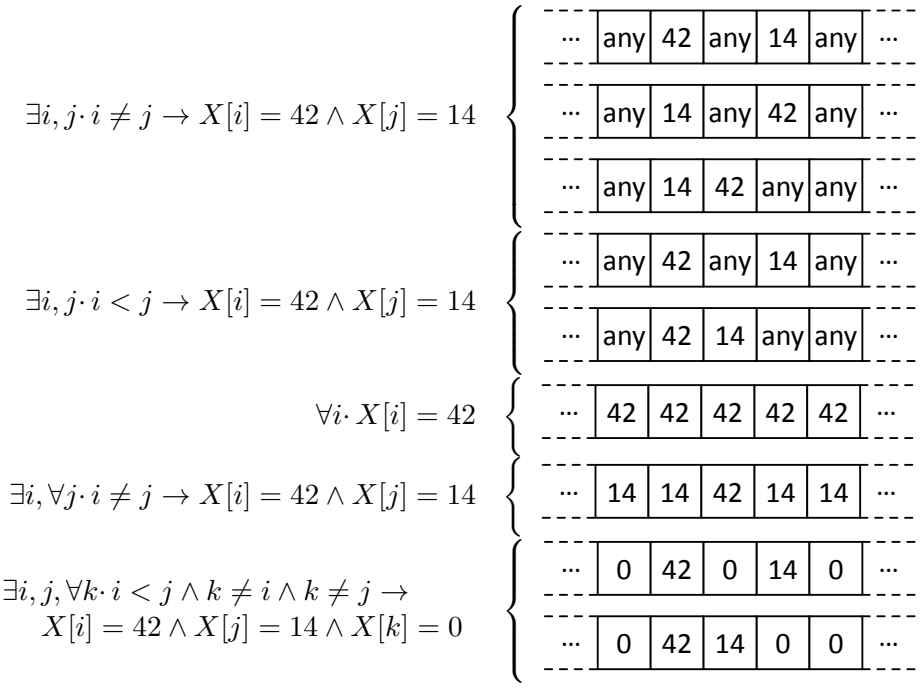


FIGURE 3.1 – Exemples de tableaux décrits par des formules logiques

3.1.2 Langage d'entrée

Cubicle permet d'exprimer et de vérifier des propriétés de sûreté de programmes concurrents décrits à l'aide de systèmes de transitions, paramétrés par un nombre arbitraire de processus. On décrit dans cette section le langage d'entrée de l'outil de manière informelle.

Cubicle permet de manipuler des variables de type entier (`int`), réel (`real`), booléen (`bool`), ainsi que des identificateurs de processus (`proc`). La cardinalité du type `proc` n'est pas précisée : elle correspond au paramètre du système. On ne peut manipuler directement les valeurs de ce type, toutefois il est muni d'une relation d'ordre : on peut donc comparer deux variables de type `proc`. L'utilisateur peut également déclarer des types énumérés à l'aide du mot-clé `type`. Dans le code suivant, on déclare un type `loc` muni des constructeurs `L1`, `L2`, `L3`, `CS` et `End`.

```
type loc = L1 | L2 | L3 | CS | End
```

L'état du système se compose de variables et de tableaux indicés par des processus. Par exemple, dans le code qui suit, on déclare une variable `X` de type `int`, un tableau `PC` du type `loc` déclaré précédemment, et un tableau `T` de type `bool`.

```
array PC[proc] : loc
array T[proc] : bool
var X : int
```

De façon similaire, on peut également déclarer des constantes ou des tableaux de constantes, à l'aide du mot-clé `const`. La valeur de ces constantes peut être précisée dans la description des états initiaux du système, ou bien grâce à un invariant spécifique. Dans le code qui suit, on déclare une constante `C` de type `int` et un tableau de constantes `TC` de type `bool`.

```
const C : int
const TC[proc] : bool
```

Les états initiaux du système sont décrits par une conjonction de littéraux introduite par le mot-clé `init` et éventuellement paramétrée par une ou plusieurs variables de type `proc`, qui sont implicitement quantifiées universellement. Ils permettent de préciser les contraintes initiales portant sur les variables et tableaux du système. Il s'agit essentiellement de relations entre variables, tableaux et constantes. L'exemple suivant définit les états initiaux comme l'ensemble des états tels que quel que soit `p`, `X = 0`, `PC[p] = L1` et `T[p] = False`.

```
init (p) { PC[p] = L1 && X = 0 && T[p] = False }
```

Les états dangereux du système sont décrits par une ou plusieurs conjonctions de littéraux introduites par le mot-clé `unsafe`, et paramétrées par une ou plusieurs variables de type `proc` qui sont implicitement quantifiées existentiellement. Comme pour les états initiaux, ces littéraux sont des relations entre variables, tableaux et constantes. Par exemple, le code suivant définit les états dangereux comme l'ensemble des états tel qu'il existe deux processus distincts `p` et `q` pour lesquels `PC[p] = CS`, `PC[q] = CS`, et `X = 42`.

```
unsafe (p q) { PC[p] = CS && PC[q] = CS && X = 42 }
```

De façon similaire, on peut également donner des invariants du système, sous forme négative, à l'aide du mot-clé `invariant`. Ces invariants permettent de considérer certains états comme non-atteignables. Il appartient à l'utilisateur de s'assurer que les invariants fournis soient corrects. Ces invariants permettent aussi de préciser les contraintes relatives aux constantes définies plus tôt. Par exemple, l'invariant suivant précise que le tableau de constantes `TC` ne contient pas plus d'une fois la valeur `True`.

```
invariant (p q) { TC[p] = True && TC[q] = True }
```

Le cœur du système est décrit par un ensemble de transitions composées d'une garde et d'un ensemble d'actions, éventuellement paramétrées par des variables de type `proc` implicitement quantifiées existentiellement et toutes distinctes. La garde est une conjonction de littéraux représentant les conditions devant être vérifiées pour que la transition soit prise. Comme précédemment, il s'agit de relations entre variables, tableaux et constantes. Les actions sont des mises à jour de variables et tableaux. Le code suivant représente une transition paramétrée par deux processus `i` et `j` *différents*. Cette transition est prise si `PC[i] = L1` et `T[j] = False`, et a pour effet de mettre à jour `PC[i]` avec la valeur `L2` et `T[j]` avec la valeur `True`, ainsi que d'incrémenter `X` de 1.

```
transition t1 (i j)
requires { PC[i] = L1 && T[j] = False }
{ PC[i] := L2; T[j] := True; X := X + 1 }
```

La garde peut aussi préciser l'état des cases d'un tableau pour l'ensemble de ses indices autres que les indices correspondant aux paramètres de la transition, à l'aide d'une sous-formule quantifiée universellement introduite par le mot-clé `forall_other`. Par exemple, la transition suivante est prise s'il existe un processus `i` tel que `PC[i] = L1` est vrai et pour tout processus `j` *différent* de `i`, `T[j] = False`.

```
transition t2 (i)
requires { PC[i] = L1 && forall_other j. T[j] = False }
{ PC[i] := L2 }
```

Les mises à jour de tableaux peuvent également porter sur la totalité des indices, en utilisant une quantification universelle introduite par la construction `case`. Chaque branche de cette construction porte une condition qui permet de restreindre les indices sur lesquels elle s'applique, mais aussi de donner des conditions supplémentaires. Elle doit obligatoirement se terminer par une branche par défaut, dont la condition est exprimée par `_`, et qui indique le cas où aucune autre branche ne s'applique. Par ailleurs, dès lors que l'on souhaite mettre à jour au moins deux cases d'un même tableau dans une même transition, l'utilisation de la construction `case` devient obligatoire. Dans le code suivant, lorsque la transition est prise, elle met à jour `PC[i]` avec la valeur `L2` et toutes les cases du tableau `T` selon les indices : toutes les cases d'indice inférieur à `i` prennent pour valeur `True`, et toutes les autres cases prennent pour valeur `False`.

```
transition t3 (i)
requires { PC[i] = L1 }
{ PC[i] := L2; T[j] := case | j < i : True | _ : False }
```

Les transitions sont exécutées de façon non-déterministe, et de manière atomique : entre l'évaluation de la garde et l'application des mises à jour, aucune autre transition ne peut commencer son exécution.

3.1.3 Exemple 1 : Mutex Naïf

Pour illustrer le langage et la sémantique de Cubicle, on commence par donner comme exemple un algorithme d'exclusion mutuelle naïf (inefficace et avec interblocage). Dans cet algorithme paramétré, chaque processus se comporte selon l'automate donné en Figure 3.2.

Chaque processus i possède une variable booléenne $\texttt{X}_i$ valant initialement *faux*. Un processus peut être soit dans l'état `Idle`, c'est-à-dire en attente, soit dans l'état `Want` pour signifier qu'il veut entrer en section critique, soit dans l'état `Crit` pour indiquer qu'il est en section critique. Pour passer de `Idle` à `Want`, un processus i passe sa variable $\texttt{X}_i$ à *vrai*. Pour passer de `Want` à `Crit`, le processus i vérifie que $\texttt{X}_j$ est *faux* pour tous les autres processus j. Enfin pour passer de `Crit` à `Idle`, le processus i repasse $\texttt{X}_i$ à *faux*.

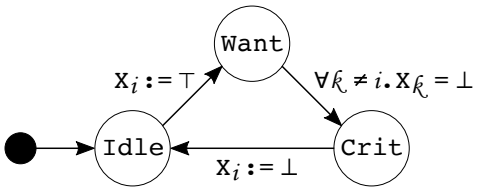


FIGURE 3.2 – Algorithme de Mutex Naïf

Pour modéliser cet algorithme avec Cubicle (Figure 3.3), on commence par définir un type `state` pour représenter les états de l'automate, c'est-à-dire `Idle`, `Want` et `Crit`, ainsi qu'un tableau `State` de ce type pour indiquer l'état courant de chaque processus. On déclare ensuite un tableau `X` de type `bool`. La formule initiale précise que pour tous les processus, `X` vaut initialement `False` et que `State` est égal à `Idle`. On s'intéresse à la propriété d'exclusion mutuelle de cet algorithme. Les états dangereux sont donc définis comme les états tels que `State` est égal à `Crit` pour au moins deux processus.

```
type state = Idle | Want | Crit

array State[proc] : state

array X[proc] : bool

init (i) {
  State[i] = Idle && X[i] = False }

unsafe (i j) {
  State[i] = Crit && State[j] = Crit }
```

```
transition t_req (i)
requires { State[i] = Idle }
{ State[i] := Want; X[i] := True }

transition t_enter (i)
requires { State[i] = Want &&
           forall_other k. X[k] = False }
{ State[i] := Crit }

transition t_exit (i)
requires { State[i] = Crit }
{ State[i] := Idle; X[i] := False }
```

FIGURE 3.3 – Code Cubicle du Mutex Naïf

Les transitions du système correspondent exactement aux transitions de l'automate. Elle sont toutes paramétrées par un unique processus `i`. La transition `t_req` modélise le passage de `Idle` à `Want` : pour que la transition soit prise par un processus `i`, il faut que `State[i]` soit égal à `Idle`, et son effet est de mettre à jour `State[i]` avec la valeur `Want` et `X[i]` avec la valeur `True`. `t_enter` représente le passage de `Want` à `Crit` : la transition est prise par `i` lorsque `State[i]` est égal à `Want` et que pour tout processus `k` *différent* de `i`, `X[k]` vaut `False`, et elle met à jour `State[i]` avec la valeur `Crit`, indiquant que le processus `i` entre en section critique. La dernière transition `t_exit` modélise la sortie de la section critique, c'est-à-dire le passage de `Crit` à `Idle` : elle est prise par `i` lorsque `State[i]` vaut `Crit`, et met à jour `State[i]` avec la valeur `Idle` et `X[i]` avec la valeur `False`.

L'exécution de Cubicle sur ce système produit la sortie suivante :

```
The system is SAFE
```

Ceci nous indique que ce système est sûr quant à la propriété de sûreté exprimée.

Il faut être vigilant quant aux paramètres que l'on donne à une transition. En particulier, un paramètre non utilisé n'est pas anodin : la construction `forall_other` quantifie sur une variable de processus *différente* de tous les paramètres de la transition, qu'ils soient utilisés ou non. Imaginons que la transition `t_enter` soit écrite comme suit :

```
transition t_enter (i j)
requires { State[i] = Want &&
           forall_other k. X[k] = False }
{ State[i] := Crit }
```

Dans ce cas, l'exécution de Cubicle sur le système ainsi modifié produit la sortie suivante :

```
Unsafe trace: t_req(#1) -> t_req(#2) -> t_enter(#1, #2) ->
              t_enter(#2, #1) -> unsafe[1]

unsafe[1] is reachable !

UNSAFE !
```

La trace fournie indique que deux processus `#1` et `#2` ont tous deux pu entrer en section critique en prenant tous les deux `t_req` puis `t_enter`. En effet, de par l'introduction du paramètre supplémentaire `j`, le `forall_other` ne s'applique pas à la case `j` du tableau `X`, qui peut contenir n'importe quelle valeur. La transition peut donc être prise, et le système devient incorrect.

3.1.4 Exemple 2 : Protocole MESI

Notre deuxième exemple est une abstraction du protocole de cohérence de cache MESI, où l'on ne considère l'état que d'une ligne de cache. Dans ce protocole, chaque processus agit selon l'automate donné en Figure 3.4. Chaque processus est soit dans l'état `I`, signifiant que la ligne de cache est invalide, soit dans l'état `E`, signifiant que la ligne de cache est valide et que le processus est seul à la posséder, soit dans l'état `S`, signifiant que la ligne de cache est valide et que d'autres processus en possèdent une copie, soit dans l'état `M`, signifiant que le processus a modifié sa copie de la ligne de cache et qu'aucun autre processus n'en possède une copie.

Un processus change d'état sous l'effet d'une de ses lecture ou écriture, qui sont matérialisées par les arcs pleins étiquetés. L'étiquette `read+` indique une lecture non-exclusive, c'est-à-dire une lecture qui a lieu lorsqu'au moins un autre processus n'est pas en `I`. `readx` indique quant à elle une lecture exclusive, c'est-à-dire une lecture ayant lieu lorsque tous les autres processus sont en `I`. `write` indique une écriture. En réaction, les autres processus sont affectés selon leur état courant et l'opération réalisée, ce qui est matérialisé par les arcs en pointillés. L'étiquette `read*` indique une réaction à une lecture d'un autre processus, tandis que `write*` indique une réaction à une écriture.

Pour modéliser cet algorithme avec Cubicle (Figure 3.5), on définit un type `location` représentant les états de l'automate, c'est-à-dire `M`, `E`, `S` et `I`, et un tableau `State` de ce type

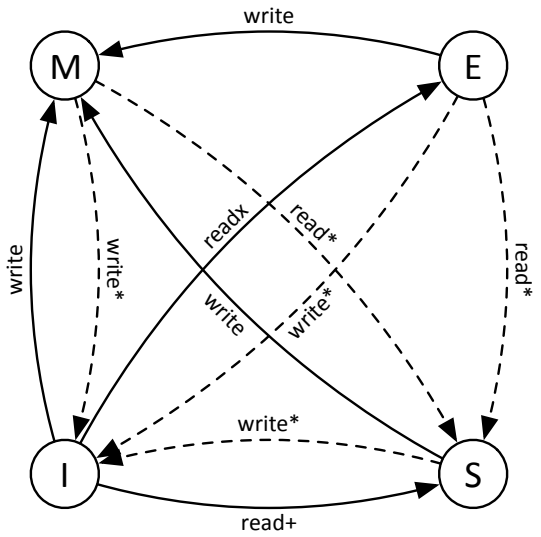


FIGURE 3.4 – Protocole MESI simplifié

pour représenter l'état courant de chaque processus. Initialement, les processus sont tous à l'état `I`. Cet algorithme garantit plusieurs propriétés, mais on s'intéresse uniquement à la propriété de sûreté suivante : deux processus ne peuvent être simultanément en `M`.

```
type location = M | E | S | I

array State[proc] : location

init (p) { State[p] = I }

unsafe (p1 p2) {
  State[p1] = M && State[p2] = M }

transition t_I_S (p q)
requires { State[p] = I && State[q] <> I }
{ State[r] := case
  | r = p : S
  | State[r] = E : S
  | State[r] = M : S
  | _ : State[r] }

transition t_I_E (p)
requires { State[p] = I &&
           forall_other q. State[q] = I }
{ State[p] := E }

transition t_I_M (p)
requires { State[p] = I }
{ State[r] := case
  | r = p : M
  | State[r] <> I : I
  | _ : State[r] }

transition t_S_M (p)
requires { State[p] = S }
{ State[r] := case
  | r = p : M
  | State[r] = S : I
  | _ : State[r] }

transition t_E_M (p)
requires { State[p] = E }
{ State[p] := M }
```

FIGURE 3.5 – Code Cubicle du protocole MESI

Le changement d'état d'un processus et la réaction des autres devant être atomique (en apparence), on les combine dans chaque transition. La première transition `t_I_S`

représente le passage de `I` à `S` lorsqu'un processus effectue une lecture non-exclusive, c'est-à-dire lorsqu'au moins un autre processus est dans un état autre que `I`. En réaction, les processus dans les états `E` et `M` passent à l'état `S`. La construction `case` permet d'effectuer la mise à jour simultanée des états de tous les processus en fonction de ces critères. La seconde transition `t_I_E` représente le passage de `I` à `E` lorsqu'un processus effectue une lecture exclusive, c'est-à-dire lorsque tous les processus sont dans l'état `I`. Dans ce cas, les autres processus ne sont pas affectés puisque déjà à l'état `I`. La troisième transition `t_I_M` représente le passage de `I` à `M` lorsqu'un processus effectue une écriture. En réaction, tous les processus n'étant pas déjà à l'état `I` passent à cet état. La quatrième transition `t_S_M` représente le passage de `S` à `M` lorsqu'un processus effectue une écriture. En réaction, tous les processus à l'état `S` passent à l'état `I`. Il n'est pas nécessaire de gérer les cas `M` et `E` : en effet, dès lors qu'au moins un processus est à `S`, aucun autre processus ne peut être à `M` ou `E`. Enfin la cinquième transition `t_E_M` représente le passage de `E` à `M` lorsqu'un processus effectue une écriture. Dans ce cas, les autres processus ne sont pas affectés puisque déjà à l'état `I`.

L'exécution de Cubicle sur cet algorithme nous indique qu'il est correct. Cet algorithme peut facilement devenir incorrect si on oublie de mettre à jour correctement l'état d'un processus. Par exemple, supposons que dans `t_S_M` on oublie de repasser à `I` les processus étant en `S`, comme indiqué dans cette transition :

```
transition t_S_M (p)
requires { State[p] = S }
{ State[r] := case
  | r = p : M
  | _ : State[r] }
```

Dans ce cas, l'exécution de Cubicle sur le système ainsi modifié produit la sortie suivante :

```
Unsafe trace: t_I_M(#1) -> t_I_S(#2, #1) ->
              t_S_M(#1) -> t_S_M(#2) -> unsafe[1]

unsafe[1] is reachable !

UNSAFE !
```

3.1.5 Exemple 3 : Two-Phase Commit

Notre dernier exemple est une abstraction du protocole de Commit à deux Phases, présenté en Figure 3.6. Ce protocole permet à un ensemble de processus de s'accorder sur la validation ou l'abandon d'une transaction. Un processus coordinateur envoie une requête en validation aux autres processus, qui vont chacun indiquer s'ils souhaitent valider ou annuler la transaction. Si au moins un processus souhaite annuler, le coordinateur envoie aux autres processus un ordre d'annulation ; dans le cas contraire, il envoie un ordre de validation. Les processus confirment alors avoir traité l'ordre.

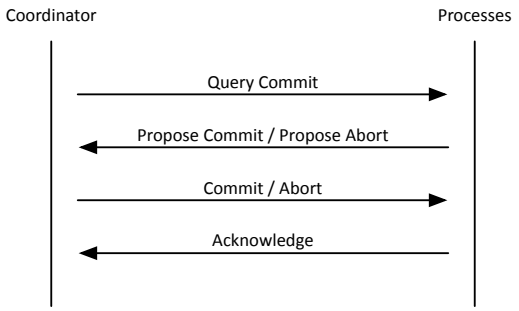


FIGURE 3.6 – Two-Phase Commit

On considère une modélisation très abstraite de ce protocole, où les processus communiquent par mémoire partagée. Par ailleurs on considère que les processus ont déjà reçu la requête en commit, et n'ont pas besoin de confirmer avoir traité l'ordre.

On commence par définir un type `location` et un tableau `PC` de ce type pour représenter l'étape courante de chaque processus. On définit également un type `state` pour représenter les différents états possible : `ReadyCommit` et `ReadyAbort`, signifiant qu'un

```
type location = A1 | A2 | Done

type state = Unknown | ReadyCommit
  | ReadyAbort | Committed | Aborted

array PC[proc] : location
array Astate[proc] : state
var Cstate : state

init (n) {
  PC[n] = A1 &&
  Astate[n] = Unknown &&
  Cstate = Unknown
}

unsafe(m n) {
  Astate[m] = Committed &&
  Astate[n] = Aborted }

transition proposeCommit(n)
requires { PC[n] = A1 }
{
  PC[n] := A2;
  Astate[n] := ReadyCommit
}

transition proposeAbort(n)
requires { PC[n] = A1 }
{
  PC[n] := A2;
  Astate[n] := ReadyAbort
}

transition executeDecision(n)
requires { PC[n] = A2 && Cstate <> Unknown }
{
  PC[n] := Done;
  Astate[n] := Cstate
}

transition decideCommit(i)
requires { Astate[i] = ReadyCommit &&
  forall_other n. Astate[n] = ReadyCommit }
{
  Cstate := Committed
}

transition decideAbort(n)
requires { Astate[n] = ReadyAbort }
{
  Cstate := Aborted
}
```

FIGURE 3.7 – Code Cubicle du Two-Phase Commit

processus souhaite valider ou annuler, `Committed` et `Aborted`, indiquant la validation ou l'annulation, et `Unknown` pour un état quelconque. Le tableau `Astate` de ce type représentera les états des différents processus, tandis que la variable `Cstate` indiquera la décision prise par le coordinateur. Initialement, tous les processus sont en `A1`, et les états sont tous à `Unknown`. Un état sera considéré comme dangereux si deux processus ont pris des décisions différentes, c'est-à-dire si un processus est dans l'état `Committed` et un autre dans l'état `Aborted`.

Les transitions `proposeCommit` et `proposeAbort` permettent aux processus de proposer soit la validation soit l'annulation, en mettant à jour leur tableau `Astate`. La transition suivante, `executeDecision`, est prise lorsque le coordinateur a renseigné son ordre dans `Cstate` : les processus mettent alors à jour leur état dans `Astate` avec la valeur de `Cstate`. Les transitions `decideCommit` et `decideAbort` correspondent au coordinateur : si tous les processus proposent la validation, alors le coordinateur donne l'ordre de validation en mettant à jour `Cstate` avec la valeur `Committed`, tandis que si au mois un processus propose l'annulation, le coordinateur ordonne l'annulation en mettant à jour `Cstate` avec la valeur `Aborted`.

3.1.6 Analyse d'atteignabilité de Cubicle

Dans cette section, on explique brièvement le fonctionnement interne de Cubicle. Le cœur de Cubicle consiste en une analyse d'atteignabilité, qui recherche s'il existe un chemin entre les états initiaux et les états dangereux d'un système. Cette analyse est mise en œuvre par un algorithme d'atteignabilité *en arrière*, c'est-à-dire, partant des états dangereux et prenant les transitions *à l'envers* pour essayer de remonter jusqu'aux états initiaux. Les états et transitions sont manipulés de façon symbolique, grâce à une représentation sous forme de formules logiques.

Cet algorithme d'atteignabilité est rappelé en Algorithme 1. Il prend en entrée un système de transitions $S = (X, I, \tau)$ et un cube Θ, tels que X est l'ensemble des variables, I est la formule décrivant les états initiaux du système, τ est l'ensemble de toutes les transitions, et Θ une formule décrivant les états dangereux. Il maintient un ensemble d'états visités $\mathcal{V}$ et une file de cubes à visiter $\mathcal{Q}$. Initialement, $\mathcal{Q}$ ne contient que Θ (ligne 3). Tant qu'il reste des états à visiter, on récupère un état décrit par une formule φ depuis la file $\mathcal{Q}$ (ligne 5), et on effectue successivement les opérations suivantes :

- test de sûreté (ligne 6) : si $\varphi \wedge I$ est satisfiable, cela signifie que l'état décrit par φ intersecte avec les états initiaux, le système n'est donc *pas sûr*
- test de point fixe (ligne 9) : si $\varphi \not\models \mathcal{V}$ cela signifie que φ décrit un état nouveau par rapport aux états contenus dans $\mathcal{V}$, on l'ajoute alors aux états visités et on effectue l'étape suivante
- calcul de pré-image (ligne 11) : on calcule tous les états qui peuvent mener à φ en une transition, ce que l'on exprime par $\textsc{Pre}_\tau(\varphi)$

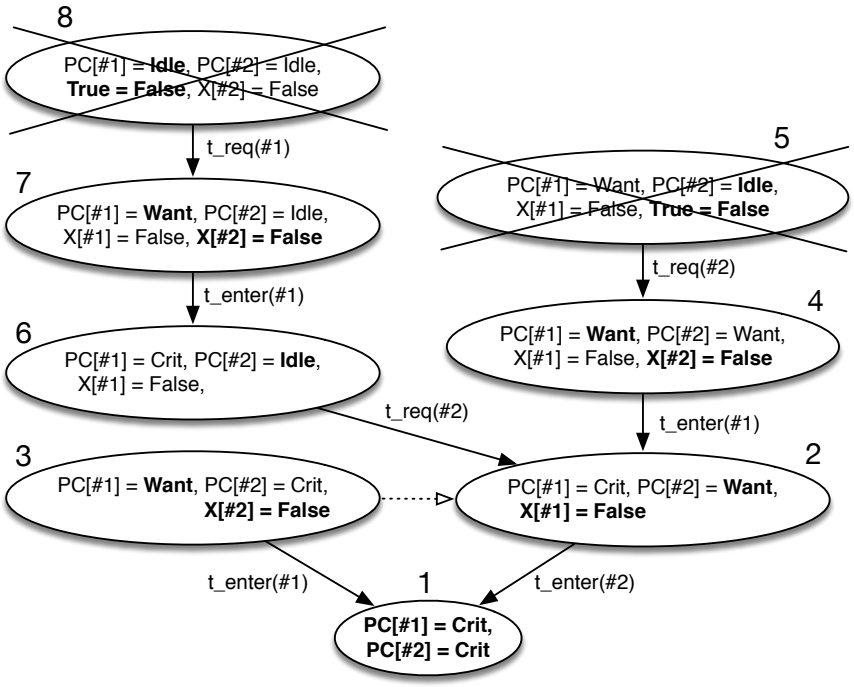
Si à un moment donné la file $\mathcal{V}$ est vide, alors on a exploré tout l'espace d'états sans rencontrer l'état initial, le système est alors considéré comme *sûr*.

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

Algorithme 1 : Algorithme d'atteignabilité arrière de Cubicle

Exemple d'exploration en arrière

On illustre le fonctionnement de cet algorithme dans la figure ci-dessous, qui représente une partie de l'exploration en arrière de l'algorithme de mutex naïf présenté en



Section 3.1.3. Les numéros à côté des états représentent un ordre de parcours possible. Les variables de processus, de la forme `#i`, sont implicitement quantifiées existentiellement.

On part de l'état 1, qui correspond à la formule décrivant les états dangereux du système, ici, deux processus pour lesquels `PC` vaut `Crit`. La transition `t_enter` peut mener à cet état. L'état 2 représente la pré-image de l'état 1 par `t_enter(#2)`. Dans ce nouvel état, `PC[#2] = Want` et `X[#1] = False`. Ces conditions sont apportées par la garde de `t_enter`. Symétriquement, on peut calculer la pré-image de l'état 1 par `t_enter(#1)`, ce qui produit l'état 3. On note que cet état est identique à l'état 3, à renommage près des variables de processus (quantifiées existentiellement). Le test de point fixe va bien entendu détecter que cet état est subsumé par l'état 2, ce qui permet de ne pas développer inutilement cette branche (ce qu'on représente par la flèche en pointillés).

On peut ensuite s'intéresser à la pré-image de l'état 2 par `t_enter(#1)`, ce qui produit l'état 4. En prenant ensuite la transition `t_req(#2)`, on obtient un état incohérent : en effet, dans 4, on a `X[#2] = False`, or `t_req(#2)` a pour effet de mettre à jour `X[#2]` avec la valeur `True`. Dans l'état 5 ainsi produit, il faudrait donc avoir `True = False`, ce qui est bien entendu faux. Cet état n'est donc pas conservé.

On peut également s'intéresser à la pré-image de l'état 2 par `t_req(#2)`, ce qui produit l'état 6. Comme `X[#2]` n'apparaît pas dans l'état 2, il n'y a pas d'incohérence comme dans l'état 5. On peut alors prendre la transition `t_enter(#1)`, qui produit l'état 7. Si on prend maintenant la transition `t_req(#1)`, on a de nouveau un état incohérent, pour les mêmes raisons que précédemment. A noter que sans cette incohérence, l'état 8 s'intersecterait avec les états initiaux, et le système ne serait pas sûr.

3.2 Des compteurs de processus

Dans le contexte paramétré, on se pose parfois le problème de savoir représenter et manipuler des compteurs de processus. En particulier, puisque le nombre de processus est arbitraire, comment exprimer le fait que l'on a compté tous les processus ? Autrement dit, quelle constante permettrait de représenter la quantité N correspondant à la totalité des processus ? Bien entendu, aucune constante (sous-entendu entière) ne peut convenir.

Une solution consiste donc à encoder ce type de compteur comme un tableau de booléens indicé par des identificateurs de processus, et tel que le nombre de cases à *vrai* représente le nombre de processus comptés. Ainsi, un tableau dont toutes les cases sont à *faux* représente la valeur 0, tandis qu'un tableau dont toutes les cases sont à *vrai* représente la totalité des processus. La Figure 3.8 donne quelques exemples de tels compteurs.

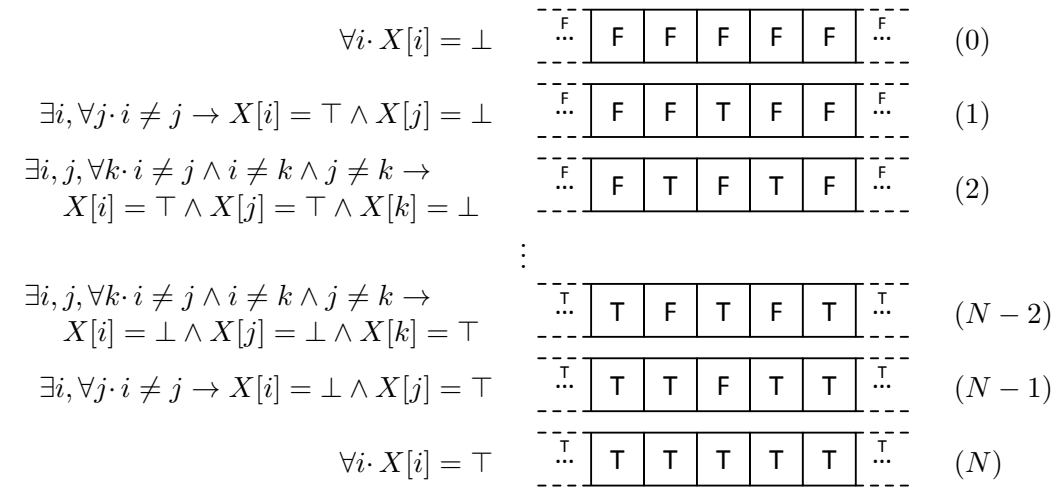


FIGURE 3.8 – Exemples de tableaux simulant des compteurs de processus

3.2.1 Expression des compteurs sous Cubicle

Un compteur sera donc représenté sous Cubicle par un tableau de booléens indicés par des processus, comme indiqué par la définition suivante :

```
array Counter[proc] : bool
```

Un compteur pourra être initialisé soit à 0 (toutes les cases à *faux*), soit à N (toutes les cases à *vrai*), comme le montrent les deux formules initiales ci-dessous :

```
init (p) { Counter[p] = False }
```

```
init (p) { Counter[p] = True }
```

Les transitions vont permettre de tester et manipuler ces compteurs. On incrémente un compteur en passant une de ses cases de *faux* à *vrai*, et on le décrémente en faisant l'opération inverse. Les deux transitions suivantes illustrent ces deux opérations :

```
transition t_increment (p)
requires { Counter[p] = False }
{ Counter[p] := True }
```

```
transition t_decrement (p)
requires { Counter[p] = True }
{ Counter[p] := False }
```

On peut également si on le souhaite ajouter ou retrancher des valeurs plus grandes que 1, en utilisant plusieurs paramètres et une construction `case`. Dans l'exemple qui suit, on ajoute ou on retranche 2 :

```
transition t_add_2 (p q)
requires { Counter[p] = False &&
           Counter[q] = False }
{ Counter[r] := case
   | r = p : True
```

```
transition t_sub_2 (p q)
requires { Counter[p] = True &&
           Counter[q] = True }
{ Counter[r] := case
   | r = p : False
```

```
  | r = q : True                    | r = q : False
  | _ : Counter[r] }                | _ : Counter[r] }
```

Pour réinitialiser un compteur à 0 ou à N, il suffit de repasser toutes les cases du compteur à *faux* ou à *vrai* :

```
transition t_reset_0 ()              transition t_reset_N ()
requires { ... }                     requires { ... }
{ Counter[r] := case | _ : False }    { Counter[r] := case | _ : True }
```

Pour comparer un compteur à 0 ou à N, on devra préciser la valeur pour toutes les cases du tableau correspondant, en utilisant la construction `forall_other` : toutes à *faux* pour 0, ou toutes les cases à *vrai* pour N, comme indiqué ci-dessous :

```
transition t_cmp_0 ()                transition t_cmp_N ()
requires { forall_other p.           requires { forall_other p.
             Counter[p] = False }                  Counter[p] = True }
{ ... }                              { ... }
```

Bien entendu, si les transitions utilisent par ailleurs des paramètres, il faudra préciser séparément les valeurs des cases pour ces paramètres, par exemple :

```
transition t_cmp_0 (p)               transition t_cmp_N (p)
requires { Counter[p] = False &&     requires { Counter[p] = True &&
           forall_other q.                      forall_other q.
             Counter[q] = False }                 Counter[q] = True }
{ ... }                              { ... }
```

On peut aussi comparer la valeur d'un compteur en précisant sa différence par rapport à 0 ou à N : on indique pour cela en paramètre de la transition le nombre de processus en plus ou en moins et on précise que la valeur associée à ces cases est différente des autres. Par exemple, la comparaison à 1 ou à N-1 est exprimée comme suit :

```
transition t_cmp_1 (p)               transition t_cmp_N_min_1 (p)
requires { Counter[p] = True &&      requires { Counter[p] = False &&
           forall_other q.                      forall_other q.
             Counter[q] = False }                 Counter[q] = True }
{ ... }                              { ... }
```

Là encore, il faudra tenir compte des éventuels paramètres supplémentaires de la transition, et préciser leur valeur en conséquence, comme dans le cas précédent.

3.2.2 Exemple de système à compteurs : Sense-Reversing Barrier

Pour illustrer le fonctionnement des compteurs de processus sur un cas concret, on prend comme exemple la barrière de synchronisation à inversion de sens, ou Sense-Reversing Barrier [88]. Son principe est illustrée en Figure 3.9. Un nombre `N` de processus doivent se

synchroniser en un point précis d'un programme, appelé barrière. Cette barrière ne peut être franchie que dans un seul sens. Par ailleurs, on attribue à chaque processus un sens de déplacement. Initialement, les processus et la barrière ont le même sens, et les processus sont situés "après" la barrière (comme s'ils venaient de la franchir), comme indiqué sur le premier schéma. Lorsque les processus doivent se rejoindre à la barrière, on commence par inverser leur sens (second schéma), ils peuvent alors avancer jusqu'à la barrière (troisième schéma). Quand il ne reste plus qu'un seul processus n'ayant pas atteint la barrière (quatrième schéma), celui-ci se voit attribuer un rôle particulier : lorsqu'il atteint la barrière (cinquième schéma), il en inverse le sens, ce qui autorise alors tous les processus à la franchir (sixième et septième schémas). Lorsque tous les processus ont franchi la barrière (dernier schéma), on est alors dans une situation symétrique à la situation initiale, et on peut recommencer le procédé. A noter que dans les faits, les processus venant de franchir la barrière peuvent immédiatement changer de sens, même si tous les processus n'ont pas encore franchi la barrière, sans que cela n'affecte la correction de l'algorithme (ces processus vont simplement attendre à la barrière).

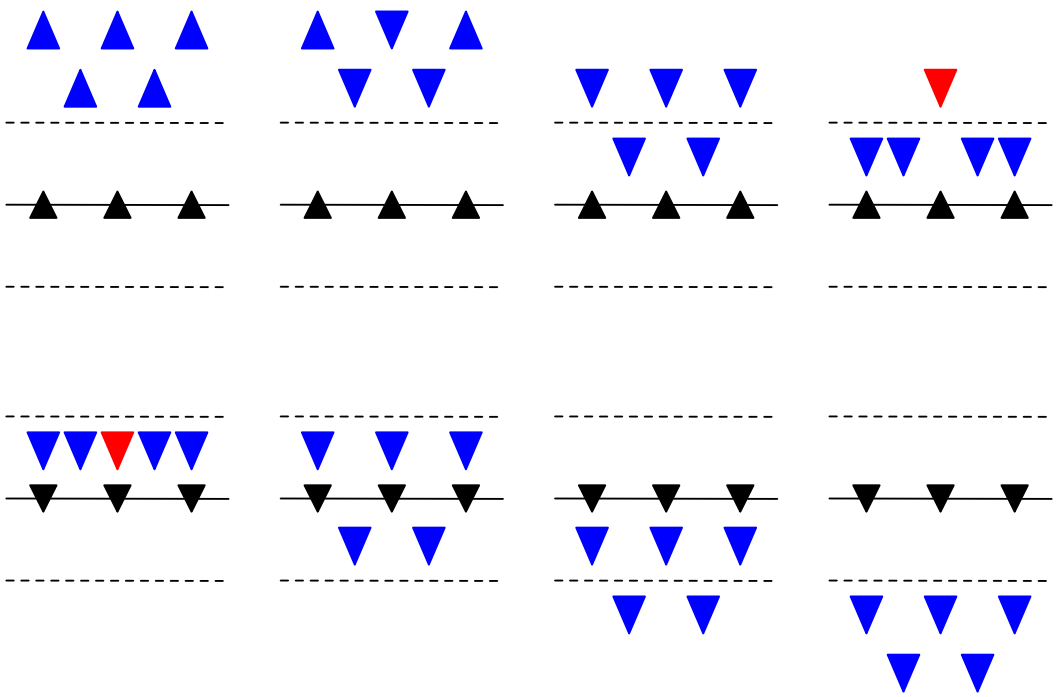


FIGURE 3.9 – Illustration de la Sense-Reversing Barrier

On donne en Figure 3.10 le pseudo-code de cette barrière, paramétrée par un nombre quelconque de processus `N`.

```
; Initialement, Local_sense = false, Sense = false et Count = N
L1: Local_sense := not Local_sense
L2: R := decrAndGet(Count);
L3: if R = 0 then (Count := N; Sense := Local_sense; goto L5);
L4: if Local_sense <> Sense then goto L4;
L5: goto L1
```

FIGURE 3.10 – Pseudo-code d'une Sense-Reversing Barrier

On utilise une variable partagée `Count` pour compter les processus arrivant à la barrière, une variable partagée `Sense` pour représenter le sens de franchissement de la barrière, une variable locale `Local_sense` pour représenter le sens de déplacement de chaque processus, et une variable locale `R` faisant office de variable temporaire. Initialement, `Sense` et `Local_sense` ont pour valeur `False` (les processus et la barrière ont le même sens) et `Count` vaut `N`, c'est-à-dire un nombre arbitraire de processus. Un processus voulant franchir la barrière commence par inverser `Local_sense`, puis, de façon *atomique*, décrémente `Count` et récupère sa valeur dans `R`. Les processus lisant une valeur autre que 0 entrent alors en phase d'attente : ils restent en attente tant que leur variable `Local_sense` est différente de `Sense` (c'est-à-dire tant que leur sens de franchissement ne correspond pas à celui de la barrière). Le processus qui récupère 0 dans `R` est le dernier à atteindre la barrière : il réinitialise `Count` à `N` et inverse le sens de la barrière en mettant `Sense` à jour avec la valeur qu'il possède dans `Local_Sense`, ce qui a pour effet de libérer les processus en attente.

Pour modéliser cet algorithme sous Cubicle (Figure 3.11), on commence par définir un type `loc` pour représenter les points de programme, et une variable `PC` pour simuler le pointeur d'instructions. Les variables locales `Local_sense` sont exprimées à l'aide d'un tableau du même nom et de type `bool`. La variable `Sense` est exprimée à l'aide d'une variable de type `bool`. La variable `Count` étant un compteur de processus, on la représente à l'aide d'un tableau d'éléments de type `bool`, comme présenté en Section 3.2.1. Dans l'état initial, `PC` vaut `L1` pour tous les processus, toutes les cases de `Count` valent `True` (le compteur est initialisé à N), toutes les cases de `Local_sense` valent `False`, et `Sense` vaut `False`. On considérera qu'un état est dangereux lorsqu'au moins deux processus ont leur `PC` en `L1` et que leur valeur de `Local_sense` est différente (ce qui permettrait à un processus de sauter la barrière).

La première instruction est traduite dans la transition `enter` à l'aide d'une construction `case` appliquée au tableau `Local_sense`. Cette construction permet d'inverser la valeur de `Local_sense` en une seule transition. Pour traduire la seconde instruction, on utilise deux transitions `decr_last` et `decr`. La première exprime le cas où la décrémentation de `Count` génère le résultat 0 : si `Count[i]` est vrai, et que pour tout autre processus `j`, `Count[j]` est faux, alors on met `Count[i]` à faux et on va au point de programme `L3`. La second exprime le cas où la décrémentation produit un résultat positif : si `Count[i]` est vrai et qu'il existe un autre processus `j` pour lequel `Count[j]` est vrai, alors on met `Count[i]` à faux et on se rend au point de programme `L4`. L'instruction suivante est traduite par la

```
type loc = L1 | L2 | L3 | L4 | L5

array PC[proc] : loc
array Local_sense[proc] : bool
array Count[proc] : bool
var Sense : bool

init (z) { Sense = False &&
  Local_sense[z] = False &&
  Count[z] = True && PC[z] = L1 }

unsafe (z1 z2) {
  PC[z1] = L1 && PC[z2] = L1 &&
  Local_sense[z1] <> Local_sense[z2] }

transition enter (i)
requires { PC[i] = L1 }
{ PC[i] := L2;
  Local_sense[j] := case
  | i = j && Local_sense[j] = True : False
  | i = j && Local_sense[j] = False : True
  | _ : Local_sense[j] }

transition decr_last (i)
requires { PC[i] = L2 &&
  Count[i] = True &&
  forall_other j. Count[j] = False }
{ PC[i] := L3; Count[i] := False }

transition decr (i j)
requires { PC[i] = L2 &&
  Count[i] = True && Count[j] = True }
{ PC[i] := L4; Count[i] := False }

transition release (i)
requires { PC[i] = L3 }
{ PC[i] := L5;
  Count[j] := case | _ : True;
  Sense := Local_sense[i] }

transition wait (i)
requires { PC[i] = L4 &&
  Local_sense[i] <> Sense }
{ PC[i] := L4 }

transition exit (i)
requires { PC[i] = L4 &&
  Local_sense[i] = Sense }
{ PC[i] := L5 }

transition end (i)
requires { PC[i] = L5 }
{ PC[i] := L1 }
```

FIGURE 3.11 – Code Cubicle de la Barrière Sense-Reversing

transition `release`. On met à jour `Sense` avec la valeur de `Local_sense` pour le processus courant, et on réinitialise toutes les cases de `Count` à `True`, ce qui équivaut à remettre le compteur à `N`. L'instruction suivante est traduite par les deux transitions `wait` et `exit`, représentant les deux résultats possibles de la comparaison. La dernière instruction est traduite à l'identique du pseudo-code.

3.3 Modèle mémoire opérationnel de Cubicle-$\mathcal{W}$

On présente maintenant le modèle mémoire implémenté dans Cubicle-$\mathcal{W}$. Il s'agit d'une extension du modèle TSO [126, 116], c'est-à-dire qu'il met en œuvre le relâchement W $\rightarrow$ R et permet la lecture précoce intra-processus, comme nous l'avons vu au Chapitre 1. Il est donc représentable par une machine abstraite à tampons, comme celle présentée en Figure 3.12.

Dans notre modèle, tout comme dans TSO, on dispose d'une unique mémoire partagée par un nombre arbitraire de processus. Chaque processus possède un ensemble de variables locales appelées registres, ainsi qu'un tampon d'écriture de type FIFO, accessible par ce processus uniquement. Toutes les écritures d'un processus transitent par son tampon avant

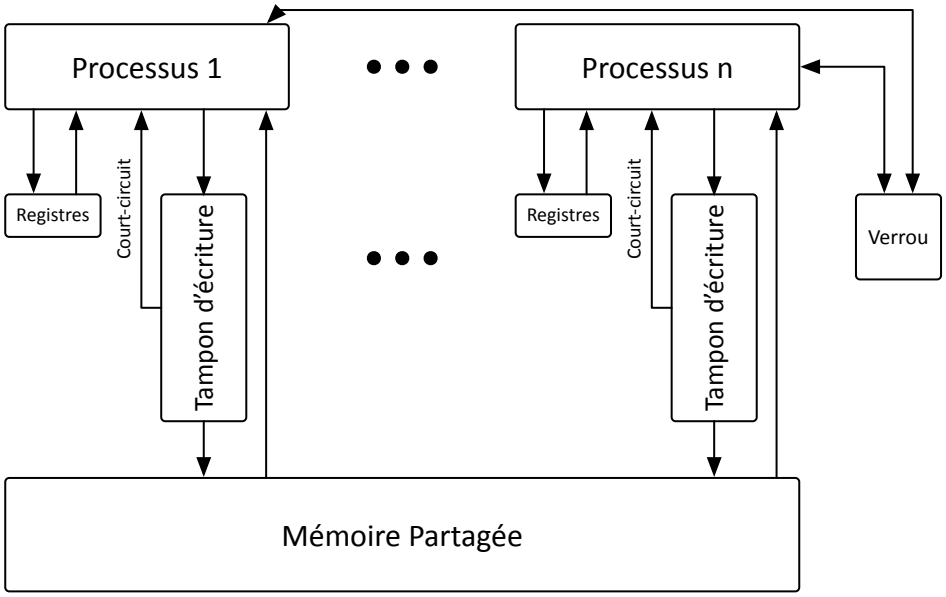


FIGURE 3.12 – Machine abstraite TSO

d'être propagées dans la mémoire partagée de façon *non déterministe*. Lorsqu'un processus effectue une lecture d'une variable X, si X est présente dans son tampon, alors le processus lit la valeur la plus récente pour X dans son tampon (ce qui est représenté par les flèches `Court-circuit`), sinon il lit la valeur depuis la mémoire partagée.

Par ailleurs, un *verrou global* permet la réalisation d'instructions de type *lecture-modification-écriture* atomiques. Un processus peut acquérir le verrou lorsque son tampon est vide et qu'aucun autre processus ne détient déjà le verrou. Tant qu'un processus détient le verrou, les autres processus ne peuvent plus effectuer de lecture, et les écritures en attente dans leur tampon ne peuvent plus être propagées en mémoire (il leur est en revanche permis d'écrire dans leur tampon). Enfin, pour qu'un processus puisse libérer le verrou, son tampon doit être vide.

Toutefois, notre modèle diffère légèrement de TSO. En effet, afin de prendre en compte le fait que certaines opérations sur les compteurs de processus se font en lisant ou en écrivant simultanément dans plusieurs cases d'un tableau, nous l'avons sensiblement étendu : les éléments des tampons peuvent être *composites*, c'est-à-dire qu'un élément peut contenir plusieurs écritures simultanées. Par ailleurs, il sera possible d'effectuer plusieurs lectures en un même instant. Ces extensions restent toutefois compatibles avec TSO : l'utilisateur peut s'abstenir de l'utiliser (ou ne les utiliser que pour les compteurs), auquel cas le modèle mémoire coïncidera avec le modèle TSO original.

3.4 Extensions et restrictions du langage d'entrée

Afin d'exprimer des systèmes de transition sur des modèles mémoire faibles, il est nécessaire d'introduire quelques nouveaux mots-clés dans le langage, et de restreindre certaines constructions.

Les processus pouvant chacun avoir une vision différente de la mémoire en raison des tampons d'écriture, il faut à présent différencier les variables locales (ou registres), spécifiques à un processus, et les variables partagées. Les variables partagées étant susceptibles d'être affectées par le modèle mémoire faible, on les appelle aussi variables faibles. A l'inverse, les variables locales n'étant accessibles que par un seul processus, elles se comportent de manière séquentiellement cohérente (SC), on les nomme donc variables SC. Pour exprimer ces variables SC, on utilise les tableaux indicés par des processus de Cubicle, toutefois on restreint les accès à ces tableaux de sorte qu'un processus ne puisse accéder qu'à la case correspondant à son identificateur. Les variables faibles, quant à elles, sont décrites par des variables ou des tableaux indicés par des processus, dont les cases sont accessibles par n'importe quel processus. Leur déclaration doit être précédée du mot-clé `weak`. Dans le code qui suit, on déclare un tableau SC (ou tableau de variables locales) `PC` de type `loc`, une variable faible `X` de type `int`, et un tableau faible `T` de type `bool`.

```
array PC[proc] : loc
weak var X : int
weak array T[proc] : bool
```

La formulation des états initiaux du système est inchangée. En effet, dans cet état, tous les processus partagent une même vision de la mémoire, les tampons d'écriture étant vides.

En revanche, la formulation des états dangereux du système change, chaque processus pouvant avoir sa propre vision de la mémoire en raison des tampons d'écriture. Il faut donc préciser pour chaque accès à une variable ou à un tableau faible le processus depuis lequel on se place pour l'*observer*, en préfixant l'accès par la variable de processus correspondante. Il en va de même pour les invariants. Par exemple, le code suivant définit les états dangereux comme l'ensemble des états tel qu'il existe deux processus distincts `p` et `q` pour lesquels `PC[p] = CS`, `PC[q] = CS`, et `X = 42` *du point de vue* du processus `p`.

```
unsafe (p q) { PC[p] = CS && PC[q] = CS && p @ X = 42 }
```

De même, les transitions sont affectées par le nouveau modèle mémoire. Les transitions doivent maintenant être paramétrées par *au moins* une variable de type `proc`. Puisque chaque processus possède un tampon d'écriture ainsi que des variables locales qu'il est le seul à pouvoir manipuler, il faut indiquer pour chaque transition quel est le processus évaluant la garde et effectuant les actions. Pour ce faire, l'un des paramètres de la transition doit être marqué comme étant le processus *principal*. Tous les accès aux variables ou tableaux faibles sont alors réalisés par ce processus en prenant en compte les effets des tampons, et tous les accès aux tableaux SC sont alors restreints à la seule case appartenant à ce processus. Le code suivant représente une transition paramétrée par deux processus

`i` et `j` *différents*, où `i` est marqué comme étant le processus *principal*. Cette transition est prise si `PC[i] = L1` et `T[j] = False`, et a pour effet de mettre à jour `PC[i]` avec la valeur `L2` et `T[j]` avec la valeur `True`, ainsi que d'incrémenter `X` de 1. Conformément à notre modèle de mémoire faible, les lectures de `T[j]` et `X` prennent en compte les éventuelles écritures déjà présentes dans le tampon du processus `i`, et les mises à jour de `X` et `T[j]` sont placées simultanément dans le tampon du processus principal `i`.

```
transition t1 ([i] j)
requires { PC[i] = L1 && T[j] = False }
{ PC[i] := L2; T[j] := True; X := X + 1 }
```

Les constructions `forall_other` et `case` peuvent toujours être utilisées sur les variables et tableaux faibles, sans restriction particulière. En revanche, `forall_other` ne peut pas s'utiliser sur les tableaux SC, puisque seule la case appartenant au processus principal est accessible. La construction `case` reste utilisable sur les tableaux SC, à condition que seule la case du processus principal soit modifiée et que la branche par défaut réaffecte à l'identique le contenu des autres cases (ces conditions sont vérifiées statiquement). Dans le code suivant, où `R` est un tableau SC de type `bool`, on utilise cette possibilité pour inverser la valeur contenue dans la case du processus principal (une autre solution consisterait à utiliser deux transitions, conditionnées par la valeur de `R[i]`).

```
transition t4 ([i])
requires { PC[i] = L1 }
{ R[j] := case
   | i = j && R[j] = False : True
   | i = j && R[j] = True : False
   | _ : R[j] }
```

Enfin, la garde d'une transition peut exiger que le tampon associé au processus principal soit vide pour que la transition puisse être prise. On exprime cela à l'aide du prédicat `fence`. Par exemple, la transition suivante ne peut être prise que si `PC[i] = L1` et si le tampon du processus `i` est vide.

```
transition t5 ([i])
requires { PC[i] = L1 && fence() }
{ PC[i] := L2 }
```

Par ailleurs, lorsqu'une transition contient à la fois une lecture et une mise à jour d'une variable faible, le processus principal acquiert le *verrou global*, comme indiqué en Section 3.3 : le tampon de ce processus doit donc être vide avant et après l'exécution de la transition.

3.5 Exemples

Dans cette section, on montre l'utilisation de Cubicle-$\mathcal{W}$ sur différents algorithmes concurrents paramétrés.

3.5.1 Mutex Naïf

On reprend l'exemple du mutex naïf présenté en Section 3.1.3. La modélisation de cet algorithme est quasiment inchangée, mais il faut toutefois préciser quelles sont les variables faibles et les processus effectuant les actions. Les éléments du tableau `State` ne concernant qu'un seul processus, ce tableau devient un tableau SC. Le tableau `X` en revanche est accédé par tous les processus : il devient un tableau faible et on ajoute le mot-clé `weak` dans sa définition. Quant aux transitions, leur unique paramètre devient le processus effectuant les actions, c'est-à-dire le processus *principal*. On lui ajoute donc les crochets `[]`.

```
type state = Idle | Want | Crit

array State[proc] : state

weak array X[proc] : bool

init (i) {
  State[i] = Idle && X[i] = False }

unsafe (i j) {
  State[i] = Crit && State[j] = Crit }
```

```
transition t_req ([i])
requires { State[i] = Idle }
{ State[i] := Want; X[i] := True }

transition t_enter ([i])
requires { State[i] = Want &&
           forall_other j. X[j] = False }
{ State[i] := Crit }

transition t_exit ([i])
requires { State[i] = Crit }
{ State[i] := Idle; X[i] := False }
```

FIGURE 3.13 – Code Cubicle-$\mathcal{W}$ du Mutex Naïf

L'exécution de Cubicle sur ce système produit la sortie suivante :

```
Unsafe trace: t_req(#1) -> t_req(#2) -> t_enter(#1) ->
              t_enter(#2) -> unsafe[1]

unsafe[1] is reachable !

UNSAFE !
```

Ceci nous indique que ce système n'est pas sûr quant à la propriété de sûreté exprimée. La trace fournie indique que deux processus `#1` et `#2` ont tous deux pu entrer en section critique en prenant tous les deux `t_req` puis `t_enter`. En effet, à cause de notre modèle de mémoire faible, les écritures `X[i] := True` dans `t_req` peuvent être encore dans les tampons lorsque les processus évaluent la garde de `t_enter`. Les processus risquent donc de voir `X` à `False` pour tous les autres processus, ce qui les autorise à prendre la transition.

Pour corriger cet algorithme, il faut empêcher qu'un processus puisse lire les autres cases de `X` tant que sa propre écriture dans `X` n'a pas été propagée en mémoire depuis son tampon. Pour ce faire, il faut ajouter une barrière mémoire, c'est-à-dire un prédicat `fence`, dans la transition entre `Want` et `Crit`, comme indiqué en Figure 3.14.

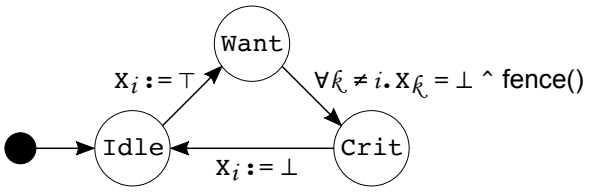


FIGURE 3.14 – Algorithme de Mutex Naïf corrigé

La transition `t_enter` est alors modifiée comme suit :

```
transition t_enter ([i])
requires { State[i] = Want && fence()
           forall_other j. X[j] = False }
{ State[i] := Crit }
```

Et la sortie de Cubicle-$\mathcal{W}$ indique maintenant que le système est correct vis à vis de la propriété d'exclusion mutuelle :

```
The system is SAFE
```

3.5.2 Spinlock Linux

Le second exemple est un verrou tournant ou Spinlock, utilisé dans la version 2.6.24.7 du noyau Linux et présenté dans l'article de référence sur x86-TSO [126]. Son code x86 est donné en Figure 3.15 (syntaxe Intel).

```
; Initialement, Lock = 1
acquire: lock dec dword [Lock]
         jns cs
    spin: cmp dword [Lock], 0
         jle spin
         jmp acquire
      cs: ; section critique
release: mov dword [Lock], 1
         jmp acquire
```

FIGURE 3.15 – Code assembleur x86 du Spinlock Linux

Initialement, `Lock` a pour valeur 1. Lorsqu'un processus souhaite entrer en section critique (`acquire`), il décrémente la variable `Lock` *de façon atomique*. Si le résultat de cette opération est positif ou nul, cela signifie qu'aucun autre processus n'a demandé la section critique, le branchement `jns` est pris et le processus entre en section critique (`cs`). Dans le cas contraire (résultat négatif), le branchement n'est pas pris, et le processus passe à la phase d'attente active (`spin`) : tant que la valeur de `Lock` est inférieure ou égale à 0, le branchement vers `spin` est pris. Lorsque la valeur devient supérieure à 0, cela signifie que plus aucun processus n'est en section critique, et le processus peut retenter l'entrée

en exécutant le branchement suivant vers `acquire`. Lorsqu'un processus quitte la section critique (`release`), il réinitialise la valeur de `Lock` à 1, ce qui a pour effet de débloquer les autres processus éventuellement en attente. Pour que cet algorithme soit correct, il est essentiel que la décrémentation de `Lock` soit atomique.

```
type loc = Acquire | Spin | CS | Release

array PC[proc] : loc

weak var Lock : int


init (p) {
  PC[p] = Acquire && Lock = 1 }

unsafe (p1 p2) {
  PC[p1] = CS && PC[p2] = CS }


transition t1_Acquire_CS ([p])
requires { PC[p] = Acquire && 0 < Lock }
{ PC[p] := CS; Lock := Lock - 1 }
```

```
transition t1_Acquire_Spin ([p])
requires { PC[p] = Acquire && Lock <= 0 }
{ PC[p] := Spin; Lock := Lock - 1 }

transition t2_Spin_Spin ([p])
requires { PC[p] = Spin && Lock <= 0 }
{ PC[p] := Spin }

transition t2_Spin_Acquire ([p])
requires { PC[p] = Spin && 0 < Lock }
{ PC[p] := Acquire }

transition t3_CS_Release ([p])
requires { PC[p] = CS }
{ PC[p] := Release }

transition t4_Release_Acquire ([p])
requires { PC[p] = Release }
{ PC[p] := Acquire; Lock := 1 }
```

FIGURE 3.16 – Code Cubicle-$\mathcal{W}$ du Spinlock Linux

Pour modéliser cet algorithme avec Cubicle-$\mathcal{W}$ (Figure 3.16), on commence par définir un type `loc` pour représenter les points de programme pertinents, soit `Acquire`, `Spin`, `CS` et `Release`, correspondant aux étiquettes du programme assembleur, et un tableau SC `PC` de ce type pour simuler le pointeur d'instruction de chaque processus. On déclare ensuite une unique variable faible `Lock` de type `int`. La formule initiale précise que `Lock` vaut initialement 1 et que `PC` est égal à `Acquire` pour tous les processus. On s'intéresse à la propriété d'exclusion mutuelle de cet algorithme. Les états dangereux sont donc définis comme les états tels que `PC` est égal à `CS` pour au moins deux processus.

Reste à exprimer les instructions grâce à des transitions. Selon la granularité souhaitée, on peut utiliser plus ou moins de transitions, chaque transition pouvant exprimer une ou plusieurs instructions. La granularité doit être suffisamment fine pour capturer les comportements essentiels du programme que l'on modélise, mais pas excessivement, au risque d'augmenter le temps d'analyse. De manière générale, il peut être pertinent de placer dans des transitions distinctes des accès successifs à des variables faibles, et de rassembler dans une même transition des accès successifs à des variables SC. Dans notre cas, on rassemble les instructions arithmétiques et de comparaison avec les instructions de branchement qui suivent. Par exemple, la traduction des deux premières instructions `dec` et `jns` donne lieu aux deux transitions `t1_Acquire_CS` et `t1_Acquire_Spin`. La première transition exprime le cas où la décrémentation de `Lock` renvoie un résultat positif ou nul et où l'on effectue le branchement vers le point de programme `CS`. La seconde exprime le cas où la décrémentation renvoie un résultat négatif et où l'on effectue le branchement

vers le point de programme `Spin`. Similairement, les deux transitions `t2_Spin_Spin` et `t2_Spin_Acquire` traduisent les trois instructions à partir du point de programme `Spin` : selon le résultat de la comparaison de `Lock` à 0, on effectue un branchement vers les points de programme `Spin` ou `Acquire`. Les deux dernières transitions traduisent à l'identique les deux dernières instructions du programme.

3.5.3 L'Arbitre

Le dernier exemple est un algorithme d'exclusion mutuelle nommé Arbitre [81], dont le pseudo-code est donné en Figure 3.17.

```
attention: array 1..N of boolean (false)
 answered: array 1..N of boolean (false)
```

```
; Arbitre (1)
for each process i
  if attention(i) then
    answered(i) := true;
    await attention(i) = false;
    answered(i) := false
```

```
; Processus (N)
; j = processus courant
repeat
  await answered(j) = false;
  attention(j) := true;
  await answered(j) = true;
  ; section critique
  attention(j) := false;
```

FIGURE 3.17 – Pseudo-code de l'Arbitre

Dans cet algorithme, un unique processus "arbitre" gère les entrées en section critique d'un nombre quelconque `N` d'autres processus. Il utilise pour cela deux tableaux de `N` booléens `attention` et `answered` initialisés à faux. Un processus `j` signale son intention d'entrer en section critique en mettant `attention(j)` à vrai, puis attend d'avoir la permission, indiquée par la valeur vrai dans `answered(j)`. Le processus est alors en section critique. Lorsqu'il quitte la section critique, le processus repasse `attention(j)` à faux. A noter que le processus ne peut pas demander à entrer en section critique tant que `answered(j)` n'est pas repassé à faux. L'arbitre parcourt continuellement le tableau `attention` jusqu'à ce que `attention(i)` soit vrai pour un processus `i`, signifiant que le processus `i` veut entrer en section critique. Il donne alors la permission à `i` en mettant `answered(i)` à vrai, puis attend que `i` quitte la section critique, ce qui sera indiqué par la valeur faux dans `attention(i)`. L'arbitre replace alors `answered(i)` à faux.

La principale difficulté dans la modélisation de cet algorithme avec Cubicle-$\mathcal{W}$ réside dans le fait qu'un processus a un comportement très différent des autres. Il faut donc trouver un moyen de différencier ce processus. Pour ce faire, on peut déclarer un type énuméré `kind`, avec les constructeurs `Arb` et `Proc`, et un tableau constant `Kind` dont les éléments sont de type `kind`. On ajoutera dans la garde des transitions la condition `Kind[p] = Arb` ou `Kind[p] = Proc` selon que la transition concerne l'arbitre ou un autre processus. Puisque toutes les valeurs de ce tableau ne sont pas identiques, on ne peut

l'initialiser avec la formule initiale. On pourra en revanche utiliser l'invariant suivant : il n'existe pas deux processus `p1` et `p2` tels que `Kind[p1] = Arb` et `Kind[p2] = Arb`.

```
type kind = Arb | Proc
type loc = L1 | L2 | L3 | L4 | CS

const Kind[proc] : kind
array PC[proc] : loc
array Pr[proc] : proc

weak array Attn[proc] : bool
weak array Answ[proc] : bool


invariant (p1 p2) {
  Kind[p1] = Arb && Kind[p2] = Arb }

init (p) { PC[p] = L1 &&
  Attn[p] = False && Answ[p] = False }

unsafe (p1 p2) {
  PC[p1] = CS && PC[p2] = CS }


(* Arbitrer (1) *)

transition t_arb_L1_L2 ([p] q)
requires { Kind[p] = Arb &&
  PC[p] = L1 && Attn[q] = True }
{ PC[p] := L2; Pr[p] := q }

transition t_arb_L2_L3 ([p] q)
requires { Kind[p] = Arb &&
  PC[p] = L2 && Pr[p] = q }
{ PC[p] := L3; Answ[q] := True }

transition t_arb_L3_L4 ([p] q)
requires { Kind[p] = Arb && PC[p] = L3 &&
  Pr[p] = q && Attn[q] = False }
{ PC[p] := L4 }

transition t_arb_L4_L1 ([p] q)
requires { Kind[p] = Arb && PC[p] = L4 &&
  Pr[p] = q }
{ PC[p] := L1; Answ[q] := False }

(* Processes (N) *)

transition t_proc_L1_L2 ([p])
requires { Kind[p] = Proc && PC[p] = L1 &&
  Answ[p] = False }
{ PC[p] := L2 }

transition t_proc_L2_L3 ([p])
requires { Kind[p] = Proc && PC[p] = L2 }
{ PC[p] := L3; Attn[p] := True }

transition t_proc_L3_CS ([p])
requires { Kind[p] = Proc && PC[p] = L3 &&
  Answ[p] = True }
{ PC[p] := CS }

transition t_proc_CS_L1 ([p])
requires { Kind[p] = Proc && PC[p] = CS }
{ PC[p] := L1; Attn[p] := False }
```

FIGURE 3.18 – Code Cubicle-$\mathcal{W}$ de l'Arbitre

Les tableaux de booléens `attention` et `answered` sont exprimés à l'aide des deux tableaux faibles `Attn` et `Answ` de type `bool`. L'arbitre a également besoin d'une variable locale pour mémoriser l'identificateur du processus qu'il est en train de gérer. On le représente à l'aide du tableau SC `Pr` de type `proc` (les autres processus ne l'utiliseront pas). Comme dans les exemples précédents, on utilise également un type `loc` et un tableau `PC` pour simuler le pointeur d'instructions. La formule initiale précise que `PC` vaut `L1` et que les deux tableaux `Attn` et `Answ` contiennent la valeur `False` pour tous les processus. Les états dangereux sont les états tels qu'il existe deux processus pour lesquels `PC` vaut `CS`.

Dans le pseudo-code, l'arbitre utilise une boucle *for* pour parcourir les différents processus. Pour simuler ce comportement, on utilise simplement un paramètre de transition supplémentaire, qui permettra de représenter tout processus autre que l'arbitre (on rappelle que les paramètres des transitions sont des variables de processus quantifiées existen-

tiellement). Les propriétés temporelles de cette modélisation ne sont donc pas identiques à celles de l'algorithme présenté, toutefois on ne s'intéresse qu'aux propriétés de sûreté.

Ainsi, la première instruction `if attention(i) then` est modélisée par la transition `t_arb_L1_L2`. Cette transition possède deux paramètres `p` et `q`, où `p` est le processus principal et `q` un autre processus. La garde de la transition précise que `p` est l'arbitre, grâce à la condition `Kind[p] = Arb`, et que `Attn[q]` doit être à `True` pour que la transition soit prise. Les actions consistent à mettre à jour `PC` et à mémoriser dans `Pr[p]` la valeur de `q`, c'est-à-dire un processus demandant à entrer en section critique.

La seconde instruction `answered(i) := true` est modélisée par la transition `t_arb_L2_L3`. Comme précédemment, cette transition prend deux paramètres `p` et `q`, avec `p` le processus principal, et sa garde précise que `p` est l'arbitre. Le deuxième paramètre `q` est nécessaire car les actions doivent mettre à jour `Answ[[Pr[p]]]` avec la valeur `True`, or un tableau ne peut être indicé que par une variable de processus. Pour cette raison, la garde indique que `Pr[p] = q` et la mise à jour s'écrit `Answ[q] := True`.

Le reste de la modélisation s'effectue naturellement selon le même procédé, chaque instruction correspondant à une transition, en prenant soin de bien préciser `Kind[p] = Arb` ou `Kind[p] = Proc` dans les gardes des transitions.

4

Model checking modulo mémoire faible

Sommaire

Dans cette section, on présente les concepts sous-jacents à notre approche. On commence par développer une approche avec modélisation explicite des tampons, et on montre qu'une telle approche n'est pas efficace en pratique. On introduit alors une version axiomatique de notre modèle mémoire faible, qui nous permet ensuite d'exprimer la théorie des tableaux faibles. On explique alors comment se construisent les systèmes de transition à tableaux faibles à partir de cette théorie, et les algorithmes mis en jeu pour leur vérification.

4.1 Expérimentation : modélisation directe des tampons TSO

Nous avons vu en Section 3.1 du Chapitre 3 que Cubicle manipulait des tableaux indicés par des identificateurs de processus. Ces tableaux étant non-bornés (puisque le nombre de processus n'est pas fixé), on peut imaginer les utiliser pour représenter les tampons d'écriture du modèle TSO : certains identificateurs de processus identifieraient des cases de tampon plutôt que des processus.

Les tampons contiennent des paires *(variable,valeur)*. Par souci de simplicité, on ne manipule que des variables simples (pas de tableaux) et de type entier, mais l'approche est généralisable aux tableaux et aux autres types. Pour représenter les différentes variables partagées, on a besoin d'un type énuméré dont les constructeurs correspondent aux différentes variables. Il faut également un constructeur pour indiquer qu'une case d'un tampon est vide. Dans l'exemple suivant, on a deux variables partagées X et Y, le type énuméré `vars` contient donc les deux constructeurs correspondants, ainsi qu'un constructeur pour les cases vides :

```
type vars = VEmpty | VX | VY
var X : int
var Y : int
```

On représente alors les tampons par deux tableaux à deux dimensions : le premier tableau correspond aux variables et le second aux valeurs. La première dimension de chaque tableau correspond au processus possédant le tampon, la deuxième correspond à la case du tampon. Indépendamment du choix des constructeurs pour le type `vars`, les deux tableaux sont donc exprimés comme suit :

```
array Var[proc,proc] : vars
array Val[proc,proc] : int
```

On considère que les tampons se comportent comme des files FIFO. On aura à l'esprit une vue horizontale des tampons, se remplissant par la gauche et se vidant par la droite. Il nous faut maintenant modéliser les différentes opérations mémoire en prenant en compte l'existence de ces tampons :

— écriture (par le biais du tampon)
— lecture depuis le tampon
— lecture depuis la mémoire
— lecture-modification-écriture atomique
— propagation d'une écriture depuis le tampon vers la mémoire
— barrière mémoire

Ecriture Une écriture se faisant systématiquement par le biais d'un tampon, il faut commencer par rechercher une case vide en tête de tampon, c'est-à-dire une case vide telle que toutes les cases à sa gauche soient également vides. On ajoute alors l'écriture dans cette case. L'exemple suivant décrit une écriture sur X avec la valeur 42 :

```
transition t_write (p b)
requires { Var[p,b] = VEmpty &&
           forall_other k. (b < k || Var[p,k] = VEmpty) }
{ Var[p,b] := VX; Val[p,b] := 42 }
```

Lecture depuis le tampon Lorsqu'une lecture a lieu, elle prend sa valeur en priorité depuis l'écriture la plus récente du tampon, le cas échéant. Il faut donc rechercher la dernière case dans laquelle a eu lieu une écriture sur la variable lue : il s'agit de la case concernant la variable et telle qu'aucune autre case à sa gauche ne concerne cette variable. L'exemple suivant décrit une lecture sur X depuis le tampon :

```
transition t_readbuf (p b)
requires { Var[p,b] = VX &&
           forall_other k. (b < k || Var[p,k] <> VX) }
{ R[p] := Val[p,b] }
```

Lecture depuis la mémoire Lorsqu'une lecture a lieu et que le tampon ne contient aucune écriture sur la variable à lire, la lecture s'effectue directement depuis la mémoire. Il faut donc simplement vérifier que le tampon ne contienne pas une telle écriture. L'exemple suivant décrit une lecture sur X depuis la mémoire :

```
transition t_readmem (p)
requires { forall_other k. Var[p,k] <> VX }
{ R[p] := X }
```

Lecture-modification-écriture atomique Pour effectuer une opération de lecture-modification-écriture atomique, le tampon du processus effectuant l'opération doit être vide, et le processus lit et écrit directement en mémoire, en une seule transition. L'exemple suivant décrit une opération d'incrémentation atomique sur X :

```
transition t_atomic_rmw (p)
requires { forall_other k. Var[p,k] = VEmpty }
{ X := X + 1 }
```

Propagation tampon-mémoire Pour propager une écriture d'un tampon vers la mémoire, il faut rechercher le case non-vide la plus à droite : il s'agit de la case non-vide telle que toutes les cases à sa droite soient vides. La variable reçoit alors la valeur correspondante, et la case est vidée. L'exemple suivant décrit la propagation d'une écriture dans X :

```
transition t_flush_X (p b)
requires { Var[p,b] = VX &&
           forall_other k. (k < b || Var[p,k] = VEmpty) }
{ X := Val[p,b]; Var[p,b] := VEmpty }
```

Barrière mémoire Pour réaliser une barrière mémoire, il suffit de tester si toutes les cases du tampon sont vides. L'exemple suivant décrit cette opération :

```
transition t_fence (p)
requires { forall_other k. Var[p,k] = VEmpty }
{ ... }
```

On souhaite maintenant expérimenter cette approche sur un algorithme concret. On reprend pour cela le Spinlock Linux présenté en Section 3.5.2 du Chapitre 3, auquel on ajoute un type `vars` muni des seuls constructeurs `VEmpty` et `VLock` (seule variable partagée) et les deux tableaux `Var` et `Val` permettant de représenter les tampons. On modifie les transitions de façon à ce que les opérations mémoires passent par les tampons. Le résultat

```
type loc = Acquire | Spin | CS | Release
type vars = VEmpty | VLock

array PC[proc] : loc
array Var[proc,proc] : vars
array Val[proc,proc] : int

var Lock : int


init (p b) {
  PC[p] = Acquire && Lock = 1 &&
  Var[p,b] = VEmpty }

unsafe (p1 p2) {
  PC[p1] = CS && PC[p2] = CS }


transition t1_Acquire_CS_armw (p)
requires { PC[p] = Acquire && 0 < Lock &&
   forall_other k. Var[p,k] = VEmpty }
{ PC[p] := CS; Lock := Lock - 1 }

transition t1_Acquire_Spin_armw (p)
requires { PC[p] = Acquire && Lock <= 0 &&
   forall_other k. Var[p,k] = VEmpty }
{ PC[p] := Spin; Lock := Lock - 1 }

transition t2_Spin_Spin_rmem (p)
requires { PC[p] = Spin && Lock <= 0 &&
   forall_other k. Var[p,k] <> VLock }
{ PC[p] := Spin }
```

```
transition t2_Spin_Spin_rbuf (p b)
requires { PC[p] = Spin &&
   Var[p,b] = VLock && Val[p,b] <= 0 &&
   forall_other k. (b < k
       || Var[p,k] <> VLock) }
{ PC[p] := Spin }

transition t2_Spin_Acquire_rmem (p)
requires { PC[p] = Spin && 0 < Lock &&
   forall_other k. Var[p,k] <> VLock }
{ PC[p] := Acquire }

transition t2_Spin_Acquire_rbuf (p b)
requires { PC[p] = Spin &&
   Var[p,b] = VLock && 0 < Val[p,b] &&
   forall_other k. (b < k
       || Var[p,k] <> VLock) }
{ PC[p] := Acquire }

transition t3_CS_Release (p)
requires { PC[p] = CS }
{ PC[p] := Release }

transition t4_Release_Acquire_w (p b)
requires { PC[p] = Release &&
   Var[p,b] = VEmpty &&
   forall_other k. (b < k
       || Var[p,k] = VEmpty) }
{ PC[p] := Acquire;
  Var[p,b] := VLock; Val[p,b] := 1 }

transition t_flush_lock (p b)
requires { Var[p,b] = VLock &&
   forall_other k. (k < b
       || Var[p,k] = VEmpty) }
{ Lock := Val[p,b]; Var[p,b] := VEmpty }
```

FIGURE 4.1 – Code Cubicle du Spinlock Linux avec tampons explicites

obtenu est présenté en Figure 4.1.

Nous avons ensuite lancé l'analyse par Cubicle sur ce programme. Alors que l'analyse de sa version SC prenait une fraction de seconde, avec cette modélisation directe des tampons TSO, l'analyse prend maintenant plus d'une heure et explore un espace d'états considérable. Ce résultat est d'autant plus significatif que le programme considéré ne manipule qu'une seule variable faible, possède peu de transitions, et effectue une opération de lecture-modification-écriture atomique qui a pour effet secondaire de limiter le nombre d'écritures en attente dans le tampon d'un processus. Deux raisons principales peuvent expliquer cette explosion : d'une part, les transitions correspondant à la propagation des écritures depuis un tampon peuvent être exécutées à n'importe quel moment, ce qui augmente le facteur de branchement, et d'autre part l'utilisation de tableaux indicés par des processus pour représenter des tampons non-bornés a pour inconvénient d'augmenter artificiellement le nombre de processus du système étudié. Pour ces raisons, il ne semble donc pas réaliste d'avoir recours à une modélisation directe des tampons TSO.

4.2 Modèle mémoire axiomatique de Cubicle-$\mathcal{W}$

La machine à tampons présentée en Section 3.3 permet d'expliquer simplement la sémantique du modèle mémoire faible de Cubicle-$\mathcal{W}$. En revanche, comme nous l'avons vu en Section 4.1, elle n'est pas efficace lorsqu'il s'agit de réaliser une analyse automatique de programmes sur ce modèle. De ce fait, notre approche repose sur l'utilisation d'un modèle axiomatique. Le formalisme que nous utilisons est largement inspiré de celui présenté par Alglave *et al.* dans [20] et [25] pour TSO : il repose sur l'utilisation d'événements et de relations sur ces événements pour déterminer si une exécution donnée d'un programme est valide. On présente dans un premier temps le modèle axiomatique de TSO original, puis on explique comment on l'adapte à notre approche.

4.2.1 Modèle mémoire axiomatique original de TSO

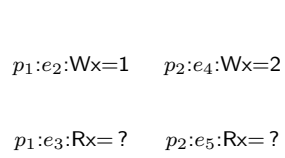
L'approche axiomatique repose sur l'utilisation de traces d'exécution pour décrire la sémantique d'un programme. Les instructions d'un programme génèrent des événements, possédant un identificateur d'événement unique et caractérisés par leur direction (W pour une écriture, R pour une lecture), le processus effectuant l'opération, la variable manipulée [1] et la valeur lue ou écrite. Lorsqu'une instruction est répétée plusieurs fois, par exemple lorsqu'elle se trouve dans une boucle, elle génère autant d'événements que le nombre de fois où elle est exécutée, chacun doté d'un identificateur distinct. Par ailleurs, il existe également un événement d'écriture initial pour chaque variable ; cet événement n'est associé à aucun processus. On appelle $\mathbb{E}$ l'ensemble des événements générés.

Prenons par exemple le programme suivant, constitué de deux processus p_1 et p_2, dans lequel $R1$ et $R2$ sont des registres et X est une variable partagée valant initialement 0 :

1. par variable, on entend un emplacement mémoire distinct

p_1	p_2
X ← 1	X ← 2
R1 ← X	R2 ← X

Les événements générés par ce programme sont les suivants :

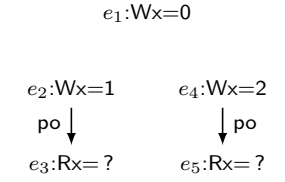


L'instruction X ← 1 du processus p_1 est une écriture (W) sur la variable partagée X avec la valeur 1. De ce fait, elle génère l'événement p_1:e_2:Wx=1, où e_2 représente son identifiant unique. L'instruction R1 ← X du processus p_1 est une lecture de la variable partagée X, dont on ne connait pas (encore) la valeur. Elle génère donc l'événement p_1:e_3:Rx= ?. Le raisonnement est identique pour les deux instructions du processus p_2. Enfin, la valeur initiale de X étant 0, on a l'événement supplémentaire e_1:Wx=0, associé à aucun processus. Par la suite, afin de ne pas surcharger les schémas, on ne préfixera plus les événements par le processus qui les a générés : les événements d'un même processus étant présentés dans une même colonne, il n'y aura pas d'ambigüité possible.

Pour donner une sémantique à ces événements, on définit ensuite un certain nombre de relations, qui représentent différents types d'interactions entre les événements.

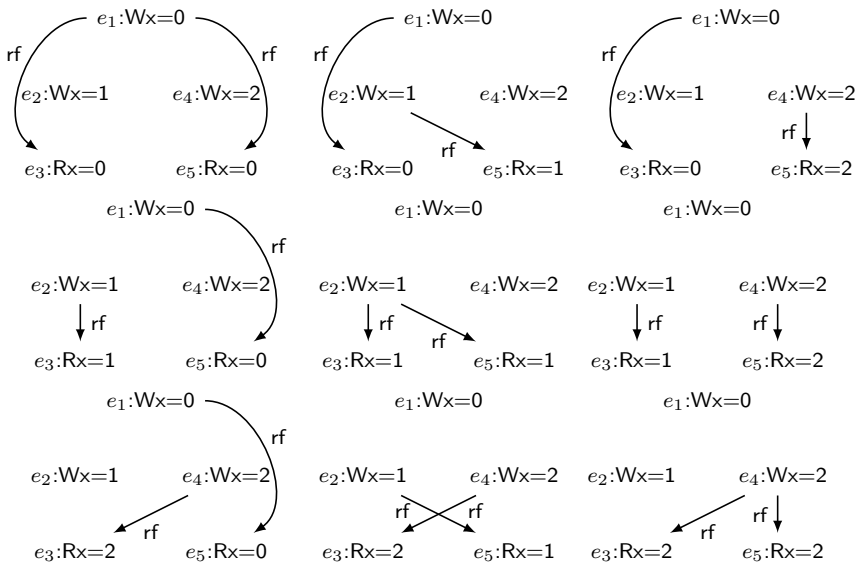
po (program order) Tous les événements issus d'un même processus sont ordonnés dans la relation d'ordre strict *po* suivant l'ordre dans lequel les instructions qui les ont générés apparaissent dans le code source du programme. Elle est donc partielle sur l'ensemble des événements, mais totale sur les événements d'un même processus. Les événements correspondant aux écritures initiales ne sont bien évidement pas inclus dans cette relation.

Si on reprend les événements décrits dans le schéma précédent, on obtient la relation *po* présentée ci-dessous :



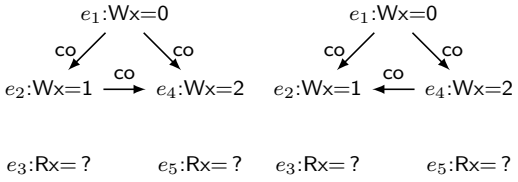
rf (read-from) Tout événement de lecture doit prendre une valeur depuis une unique écriture. La relation *rf* permet donc de relier chaque lecture avec l'écriture qui lui fournit sa valeur. Il existe donc plusieurs façons différentes de construire cette relation. Bien entendu, une même écriture peut fournir sa valeur à plusieurs lectures différentes.

En reprenant de nouveau les événements décrits dans le schéma précédent, on obtient les différentes relations *rf* présentées ci-dessous :



co (coherence) Quel que soit le modèle mémoire sous-jacent, il existe un ordre total dans lequel toutes les écritures sur une même variable deviennent globalement visibles à tous les processus, indépendamment de toute vision locale que pourrait en avoir chaque processus. On peut le voir comme l'ordre dans lequel ces écritures sont effectivement réalisées dans la mémoire partagée. La relation *co* représente donc cet ordre. On peut noter que les écritures initiales sont nécessairement avant toutes les autres écritures dans cette relation.

Si on reprend une nouvelles fois les événements décrits dans le schéma précédent, on obtient les deux relations *co* présentées ci-dessous :



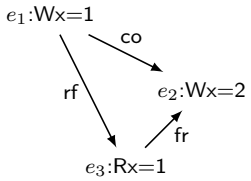
Une exécution sera alors décrite par un tuple $\mathcal{E} = (\mathbb{E}, po, rf, co)$. Les définitions précédentes étant minimalistes, elle laissent beaucoup de liberté quant aux exécutions que l'on peut construire. Pour autant, les exécutions ainsi construites ne sont pas nécessairement toutes correctes. On parlera donc d'*exécutions candidates*. Des contraintes supplémentaires vont permettre de filtrer ces exécutions selon qu'elles soient effectivement réalisables ou non. On divise ces contraintes en deux catégories, selon qu'elles dépendent ou non du modèle mémoire sous-jacent.

On s'intéresse d'abord aux contraintes indépendantes du modèle mémoire. Pour exprimer ces contraintes, il est nécessaire de définir de nouvelles relations, dérivées des précédentes.

fr (from-read) On a vu précédemment que dans toute exécution, il existe un ordre total dans lequel toutes les écritures vers une même variable deviennent globalement visible à tous les processus (*co*). En conséquence de ceci, lorsqu'une lecture prend sa valeur depuis une écriture, alors toutes les écritures situées après cette écriture dans *co* doivent *nécessairement* se situer dans le temps après la lecture. Il s'agit de la plus petite relation *fr* qui satisfait l'axiome suivant :

$$\forall e_1, e_2, e_3 \cdot co(e_1, e_2) \wedge rf(e_1, e_3) \rightarrow fr(e_3, e_2) \qquad \text{FR}$$

Ainsi, si on prend deux événements d'écriture quelconques e_1 et e_2 tels que $co(e_1,e_2)$ et un événement de lecture e_3 prenant sa valeur de e_1, on obtient la relation *fr* décrite dans le schéma suivant :



Par ailleurs, les relations *rf* et *fr* peuvent être scindées en deux, selon qu'elles relient des événements issus d'un même processus ou de processus différents. On aura donc *rfi* et *fri* pour les sous-relations reliant des événements appartement au même processus, et *rfe* et *fre* pour les sous-relations reliant des événements issus de processus différents [2].

On peut alors préciser les contraintes portant sur *rf* et *fr*, et indépendantes du modèle mémoire. Dans un même processus, une lecture ne peut pas prendre sa valeur depuis une écriture située après. De façon similaire, une écriture située avant une lecture ne peut avoir lieu après, du point de vue du processus qui l'effectue. Autrement dit, ces contraintes précisent que les relations *rfi* et *fri* doivent être compatibles avec *po*. On peut les exprimer de la façon suivante :

$$\forall e_1, e_2 \cdot rfi(e_1, e_2) \rightarrow po(e_1, e_2) \qquad \text{UNIPROCRW}$$
$$\forall e_1, e_2 \cdot fri(e_1, e_2) \rightarrow po(e_1, e_2) \qquad \text{UNIPROCWR}$$

On s'intéresse maintenant aux contraintes spécifiques au modèle mémoire. Pour exprimer ces contraintes, il est nécessaire de définir de nouvelles relations. Ces relations dépendent du modèle mémoire sous-jacent : on les présentera en adoptant le point de vue du modèle mémoire TSO.

ppo (preserved program order) Un des effets principaux d'un modèle mémoire faible est de relâcher l'ordre dans lequel les opérations mémoire issues d'un même processus sont effectuées. En d'autres termes, l'ordre des opérations mémoires ne suit pas nécessairement *po*. Comme on l'a vu au Chapitre 1, différents relâchements sont possibles. Dans le modèle TSO, c'est l'ordre W → R qui est relâché. On définit donc *ppo* comme la restriction de *po* aux paires d'événements autres que WR.

2. e = external, i = internal

fence Puisqu'un modèle mémoire faible ne conserve pas dans *ppo* toutes les paires d'événements de *po*, la sémantique d'un programme concurrent s'en trouve modifiée par rapport à SC, ce qui peut amener à des comportements incorrects. Pour rétablir l'ordre de certaines paires d'événements non préservé dans *ppo*, le programmeur peut ajouter des barrières mémoires dans son programme. Différent types de barrières mémoires permettent d'empêcher différentes sortes de relâchements. En TSO, seul le relâchement W $\rightarrow$ R est possible. Il n'existe donc qu'un seul type de barrière. La relation *fence* représente alors les paires d'événements WR issus d'un même processus et séparés par une barrière mémoire.

ghb (global happens-before) Les relations précédentes nous permettent de définir une relation d'ordre strict (partiel) *ghb*, qui représente l'ordre dans lequel les opérations mémoires sont effectuées d'un point de vue global. Toute les relations ne contribuent pas nécessairement à cet ordre global, qui dépend par ailleurs du modèle mémoire employé. Pour TSO, on définit *ghb* comme la plus petite relation qui satisfait les axiomes suivants :

$$
\begin{array}{rcll}
\forall e\cdot & \neg ghb(e,e) & & \text{GHB-IR} \\
\forall e_1,e_2,e_3\cdot ghb(e_1,e_2)\wedge ghb(e_2,e_3) & \rightarrow & ghb(e_1,e_3) & \text{GHB-T} \\
\forall e_1,e_2\cdot ppo(e_1,e_2) & \rightarrow & ghb(e_1,e_2) & \text{GHB-PPO} \\
\forall e_1,e_2\cdot fence(e_1,e_2) & \rightarrow & ghb(e_1,e_2) & \text{GHB-FENCE} \\
\forall e_1,e_2\cdot rfe(e_1,e_2) & \rightarrow & ghb(e_1,e_2) & \text{GHB-RFE} \\
\forall e_1,e_2\cdot co(e_1,e_2) & \rightarrow & ghb(e_1,e_2) & \text{GHB-CO} \\
\forall e_1,e_2\cdot fr(e_1,e_2) & \rightarrow & ghb(e_1,e_2) & \text{GHB-FR}
\end{array}
$$

On note que c'est la relation *ppo* qui est utilisée et non *po* : en effet, les paires d'événements WR de *po* n'étant pas préservées par TSO, elles ne participent pas à l'ordre global des événements mémoires. On note également que l'on considère la relation *rfe* et non *rf* : lorsqu'un processus effectue une lecture intra-processus, celle-ci peut avoir lieu depuis son tampon d'écriture, ce qui n'a aucun impact sur l'ordre global des événements. De ce fait, *rfi* ne participe pas non plus à la relation *ghb*.

A présent, il reste à préciser quand une exécution candidate est une exécution valide (ou réalisable). Une exécution candidate est valide dès lors que la relation *ghb* qui en découle est une relation d'ordre partiel valide, c'est-à-dire acyclique, et que les propriétés UNIPROCRW et UNIPROCWR sont vérifiées.

4.2.2 Prise en compte de l'ordonnancement dans le modèle TSO axiomatique

Dans Cubicle-$\mathcal{W}$, on effectue une exécution symbolique des systèmes de transitions analysés, tout en essayant de construire une relation *ghb* cohérente. Cette exécution symbolique simule un ordonnancement, or le modèle axiomatique défini précédemment n'utilise pas cette information. On souhaite alors utiliser les informations apportées par cet ordonnancement afin d'apporter des contraintes supplémentaires dans la construction de le relation *ghb* et de simplifier la vérification des propriétés UNIPROC*.

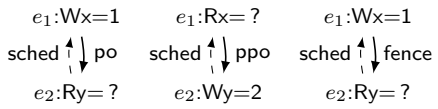
Afin de simplifier les explications qui vont suivre, on décrit l'ordonnancement à l'aide d'une relation d'ordre strict *sched*, qui ordonne totalement les événements selon l'ordre

dans lequel les instructions qui les ont générés ont été exécutées.

On définit également une notion de compatibilité avec l'ordonnancement :

Définition 1. *Une relation r est dite compatible avec un ordonnancement sched s'il n'existe pas de paire d'événements (e_1, e_2) telle que sched(e_1, e_2) et r(e_2, e_1) soient tous les deux vrais.*

On s'intéresse maintenant aux interactions entre *sched* et les différentes relations. Premièrement, on constate que la relation *po* et toute relation dérivée de celle-ci, *i.e. ppo* et *fence*, doivent nécessairement être compatibles avec l'ordonnancement, puisqu'elles matérialisent l'ordre des instructions dans le code source du programme. Les exécutions suivantes sont donc invalides vis-à-vis de *sched* :



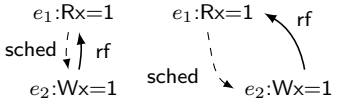
Autrement dit, ces relations doivent respecter les axiomes suivants :

$$\begin{array}{rcll} \forall e_1, e_2 \cdot po(e_1, e_2) & \rightarrow & sched(e_1, e_2) & \text{SCHED-PO} \\ \forall e_1, e_2 \cdot ppo(e_1, e_2) & \rightarrow & sched(e_1, e_2) & \text{SCHED-PPO} \\ \forall e_1, e_2 \cdot fence(e_1, e_2) & \rightarrow & sched(e_1, e_2) & \text{SCHED-FENCE} \end{array}$$

Par ailleurs, une lecture ne pouvant lire que depuis une écriture ayant été ordonnancée avant, l'axiome suivant doit également être respecté :

$$\forall e_1, e_2 \cdot rf(e_1, e_2) \quad \rightarrow \quad sched(e_1, e_2) \qquad \text{SCHED-RF}$$

Ceci interdit les exécutions suivantes :



En conséquence, seule la relation *co* est indépendante de l'ordonnancement. Cependant, on va voir qu'en pratique, il est possible de forcer *co* à être compatible avec l'ordonnancement, sans perdre d'exécutions réalisables.

Pour ce faire, on commence par redéfinir *ghb* de façon à isoler les relations qui dépendent de l'ordonnancement, ainsi que *co*. On définit une nouvelle relation *hb* comme étant la plus petite relation qui satisfait les axiomes suivants :

$$\begin{array}{rcll} \forall e \cdot & \neg hb(e, e) & & \text{HB-IR} \\ \forall e_1, e_2, e_3 \cdot hb(e_1, e_2) \wedge hb(e_2, e_3) & \rightarrow & hb(e_1, e_3) & \text{HB-T} \\ \forall e_1, e_2 \cdot ppo(e_1, e_2) & \rightarrow & hb(e_1, e_2) & \text{HB-PPO} \\ \forall e_1, e_2 \cdot fence(e_1, e_2) & \rightarrow & hb(e_1, e_2) & \text{HB-FENCE} \\ \forall e_1, e_2 \cdot rfe(e_1, e_2) & \rightarrow & hb(e_1, e_2) & \text{HB-RFE} \\ \forall e_1, e_2 \cdot co(e_1, e_2) & \rightarrow & hb(e_1, e_2) & \text{HB-CO} \end{array}$$

Enfin, la relation *ghb* est redéfinie comme étant la plus petite relation qui satisfait les axiomes suivants :

$$\begin{array}{rcll} \forall e\cdot & \neg ghb(e,e) & & \text{GHB-IR} \\ \forall e_1,e_2,e_3\cdot ghb(e_1,e_2)\wedge ghb(e_2,e_3) & \rightarrow & ghb(e_1,e_3) & \text{GHB-T} \\ \forall e_1,e_2\cdot hb(e_1,e_2) & \rightarrow & ghb(e_1,e_2) & \text{GHB-HB} \\ \forall e_1,e_2\cdot fr(e_1,e_2) & \rightarrow & ghb(e_1,e_2) & \text{GHB-FR} \end{array}$$

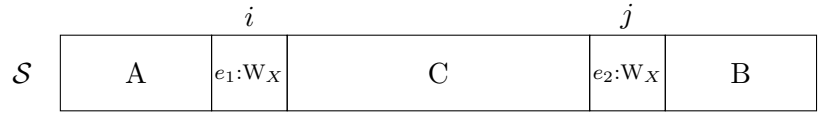
On va maintenant montrer que pour toute relation *co* participant d'une exécution valide, il existe un ordonnancement qui est compatible avec cette relation *co*. On commence par énoncer et démontrer les lemmes intermédiaires utilisés, puis on réalise la preuve de ce théorème.

Définition 2. *Soit $\mathbb{E}$ un ensemble d'événements et sched un ordonnancement de la totalité des événements de $\mathbb{E}$. On appelle séquence d'ordonnancement la séquence $\mathcal{S}$ de longueur $l = |\mathbb{E}|$ contenant une et une seule fois chaque événement de $\mathbb{E}$ et telle que $\forall i,j \in \{1..l\}\cdot i < j \leftrightarrow sched(\mathcal{S}[i],\mathcal{S}[j])$.*

Lemme 1. *Pour toute exécution $\mathcal{E} = (\mathbb{E}, po, rf, co)$ et tout ordonnancement sched d'un programme $\mathcal{P}$, pour toute paire d'événements d'écriture (e_1,e_2) issus de deux processus différents et telle que $sched(e_1,e_2)$ est vrai, si $hb(e_1,e_2)$ est faux, alors la séquence d'ordonnancement $\mathcal{S}$ issue de sched peut être divisée en deux sous-séquences $\mathcal{S}_1$ et $\mathcal{S}_2$ telles que :*

- $\exists i\cdot \mathcal{S}_1[i] = e_1$
- $\exists j\cdot \mathcal{S}_2[j] = e_2$
- $\forall k_1,k_2\cdot i \le k_1 \wedge k_2 \le j \rightarrow \neg hb(\mathcal{S}_1[k_1],\mathcal{S}_2[k_2])$
- $\forall k_1,k_2\cdot i \le k_1 \wedge k_2 \le j \rightarrow \neg po(\mathcal{S}_1[k_1],\mathcal{S}_2[k_2])$

Démonstration. Soit $\mathcal{S}$ la séquence d'ordonnancement issue de *sched*. On s'intéresse dans cette séquence à deux événements d'écriture sur une même variable $\mathcal{S}[i] = e_1$ et $\mathcal{S}[j] = e_2$, issus de deux processus différents et tels que $i < j$ et $hb(\mathcal{S}[i],\mathcal{S}[j])$ est faux. Le schéma suivant représente cette séquence et la position relative des deux événements $\mathcal{S}[i]$ et $\mathcal{S}[j]$:

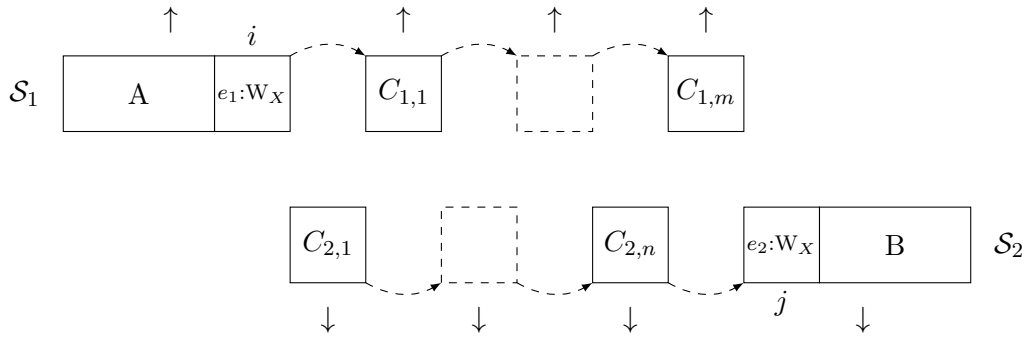


Puisque $hb(\mathcal{S}[i],\mathcal{S}[j])$ est faux, on sait qu'il n'existe pas d'indice k tel que $hb(\mathcal{S}[i],\mathcal{S}[k])$ et $hb(\mathcal{S}[k],\mathcal{S}[j])$, autrement on aurait $hb(\mathcal{S}[i],\mathcal{S}[j])$. On peut alors proposer de diviser $\mathcal{S}$ en deux *sous-séquences* S_1 et S_2 de sorte que, pour tout indice k :

- si $k \le i$ alors $\mathcal{S}[k] \in \mathcal{S}_1$
- si $k \ge j$ alors $\mathcal{S}[k] \in \mathcal{S}_2$
- si $i < k < j$ et $(\mathcal{S}[i],\mathcal{S}[k]) \in (hb \cup po)^+$ alors $\mathcal{S}[k] \in \mathcal{S}_1$
- si $i < k < j$ et $(\mathcal{S}[k],\mathcal{S}[j]) \in (hb \cup po)^+$ alors $\mathcal{S}[k] \in \mathcal{S}_2$

— si aucune des règles précédentes ne s'applique, $\mathcal{S}[k]$ peut appartenir indifféremment à $\mathcal{S}_1$ ou à $\mathcal{S}_2$, mais tous les evénements dans le même cas doivent appartenir à la même sous-séquence, on décide alors arbitrairement que $\mathcal{S}[k] \in \mathcal{S}_1$; autrement dit : si $i < k < j$ et $(\mathcal{S}[i], \mathcal{S}[k]) \notin (hb \cup po)^+$ et $(\mathcal{S}[k], \mathcal{S}[j]) \notin (hb \cup po)^+$ alors $\mathcal{S}[k] \in \mathcal{S}_1$

La figure suivante illustre le découpage ainsi obtenu :



Les règles utilisées pour construire ce découpage impliquent qu'aucun événement de C_1 n'est situé avant un événement de C_2 dans les relations *hb* et *po*, sinon ces événements devraient se situer dans la même sous-séquence, et on aurait alors e_1 et e_2 dans la même sous-séquence, ce qui contredit les règles de construction.

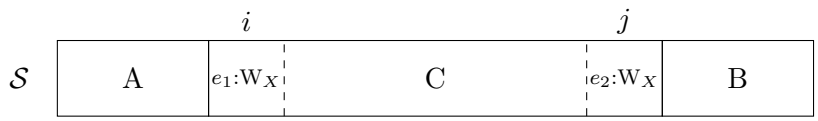
On peut alors compacter les deux sous-séquences comme suit :

En conséquence, on a donc bien deux sous-séquences $\mathcal{S}_1$ et $\mathcal{S}_2$ telles que :
— $\exists i \cdot \mathcal{S}_1[i] = e_1$
— $\exists j \cdot \mathcal{S}_2[j] = e_2$
— $\forall k_1, k_2 \cdot i \leq k_1 \wedge k_2 \leq j \rightarrow \neg hb(\mathcal{S}_1[k_1], \mathcal{S}_2[k_2])$
— $\forall k_1, k_2 \cdot i \leq k_1 \wedge k_2 \leq j \rightarrow \neg po(\mathcal{S}_1[k_1], \mathcal{S}_2[k_2])$

□

Lemme 2. *Pour toute exécution $\mathcal{E} = (\mathbb{E}, po, rf, co)$ et tout ordonnancement sched d'un programme $\mathcal{P}$, pour toute paire d'événements d'écriture (e_1, e_2) issus de deux processus différents et telle que sched(e_1, e_2) est vrai, si co(e_2, e_1) est également vrai, alors il existe* nécessairement *un autre ordonnancement sched$'$ tel que :*
— *sched$'(e_2, e_1)$ est vrai*
— *pour toute paire d'événements (e_3, e_4) telle que hb(e_3, e_4) et sched(e_3, e_4) sont tous les deux vrais, sched$'(e_3, e_4)$ est vrai*

Démonstration. Soit $\mathcal{S}$ la séquence d'ordonnancement issue de *sched*. On s'intéresse dans cette séquence à deux événements d'écriture sur une même variable $\mathcal{S}[i] = e_1$ et $\mathcal{S}[j] = e_2$, issus de deux processus différents et tels que $i < j$ et $co(\mathcal{S}[j], \mathcal{S}[i])$ est vrai. Le schéma suivant représente cette séquence et la position relative des deux événements $\mathcal{S}[i]$ et $\mathcal{S}[j]$:

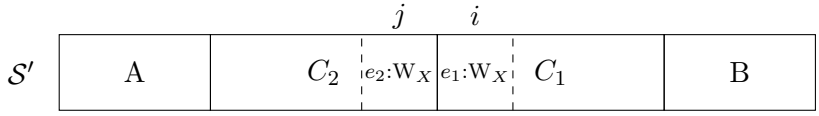


Puisque $co(\mathcal{S}[j], \mathcal{S}[i])$ est vrai, alors nécessairement $hb(\mathcal{S}[j], \mathcal{S}[i])$ est vrai, et donc $hb(\mathcal{S}[i], \mathcal{S}[j])$ est faux. On peut alors utiliser le Lemme 1, qui nous permet de diviser cette séquence $\mathcal{S}$ en deux sous-séquences $\mathcal{S}_1$ et $\mathcal{S}_2$ telles que :

— $\exists i \cdot \mathcal{S}_1[i] = e_1$
— $\exists j \cdot \mathcal{S}_2[j] = e_2$
— $\forall k_1, k_2 \cdot i \leq k_1 \wedge k_2 \leq j \rightarrow \neg hb(\mathcal{S}_1[k_1], \mathcal{S}_2[k_2])$
— $\forall k_1, k_2 \cdot i \leq k_1 \wedge k_2 \leq j \rightarrow \neg po(\mathcal{S}_1[k_1], \mathcal{S}_2[k_2])$

Un tel découpage est illustré dans le schéma suivant :

Pour toute paire d'événements (e_{C_1}, e_{C_2}) telle que $e_{C_1} \in C_1$ et $e_{C_2} \in C_2$, on a que $hb(e_{C_1}, e_{C_2})$ et $po(e_{C_1}, e_{C_2})$ sont faux tous les deux (par définition du découpage produit par le Lemme 1). Ainsi, (e_{C_1}, e_{C_2}) peut être librement ordonnancée dans un sens ou dans l'autre sans contredire *hb* ni *po*. En conséquence, tout entrelacement des événements de C_1 avec les événements de C_2 constitue un ordonnancement valide, puisque l'ordre relatif des événements issus de C_1 et de C_2 est préservé dans l'ordonnancement obtenu. En particulier, on peut réordonnancer l'intégralité des événements de C_2 avant les événements de C_1, comme illustré dans le schéma suivant :



Cette séquence $\mathcal{S}'$ constitue un nouvel ordonnancement *sched*$'$ dans lequel on a bien *sched*$'(e_2, e_1)$, tout comme on a $co(e_2, e_1)$, et pour toute paire d'événements (e_3, e_4) telle que $hb(e_3, e_4)$ et *sched*(e_3, e_4) sont vrais tous les deux, *sched*$'(e_3, e_4)$ est vrai.

□

Théorème 1. *Pour toute exécution $\mathcal{E} = (\mathbb{E}, po, rf, co)$ et tout ordonnancement sched d'un programme $\mathcal{P}$, il existe un ordonnancement sched$'$ tel que co soit compatible avec sched$'$.*

Démonstration. On procède par induction sur le nombre de paires d'événements dans *co* incompatibles avec *sched*. Soit n le nombre de paires d'événements (e_1, e_2) telles que $co(e_2, e_1)$ et *sched*(e_1, e_2) sont tous les deux vrais.

Cas de base Si $n = 0$, alors *sched*$' =$ *sched* et le théorème est vrai

Cas inductif Si $n > 0$, alors il existe une paire d'événements d'écriture (e_1, e_2) telle que $co(e_2, e_1)$ et $sched(e_1, e_2)$ sont tous les deux vrais. Alors, d'après le Lemme 2, il existe un autre ordonnancement $sched'$ tel que $sched'(e_2, e_1)$ est vrai et pour toute paire d'événements (e_3, e_4) telle que $hb(e_3, e_4)$ et $sched(e_3, e_4)$ sont vrais tous les deux, $sched'(e_3, e_4)$ est vrai. En conséquence, toutes les paires d'événements qui étaient à la fois dans *co* et *sched* sont toujours dans $sched'$, et une paire qui était dans *co* mais pas dans *sched* est maintenant dans $sched'$. On substitue alors l'ordonnancement $sched'$ à l'ordonnancement *sched*. On a maintenant une paire de moins telle que $co(e_2, e_1)$ et $sched(e_1, e_2)$ sont tous les deux vrais, autrement dit n a diminué.

□

Grâce à ce théorème, on peut donc s'autoriser à construire une relation *co* compatible avec l'ordonnancement. De cette façon, on pourra construire plus efficacement la relation *ghb*, comme on le verra en Section 4.2.4.

4.2.3 Prise en compte des opérations atomiques

Dans Cubicle-$\mathcal{W}$, on peut être amené à manipuler plusieurs variables dans une même transition, par exemple :

— lorsqu'un tableau représente un compteur de processus, on peut lire ou écrire plusieurs de ses cases simultanément

— lorsqu'on souhaite effectuer une opération de *lecture-modification-écriture* atomique

On aura donc plusieurs événements générés pour une même transition. On voudrait pouvoir exprimer le fait que ces événements puissent avoir lieu de façon atomique. Pour ce faire, on aura recours à deux mécanismes.

Premièrement, tous les événements d'un même type (lecture ou écriture) et issus d'un même processus se verront attribuer le même identificateur d'événement. Un identificateur ne correspondra donc plus à un seul événement, mais à un ensemble d'événements, différentiables uniquement par la variable sur laquelle ils portent. Les relations définies précédemment porteront sur ces ensembles, et seront établies dès lors que deux événements issus chacun de deux ensembles distincts satisferont les conditions de ces relations. Par exemple, si e_w identifie un ensemble d'écritures et e_r identifie un ensemble de lectures d'un processus différent, on aura $rf(e_w, e_r)$ dès lors qu'au moins une des lectures identifiées par e_r est satisfaite par une des écritures identifiée par e_w. De cette façon, on n'aura besoin que de deux identificateurs d'événements par transition : un pour les lectures, et un pour les écritures.

Deuxièmement, lorsqu'une transition contient à la fois des lectures et des écritures, on a affaire à une opération de *lecture-modification-écriture*. Dans certains cas, on peut vouloir rendre cette opération atomique. Pour ce faire, on définit une relation d'équivalence *atom*. L'ensemble des lectures e_r et des écritures e_w d'une même transition sera rendu atomique en écrivant $atom(e_r, e_w)$. Cette relation peut également servir à définir un point de synchronisation entre les événements de deux processus distincts : on utilisera cette possibilité dans les formules décrivant les états dangereux, lorsque ceux-ci portent sur des processus différents.

Pour prendre en compte cette nouvelle relation, on ajoute simplement les deux axiomes suivants dans la construction de *ghb* :

$$\begin{array}{lcll} \forall e_1, e_2, e_3 \cdot ghb(e_1, e_2) \wedge atom(e_2, e_3) & \rightarrow & ghb(e_1, e_3) & \text{GHB-ATOM-R} \\ \forall e_1, e_2, e_3 \cdot atom(e_1, e_2) \wedge ghb(e_2, e_3) & \rightarrow & ghb(e_1, e_3) & \text{GHB-ATOM-L} \end{array}$$

4.2.4 Stratégie de construction de *ghb* en arrière

Dans Cubicle-$\mathcal{W}$, l'analyse d'atteignabilité se fait en arrière. L'ordonnancement ainsi simulé se fait donc également en arrière. En utilisant les connaissances supplémentaires apportées par l'ordonnancement *sched*, on va pouvoir construire plus efficacement la relation *ghb*. Dans ce qui suit, on parle de nouvel événement pour décrire un événement apporté par une nouvelle itération de l'algorithme, et d'ancien événement pour décrire tout événement déjà connus avant cette itération. Les nouveaux événements sont donc avant les anciens dans *sched*. On établit alors comme suit les règles de construction de *ghb* :

- une nouvelle lecture e_r sera avant tout ancien événement e_2 du même processus dans *ghb*, selon GHB-PPO
- une nouvelle écriture e_{w1} sera avant toute ancienne écriture e_{w2} du même processus dans *ghb*, selon GHB-PPO
- une nouvelle écriture e_w sera avant toute ancienne lecture e_r du même processus dans *ghb* si elles sont séparées par une barrière mémoire, selon GHB-FENCE
- une nouvelle écriture e_{w1} sera avant toute ancienne écriture e_{w2} sur la même variable dans *ghb*, selon GHB-CO et grâce à la compatibilité entre *co* et *sched*
- une nouvelle écriture e_w sera avant toute ancienne lecture e_r d'un processus différent qu'elle satisfait dans *ghb*, selon GHB-RFE
- une nouvelle écriture e_w sera après toute ancienne lecture e_r d'un processus différent qu'elle ne satisfait pas dans *ghb*, selon GHB-FR
- une nouvelle lecture e_r sera avant toute ancienne écriture e_w sur la même variable dans *ghb*, selon GHB-FR

En construisant *ghb* de cette façon, on garantit également que l'axiome UNIPROCRW est toujours vérifié : une nouvelle lecture e_r ne pourra jamais prendre sa valeur d'une ancienne écriture e_w, on n'aura donc jamais $po(e_r, e_w)$ et $rf(e_w, e_r)$. On n'a donc pas besoin de vérifier explicitement UNIPROCRW.

On peut également s'épargner de vérifier explicitement UNIPROCWR en choisissant judicieusement les paires *rf* que l'on construit : lorsqu'on découvre une nouvelle écriture e_w, celle-ci doit satisfaire toutes les lectures e_r non encore satisfaites du même processus. En effet, si on a $po(e_w, e_r)$ mais pas $rf(e_w, e_r)$ cela signifie que la lecture e_r doit prendre sa valeur depuis une autre écriture que l'on a pas encore rencontrée. Lorsque cette nouvelle écriture e_{w2} sera découverte, on aura nécessairement $co(e_{w2}, e_w)$ (de par la compatibilité de *co* avec *sched*), et par FR on aura également $fr(e_r, e_w)$, ce qui contredirait $po(e_w, e_r)$.

Grâce à toutes ces simplifications, on peut dorénavant se contenter de raisonner uniquement sur *ghb*. Il n'est pas nécessaire de conserver les relations intermédiaires, ni de se soucier des axiomes UNIPROCRW et UNIPROCWR, dont on garantit la correction par construction.

4.3 Théorie des tableaux faibles

Dans Cubicle, un unique langage logique est utilisé pour représenter les états et pour décrire les systèmes de transitions. Pour réaliser notre extension, on a besoin d'ajouter au langage les notions d'événements et de relations décrites précédemment. Toutefois, si les états vont effectivement décrire des événements et des relations, les transitions décrivent des opérations sur des variables et tableaux. On ne peut donc plus utiliser le même langage pour décrire les états et les transitions. Les états seront alors décrits dans un langage interne que l'on nomme langage à événements $\mathcal{L}_{\mathcal{E}}$, présenté dans cette section, tandis que les transitions seront exprimées dans un langage de description $\mathcal{L}_{\mathcal{D}}$, qui sera présenté dans la section suivante.

4.3.1 Langage à événements

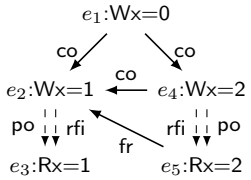
Dans cette section, on présente la syntaxe du langage $\mathcal{L}_{\mathcal{E}}$. On commence par définir les différents objets manipulés : des constantes de différents types (entiers, réels, booléens, constructeurs), des variables de processus, des identificateurs d'événements, des tableaux SC et des variables et tableaux faibles (instanciés et non instanciés).

$$
\begin{aligned}
\textit{const, } c &::= \text{constantes} \\
\textit{proc, } i, j, k &::= \text{variables de processus} \\
\textit{eid, } e &::= \text{identificateurs d'événements} \\
x, y, z &::= \text{tableaux SC} \\
\alpha, \beta, \gamma &::= \text{variables et tableaux faibles} \\
\tilde{\alpha}, \tilde{\beta}, \tilde{\gamma} &::= \text{variables et tableaux faibles instanciés} \\
\textit{op} &::= = \mid \neq \mid < \mid \leq \\
\textit{term, } t &::= c \mid i \mid x[j] \mid \tilde{\alpha}[e] \mid \tilde{\alpha}[e, j] \\
\textit{atom, } a &::= t \ \textit{op} \ t \mid \mathsf{true} \mid \mathsf{false} \\
&\quad \mid \mathsf{R}_\alpha(e, i) \mid \mathsf{R}_\alpha(e, i, j) \mid \mathsf{W}_\alpha(e, i) \mid \mathsf{W}_\alpha(e, i, j) \\
&\quad \mid \mathsf{fence}(e, i) \mid \mathsf{ghb}(e, e) \mid \mathsf{atom}(e, e) \\
\textit{literal, } l &::= a \mid \neg a \\
\textit{qf\_form, } qff &::= l \mid qff \wedge qff \mid qff \vee qff \\
\textit{formula, } f &::= qff \mid \forall \vec{x} : \textit{type}\cdot f \mid \exists \vec{x} : \textit{type}\cdot f
\end{aligned}
$$

Les événements sont décrits par des littéraux du type R_α et W_α. Plus précisément, $\mathsf{R}_\alpha(e, i)$ représente un événement de lecture sur la variable α, d'identificateur e et effectué par le processus d'identificateur i. $\mathsf{R}_\alpha(e, i, j)$ représente un événement de lecture sur $\alpha[j]$ d'identificateur e et effectué par le processus d'identificateur i. Les événements d'écriture W_α sont définis de la même façon. Ces termes sont définis pour toutes les variable et tableaux α. Par ailleurs, pour chaque variable ou tableau faible α, il existe une variable ou un tableau faible instancié, de la forme $\tilde{\alpha}$, qui permet de décrire la valeur associée à chaque événement. Le prédicat $\mathsf{fence}(e, i)$ permet d'indiquer la présence d'une barrière mémoire avant l'événement e du processus i. $\mathsf{ghb}(e, e)$ et $\mathsf{atom}(e, e)$ permettent de représenter les relations *ghb* et *atom*.

Exemple

Prenons une exécution possible du programme présenté en Section 4.2.1 et illustrée par le schéma suivant :



On rappelle que la relation *ghb* est constituée des diverses relations *ppo*, *fence*, *co*, *rfe* et *fr*, et que *ppo* est la restriction de *po* aux paires d'événements autres que celles constituées d'une écriture suivie d'une lecture. La formule correspondant à ce schéma est alors la suivante :

$$\begin{aligned}\exists p1, p2 : proc, \exists e_1, e_2, e_3, e_4, e_5 : eid \cdot \quad & \mathsf{W}_x(e_1) \wedge \tilde{x}[e_1] = 0 \wedge \\ & \mathsf{W}_x(e_2, p_1) \wedge \tilde{x}[e_2] = 1 \wedge \\ & \mathsf{R}_x(e_3, p_1) \wedge \tilde{x}[e_3] = 1 \wedge \\ & \mathsf{W}_x(e_4, p_2) \wedge \tilde{x}[e_4] = 2 \wedge \\ & \mathsf{R}_x(e_5, p_2) \wedge \tilde{x}[e_5] = 2 \wedge \\ & \mathsf{ghb}(e_1, e_2) \wedge \mathsf{ghb}(e_1, e_4) \wedge \\ & \mathsf{ghb}(e_4, e_2) \wedge \mathsf{ghb}(e_5, e_2)\end{aligned}$$

4.3.2 Sémantique des formules logiques

On présente à présent la sémantique du langage $\mathcal{L}_{\mathcal{E}}$. Etant donné un modèle $\mathcal{M}$, le langage $\mathcal{L}_{\mathcal{E}}$ s'interprète selon les règles suivantes :

$$
\begin{array}{lcll}
\mathcal{M}[c] & = & \mathcal{M}(c) & \\
\mathcal{M}[i] & = & \mathcal{M}(i) & \\
\mathcal{M}[e] & = & \mathcal{M}(e) & \\
\mathcal{M}[x[j]] & = & x^{\mathcal{M}}(\mathcal{M}[j]) & \\
\mathcal{M}[\tilde{\alpha}[e]] & = & \tilde{\alpha}^{\mathcal{M}}(\mathcal{M}[e]) & \\
\mathcal{M}[\tilde{\alpha}[e,j]] & = & \tilde{\alpha}^{\mathcal{M}}(\mathcal{M}[e],\mathcal{M}[j]) & \\
\mathcal{M} \models t_1 \ op \ t_2 & = & \mathcal{M}[t_1] \ op \ \mathcal{M}[t_2] & \\
\mathcal{M} \models \mathsf{ghb}(e_1,e_2) & = & (\mathcal{M}[e_1],\mathcal{M}[e_2]) \in \mathsf{ghb}^{\mathcal{M}} & \\
\mathcal{M} \models \mathsf{atom}(e_1,e_2) & = & (\mathcal{M}[e_1],\mathcal{M}[e_2]) \in \mathsf{atom}^{\mathcal{M}} & \\
\mathcal{M} \models \mathsf{Rd}_\alpha(e,i) & = & (\mathcal{M}[e],\mathcal{M}[i]) \in {\mathsf{Rd}_\alpha}^{\mathcal{M}} & \\
\mathcal{M} \models \mathsf{Rd}_\alpha(e,i,j) & = & (\mathcal{M}[e],\mathcal{M}[i],\mathcal{M}[j]) \in {\mathsf{Rd}_\alpha}^{\mathcal{M}} & \\
\mathcal{M} \models \mathsf{Wr}_\alpha(e,i) & = & (\mathcal{M}[e],\mathcal{M}[i]) \in {\mathsf{Wr}_\alpha}^{\mathcal{M}} & \\
\mathcal{M} \models \mathsf{Wr}_\alpha(e,i,j) & = & (\mathcal{M}[e],\mathcal{M}[i],\mathcal{M}[j]) \in {\mathsf{Wr}_\alpha}^{\mathcal{M}} & \\
\mathcal{M} \models \mathsf{fence}(e,i) & = & (\mathcal{M}[e],\mathcal{M}[i]) \in \mathsf{fence}^{\mathcal{M}} & \\
\mathcal{M} \models \neg a & = & \mathcal{M} \not\models a & \\
\mathcal{M} \models \mathit{qff}_1 \wedge \mathit{qff}_2 & = & \mathcal{M} \models \mathit{qff}_1 \text{ et } \mathcal{M} \models \mathit{qff}_2 & \\
\mathcal{M} \models \mathit{qff}_1 \vee \mathit{qff}_2 & = & \mathcal{M} \models \mathit{qff}_1 \text{ ou } \mathcal{M} \models \mathit{qff}_2 & \\
\mathcal{M} \models \forall x : type \cdot f & = & \mathcal{M}\{x \mapsto v\} \models f & \text{pour tout } v \in \mathcal{D}^{type} \\
\mathcal{M} \models \exists x : type \cdot f & = & \mathcal{M}\{x \mapsto v\} \models f & \text{pour un } v \in \mathcal{D}^{type}
\end{array}
$$

Le domaine du modèle $\mathcal{D}_{\mathcal{M}}$ est partitionné selon les types *proc*, représentant les identificateurs de processus, *eid*, représentant les identificateurs d'événements, et *val*, pour les valeurs que peuvent prendre les différentes variables et tableaux. On a donc $\mathcal{D}_{\mathcal{M}} = \mathcal{D}^{proc} \uplus \mathcal{D}^{eid} \uplus \mathcal{D}^{val}$.

4.3.3 Etats

Les états représentent les contraintes portant sur les tableaux SC, les événements, leurs valeurs et leurs relations. Un état est exprimé dans le langage $\mathcal{L}_{\mathcal{E}}$ par une formule paramétrée par les ensembles X et $\tilde{A}$, X représentant l'ensemble des tableaux SC et $\tilde{A}$ l'ensemble des variables et tableaux faibles instanciés. La formule est préfixée par un ensemble d'identificateurs d'événements quantifiés existentiellement et tous différents :

$$\varphi(X,\tilde{A}) = \exists \vec{e} : eid \cdot \Delta(\vec{e}) \wedge \Phi(\vec{e},X,\tilde{A})$$

La seconde partie de la formule, Φ, est paramétrée par l'ensemble d'identificateurs d'événements et les ensembles les ensembles X et $\tilde{A}$ définis précédemment, et est préfixée par un ensemble d'identificateurs de processus tous différents :

$$\Phi(\vec{e},X,\tilde{A}) = \exists \vec{j} : proc \cdot \Delta(\vec{j}) \wedge \phi(\vec{e},\vec{j},X,\tilde{A})$$

Enfin, $\phi(\vec{e},\vec{j},X,\tilde{A})$ est une conjonction de littéraux qui décrit effectivement les contraintes portant sur les tableaux SC, les événements, leurs valeurs et leurs relations.

4.4 Systèmes de transitions à tableaux faibles

Dans notre approche, les systèmes étudiés sont des systèmes de transition paramétrés, c'est-à-dire des ensembles de transitions manipulant des variables et tableaux indicés par des identificateurs de processus. Du point de vue de l'utilisateur, la notion d'événements présentée précédemment importe peu : les accès aux variables et tableaux sont compris comme tels. En revanche, dans notre analyse, on manipule des états exprimés dans le langage à événements $\mathcal{L}_{\mathcal{E}}$, c'est-à-dire des états comportant des événements et des relations. Pour concilier ces deux points de vue, on exprime les systèmes de transition dans un langage de description $\mathcal{L}_{\mathcal{D}}$, dans lequel les événements n'apparaissent pas, et on utilise des fonctions de traduction pour transposer un système du langage $\mathcal{L}_{\mathcal{D}}$ vers le langage à événements $\mathcal{L}_{\mathcal{E}}$. On présente à présent les différents éléments constitutifs d'un système de transition avec mémoire faible. A chaque étape, on donne un exemple issu du Mutex Naïf présenté précédemment en Figure 3.3.

4.4.1 Langage de description

Le langage de description $\mathcal{L}_{\mathcal{D}}$ permet à l'utilisateur d'exprimer les états et transitions en faisant abstraction des événements et des relations. Une grande partie de ce langage est commune avec le langage à événements $\mathcal{L}_{\mathcal{E}}$.

$$
\begin{aligned}
\textit{const, } c &::= \text{constantes} \\
\textit{proc, } i, j, k &::= \text{variables de processus} \\
x, y, z &::= \text{tableaux SC} \\
\alpha, \beta, \gamma &::= \text{variables et tableaux faibles} \\
op &::= = \mid \neq \mid < \mid \leq \\
\textit{term, } t &::= c \mid i \mid x[j] \mid \alpha \mid \alpha[j] \mid i \mathbin{@} \alpha \mid i \mathbin{@} \alpha[j] \\
\textit{atom, } a &::= t \ op \ t \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{fence}() \\
\textit{literal, } l &::= a \mid \neg a \\
\textit{qf\_form, } qff &::= l \mid qff \wedge qff \\
\textit{uformula, } uf &::= \forall \vec{j} : proc \cdot \ qff \\
\textit{eformula, } ef &::= \exists \vec{j} : proc \cdot \ qff
\end{aligned}
$$

La principale différence avec le langage $\mathcal{L}_{\mathcal{E}}$ réside dans l'accès aux variables et tableaux faibles. Les termes $i \mathbin{@} \alpha$ et $i \mathbin{@} \alpha[j]$ représentent des accès réalisés par un processus i, tandis que α et $\alpha[j]$ ne précisent pas quel processus effectue l'accès. Le contexte permet de décider quelle forme doit être utilisée. Le prédicat $\mathsf{fence}()$ se comporte également différemment par rapport au langage $\mathcal{L}_{\mathcal{E}}$: il ne représente plus une relation mais indique si le tampon d'un processus est vide ou non, autrement dit, il est vrai pour un processus lorsque toutes ses écritures sont devenues visibles globalement pour les autres processus.

Par ailleurs, dans un souci de praticité, on utilise les notations suivantes pour représenter différents ensembles de variables fréquemment utilisés :

— X : l'ensemble des tableaux SC $(x, y...)$

— $\tilde{A}$: l'ensemble des variables et tableaux faibles instanciés

— A^0 : l'ensemble de toutes les variables faibles
— A^1 : l'ensemble de tous les tableaux faibles
— A : l'ensemble de toutes les variables et tableaux faibles ($A^0 \cup A^1$)
— A^0_t, A^1_t et A_t : restriction des ensembles A^0, A^1 et A, aux variables et tableaux faibles manipulés par la transition t

4.4.2 Etats initiaux

Les états initiaux décrivent les contraintes portant sur les tableaux SC et les variables et tableaux faibles au démarrage du système. Ils sont décrits dans le langage $\mathcal{L}_\mathcal{D}$ par une formule I paramétrée par un ensemble de tableaux SC X et préfixée par une variable de processus quantifiée universellement. Puisque dans les états initiaux, tous les processus ont la même vision de la mémoire, les accès aux variables et tableaux faibles doivent se faire en utilisant les termes de la forme α ou $\alpha[j]$ (i @ α et i @ $\alpha[j]$ ne sont pas permis). Par ailleurs, fence() n'est pas non plus utilisable dans ce contexte.

$$I(X) = \forall j : proc \cdot \mathcal{I}(j, X)$$

La formule équivalente $\tilde{I}$ dans le langage $\mathcal{L}_\mathcal{E}$ est obtenue en appliquant la fonction de traduction $[\![]\!]_I$:

$$\tilde{I}(X, \tilde{A}) = [\![I(X)]\!]_I = \forall i, j : proc, \forall \vec{e_\alpha} : ur \cdot \bigwedge_{\alpha \in A^0} \mathsf{R}_\alpha(e_\alpha, i) \ \wedge \bigwedge_{\alpha \in A^1} \mathsf{R}_\alpha(e_\alpha, i, j) \ \wedge \ [\![\mathcal{I}(j, X)]\!]_I$$

Cette fonction génère un événement de lecture pour chaque variable ou tableau faible du système, et pas uniquement ceux effectivement utilisés dans I (toutefois, seulement ces derniers seront associés à une valeur). On dispose d'un identificateur d'événement e_α par variable ou tableau faible α. Les identificateurs d'événements sont choisis dans le domaine $\mathcal{D}^{ur}$, qui restreint les événements à ceux devant prendre leur valeur depuis l'état initial. Le processus effectuant l'opération est représenté par la variable de processus quantifiée universellement i. Cela signifie que dans l'état initial, tout processus effectuant une lecture d'une variable ou un tableau faible obtiendra la même valeur. La formule $\tilde{I}$ obtenue contenant à présent des variables et tableaux faibles instanciés, elle est paramétrée par l'ensemble $\tilde{A}$.

La fonction de traduction $[\![]\!]_I : \mathcal{L}_\mathcal{D} \to \mathcal{L}_\mathcal{E}$ est définie comme suit :

$$\begin{array}{rcll}
[\![qff_1 \wedge qff_2]\!]_I & = & [\![qff_1]\!]_I \wedge [\![qff_2]\!]_I & \\
[\![\neg a]\!]_I & = & \neg [\![a]\!]_I & \\
[\![\mathsf{true}]\!]_I & = & \mathsf{true} & \\
[\![\mathsf{false}]\!]_I & = & \mathsf{false} & \\
[\![t_1 \ op \ t_2]\!]_I & = & [\![t_1]\!]_I \ op \ [\![t_2]\!]_I & \\
[\![\alpha]\!]_I & = & \tilde{\alpha}[e_\alpha] & \\
[\![\alpha[j]]\!]_I & = & \tilde{\alpha}[e_\alpha, j] & \\
[\![t]\!]_I & = & t & \text{quand } t \neq \alpha \text{ et } t \neq \alpha[j]
\end{array}$$

Exemple

Dans le Mutex Naïf, l'état initial est défini comme suit :

$$I = \forall j : proc \cdot State[j] = Idle \wedge X[j] = False$$

En appliquant la fonction de traduction $[\![]\!]_I$, on obtient la formule suivante, dans laquelle les identificateurs d'événements sont explicitement mentionnés :

$$\tilde{I} = \forall i, j : proc, \forall e_X : ur \cdot State[j] = Idle \wedge R_X(e_X, i, j) \wedge \tilde{X}[e_X, j] = False$$

Domaine des événements de lecture non satisfaits

On a vu que la formule initiale obtenue par traduction quantifie sur les événements de type ur, c'est-à-dire les événements de lecture non satisfaits. Pour caractériser ces événements, on définit le domaine $\mathcal{D}^{ur} \subseteq \mathcal{D}^{eid}$ comme le sous-ensemble des identificateurs d'événements correspondant uniquement aux événements de lecture non satisfaits, *i.e.* les événements de lecture qui ne sont reliés à aucun événement d'écriture, et on a :

$$\forall e_r \in \mathcal{D}^{eid} \cdot unsat\_read(e_r) \leftrightarrow e_r \in \mathcal{D}^{ur}$$

Le prédicat $unsat\_read$ indique si un événement de lecture est relié ou non à un événement d'écriture quelconque :

$$\begin{aligned} unsat\_read(e_r) \quad = \quad & \forall i, j, k \in \mathcal{D}^{proc}, \forall e_w \in \mathcal{D}^{eid}. \\ & \Big(\bigvee_{\alpha \in A^0} (e_r, i) \in \mathsf{R}_\alpha \wedge (e_w, j) \in \mathsf{W}_\alpha \Big) \vee \\ & \Big(\bigvee_{\alpha \in A^1} (e_r, i, k) \in \mathsf{R}_\alpha \wedge (e_w, j, k) \in \mathsf{W}_\alpha \Big) \rightarrow (e_r, e_w) \in \mathsf{ghb} \end{aligned}$$

4.4.3 Etats dangereux

Les états dangereux du système sont décrits par des contraintes portant sur les tableaux SC ainsi que les variables et tableaux faibles. Ces états sont décrits dans le langage $\mathcal{L}_\mathcal{D}$ par une formule Θ, paramétrée par un ensemble de tableaux SC X. Ils utilisent un certain nombre de variables de processus, quantifiées existentiellement. Contrairement à l'état initial, différents processus peuvent avoir une vision différente de la mémoire, de ce fait les accès aux variables et tableaux faibles doivent se faire en utilisant les termes i @ α et i @ $\alpha[j]$ (α et $\alpha[j]$ ne sont pas permis). Par ailleurs, fence() n'est pas non plus utilisable dans ce contexte.

$$\Theta(X) = \exists \vec{j} : proc \cdot \Delta(\vec{j}) \wedge \vartheta(\vec{j}, X)$$

La formule équivalente $\tilde{\Theta}$ dans le langage $\mathcal{L}_\mathcal{E}$ est obtenue en appliquant la fonction de traduction $[\![]\!]_u$:

$$\tilde{\Theta}(X, \tilde{A}) = [\![\Theta(X)]\!]_u = \exists \vec{e} : eid \cdot \Delta(\vec{e}) \wedge \Diamond(\vec{e}) \wedge \theta(\vec{e}, X, \tilde{A})$$

Avec :

$$\theta(\vec{e}, X, \tilde{A}) = \exists \vec{j} : proc \cdot \Delta(\vec{j}) \wedge [\![\vartheta(\vec{j}, X)]\!]_u^{\vec{e}}$$

La notation $\Diamond(\vec{e})$ est un raccourci pour exprimer le fait que tous les identificateurs d'événements dans $\vec{e}$ sont dans la relation *atom* :

$$\Diamond(\vec{e}) \quad = \quad \bigwedge_{(e_1,e_2)\in\vec{e}} \mathsf{atom}(e_1, e_2)$$

La fonction de traduction instancie tous les accès à des variables ou tableaux faibles, et génère un identificateur d'événement e_i par processus.

La fonction de traduction $[\![]\!]_u : \mathcal{L}_\mathcal{D} \times \vec{eid} \to \mathcal{L}_\mathcal{E}$ est définie comme suit :

$$\begin{array}{rcll}
[\![qff_1 \wedge qff_2]\!]_u^{\vec{e}} & = & [\![qff_1]\!]_u^{\vec{e}} \wedge [\![qff_2]\!]_u^{\vec{e}} & \\
[\![a]\!]_u^{\vec{e}} & = & [\![a]\!]_{u_e}^{\vec{e}} \wedge [\![a]\!]_{u_v}^{\vec{e}} & \\
[\![\neg a]\!]_u^{\vec{e}} & = & [\![a]\!]_{u_e}^{\vec{e}} \wedge \neg [\![a]\!]_{u_v}^{\vec{e}} & \\
[\![t_1 \ op \ t_2]\!]_{u_e}^{\vec{e}} & = & [\![t_1]\!]_{u_e}^{\vec{e}} \wedge [\![t_2]\!]_{u_e}^{\vec{e}} & \\
[\![t_1 \ op \ t_2]\!]_{u_v}^{\vec{e}} & = & [\![t_1]\!]_{u_v}^{\vec{e}} \ op \ [\![t_2]\!]_{u_v}^{\vec{e}} & \\
[\![i \ @ \ \alpha]\!]_{u_e}^{\vec{e}} & = & \mathsf{R}_\alpha(e_i, i) & e_i \in \vec{e} \\
[\![i \ @ \ \alpha]\!]_{u_v}^{\vec{e}} & = & \tilde{\alpha}[e_i] & e_i \in \vec{e} \\
[\![i \ @ \ \alpha[j]]\!]_{u_e}^{\vec{e}} & = & \mathsf{R}_\alpha(e_i, i, j) & e_i \in \vec{e} \\
[\![i \ @ \ \alpha[j]]\!]_{u_v}^{\vec{e}} & = & \tilde{\alpha}[e_i, j] & e_i \in \vec{e} \\
[\![t]\!]_{u_e}^{\vec{e}} & = & \mathsf{true} & \text{quand } t \neq i \ @ \ \alpha \text{ et } t \neq i \ @ \ \alpha[j] \\
[\![t]\!]_{u_v}^{\vec{e}} & = & t & \text{quand } t \neq i \ @ \ \alpha \text{ et } t \neq i \ @ \ \alpha[j]
\end{array}$$

La fonction $[\![]\!]_u$ utilise deux sous-fonctions $[\![]\!]_{u_e}$ et $[\![]\!]_{u_v}$. La première permet de construire les littéraux décrivant les événements, tandis que la seconde permet d'instancier les accès aux variables et tableaux faibles.

Exemple

Dans le Mutex Naïf, la propriété de sûreté proposée ne fait pas intervenir de variable faible. Pour disposer d'un exemple plus intéressant, on s'intéresse alors à une propriété quelque peu différente : un processus i ne peut pas être à l'état `Crit` si sa variable `X` est à faux.

$$\Theta = \exists i : proc \cdot State[i] = Crit \wedge X[i] = False$$

En appliquant la fonction de traduction $[\![]\!]_u$, on obtient la formule suivante, dans laquelle les identificateurs d'événements sont explicitement mentionnés :

$$\tilde{\Theta} = \exists e_i : eid, \exists i : proc \cdot State[i] = Crit \wedge R_X(e_i, i, i) \wedge \tilde{X}[e_i, i] = False$$

4.4.4 Transitions

Les transitions décrivent l'évolution du système. Elles se composent d'une garde, qui précise les conditions devant être vérifiées pour que la transition soit prise, et d'actions, qui sont des mises à jour de tableaux SC ou de variables et tableaux faibles. Elles sont

décrites dans le langage $\mathcal{L}_{\mathcal{D}}$, et puisqu'un seul processus effectue toutes les opérations mémoire dans une transition donnée, les accès aux variables faibles doivent utiliser les termes α ou $\alpha[j]$ (i @ α et i @ $\alpha[j]$ ne sont pas permis). Par ailleurs, le prédicat fence() ne peut être utilisé que dans la garde.

Une transition t est décrite dans le langage $\mathcal{L}_{\mathcal{D}}$ par une formule paramétrée par deux ensembles X et X', représentant les tableaux SC avant et après application de la transition.

$$\begin{aligned} t(X,X') \quad = \quad & \exists i, \vec{j} : proc \cdot \Delta(\vec{j}) \wedge \gamma(i,\vec{j},X) \wedge \\ & \bigwedge_{x \in X} x'[i] = \delta_x(i,\vec{j},X) \wedge \\ & \bigwedge_{\alpha \in A_t^0} \alpha' = \delta_\alpha(i,\vec{j},X) \wedge \\ & \bigwedge_{\alpha \in A_t^1} \bigwedge_{k \in \vec{j}} \alpha'[k] = \delta_\alpha(i,\vec{j},X) \end{aligned}$$

La variable de processus quantifiée existentiellement i représente le processus effectuant les accès aux variables et tableaux faibles. Cela signifie également que les termes d'accès aux tableaux SC sont restreints à ceux de la forme $x[i]$. Le processus i peut être égal ou non à un processus dans $\vec{j}$. La garde est représentée par la sous-formule $\gamma(i,\vec{j},X)$, les mises à jour de tableaux SC par la sous-formule $\delta_x(i,\vec{j},X)$ et les mises à jour de variables et tableaux faibles par les sous-formules $\delta_\alpha(i,\vec{j},X)$.

La transition équivalente $\tilde{t}$ dans le langage $\mathcal{L}_{\mathcal{E}}$ est obtenue en appliquant la fonction de traduction $[\![]\!]_t$, qui prend deux paramètres additionnels e_r et e_w, représentant des identificateurs d'événements frais :

$$\tilde{t}(X,X',\tilde{A},\tilde{A}',e_r,e_w) \quad = \quad [\![t(X,X')]\!]_t^{e_r,e_w}$$

Et on a :

$$\begin{aligned} [\![t(X,X')]\!]_t^{e_r,e_w} \quad = \quad & \exists i, \vec{j} : proc \cdot \Delta(\vec{j}) \wedge e_r \neq e_w \wedge \mathsf{atom}(e_r,e_w) \wedge [\![\gamma(i,\vec{j},X)]\!]_\gamma^{i,e_r} \wedge \\ & \bigwedge_{x \in X} \Big([\![x'[i] = \delta_x(i,\vec{j},X)]\!]_\gamma^{i,e_r} \wedge (\forall k.k = i \ \vee \ x'[k] = x[k]) \Big) \wedge \\ & \bigwedge_{\alpha \in A_t^0} \mathsf{W}_\alpha(e_w,i) \wedge [\![\tilde{\alpha}'[e_w] = \delta_\alpha(i,\vec{j},X)]\!]_\gamma^{i,e_r} \wedge \\ & \bigwedge_{\alpha \in A_t^1} \bigwedge_{k \in \vec{j}} \mathsf{W}_\alpha(e_w,i,k) \wedge [\![\tilde{\alpha}'[e_w,k] = \delta_\alpha(i,\vec{j},X)]\!]_\gamma^{i,e_r} \end{aligned}$$

La transition s'assure que les identificateurs d'événements e_r et e_w soient différents, et les relie dans la relation *atom*. Les mises à jour de tableaux SC sont étendues de sorte que toutes les cases autres que i reçoivent une valeur égale à la précédente. Les mises à jour de variables et tableaux faibles génèrent des événements d'écriture. Les traductions de la garde et des mises à jour sont réalisées par une nouvelle fonction de traduction

$[\![]\!]_\gamma : \mathcal{L}_\mathcal{D} \times proc \times eid \to \mathcal{L}_\mathcal{E}$.

$$
\begin{array}{rcll}
[\![qff_1 \wedge qff_2]\!]^{i,e_r}_\gamma & = & [\![qff_1]\!]^{i,e_r}_\gamma \wedge [\![qff_2]\!]^{i,e_r}_\gamma & \\
[\![a]\!]^{i,e_r}_\gamma & = & [\![a]\!]^{i,e_r}_{\gamma_e} \wedge [\![a]\!]^{i,e_r}_{\gamma_v} & \\
[\![\neg a]\!]^{i,e_r}_\gamma & = & [\![a]\!]^{i,e_r}_{\gamma_e} \wedge \neg [\![a]\!]^{i,e_r}_{\gamma_v} & \\
[\![\mathsf{fence}()]\!]^{i,e_r}_{\gamma_e} & = & \mathsf{fence}(e_r, i) & \\
[\![\mathsf{fence}()]\!]^{i,e_r}_{\gamma_v} & = & \mathsf{true} & \\
[\![t_1 \ op \ t_2]\!]^{i,e_r}_{\gamma_e} & = & [\![t_1]\!]^{i,e_r}_{\gamma_e} \wedge [\![t_2]\!]^{i,e_r}_{\gamma_e} & \\
[\![t_1 \ op \ t_2]\!]^{i,e_r}_{\gamma_v} & = & [\![t_1]\!]^{i,e_r}_{\gamma_v} \ op \ [\![t_2]\!]^{i,e_r}_{\gamma_v} & \\
[\![\alpha]\!]^{i,e_r}_{\gamma_e} & = & \mathsf{R}_\alpha(e_r, i) & \\
[\![\alpha]\!]^{i,e_r}_{\gamma_v} & = & \tilde{\alpha}[e_r, i] & \\
[\![\alpha[j]]\!]^{i,e_r}_{\gamma_e} & = & \mathsf{R}_\alpha(e_r, i, j) & \\
[\![\alpha[j]]\!]^{i,e_r}_{\gamma_v} & = & \tilde{\alpha}[e_r, i, j] & \\
[\![t]\!]^{i,e_r}_{\gamma_e} & = & \mathsf{true} & \text{quand } t \neq \alpha \text{ et } t \neq \alpha[j] \\
[\![t]\!]^{i,e_r}_{\gamma_v} & = & t & \text{quand } t \neq \alpha \text{ et } t \neq \alpha[j]
\end{array}
$$

Exemple

Dans le Mutex Naïf, on a les trois transitions suivantes :

$$
\begin{array}{rcl}
t_{req} & = & \exists i : proc.\, State[i] = Idle \wedge \\
 & & \quad State'[i] = Want \wedge X'[i] = True \\
t_{enter} & = & \exists i : proc.\, State[i] = Want \wedge \mathsf{fence}() \wedge (\forall k.k = i \vee X[i] = False) \wedge \\
 & & \quad State'[i] = Crit \\
t_{exit} & = & \exists i : proc.\, State[i] = Crit \wedge \\
 & & \quad State'[i] = Idle \wedge X'[i] = False
\end{array}
$$

En appliquant la fonction de traduction $[\![]\!]_t$, on obtient les formules suivantes, dans lesquelles les identificateurs d'événements sont explicitement mentionnés :

$$
\begin{array}{rcl}
\tilde{t}_{req}(e_r, e_w) & = & \exists i : proc.\, e_r \neq e_w \wedge \mathsf{atom}(e_r, e_w) \wedge \\
 & & State[i] = Idle \wedge \\
 & & State'[i] = Want \wedge W_X(e_w, i, i) \wedge \tilde{X}'(e_w, i) = True \\
\tilde{t}_{enter}(e_r, e_w) & = & \exists i : proc.\, e_r \neq e_w \wedge \mathsf{atom}(e_r, e_w) \wedge \mathsf{fence}(e_r, i) \wedge \\
 & & State[i] = Want \ \wedge (\forall k.k = i \vee R_X(e_r, i, k) \wedge \tilde{X}(e_r, i) = False) \wedge \\
 & & State'[i] = Crit \\
\tilde{t}_{exit}(e_r, e_w) & = & \exists i : proc.\, e_r \neq e_w \wedge \mathsf{atom}(e_r, e_w) \wedge \\
 & & State[i] = Crit \wedge \\
 & & State'[i] = Idle \wedge W_X(e_w, i, i) \wedge \tilde{X}'(e_w, i) = False
\end{array}
$$

4.5 Analyse d'atteignabilité

4.5.1 Algorithme d'atteignabilité arrière

Dans notre approche, on utilise un algorithme d'atteignabilité en arrière très similaire à l'algorithme initial de Cubicle donné en Section 3.1.6. Ce nouvel algorithme est présenté

en Algorithme 2.

```
1  function Bwd(S, Θ) : begin
2  |  V := ∅;
3  |  push(Q, [[Θ]]_u);
4  |  while not_empty(Q) do
5  |  |  φ := pop(Q);
6  |  |  if φ ∧ [[I]]_I satisfiable then
7  |  |  |  return unsafe
8  |  |  end
9  |  |  else if φ ⊭ V then
10 |  |  |  V := V ∪ {φ};
11 |  |  |  push(Q, Pre_τ(φ)) ;
12 |  |  end
13 |  end
14 |  return safe
15 end
```

Algorithme 2 : Algorithme d'atteignabilité arrière de Cubicle-$\mathcal{W}$

Ses différences avec l'algorithmes original de Cubicle sont les suivantes :
— initialisation de la file $\mathcal{Q}$ (ligne 3) : la formule Θ est maintenant traduite du langage $\mathcal{L}_{\mathcal{D}}$ vers le langage $\mathcal{L}_{\mathcal{E}}$ de façon à rendre explicites les événements
— test de sûreté (ligne 6) : la formule I est maintenant traduite du langage $\mathcal{L}_{\mathcal{D}}$ vers le langage $\mathcal{L}_{\mathcal{E}}$ de façon à rendre explicites les événements
— calcul de pré-image (ligne 11) : le calcul est étendu afin de produire les événements et gérer les différentes relations, suivant notre modèle de mémoire faible
Un exemple d'exploration en arrière sera donné en Section 4.5.3

4.5.2 Calcul de préimage

Dans cette section, on présente une nouvelle façon d'effectuer le calcul de pré-image, de façon à prendre en compte les événements et les relations du modèle mémoire.

On rappelle qu'une formule décrivant des états est de la forme :

$$\varphi(X, \tilde{A}) = \exists \vec{e} : eid \cdot \Delta(\vec{e}) \wedge \Phi(\vec{e}, X, \tilde{A})$$

Pour calculer la préimage d'une formule φ par un ensemble de transitions τ, il suffit de calculer la pré-image de φ pour chaque transition de τ. Ainsi, la pré-image s'exprime comme suit :

$$\textsc{Pre}_\tau(\varphi)(X, \tilde{A}) = \bigvee_{t \in \tau} \textsc{Pre}_t(\varphi)(X, \tilde{A})$$

Quant à la préimage d'une formule φ par une unique transition t, elle est exprimée de

la façon suivante :

$$\begin{aligned}\textsc{Pre}_t(\varphi)(X,\tilde{A}) \;=\; & \exists X', \exists \tilde{A}', \exists e_r, e_w, \vec{e} : \mathit{eid}\cdot \Delta(e_r\cdot e_w\cdot \vec{e}) \wedge \\ & [\![t(X,X')]\!]_t^{e_r,e_w} \wedge \Phi(\vec{e},X',\tilde{A}') \wedge \\ & \mathit{extend\_ghb}(e_r,e_w,\vec{e}) \wedge \mathit{rffr}(\tilde{A},\tilde{A}',e_w,\vec{e})\end{aligned}$$

Ce calcul de préimage génère deux nouveaux identificateurs d'événements e_r et e_w. On s'assure que ces identificateurs soient différents de ceux dans φ à l'aide de l'expression $\Delta(e_r\cdot e_w\cdot \vec{e})$. Ces identificateurs sont donnés en paramètres à la fonction de traduction $[\![]\!]_t$, ce qui garantit que la transition ne manipule que des nouveaux événements. On cherche alors à construire *ghb*, selon la stratégie décrite en Section 4.2.4.

Fonction rffr La fonction *rffr* détermine si les nouvelles écritures e_w de la transition t peuvent satisfaire des lectures compatibles non satisfaites dans $\Phi(\vec{e},X',\tilde{A}')$, et les ordonne correctement dans la relation *ghb*, le cas échéant. Cela correspond à la relation *rf* et à une partie de la relation *fr* définies dans le modèle axiomatique en Section 4.2. Une lecture et une écriture sont compatibles si elles concernent la même variable ou la même case de tableau (la valeur associée aux événements n'entre pas en compte dans cette notion de compatibilité). Pour chaque lecture non satisfaite dans $\vec{e}$, soit :

- il existe une écriture compatible e_w provenant du *même* processus, dans ce cas la lecture DOIT prendre sa valeur depuis cette écriture
- il existe une écriture compatible e_w provenant d'un processus *différent*, dans ce cas la lecture PEUT prendre sa valeur depuis cette écriture ; si elle le fait, e_w est ordonné avant e_r dans *ghb*, dans le cas contraire c'est e_r qui est ordonné avant e_w dans *ghb*
- il n'existe pas d'écriture compatible : la lecture reste insatisfaite

On adopte donc la définition logique suivante pour *rffr*, en faisant usage des sous-fonctions $\mathit{int\_rffr}_\alpha$, $\mathit{ext\_rffr}_\alpha$ et $\mathit{no\_rf}_\alpha$ pour représenter ces trois cas :

$$\begin{aligned}\mathit{rffr}(\tilde{A},\tilde{A}',e_w,\vec{e}) \;=\; & \bigwedge_{\alpha\in A}\bigwedge_{e_r\in\vec{e}}\Big(\exists i,k : \mathit{proc}\cdot \mathit{unsat\_read}_\alpha(e_r,i,k,\vec{e}) \rightarrow \\ & \qquad \mathit{int\_rffr}_\alpha(\tilde{A},\tilde{A}',e_r,e_w,i,k) \veebar \\ & \qquad \mathit{ext\_rffr}_\alpha(\tilde{A},\tilde{A}',e_r,e_w,i,k) \veebar \\ & \qquad \mathit{no\_rf}_\alpha(\tilde{A},\tilde{A}',e_r,e_w,k)\Big)\end{aligned}$$

$$\mathit{int\_rffr}_\alpha(\tilde{A},\tilde{A}',e_r,e_w,i,k) \;=\; \mathit{write}_\alpha\_\mathit{by}(e_w,i,k) \wedge \tilde{\alpha}'[e_r,k] = \tilde{\alpha}'[e_w,k]$$

$$\begin{aligned}\mathit{ext\_rffr}_\alpha(\tilde{A},\tilde{A}',e_r,e_w,i,k) \;=\; & \exists j : \mathit{proc}\cdot j \neq i \wedge \mathit{write}_\alpha\_\mathit{by}(e_w,j,k) \wedge \\ & (\tilde{\alpha}'[e_r,k] = \tilde{\alpha}'[e_w,k] \wedge \mathsf{ghb}(e_w,e_r) \vee \\ & \tilde{\alpha}'[e_r,k] = \tilde{\alpha}[e_r,k] \wedge \mathsf{ghb}(e_r,e_w)\end{aligned}$$

$$\mathit{no\_rf}_\alpha(\tilde{A},\tilde{A}',e_r,e_w,i,k) \;=\; (\nexists j : \mathit{proc}\cdot \mathit{write}_\alpha\_\mathit{by}(e_w,j,k)) \wedge \tilde{\alpha}'[e_r,k] = \tilde{\alpha}[e_r,k]$$

rffr utilise également une fonction $unsat\_read_\alpha$ pour déterminer si un événement de lecture e_r n'est satisfait par aucun événement d'écriture dans $\vec{e}$ pour une certaine variable α :

$$\begin{aligned} &unsat\_read_\alpha(e_r, i, k, \vec{e}) \;\; = \\ &\quad read_\alpha\_by(e_r, i, k) \;\wedge \bigwedge_{e_w \in \vec{e}} \Big(e_r \neq e_w \rightarrow \big(\forall j : proc \cdot\, write_\alpha\_by(e_w, j, k) \rightarrow \mathsf{ghb}(e_r, e_w) \big) \Big) \end{aligned}$$

Fonction extend_ghb La fonction *extend_ghb* étend la relation *ghb* en ajoutant des littéraux de la forme $\mathsf{ghb}(e_1,\, e_2)$ dans la formule, en fonction des dépendances entre les nouveaux événements e_r et e_w et les anciens événements dans $\vec{e}$.

$$extend\_ghb(e_r, e_w, \vec{e}) \;\; = \;\; ppo(e_r \cdot e_w, \vec{e}) \wedge fence(e_w, \vec{e}) \wedge co(e_w, \vec{e}) \wedge fr(e_r, \vec{e})$$

fr ordonne dans *ghb* les nouveaux événements de lecture e_r avant les anciens événements d'écriture.

$$fr(e_r, \vec{e}) \;\; = \bigwedge_{\alpha \in A} \bigwedge_{e_w \in \vec{e}} \Big(\exists k : proc \cdot\, read_\alpha(e_r, k) \wedge write_\alpha(e_w, k) \rightarrow \mathsf{ghb}(e_r, e_w) \Big)$$

co ordonne dans *ghb* les nouveaux événements d'écriture e_w avant les anciens événements d'écriture.

$$co(e_{w_1}, \vec{e}) \;\; = \bigwedge_{\alpha \in A} \bigwedge_{e_{w_2} \in \vec{e}} \Big(\exists k : proc \cdot\, write_\alpha(e_{w_1}, k) \wedge write_\alpha(e_{w_2}, k) \rightarrow \mathsf{ghb}(e_{w_1}, e_{w_2} \Big)$$

fence ordonne dans *ghb* les nouveaux événements d'écriture e_w avant les anciens événements de lecture issues d'un même processus si elles sont séparées par un prédicat fence.

$$\begin{aligned} &fence(e_w, \vec{e}) \;\; = \\ &\quad \bigwedge_{e_r \in \vec{e}} \Big(\exists i : proc \cdot\, \mathsf{fence}(e_r, i) \wedge write\_by(e_w, i) \wedge read\_by(e_r, i) \rightarrow \mathsf{ghb}(e_w, e_r) \Big) \end{aligned}$$

ppo ordonne dans *ghb* les nouveaux événements e_a et les événements suivants e_b issus d'un même processus à l'exception des écritures suivies de lectures.

$$ppo(\vec{e_a}, \vec{e_b}) \;\; = \bigwedge_{e_1 \in \vec{e_a}} \bigwedge_{e_2 \in \vec{e_b}} \Big(ppo\_RR(e_1, e_2) \wedge ppo\_RW(e_1, e_2) \wedge ppo\_WW(e_1, e_2) \Big)$$

$$ppo\_RR(e_1, e_2) \;\; = \;\; \exists i : proc \cdot\, read\_by(e_1, i) \;\wedge\; read\_by(e_2, i) \rightarrow \mathsf{ghb}(e_1, e_2)$$

$$ppo\_RW(e_1, e_2) \;\; = \;\; \exists i : proc \cdot\, read\_by(e_1, i) \;\wedge\; write\_by(e_2, i) \rightarrow \mathsf{ghb}(e_1, e_2)$$

$$ppo\_WW(e_1, e_2) \;\; = \;\; \exists i : proc \cdot\, write\_by(e_1, i) \;\wedge\; write\_by(e_2, i) \rightarrow \mathsf{ghb}(e_1, e_2)$$

Prédicats auxiliaires Les définitions précédentes utilisent les prédicats auxiliaires suivants. Il s'agit essentiellement de sucre syntaxique destiné à rendre les formules plus lisibles.

Les prédicats $read_\alpha$ et $write_\alpha$ permettent de vérifier qu'un identificateur d'événement e correspond à un événement de lecture ou d'écriture sur une variable α ou une case de tableau $\alpha[k]$.

$$\begin{aligned} read_\alpha(e,k) &= \exists i : proc \cdot \mathsf{R}_\alpha(e,i) \vee \mathsf{R}_\alpha(e,i,k) \\ write_\alpha(e,k) &= \exists i : proc \cdot \mathsf{W}_\alpha(e,i) \vee \mathsf{W}_\alpha(e,i,k) \end{aligned}$$

De façon similaire, les prédicats *read_by* et *write_by* permettent de vérifier qu'un identificateur d'événement e corresponde à un événement de lecture ou d'écriture d'un certain processus i.

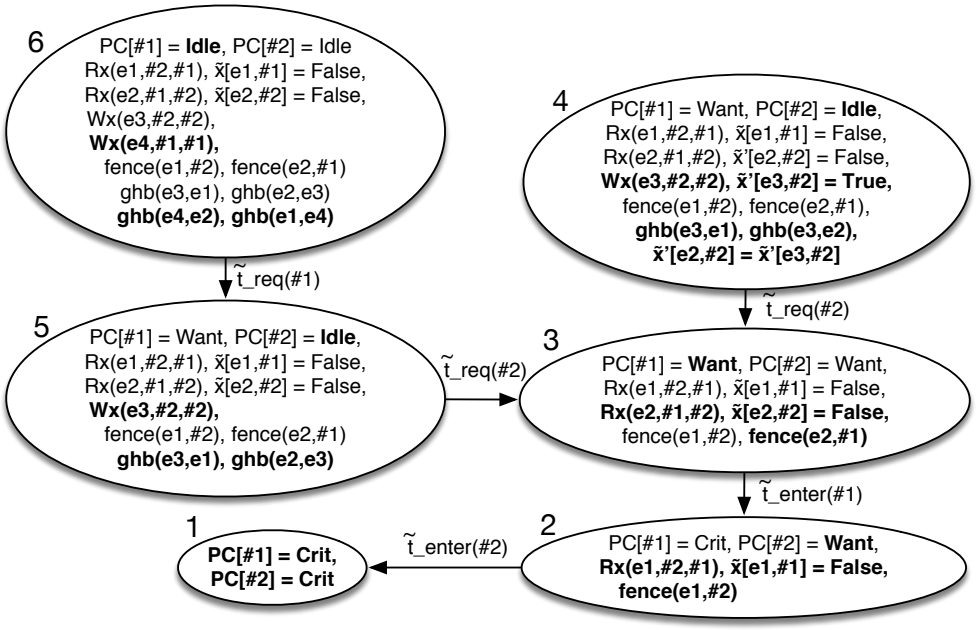
$$\begin{aligned} read\_by(e,i) &= \exists k : proc \cdot \bigvee_{\alpha \in A^0} \mathsf{R}_\alpha(e,i) \vee \bigvee_{\alpha \in A^1} \mathsf{R}_\alpha(e,i,j) \\ write\_by(e,i) &= \exists k : proc \cdot \bigvee_{\alpha \in A^0} \mathsf{W}_\alpha(e,i) \vee \bigvee_{\alpha \in A^1} \mathsf{W}_\alpha(e,i,j) \end{aligned}$$

Enfin, les prédicats $read_\alpha\_by$ et $write_\alpha\_by$ permettent de vérifier qu'un identificateur d'événement e corresponde à un événement de lecture ou d'écriture sur une variable α par un certain processus i.

$$\begin{aligned} read_\alpha\_by(e,i,k) &= \mathsf{R}_\alpha(e,i) \vee \mathsf{R}_\alpha(e,i,k) \\ write_\alpha\_by(e,i,k) &= \mathsf{W}_\alpha(e,i) \vee \mathsf{W}_\alpha(e,i,k) \end{aligned}$$

4.5.3 Exemple d'exploration arrière

On illustre le fonctionnement de l'algorithme d'atteignabilité arrière sur l'exemple du Mutex Naïf. Le graphe ci-dessous illustre une exploration possible de l'espace d'états de cet algorithme. On commence par l'état 1, qui représente la formule décrivant les états dangereux $\tilde{\Theta}$. Les états suivants représentent le résultat du calcul de préimage pour une instance d'une transition donnée. Les arcs sont étiquetés par le nom de la transition et les identificateurs de processus utilisés pour instancier les variables de processus quantifiées existentiellement dans chaque transition. Dans les formules représentant les états, les processus et les identificateurs d'événements sont implicitement quantifiés existentiellement. L'identificateur d'un processus i s'écrit $\#_i$. Par ailleurs, afin de simplifier le graphe, on omet les identificateurs d'événements dans les étiquettes des arcs, et on élimine les identificateurs d'événements non utilisés dans les états (ils ne contribuent pas à la relation *ghb*), et on suppose que tous les identificateurs d'événements sont différents.



On se concentre sur l'état 3 qui résulte de la préimage de l'état 1 par `t_enter(#2)` puis `t_enter(#1)`. Dans cet état, les deux processus ont lu `False` dans `X` (evénements e_1 et e_2). De plus, puisqu'il y a une barrière mémoire dans `t_enter`, les deux lectures sont associées à un littéral `fence`. La préimage de l'état 3 par `t_req(#2)` introduit un nouvel événement d'écriture `Wx(e3,#2,#2)` associé à la valeur `x̃'(e3,#2) = True`. Puisqu'il y a une barrière mémoire `fence(e1,#2)` sur `e1` par le même processus `#2`, le prédicat *extend_ghb* provoque l'ajout de `ghb(e3,e1)` dans la formule. D'après le prédicat *rffr*, ce nouvel événement d'écriture peut ou non satisfaire la lecture `e2`, il faut donc considérer les deux possibilités (états 4 et 5).

Dans l'état 4, on considère le cas tel que l'événement d'écriture `e3` satisfait l'événement de lecture `e2`. Comme indiqué dans le prédicat *ext_rffr*, l'égalité `x̃'(e2,#2) = x̃'(e3,#2)` est ajoutée à la formule, ce qui la rend inconsistante de façon évidente. Dans l'état 5, l'événement d'écriture `e3` ne satisfait pas l'événement de lecture `e2`, de ce fait la valeur `x̃'(e3,#2)` est éliminée et `ghb(e2,e3)` est ajouté à la formule, comme précisé par le prédicat *no_rf*. De la même façon, la préimage de l'état 5 par `t_req(#1)` produit la formule donnée à l'état 6, telle que le nouvel événement d'écriture `e4` ne satisfait pas l'événement de lecture `e1`. La relation *ghb* ainsi obtenue n'est pas un ordre partiel valide, car la séquence $\texttt{ghb}(\texttt{e2},\texttt{e3}), \texttt{ghb}(\texttt{e3},\texttt{e1}), \texttt{ghb}(\texttt{e1},\texttt{e4}), \texttt{ghb}(\texttt{e4},\texttt{e2})$ constitue une relation cyclique. Cet état est donc éliminé et le programme est déclaré *sûr*.

A noter que sans les prédicats fence dans `t_enter`, nous n'aurions que $\texttt{ghb}(\texttt{e3},\texttt{e1})$, $\texttt{ghb}(\texttt{e4},\texttt{e2})$ dans l'état 6, ce qui constitue une relation d'ordre partiel valide. La formule ainsi obtenue intersecterait donc avec l'état initial et le programme deviendrait *incorrect*.

4.6 Résultats et comparaison avec d'autres outils

Nous avons évalué les performances de notre outil sur divers algorithmes paramétrés, la plupart issus de la littérature, et les avons comparées à un certain nombre d'autres outils. Afin de choisir les outils auxquels nous comparer, nous avons effectué une première évaluation succincte des outils disponibles et présentés en Chapitre 2 et avons retenu les plus performants, tout en essayant d'avoir des représentants de différentes approches. Ainsi notre choix s'est porté sur Memorax [3], Dual-TSO, Trencher et CBMC [4]. Les deux premiers permettent la vérification de propriétés de sûreté, tandis que les deux suivants sont des outils de recherche de bugs. Par ailleurs, Dual-TSO, étant la seul outil à proposer une forme de vérification paramétrée, nous nous devions de l'inclure dans ce comparatif.

Afin de comparer les performances de Cubicle-$\mathcal{W}$ à celles des autres outils, nous avons donc écrit les différents algorithmes dans leurs langages d'entrée respectifs. Ceux-ci ne supportant pas les programmes paramétrés (à une exception près), nous avons créé pour chaque algorithme plusieurs instances de tailles fixes et croissantes. Dual-TSO proposant à la fois une approche non paramétrée et une approche paramétrée, nous l'avons évalué dans les deux modes. Toutefois, la plupart des algorithmes n'ont pas pu être exprimés dans leur version paramétrée en raison des restrictions que Dual-TSO impose dans son mode paramétré (interdiction des variables locales aux processus) et de l'impossibilité de manipuler les identificateurs de processus.

Les tests ont été effectuées sur un MacBook Pro équipé d'un processeur Intel Core i7 @ 2.9 Ghz et de 8 Go de RAM, sous OSX 10.11.6. Le délai d'expiration (TO) est de 15 minutes. Le Tableau 4.1 présente les résultats obtenus. ☠ signifie que l'outil a donné une mauvaise réponse. KO indique que l'outil a provoqué une erreur. L'indication S / US dans la deuxième colonne précise si l'algorithme est correct (S) ou incorrect (US). Le nombre de processus de l'instance est précisé entre crochets (N indiquant le cas paramétré). NT signifie qu'un algorithme n'a pas pu être exprimé dans le langage d'entrée d'un outil.

Le premier constat que l'on peut faire concerne l'efficacité de Cubicle-$\mathcal{W}$: sur les algorithmes les plus simples, moins d'une seconde suffit à prouver la correction (ou à trouver un bug pour les versions incorrectes), et il faut moins d'une minute sur des algorithmes plus complexes. Ce résultat est d'autant plus intéressant que les algorithmes ont été vérifiés pour un nombre arbitraire de processus. Si les autres outils sont par ailleurs très efficaces - voir plus que Cubicle-$\mathcal{W}$ - sur des algorithmes simples et des petites instances, leurs performances se dégradent très rapidement au fur et à mesure que le nombre de processus augmente et que l'algorithme étudié devient complexe.

Le cas du mutex naïf illustre parfaitement ces résultats : c'est un algorithme extrêmement simple, faisant intervenir un tout petit nombre d'instructions. Au delà de 5 processus, il n'y a guère plus que Memorax avec l'abstraction PB qui arrive à analyser l'algorithme en un temps raisonnable, et au delà de 10 processus, plus aucun outil ne parvient à répondre en un temps raisonnable. CBMC se distingue sur les algorithmes incorrects : étant conçu pour rechercher des bugs, il n'est pas surprenant qu'il résiste bien à l'augmentation du

3. Memorax a été testé avec l'option –rff
4. CBMC a été testé avec des déroulages de boucle d'ordre 2 et 3

		Cubicle $\mathcal{W}$	Memorax SB	Memorax PB	Trencher	CBMC Unwind 2	CBMC Unwind 3	Dual TSO
naive mutex	US	0.04s [N]	– TO [6] 7m54s [5] 13.9s [4]	– TO [10] 12m02s [9] 1m36s [8]	– TO [5] 10.1s [4] 0.08s [3]	– 23.6s [11] 12.7s [10] 10.3s [9]	– 5m37s [11] 3m39s [10] 2m35s [9]	NT [N] TO [6] 1m12s [5] 2.61s [4]
naive mutex	S	0.30s [N]	– TO [5] 23.3s [4] 0.16s [3]	– TO [11] 2m28s [10] 25.2s [9]	– TO [6] 54.8s [5] 0.31s [4]	– TO [5] 2m24s [4] 30.3s [3]	– TO [3] 19.4s [2]	NT [N] TO [5] 35.7s [4] 0.15s [3]
lamport	US	0.10s [N]	– TO [4] 17.4s [3] 0.23s [2]	– TO [4] 25.4s [3] 0.16s [2]	– KO [4] 1.73s [3] 0.05s [2]	– 7m42s [11] 4m29s [10] 2m23s [9]	– TO [7] 5m12s [6] 1m21s [5]	NT [N] TO [6] 13m12s [5] 34.6s [4]
lamport	S	0.60s [N]	– TO [3] 0.14s [2]	– TO [4] 3m02s [3] 0.21s [2]	– KO [5] 3.37s [4] 0.08s [3]	– TO [4] 8m39s [3] 1.71s [2]	– TO [3] 1m55s [2]	NT [N] TO [4] 9.42s [3] 0.04s [2]
spinlock [126]	S	0.07s [N]	– TO [5] 8m51s [4] 0.29s [3]	– TO [7] 9m52s [6] 1.19s [5]	– TO [7] 21.45s [6] 2.08s [5]	– TO [3] 19.58s [2]	– TO [3] 5m08s [2]	TO [N] TO [6] 1m16s [5] 4.08s [4]
sense [88] reversing barrier	S	0.06s [N]	– TO [3] 0.34s [2]	– TO [3] 0.09s [2]	– TO [5] 1m58s ☠ [4] 0.55s ☠ [3]	– TO [9] 12m25s [8] 3m34s [7]	– TO [4] 1m43s [3] 4.97s [2]	NT [N] TO [3] 0.09s [2]
arbiter v1 [81]	S	0.18s [N]	– TO [1+2]	– TO [1+2]	– KO [1+5] 4.57s [1+4] 0.54s [1+3]	– TO [1+6] 12m02s [1+5] 4m14s [1+4]	– TO [1+3] 44.3s [1+2]	NT [N] TO [1+6] 2m45s ☠ [1+5] 28.1s ☠ [1+4]
arbiter v2 [81]	S	13.5s [N]	– TO [1+2]	– TO [1+2]	– KO [1+4] 1.62s [1+3] 0.09s [1+2]	– TO [1+4] 2m56s [1+3] 5.84s [1+2]	– TO [1+2]	NT [N] TO [1+3] 24.2s [1+2]
two phase commit	S	54.1s [N]	– TO [2]	– TO [4] 39.7s [3] 0.31s [2]	– TO [4] 7.08s ☠ [3] 0.23s ☠ [2]	– TO [11] 12m39s [10] 5m47s [9]	– TO [11] 13m41s [10] 6m28s [9]	NT [N] TO [3] 12.3s [2]

TABLEAU 4.1 – Comparaison des performances de Cubicle-$\mathcal{W}$ et d'autres outils

nombre de processus - les traces incorrectes ne faisant en général intervenir que deux ou trois processus. Trencher n'étant quant à lui pas complet, il lui arrive d'indiquer qu'un algorithme n'est pas correct alors qu'il devrait l'être : il s'agit des algorithmes non-robustes mais corrects. Par ailleurs, Trencher a tendance à présenter des dysfonctionnement (plantages) sur les instances de grande taille, ce qui ne permet pas toujours de déterminer la plus grande instance vérifiable dans le temps imparti. Quant à Dual-TSO, il ne parvient pas à vérifier la version paramétrée du spinlock dans le temps imparti ; ce résultat est assez surprenant dans la mesure où il parvient à vérifier une version non-paramétrée de ce même algorithme jusqu'à 5 processus. De plus, il indique de façon erronée que l'algorithme de l'arbitre v1 est incorrect : en effet, l'analyse de la trace d'erreur remontée par Dual-TSO montre que cette trace est incorrecte et semble indiquer une erreur dans la gestion des lectures précoces intra-processus.

5

Traduction de programmes x86 vers Cubicle-$\mathcal{W}$

Sommaire

Dans les chapitres précédents, on a vu comment Cubicle-$\mathcal{W}$ permettait de vérifier des algorithmes paramétrés exprimés sous forme de systèmes de transitions. Certains algorithmes étaient donnés directement en langage d'assemblage, et ont dû être traduits manuellement vers le langage d'entrée de Cubicle-$\mathcal{W}$. Ce processus de traduction manuelle pouvant s'avérer long et être source d'erreur, on voudrait pouvoir l'automatiser. Ce chapitre présente donc un schéma de traduction des programmes écrits dans un sous-ensemble du langage d'assemblage x86 vers les systèmes à tableaux faibles de Cubicle-$\mathcal{W}$. On commence par présenter le fragment du langage d'assemblage x86 supporté, puis on détaille

notre schéma de traduction. On prouve la correction de cette traduction par simulation, qui montre que notre traduction préserve la sémantique x86-TSO. On illustre enfin cette traduction à l'aide d'exemples traduits automatiquement par un outil implémentant cette méthode.

5.1 Langage source

Dans cette section, on présente le langage source supporté par notre approche. Il s'agit d'un sous-ensemble du langage d'assemblage x86 32-bit, auquel on ajoute une gestion des processus légers ou *threads*. Afin de guider notre traduction vers Cubicle-$\mathcal{W}$ et de prouver sa correction, on donne également une sémantique opérationnelle de ce fragment.

5.1.1 Syntaxe

Les programmes sont écrits dans une syntaxe similaire à celle de NASM. On dispose des six registres généraux `eax`, `ebx`, `ecx`, `edx`, `esi` et `edi`. Les opérandes des instructions peuvent être des registres, des données immédiates, ainsi que des références mémoires de la forme `[var]`. Les accès mémoire se comportent suivant la sémantique du modèle TSO.

Les instructions supportées sont les suivantes :

— Chargement / rangement : `mov`
— Arithmétique : `add`, `sub`, `inc`, `dec`
— Echange : `xadd`, `xchg`, `cmpxchg`
— Comparaison : `cmp`
— Branchement : `jmp`, `jCC` (avec `CC` $\in$ { `e, ne, z, nz, l, le, g, ge, s, ns` })
— Ordre mémoire : `mfence`, préfixe `lock` (sur `add`, `sub`, `inc`, `dec`, `xadd`, `xchg`, `cmpxchg`)

Pour pouvoir écrire et traduire des programmes paramétrés utilisant des compteurs de threads (présentés en Section 3.2), les déclarations de données peuvent recevoir une annotation `! as counter`, qui indique que la variable déclarée doit se comporter comme un compteur de threads. Ces compteurs sont traités comme des entiers classiques sur x86, mais ne peuvent être manipulés que par les instructions `inc`, `dec`, `cmp` et `mov`. De plus, leur traduction vers Cubicle-$\mathcal{W}$ sera effectuée d'une façon spécifique.

Afin de simplifier la présentation du fragment supporté, on n'en présente dans cette section que les aspects les plus pertinents (cf. Figure 5.1). Une grammaire plus complète est donnée en Annexe A.1.

integer, n			entier
register, r			registre
variable, x			variable
label, l			étiquette d'instruction (index dans le tableau d'instructions)
thread_id, tid			identificateur de thread
instruction, i	::=		
	|	mov *r, n*	charge une constante dans un registre
	|	mov *r, x*	charge une variable dans un registre
	|	mov *x, n*	écrit une constante dans une variable
	|	mov *x, r*	écrit le contenu d'un registre dans une variable
	|	add *x, r*	ajoute le contenu d'un registre a une variable
	|	inc *x*	incrémente une variable de 1
	|	cmp *r, r'*	compare le contenu de deux registres
	|	cmp *x, n*	compare une variable à une constante
	|	je *l*	branchement si égal (ZF=1)
	|	jne *l*	branchement si différent (ZF=0)
	|	lock *i*	préfixe lock pour effectuer des instructions RMW atomiques
	|	mfence	barrière mémoire
action, a	::=		
	|	*i*	exécution d'une instruction
	|	ϵ	propagation en mémoire d'une écriture en attente dans le tampon
thread, t	::=		
	|	(*i array*)	tableau d'instructions
program, p	::=		
	|	(*tid* $\mapsto$ *t*) *map*	dictionnaire des identificateurs de threads vers leurs instructions

FIGURE 5.1 – Syntaxe abstraite du fragment x86 supporté

Dans la syntaxe abstraite, un *thread* est décrit par un tableau d'instructions, et un *programme* est un dictionnaire associant les identificateurs de threads au tableau d'instruction correspondant. Un thread exécute une *action*, qui est soit une instruction, soit la propagation en mémoire d'une écriture en attente dans son tampon.

5.1.2 Représentation des états x86-TSO

Pour donner la sémantique du fragment x86 choisi, on définit les états mémoires x86-TSO S. Ces états se composent de deux parties : l'ensemble des états locaux des différents threads LS, et la mémoire partagée M. Chaque état local d'un thread ls se compose de son *pointeur d'instruction* `eip`, son *zero flag* `zf`, l'ensemble de ses *registres* Q, et son tampon d'écriture B.

$S = (LS \times M)$	Un état machine x86-TSO
$M = (var \mapsto int)\ map$	La mémoire : dictionnaire des variables vers des entiers
$LS = (tid \mapsto ls)\ map$	Les états locaux des threads : dictionnaire des identificateurs de threads vers leurs états
$ls = (\texttt{eip} \times \texttt{zf} \times Q \times B)$	L'état local d'un thread
$Q = (reg \mapsto int)\ map$	Les registres d'un thread : dictionnaire des noms des registres vers des entiers
$B = (var \times int)\ queue$	Le tampon d'écriture TSO d'un thread
$\texttt{eip} = int$	Le pointeur d'instruction d'un thread
$\texttt{zf} = int$	Le *zero flag* d'un thread
var	L'ensemble de toutes les variables
tid	L'ensemble de tous les identificateurs de threads
reg	L'ensemble de tous les noms de registres

Le registre `eip` représente le *pointeur d'instructions* d'un thread, *i.e.* le point de programme courant d'un thread. Par simplicité, on choisit de le représenter à l'aide d'un type entier. Le *zero flag* `zf` est un registre booléen, utilisé pour stocker le résultat d'une instruction de comparaison, et plus généralement de toute instruction arithmétique. Il doit être positionné à *vrai* si la dernière instruction a produit comme résultat 0, et à *faux* dans le cas contraire. On le représente à l'aide d'un entier, avec la convention classique $0 = faux$ et $1 = vrai$. Les tampons d'écriture sont représentés par des files contenant des paires composées d'une variable et d'un entier. Initialement, la machine x86-TSO est dans un état S_{init} tel que les registres `eip` de tous les threads sont à 0, les tampons sont vides, et tous les registres ainsi que la mémoire partagée sont dans un état indéterminé.

5.1.3 Notations pour manipuler les tampons TSO

Afin de décrire la sémantique des instructions, on utilise les notations suivantes pour manipuler les tampons d'écriture :

$x \in B$	Vrai si au moins une paire dans B concerne la variable x
$x \notin B$	Vrai si aucune paire dans B ne concerne la variable x
$B = \varnothing$	Vrai si B est vide
$B_1 +\!\!+ B_2$	Concaténation de B_1 et B_2
$(x, n) +\!\!+ B$	Ajout de (x, n) en tête de B
$B +\!\!+ (x, n)$	Ajout de (x, n) en queue de B

5.1.4 Sémantique des programmes

La sémantique d'un programme est définie par la plus petite relation $\xrightarrow{tid:a}$ sur les états machine x86-TSO qui satisfait la règle SCHEDULING décrite ci-dessous.

On définit une fonction d'ordonnancement $\pi_p(S)$ qui, étant donné un programme p et un état S choisit la prochaine action devant être effectuée par un thread.

$$\frac{\pi_p(LS, M) = tid : a \quad LS(tid) = ls \quad (ls, M) \xrightarrow{a} (ls', M')}{(LS, M) \xrightarrow{tid:a} (LS[t \mapsto ls'], M')} \text{ SCHEDULING}$$

5.1.5 Sémantique des instructions

La sémantique des instructions est définie par la plus petite relation $\xrightarrow{i}$ qui satisfait les règles suivantes.

Il existe autant de règles que nécessaire pour couvrir les différentes combinaisons d'opérandes autorisées (constantes, registres, mémoire). Pour des raisons de lisibilité, on ne donne ici que quelques règles typiques qui mettent en œuvre la sémantique du modèle mémoire TSO.

La règle MOVVARCST affecte à une variable partagée x la valeur donnée par la constante n. Puisque sous TSO, les affectations sont retardées par l'effet des tampons, une nouvelle paire (x, n) est ajoutée au tampon B.

$$\frac{a = \mathsf{mov}\ x, n}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, zf, Q, (x, n) +\!\!+ B), M)} \text{ MOVVARCST}$$

Les deux règles suivantes donnent la sémantique d'une instruction `mov` r, x, qui affecte au registre r le contenu d'une variable partagée x. Lorsque le tampon du thread ne contient pas d'écriture sur x, la règle MOVREGVARM s'applique, et la valeur de x est lue directement depuis la mémoire.

$$\frac{a = \mathsf{mov}\ r, x \quad x \notin B \quad M(x) = n}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, zf, Q[r \mapsto n], B), M)} \text{ MOVREGVARM}$$

En revanche, si le tampon du thread contient une paire (x, n), c'est la règle MOVREGVARB qui s'applique, et la valeur de x est donnée par l'affectation la plus récente sur x dans B.

$$\frac{a = \mathsf{mov}\ r, x \quad B = B_1 +\!\!+ (x, n) +\!\!+ B_2 \quad x \notin B_1}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, zf, Q[r \mapsto n], B), M)} \text{ MOVREGVARB}$$

La sémantique des instructions de *lecture-modification-écriture* non atomiques telles que `add` est également donnée par des règles uniques. En effet, puisque l'écriture est placée dans un tampon, cela a le même effet que de placer la lecture et l'écriture dans deux règles distinctes.

$$\frac{a = \mathsf{add}\ x, r \quad x \notin B \quad M(x) + Q(r) = n}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, \mathsf{iszero}(n), Q, (x, n) +\!\!+ B), M)} \text{ ADDVARMREG}$$

$$\frac{a = \mathsf{add}\ x, r \quad B = B_1 +\!\!+ (x, m) +\!\!+ B_2 \quad x \notin B_1 \quad m + Q(r) = n}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, \mathsf{iszero}(n), Q, (x, n) +\!\!+ B), M)} \text{ ADDVARBREG}$$

Lorsque le préfixe *lock* est utilisé sur une instruction de *lecture-modification-écriture*, on

ajoute simplement la condition que le tampon du thread soit vide, et on effectue l'écriture directement en mémoire.

$$\frac{a = \mathsf{lock\ add}\ x, r \quad B = \varnothing \quad M(x) + Q(r) = n}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, \mathsf{iszero}(n), Q, B), M[x \mapsto n])}\ \textsc{LockAddVarReg}$$

Enfin, la règle MFence décrit l'effet d'une barrière mémoire, qui permet d'attendre que le tampon d'un thread soit vide avant de poursuivre l'exécution.

$$\frac{a = \mathsf{mfence} \quad B = \varnothing}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip + 1, zf, Q, B), M)}\ \textsc{MFence}$$

5.1.6 Synchronisation Tampon / Mémoire

Les tampons peuvent propager en mémoire l'écriture la plus ancienne qu'ils contiennent, et ce de façon asynchrone. On matérialise ce comportement à l'aide d'une règle qui n'implique que l'état des tampons, sans prendre en compte le registre *eip*.

$$\frac{a = \epsilon \quad B = B_1 +\!\!+ (x, n)}{((eip, zf, Q, B), M) \xrightarrow{a} ((eip, zf, Q, B_1), M[x \mapsto n])}\ \textsc{WriteMem}$$

5.2 Schéma de traduction

On fait correspondre la notion de thread du langage source à la notion de processus de Cubicle-$\mathcal{W}$. Les états x86-TSO sont représentés sous Cubicle-$\mathcal{W}$ par un ensemble de variables, correspondant aux variables partagées et aux états locaux de chaque thread. Les registres (ou variables locales) sont encodés comme étant les éléments d'un tableau indicé par des identificateurs de processus. Le type du tableau dépend du type des valeurs contenues dans les registres.

Pointeurs d'instructions. On les représente par un tableau `EIP`. Les points de programmes sont représentés par un type énuméré *loc*, dont les éléments sont de la forme L_0, ... L_n. Le nombre de ces éléments peut être déterminé statiquement à la compilation : il dépend de la longueur de la plus longue séquence d'instructions.

Zero flags. On les représente par un tableau `ZF` de type `int`. On utilise la convention $n = 0 \equiv true$ et $n \neq 0 \equiv false$, contrairement à ce que l'on a sous x86. De cette façon, on peut calculer ce flag plus facilement, en le positionnant à la valeur donnée par la dernière instruction arithmétique exécutée. Cela permet de réduire le nombre de transitions, puisqu'il n'y a besoin d'aucune autre opération pour effectuer ce calcul.

Variables partagées. Chaque variable partagée `X` donne lieu à une déclaration `weak X : int`. Les compteurs (voir ci-dessous) sont quant à eux exprimés à l'aide de tableaux déclarés `weak` et de type `bool`.

5.2.1 Traduction des instructions x86-TSO

Dans un souci de lisibilité, on ne présente dans cette section que la traduction des instructions présentées dans la section précédente.

On définit une fonction de compilation $\mathcal{C}$ qui prend en entrée un identificateur de thread, une instruction, et la position de l'instruction dans le tableau (cela correspond au *pointeur d'instructions*). Elle renvoie un ensemble de transitions Cubicle-$\mathcal{W}$ qui simule l'instruction. Toutes les transitions seront paramétrées par au moins un processus t, qui sera le processus effectuant les opérations.

La première règle, TMovVarCst, explique comment traduire l'écriture d'une constante dans une variable partagée. Il s'agit simplement d'utiliser l'opérateur d'affectation de Cubicle-$\mathcal{W}$ sur la variable en question.

$\mathcal{C}(\texttt{t}\ ;\ \mathsf{mov}\ x, n\ ;\ i)$ = `transition mov_var_cst`$_i$`([t])` TMovVarCst
`requires { EIP[t] =` $\texttt{L}_i$ `}`
`{ X := n; EIP[t] =` $\texttt{L}_{i+1}$ `}`

La règle suivante est l'opération inverse : le chargement du contenu d'une variable dans un registre. On utilise pour cela un simple accès à la variable.

$\mathcal{C}(t\ ;\ \mathsf{mov}\ r, x\ ;\ i)$ = `transition mov_reg_var`$_i$`([t])` TMovRegVar
`requires { EIP[t] =` $\texttt{L}_i$ `}`
`{ R[t] := X; EIP[t] =` $\texttt{L}_{i+1}$ `}`

L'ajout du contenu d'un registre à une variable est une opération de *lecture-modification-écriture*. Puisque Cubicle-$\mathcal{W}$ rend atomique toutes les opérations effectuées dans une seule et même transition, il faut utiliser deux transitions pour effectuer cette opération. La première, TAddVarReg1, lit la variable partagée `X`, l'ajoute au registre local, et stocke le résultat dans le registre temporaire `T[t]`. La seconde règle, TAddVarReg2, stocke le contenu de ce registre temporaire dans la variable `X` et met à jour le *zero flag* en conséquence.

$\mathcal{C}(t\ ;\ \mathsf{add}\ x, r\ ;\ i)$ = `transition add_var_reg_1`$_i$`([t])` TAddVarReg1
`requires { EIP[t] =` $\texttt{L}_i$ `}`
`{ T[t] := X + R[t]; EIP[t] =` $\texttt{L}_{xi}$ `}`

`transition add_var_reg_2`$_i$`([t])` TAddVarReg2
`requires { EIP[t] =` $\texttt{L}_{xi}$ `}`
`{ X := T[t]; ZF[t] := T[t];`
`EIP[t] =` $\texttt{L}_{i+1}$ `}`

Traduire la version atomique de cette opération est très simple, puisque les transitions de Cubicle-$\mathcal{W}$ sont atomiques. Il suffit d'une unique transition, comme indiqué dans la règle TLockAddVarReg.

$\mathcal{C}(t\ ;\ \mathsf{lock\ add}\ x, r\ ;\ i)$ = `transition lockadd_var_reg`$_i$`([t])` TLockAddVarReg
`requires { EIP[t] =` $\texttt{L}_i$ `}`
`{ X := X + R[t];`
`ZF[t] := X + R[t];`
`EIP[t] =` $\texttt{L}_{i+1}$ `}`

La traduction d'une barrière mémoire utilise simplement le prédicat `fence` de Cubicle-$\mathcal{W}$ pour exprimer le fait que la transition ne peut être prise que lorsque le tampon du thread est vide.

$$\mathcal{C}(t \ ; \ \mathsf{mfence} \ ; \ i) \quad = \quad \begin{array}{l} \texttt{transition mfence}_i\texttt{([t])} \\ \texttt{requires \{ EIP[t] = L}_i\texttt{ \&\& fence() \}} \\ \texttt{\{ EIP[t] = L}_{i+1}\texttt{ \}} \end{array} \qquad \text{TMFence}$$

5.2.2 Traduction des opérations sur les compteurs

Les opérations sur les compteurs sont sujettes à des restrictions, et sont traduites de façon spécifique. Lorsque la déclaration d'une variable `X` reçoit une annotation `! as counter`, cette variable ne peut plus être utilisée que dans les opérations suivantes :

`mov X, 0`	remise à zéro	`mov X,` N	remise à N
`inc X`	incrémentation	`dec X`	décrémentation
`cmp X,` N	comparaison à N	`cmp X, 0`	comparaison à 0

où N est une valeur abstraite représentant le nombre (paramétré) de threads.

Comme nous l'avons vu en Section 3.2, on ne peut traduire les compteurs directement comme des variables de type `int`, puisqu'aucun entier ne permet de représenter l'ensemble des processus (N). On représente donc les compteurs comme des tableaux faibles de booléens, indicés par des processus. Chaque opération est alors encodée dans un système numéral unaire. On présente à présent la traduction des trois premières instructions, la traduction des trois instructions suivantes étant symétrique.

Remise à 0. Pour remettre le compteur à 0, il suffit d'appliquer la transition donnée par la règle suivante, qui place la valeur `False` dans toutes les cases du tableau.

$$\mathcal{C}(t \ ; \ \mathsf{mov}\ x, 0 \ ; \ i) \quad = \quad \begin{array}{l} \texttt{transition mov\_cnt0}_i\texttt{([t])} \\ \texttt{requires \{ EIP[t] = L}_i\texttt{ \}} \\ \texttt{\{ X[k] := case | \_ : False;} \\ \texttt{\ \ EIP[t] = L}_{i+1}\texttt{ \}} \end{array} \qquad \text{TMovCnt0}$$

Incrémentation. L'incrémentation d'un compteur est quant à elle réalisée en deux étapes. La première consiste à lire le contenu du compteur, et la seconde à mettre à jour le compteur avec la valeur augmentée de 1. Dans notre système numéral unaire, ajouter 1 consiste à faire passer une case du tableau de `False` à `True`. Le but de la première transition est donc de trouver une case contenant `False`, tandis que la seconde transition effectue l'affectation à `True`. Les règles sont dupliquées, car la case en question peut être soit celle appartenant au processus effectuant les actions, soit à un autre processus.

$\mathcal{C}(t\ ;\ \mathsf{inc}\ x\ ;\ i)$ =

```
transition inc_cntS_1_i([t])                                   TIncCntS1
requires { EIP[t] = L_i && X[t] = False }
{ EIP[t] = L_xi }

transition inc_cntS_2_i([t])                                   TIncCntS2
requires { EIP[t] = L_xi }
{ X[t] := True; ZF[t] := 1; EIP[t] = L_i+1 }

transition inc_cntO_1_i([t] o)                                 TIncCntO1
requires { EIP[t] = L_i  && X[o] = False}
{ EIP[t] = L_yi; TP[t] = o }

transition inc_cntO_2_i([t] o)                                 TIncCntO2
requires { EIP[t] = L_yi && TP[t] = o}
{ X[o] := True; ZF[t] := 1; EIP[t] = L_i+1 }
```

Comparaison. On utilise trois transitions pour comparer la valeur d'un compteur au nombre (paramétré) de processus N. Pour vérifier si un compteur est égal à N, on vérifie que toutes les cases du tableau soient à `True`, en utilisant une variable de processus quantifiée universellement. Si c'est le cas, alors le compteur a atteint le nombre total de threads, et le *zero flag* est positionné à 0. Pour vérifier que le compteur n'est pas égal à N, on vérifie qu'il existe une case du tableau contenant la valeur `False`. Dans ce cas, le compteur n'a pas atteint le nombre total de threads, et le *zero flag* est positionné à 1. Il faut deux transitions pour effectuer cette dernière opération : une pour comparer la case appartement au processus effectuant les opérations, et une pour comparer la case d'un autre processus.

$\mathcal{C}(t\ ;\ \mathsf{cmp}\ x, N\ ;\ i)$ =

```
transition cmp_cnt_eqN_i([t])                                  TCmpCntEqN
requires { EIP[t] = L_i
   && X[t] = True
   && forall_other o. X[o] = True }
{ ZF[t] = 0; EIP[t] = L_xi }

transition cmp_cntS_NeqN_i([t])                                TCmpCntSNeqN
requires { EIP[t] = L_xi
   && X[t] = False }
{ ZF[t] := 1; EIP[t] = L_i+1 }

transition cmp_cntO_NeqN_i([t] o)                              TCmpCntONeqN
requires { EIP[t] = L_xi
   && X[o] = False }
{ ZF[t] := 1; EIP[t] = L_i+1 }
```

5.2.3 Traduction des programmes

Pour compiler toutes les instructions d'un thread, on définit une fonction de compilation $\mathcal{C}_t$, qui prend en entrée un identificateur de thread et un tableau d'instructions. Cette fonction renvoie l'ensemble des transitions Cubicle-$\mathcal{W}$ correspondant à la traduction de toutes les instructions du tableau.

$$\mathcal{C}_t(tid\ ;\ t) \quad = \quad \bigcup_{i=1}^{|t|} \mathcal{C}(tid\ ;\ t(i)\ ;\ i)$$

De la même façon, on définit une fonction de compilation $\mathcal{C}_p$, qui prend en entrée un programme x86 et renvoie l'ensemble des transitions correspondant à la traduction de toutes les instructions de tous les threads.

$$\mathcal{C}_p(p) \quad = \quad \bigcup_{tid\ \in\ dom(p)} \mathcal{C}_t(tid\ ;\ p(tid))$$

5.3 Correction

Pour prouver la correction de notre approche, on démontre un lemme de simulation entre les programmes x86 et les systèmes de transitions à tableaux faibles obtenus par la traduction.

Soit $S = (LS \times M)$ un état machine x86-TSO. Traduire S vers un état Cubicle-$\mathcal{W}$ A est immédiat, excepté pour la mémoire M et le contenu des tampons de chaque thread. Pour cela, on définit un prédicat $\mathcal{T}(S, A)$ sur les états Cubicle-$\mathcal{W}$ tel que $\mathcal{T}(S, A)$ est vrai si et seulement si :

— Les registres généraux, ainsi que *eip* et zf dans LS contiennent les mêmes valeurs que leur représentation sous forme de tableau dans A

— Pour chaque tampon B d'un thread `tid` et pour chaque variable partagée `X`

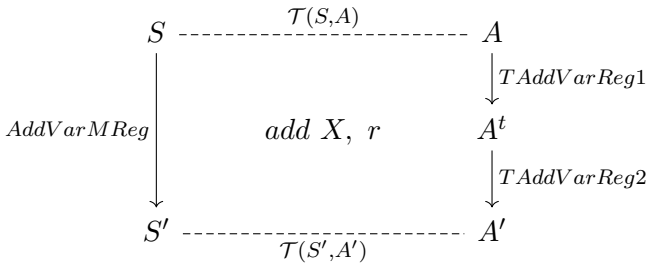
si ($\texttt{X} \notin B$ et $M(\texttt{X}) = v$) ou ($B = B_1 +\!\!+ (\texttt{X}, v) +\!\!+ B_2$ and $\texttt{X} \notin B_1$) alors

- si `X` est un compteur, alors `tid @ X[k]` = `True` est vrai pour v identificateurs de threads dans A
- sinon, `tid @ X` $= v$ est vrai dans A

Lemme 3 (Simulation)**.** *Pour tout programme p et tout état S, si $S \xrightarrow{tid:a} S'$ alors il existe un état Cubicle-$\mathcal{W}$ A tel que $\mathcal{T}(S, A)$ est vrai et $\mathcal{C}_p(p)$ peut avancer de A à A' et $\mathcal{T}(S', A')$ est également vrai.*

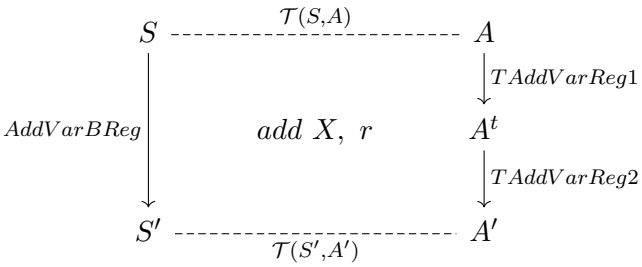
Démonstration. Par simple inspection de chaque règle de transition des instructions x86. □

Règle addition (avec lecture depuis la mémoire)



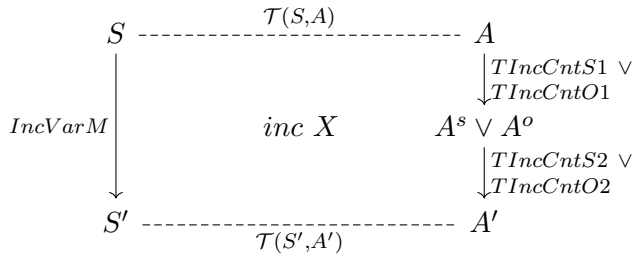
Soit t un thread et S un état x86-TSO de la forme (LS, M) avec $LS(t) = (eip, zf, Q, B)$ tel que $\texttt{X} \notin B$ et $M(\texttt{X}) = n$. D'après la règle AddVarMReg, le programme x86-TSO peut avancer vers un état S' de la forme (LS', M) avec $LS'(t) = (eip', zf', Q, B')$ tel que $B' = (\texttt{X}, n + Q(r)) +\!\!+ B$ et $\forall t' \neq t. LS'(t') = LS(t')$. Soit A un état tel que $\mathcal{T}(S, A)$ est *vrai*. En particulier, le registre `EIP[t]` dans A est équivalent à sa contrepartie dans S, et `t @ X` $= n$ est *vrai*. Alors la règle TAddVarReg1 s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A^t dans lequel `T[t]` $= n +$ `R[t]`. Depuis cet état, la règle TAddVarReg2 s'applique et effectue l'opération `t @ X := T[t]` ; le programme atteint alors un état A' tel que `t @ X` $= n +$ `R[t]` est *vrai* et donc $\mathcal{T}(S', A')$ est *vrai*.

Règle addition (avec lecture depuis un tampon)



Soit t un thread et S un état x86-TSO de la forme (LS, M) avec $LS(t) = (eip, zf, Q, B)$ tel que $B = B_1 +\!\!+ (\texttt{X}, n) +\!\!+ B_2$ et $\texttt{X} \notin B_1$. D'après la règle AddVarBReg, le programme x86-TSO peut avancer vers un état S' de la forme (LS', M) avec $LS'(t) = (eip', zf', Q, B')$ tel que $B' = (\texttt{X}, n + Q(r)) +\!\!+ B$ et $\forall t' \neq t. LS'(t') = LS(t')$. Soit A un état tel que $\mathcal{T}(S, A)$ est *vrai*. En particulier, le registre `EIP[t]` dans A est équivalent à sa contrepartie dans S, et `t @ X` $= n$ est *vrai*. Alors la règle TAddVarReg1 s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A^t dans lequel `T[t]` $= n +$ `R[t]`. Depuis cet état, la règle TAddVarReg2 s'applique et effectue l'opération `t @ X := T[t]` ; le programme atteint alors un état A' tel que `t @ X` $= n +$ `R[t]` est *vrai* et donc $\mathcal{T}(S', A')$ est *vrai*.

Règle incrémentation sur un compteur (avec lecture depuis la mémoire)



Soit t un thread et S un état x86-TSO de la forme (LS, M) avec $LS(t) = (eip, zf, Q, B)$ tel que $\texttt{X} \notin B$ et $M(\texttt{X}) = n$. D'après la règle IncVarM, le programme x86-TSO peut avancer vers un état S' de la forme (LS', M) avec $LS'(t) = (eip', zf', Q, B')$ tel que $B' = (\texttt{X}, n + 1) +\!\!+ B$ et $\forall t' \neq t. LS'(t') = LS(t')$. Soit A un état tel que $\mathcal{T}(S, A)$ est *vrai*. En

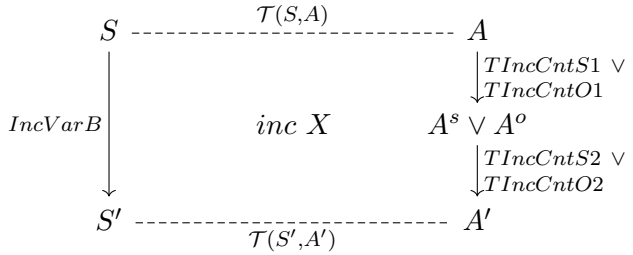
particulier, le registre `EIP[t]` dans A est équivalent à sa contrepartie dans S, et puisque `X` est un compteur, `t @ X[k] = True` est *vrai* pour exactement n threads. A présent, deux cas sont possibles :

Cas 1. `t @ X[t] = False` est *vrai*. Alors la règle TIncCntS1 s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A^s. Depuis cet état, la règle TIncCntS2 s'applique et effectue l'opération `t @ X[t] := True` ; le programme atteint alors un état A' tel que `t @ X[k] = True` est *vrai* pour exactement $n+1$ threads.

Cas 2. $\exists t' \neq t$. `t @ X[t'] = False` est *vrai*. Alors la règle TIncCntO1 s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A^o. Depuis cet état, la règle TIncCntO2 s'applique et effectue l'opération `t @ X[t'] := True` ; le programme atteint alors un état A' tel que `t @ X[k] = True` est *vrai* pour exactement $n+1$ threads.

Dans les deux cas, l'état A' est tel que $\mathcal{T}(S', A')$ est *vrai*.

Règle incrémentation sur un compteur (avec lecture depuis un tampon)



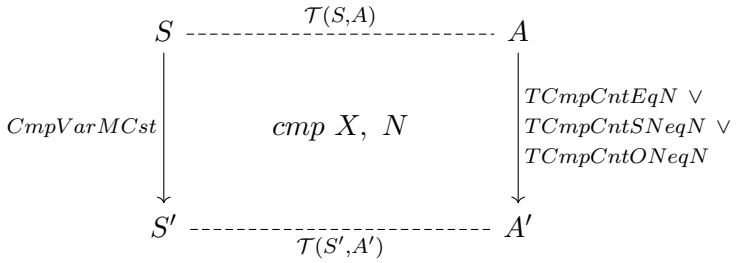
Soit t un thread et S un état x86-TSO de la forme (LS, M) avec $LS(t) = (eip, zf, Q, B)$ tel que $B = B_1 +\!\!+ (\mathtt{X}, n) +\!\!+ B_2$ et $\mathtt{X} \notin B_1$. D'après la règle IncVarB, le programme x86-TSO peut avancer vers un état S' de la forme (LS', M) avec $LS'(t) = (eip', zf', Q, B')$ tel que $B' = (\mathtt{X}, n+1) +\!\!+ B$ et $\forall t' \neq t. LS'(t') = LS(t')$. Soit A un état tel que $\mathcal{T}(S, A)$ est *vrai*. En particulier, le registre `EIP[t]` dans A est équivalent à sa contrepartie dans S, et puisque `X` est un compteur, `t @ X[k] = True` est *vrai* pour exactement n threads. A présent, deux cas sont possibles :

Cas 1. `t @ X[t] = False` est *vrai*. Alors la règle TIncCntS1 s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A^s. Depuis cet état, la règle TIncCntS2 s'applique et effectue l'opération `t @ X[t] := True` ; le programme atteint alors un état A' tel que `t @ X[k] = True` est *vrai* pour exactement $n+1$ threads.

Cas 2. $\exists t' \neq t$. `t @ X[t'] = False` est *vrai*. Alors la règle TIncCntO1 s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A^o. Depuis cet état, la règle TIncCntO2 s'applique et effectue l'opération `t @ X[t'] := True` ; le programme atteint alors un état A' tel que `t @ X[k] = True` est *vrai* pour exactement $n+1$ threads.

Dans les deux cas, l'état A' est tel que $\mathcal{T}(S', A')$ est *vrai*.

Règle comparaison sur un compteur (avec lecture depuis la mémoire)



Soit t un thread et S un état x86-TSO de la forme (LS, M) avec $LS(t) = (eip, zf, Q, B)$ tel que $\texttt{X} \notin B$ et $M(\texttt{X}) = n$. D'après la règle IncVarB, le programme x86-TSO peut avancer vers un état S' de la forme (LS', M) avec $LS'(t) = (eip', zf', Q, B')$ tel que $zf' = iszero(n - N)$ et $\forall t' \neq t. LS'(t') = LS(t')$. Soit A un état tel que $\mathcal{T}(S, A)$ est $vrai$. En particulier, le registre `EIP[t]` dans A est équivalent à sa contrepartie dans S, et puisque `X` est un compteur, `t @ X[k] = True` est $vrai$ pour exactement n threads. A présent, trois cas sont possibles :

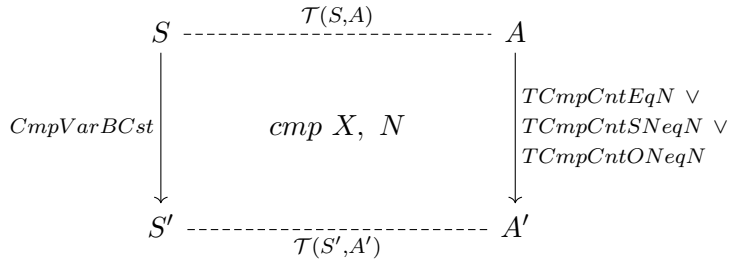
Cas 1. $\forall t'$. `t @ X[t'] = True` est $vrai$, *i.e.* $n = N$. Alors la règle TCmpCntEqN s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A' tel que `ZF[t] = 0`.

Cas 2. `t @ X[t] = False` est $vrai$, *i.e.* $n \neq N$. Alors la règle TCmpCntSNeqN s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A' tel que `ZF[t] = 1`.

Cas 3. $\exists t' \neq t$. `t @ X[t'] = False` est $vrai$, *i.e.* $n \neq N$. Alors la règle TCmpCntONeqN s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A' tel que `ZF[t] = 1`.

Dans les trois cas, l'état A' est tel que $\mathcal{T}(S', A')$ est $vrai$.

Règle comparaison sur un compteur (avec lecture depuis un tampon)



Soit t un thread et S un état x86-TSO de la forme (LS, M) avec $LS(t) = (eip, zf, Q, B)$ tel que $B = B_1 +\!\!+ (\texttt{X}, n) +\!\!+ B_2$ et $\texttt{X} \notin B_1$. D'après la règle CmpVarBCst, le programme x86-TSO peut avancer vers un état S' de la forme (LS', M) avec $LS'(t) = (eip', zf', Q, B)$ tel que $zf' = iszero(n - N)$ et $\forall t' \neq t. LS'(t') = LS(t')$. Soit A un état tel que $\mathcal{T}(S, A)$ est $vrai$. En particulier, le registre `EIP[t]` dans A est équivalent à sa contrepartie dans S, et puisque `X` est un compteur, `t @ X[k] = True` est $vrai$ pour exactement n threads. A présent, trois cas sont possibles :

Cas 1. $\forall t'$. `t @ X[t'] = True` est *vrai*, *i.e.* $n = N$. Alors la règle TCmpCntEqN s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A' tel que `ZF[t] = 0`.

Cas 2. `t @ X[t] = False` est *vrai*, *i.e.* $n \neq N$. Alors la règle TCmpCntSNeqN s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A' tel que `ZF[t] = 1`.

Cas 3. $\exists t' \neq t$. `t @ X[t'] = False` est *vrai*, *i.e.* $n \neq N$. Alors la règle TCmpCntONeqN s'applique, et le programme Cubicle-$\mathcal{W}$ atteint un état A' tel que `ZF[t] = 1`.

Dans les trois cas, l'état A' est tel que $\mathcal{T}(S', A')$ est *vrai*.

Théorème 2. *Etant donné un programme p, si Cubicle-$\mathcal{W}$ renvoie* safe *sur $\mathcal{C}_p(p)$ alors p ne peut pas atteindre d'état dangereux (tel que décrit dans la section* `unsafe_prop` *de p).*

Démonstration. Par induction sur la longueur de la réduction $p \xrightarrow{tid:a}{}^{+} \perp$ et par analyse de cas sur chaque étape (en utilisant le lemme de simulation). □

5.4 Exemples

5.4.1 Mutex avec `lock xchg`

Le premier exemple est une implémentation typique d'un mutex, tel qu'on pourrait le rencontrer dans une primitive système. La Figure 5.2 donne le code de cet algorithme : on utilise une variable partagée `Lock` valant initialement 0. Pour entrer en section critique, un processus va échanger de façon *atomique* le contenu de cette variable avec le contenu d'un registre initialisé à 1. Si suite à cet échange le registre contient la valeur 0, c'est qu'aucun autre processus avant lui n'a demandé à entrer en section critique : le processus peut donc y entrer. Au contraire, tant que la valeur récupérée lors de cet échange est 1, cela signifie que la section critique est déjà occupée par un processus. Pour quitter la section critique, le processus doit simplement remettre à 0 la variable `Lock`.

```
begin shared_data
    Lock  dd  0
end shared_data

begin unsafe_prop
    eip[$t1] = cs &&
    eip[$t2] = cs
end unsafe_prop

begin init_code
    start_threads
end init_code
```

```
begin thread_code
enter: mov eax, 1
       lock xchg dword [Lock], eax
       cmp eax, 0
       jne enter
cs:    ; critical section
exit:  mov dword [Lock], 0
       jmp enter
end thread_code
```

FIGURE 5.2 – Code assembleur x86 du mutex avec `lock xchg`

La traduction automatique de cet algorithme grâce à notre outil produit le code donnée en Figure 5.3. On observe bien que seule la variable `Lock` a été traduite par une

variable faible (`GLOB_Lock`). La traduction des instructions découle des règles données en Section 5.2. La transition la plus notable est `t_L2_L3_xchg`, qui traduit l'instruction `lock xchg dword [Lock], eax` : puisque le préfixe `Lock` est utilisé, une seule transition est générée, ce qui permet de rendre compte de l'atomicité de l'opération. Sans ce préfixe, la traduction se serait faite par deux transitions distinctes, une pour la lecture, et une pour l'écriture.

```
type loc = L1 | L2 | L3 | L4 | L5 | L6

array EIP[proc] : loc
array TP[proc] : proc
array EAX[proc] : int
array ZF[proc] : int
array TMP[proc] : int
weak var GLOB_Lock : int

init (p) { EIP[p] = L1 && GLOB_Lock = 0 }

unsafe (t1 t2) { EIP[t1] = L5 && EIP[t2] = L5 }

transition t_L1_L2_mov ([p])
requires { EIP[p] = L1 }
{ EAX[p] := 1; EIP[p] := L2 }

transition t_L2_L3_xchg ([p])
requires { EIP[p] = L2 }
{ GLOB_Lock := EAX[p];
  EAX[p] := GLOB_Lock; EIP[p] := L3 }

transition t_L3_L4_cmp ([p])
requires { EIP[p] = L3 }
{ ZF[p] := EAX[p] - 0; EIP[p] := L4 }

transition t_L4_L1_jne_tk ([p])
requires { EIP[p] = L4 && ZF[p] <> 0 }
{ EIP[p] := L1 }

transition t_L4_L5_jne_ntk ([p])
requires { EIP[p] = L4 && ZF[p] = 0 }
{ EIP[p] := L5 }

transition t_L5_L6_mov ([p])
requires { EIP[p] = L5 }
{ GLOB_Lock := 0; EIP[p] := L6 }

transition t_L6_L1_jmp ([p])
requires { EIP[p] = L6 }
{ EIP[p] := L1 }
```

FIGURE 5.3 – Code Cubicle-$\mathcal{W}$ du Mutex avec `lock xchg`

5.4.2 Spinlock Linux

Pour le second exemple, on reprend le Spinlock Linux présenté en Section 3.5.2, auquel on ajoute la déclaration de la variable `Lock` ainsi que la propriété de sûreté, comme indiqué

```
begin shared_data
    Lock  dd  1
end shared_data

begin unsafe_prop
    eip[$t1] = cs &&
    eip[$t2] = cs
end unsafe_prop

begin init_code
    start_threads
end init_code
```

```
begin thread_code
acquire: lock dec dword [Lock]
         jns cs
spin:    cmp dword [Lock], 0
         jle spin
         jmp acquire
cs:      ; critical section
exit:    mov dword [Lock], 1
         jmp acquire
end thread_code
```

FIGURE 5.4 – Code assembleur x86 du Spinlock Linux

en Figure 5.4. Le reste du code est identique.

Sa traduction automatique donne le code présenté en Figure 5.5. Comme dans l'exemple précédent, on note que l'instruction préfixée par `Lock` a donné lieu à une seule transition, `t_L1_L2_dec`. Les autres instructions ne présentent pas de caractéristiques particulières.

```
type loc = L1 | L2 | L3
         | L4 | L5 | L6 | L7

array EIP[proc] : loc
array TP[proc] : proc
array ZF[proc] : int
array TMP[proc] : int
weak var GLOB_Lock : int


init (p) {
  EIP[p] = L1 && GLOB_Lock = 1 }

unsafe (t1 t2) {
  EIP[t1] = L6 && EIP[t2] = L6 }


transition t_L1_L2_dec ([p])
requires { EIP[p] = L1 }
{ GLOB_Lock := GLOB_Lock - 1;
  ZF[p] := GLOB_Lock - 1;
  EIP[p] := L2 }

transition t_L2_L6_jns_tk ([p])
requires { EIP[p] = L2 && 0 <= ZF[p] }
{ EIP[p] := L6 }

transition t_L2_L3_jns_ntk ([p])
requires { EIP[p] = L2 && ZF[p] < 0 }
{ EIP[p] := L3 }

transition t_L3_L4_cmp ([p])
requires { EIP[p] = L3 }
{ ZF[p] := GLOB_Lock - 0; EIP[p] := L4 }

transition t_L4_L3_jle_tk ([p])
requires { EIP[p] = L4 && ZF[p] <= 0 }
{ EIP[p] := L3 }

transition t_L4_L5_jle_ntk ([p])
requires { EIP[p] = L4 && 0 < ZF[p] }
{ EIP[p] := L5 }

transition t_L5_L1_jmp ([p])
requires { EIP[p] = L5 }
{ EIP[p] := L1 }

transition t_L6_L7_mov ([p])
requires { EIP[p] = L6 }
{ GLOB_Lock := 1; EIP[p] := L7 }

transition t_L7_L1_jmp ([p])
requires { EIP[p] = L7 }
{ EIP[p] := L1 }
```

FIGURE 5.5 – Code Cubicle-$\mathcal{W}$ du Spinlock Linux

5.4.3 Sense-Reversing Barrier

Le troisième exemple reprend la Sense-Reversing Barrier présentée en Section 3.2.2, auquel on ajoute les déclarations des variables `sense` et `count`, et la propriété de sûreté, comme indiqué en Figure 5.6. La déclaration de la variable `count` est annotée avec `! as counter`, indiquant que cette variable sert de compteur de processus.

```
begin shared_data
    sense  dd  0
    count  dd  N ! as counter
end shared_data

begin unsafe_prop
    eip[$t1] = loop &&
    eip[$t2] = loop &&
    esi[$t1] <> esi[$t2]
end unsafe_prop

begin init_code
    start_threads
end init_code
```

```
begin thread_code
      mov esi, 0 ; esi = local sense
loop: not esi
      lock dec dword [count]
      jne spin
last: mov dword [count], N
      mov dword [sense], esi
      jmp loop
spin: cmp dword [sense], esi
      jne spin
      jmp loop
end thread_code
```

FIGURE 5.6 – Code assembleur x86 de la Barrière Sense-Reversing

La traduction automatique de cet algorithme réalisée par notre outil est présentée en Figure 5.7. On note que la variable `sense` a été traduite comme une variable faible de type entier, tandis que la variable `count`, représentant un compteur de processus, a été traduite par un tableau faible de booléens. La première instruction notable est l'instruction `lock dec dword [Count]` : il s'agit de la décrémentation atomique d'un compteur. Comme il s'agit d'une décrémentation, elle est susceptible de mettre à jour le *zero flag* selon le résultat de cette opération. Elle donne donc lieu à deux transitions : une pour gérer le cas où le résultat obtenu est égal à 0 (`t_L3_L4_decz`) et une pour le cas contraire (`t_L3_L4_decnz`). La seconde instruction notable est l'instruction `mov dword [Count], N` qui réinitialise le compteur à N. Dans sa traduction (`t_L5_L6_mov`), on se contente de repasser toutes les cases du tableau à `True`.

```
type loc = L1 | L2 | L3 | L4 | L5
         | L6 | L7 | L8 | L9 | L10

array ESI[proc] : int
array EIP[proc] : loc
array TP[proc] : proc
array ZF[proc] : int
array TMP[proc] : int
weak array GLOB_count[proc] : bool
weak var GLOB_sense : int

init (p) { EIP[p] = L1 &&
  GLOB_count[p] = True && GLOB_sense = 0 }

unsafe (t1 t2) { ESI[t1] <> ESI[t2] &&
  EIP[t1] = L2 && EIP[t2] = L2 }

transition t_L1_L2_mov ([p])
requires { EIP[p] = L1 }
{ ESI[p] := 0; EIP[p] := L2 }

transition t_L2_L3_not_z ([p])
requires { EIP[p] = L2 && ESI[p] = 0 }
{ ESI[p] := 1; EIP[p] := L3 }

transition t_L2_L3_not_nz ([p])
requires { EIP[p] = L2 && ESI[p] <> 0 }
{ ESI[p] := 0; EIP[p] := L3 }

transition t_L3_L4_decz ([p])
requires { EIP[p] = L3 &&
  GLOB_count[p] = True &&
  forall_other q. GLOB_count[q] = False }
{ GLOB_count[p] := False;
  ZF[p] := 0; EIP[p] := L4 }

transition t_L3_L4_decnz ([p] q)
requires { EIP[p] = L3 &&
  GLOB_count[p] = True &&
  GLOB_count[q] = True }
{ GLOB_count[p] := False;
  ZF[p] := 1; EIP[p] := L4 }

transition t_L4_L8_jne_tk ([p])
requires { EIP[p] = L4 && ZF[p] <> 0 }
{ EIP[p] := L8 }

transition t_L4_L5_jne_ntk ([p])
requires { EIP[p] = L4 && ZF[p] = 0 }
{ EIP[p] := L5 }

transition t_L5_L6_mov ([p])
requires { EIP[p] = L5 }
{ GLOB_count[q] := case | _ : True;
  EIP[p] := L6 }

transition t_L6_L7_mov ([p])
requires { EIP[p] = L6 }
{ GLOB_sense := ESI[p]; EIP[p] := L7 }

transition t_L7_L2_jmp ([p])
requires { EIP[p] = L7 }
{ EIP[p] := L2 }

transition t_L8_L9_cmp_eq ([p])
requires { EIP[p] = L8 &&
  GLOB_sense = ESI[p] }
{ ZF[p] := 0; EIP[p] := L9 }

transition t_L8_L9_cmp_ne ([p])
requires { EIP[p] = L8 &&
  GLOB_sense <> ESI[p] }
{ ZF[p] := 1; EIP[p] := L9 }

transition t_L9_L8_jne_tk ([p])
requires { EIP[p] = L9 && ZF[p] <> 0 }
{ EIP[p] := L8 }

transition t_L9_L10_jne_ntk ([p])
requires { EIP[p] = L9 && ZF[p] = 0 }
{ EIP[p] := L10 }

transition t_L10_L2_jmp ([p])
requires { EIP[p] = L10 }
{ EIP[p] := L2 }
```

FIGURE 5.7 – Code Cubicle-$\mathcal{W}$ de la Barrière Sense-Reversing

5.5 Résultats

Nous avons évalué les résultats produits par notre traduction automatique de programmes x86 vers Cubicle-$\mathcal{W}$. On donne pour chaque programme le nombre de registres, variables faibles et transitions générées. La colonne Long. CE indique la longueur du contre-exemple éventuel. Il s'agit du plus petit nombre de transitions qui mène à un état

Algorithme	Regs.	Vars. F.	Trans.	Long. CE	Temps
naive mutex (avec deadlock) (US)	3	2	11	12	0.38s
naive mutex (sans deadlock) (US)	3	2	14	12	0.85s
mutex avec *xchg* (US)	4	1	8	10	0.07s
mutex avec *xchg* (S)	3	1	7	-	0.08s
mutex avec *cmpxchg* (US)	4	1	10	10	0.12s
mutex avec *cmpxchg* (S)	4	1	8	-	0.47s
Linux spinlock (US)	4	1	10	6	0.06s
Linux spinlock (S)	4	1	9	-	0.30s
sense barrier (entrée simple) (S)	3	2	15	-	0.27s
sense barrier (entrées multiples) (S)	3	2	16	-	1m37s

TABLEAU 5.2 – Résultats obtenus sur des traductions automatiques

dangereux depuis l'état initial. La colonne Temps indique le temps mis par Cubicle-$\mathcal{W}$ pour prouver la propriété de sûreté (ou trouver un contre-exemple). On étudie aussi bien des programmes sûrs (S) que non-sûrs (US).

6

Conclusion et Perspectives

Dans cette thèse, nous avons présenté une extension du model checker Cubicle permettant la vérification de propriétés de sûreté de programmes concurrents paramétrés s'exécutant sur un modèle mémoire faible similaire à TSO. Ces travaux s'appuient sur une description axiomatique du modèle mémoire TSO, qui décrit les opérations de lecture et d'écriture par des événements, et utilise des relations sur ces événements pour déterminer si une exécution est ou non valide. Nos expériences ont montré que notre approche permettait une vérification efficace d'un certain nombre d'algorithmes concurrents paramétrés pour lesquels des outils non paramétrés sont rapidement limités par le nombre de processus impliqués.

Une possibilité d'évolution immédiate de notre approche consisterait à étendre Cubicle-$\mathcal{W}$ à d'autres modèles de mémoire faible, en modifiant la façon dont est construite la relation *ghb*. Par exemple, pour le modèle PSO, il faudrait exclure de *ghb* les paires d'écritures sur des variables différentes issues d'un même processus. Il faut également vérifier si les adaptations proposées en Section 4.2.2, notamment les simplifications relatives à la relation *co*, peuvent être appliquées dans ce contexte et les adapter si nécessaire.

Un autre prolongement intéressant de ce travail consisterait à implémenter une procédure d'inférence automatique de barrières mémoires en s'inspirant de ce qui a pu être fait dans les outils présentés en Chapitre 2. Lorsque Cubicle-$\mathcal{W}$ détecte qu'un programme viole une propriété de sûreté, il génère une trace d'exécution menant des état initiaux vers les états dangereux. Cette trace impliquant un nombre fini de processus et de variables, il est alors relativement simple de l'analyser pour déterminer à quel endroit placer des primitives de synchronisation. Une nouvelle analyse peut alors être lancée afin de vérifier s'il existe de nouvelles traces d'erreur. Ce processus d'analyse-correction peut être répété itérativement jusqu'à ce qu'il ne reste plus d'erreur (à condition que le programme soit correct en SC).

Enfin, on peut tenter d'utiliser une approche similaire sur des programmes communiquant par différentes formes de canaux de communication FIFO [57]. Les événements manipulés ne seraient plus des lectures et des écritures sur des variables, mais des envois et réceptions de messages sur des canaux. On construirait alors une relation *ghb* sur ces événements en fonction du type de canal utilisé (1-1, 1-n, n-1, n-n, *causal* ou *asynchrone*).

Dans ce qui suit, on propose une syntaxe pour manipuler ces canaux dans Cubicle. On effectue la déclaration d'un canal de la façon suivante :

```
chan ChanName[chantype] : type
```

`ChanName` est le nom du canal, `type` est le type des éléments contenus dans le canal, et `chantype` représente le type de canal parmi `1,1`, `1,N`, `N,1`, `N,N`, `CAUSAL` et `ASYNC`. Selon ce type, on aura donc un seul canal ou bien un ensemble de canaux : pour le type `N,N` par exemple, il s'agit d'un unique canal partagé par tous les processus, tandis que pour le type `1,1`, il y aura autant de canaux que de paires de processus pouvant communiquer deux à deux. Un envoi de message sera réalisé de la façon suivante dans une transition :

```
transition t_send ([p])
requires {...}
{ ChanName!msg }
```

Comme dans Cubicle-$\mathcal{W}$, l'un des paramètres de la transition, ici `p`, est marqué comme étant le processus effectuant les envois et réceptions. `ChanName` est le canal sur lequel on effectue l'envoi, et `msg` le message envoyé. N'importe quel processus peut reçevoir le message. On propose également de pouvoir restreindre dès l'envoi le processus pouvant recevoir le message, grâce à la notation (facultative) suivante :

```
transition t_send ([p] q)
requires {...}
{ ChanName'q!msg }
```

Ici, le message est envoyé à un processus `q` ; il suffit alors d'ajouter dans la garde des conditions sur `q` pour restreindre les destinataires possibles. Enfin, la réception de message s'effectue de la façon suivante :

```
transition t_recv ([p])
requires {...}
{ R[p] := ChanName? }
```

Ici, on effectue la réception sur le canal `ChanName`, et on stocke la valeur reçue dans `R[p]`. De façon analogue à l'envoi, on peut restreindre au niveau de la réception les processus depuis lesquels on s'autorise à effectuer la réception :

```
transition t_recv ([p] q)
requires {...}
{ R[p] := ChanName'q? }
```

Les messages reçus sont alors ceux émis par le processus `q`, que l'on peut contraindre en ajoutant des conditions supplémentaires dans la garde de la transition.

Le code présenté en Figure 6.1 illustre l'utilisation de tels canaux dans Cubicle avec le protocole Chandy Misra.

```
type state = Trying | In | Out

array State[proc] : state
array Request[proc,proc] : bool
array Expected[proc,proc] : bool
array Delayed[proc,proc] : bool
array Priority[proc,proc] : bool

chan NetReq[ASYNC] : bool
chan NetPerm[ASYNC] : bool

init(i j) { State[i] = Out &&
  Request[i,j] = False &&
  Delayed[i,j] = False &&
  Expected[i,j] = True &&
  Priority[i,j] = False }

unsafe(i j) { State[i] = In &&
              State[j] = In }

unsafe(i j) { State[i] = Trying &&
  State[j] = Trying &&
  Priority[i,j] = True &&
  Priority[j,i] = True }


transition request_mutex([i])
requires { State[i] = Out }
{ State[i] := Trying }

transition send_request([i] j)
requires { State[i] = Trying &&
  Request[i,j] = True }
{ Request[i,j] := False;
  NetReq'j!True }

transition acquire_mutex([i])
requires { State[i] = Trying &&
  forall_other j. Expected[i,j] = False }
{ State[i] := In;
  Priority[x,y] := case
     | x = i : True
     | _ : Priority[x,y]; }
```

```
transition release_mutex([i])
requires { State[i] = In }
{ State[i] := Out }

transition send_permission([i] j)
requires { State[i] = Out &&
  Delayed[i,j] = True }
{ Delayed[i,j] := False;
  Request[i,j] := True;
  Expected[i,j] := True;
  NetPerm'j!True;
  Priority[i,j] := False }

transition receive_permission([i] j)
requires { NetPerm'j? = True }
{ Expected[i,j] := False }

transition receive_request_in([i] j)
requires { State[i] = In &&
  NetReq'j? = True }
{ Delayed[i,j] := True }

transition receive_request_out([i] j)
requires { State[i]  = Out &&
  NetReq'j? = True }
{ Request[i,j] := True;
  Expected[i,j] := True;
  NetPerm'j!True;
  Priority[i,j] := False }

transition receive_request_trying1([i] j)
requires { State[i] = Trying &&
  NetReq'j? = True && Priority[i,j] = False }
{ Delayed[i,j] := True }

transition receive_request_trying2([i] j)
requires { State[i] = Trying &&
  NetReq'j? = True && Priority[i,j] = True }
{ NetPerm'j!True;
  NetReq'j!True;
  Request[i,j] := False;
  Expected[i,j] := True;
  Priority[i,j] := False }
```

FIGURE 6.1 – Une version abstraite du protocole Chandy Misra

A

Annexes

A.1 Grammaire du fragment x86 supporté

digit	::=	`0-9`
alpha	::=	`a-z A-Z`
ident	::=	(*alpha* | `_`) (*alpha* | `_` | *digit*)$\star$
integer	::=	*digit*+
label	::=	*ident* :
thread	::=	*$ident*
reg	::=	*eax* | *ebx* | *ecx* | *edx* | *esi* | *edi*
program	::=	*shareddata*? *threaddata*? *threadcode* *unsafeprop*
shareddata	::=	`begin` *shared_data NL* *dline*$\star$ `end` *shared_data NL*
threaddata	::=	`begin` *thread_data NL* *dline*$\star$ `end` *thread_data NL*
threadcode	::=	`begin` *thread_code NL* *cline*$\star$ `end` *thread_code NL*

unsafeprop ::= `begin` *unsafe_prop NL*
atom (*&&* *atom*)$\star$
`end` *unsafe_prop NL*

dline ::= *ident dd integer dannot*? *NL*
cline ::= *label*$\star$ *instr NL*

atom ::= *term op term*
op ::= = | <> | < | > | <= | >=
term ::= *treg* | *tlvar* | *tvar* | *ident*
treg ::= *reg* [*thread*)]
tlvar ::= *ident* [*thread*)]
tvar ::= *thread* : *ident* ([*thread*)])?

dannot ::= *!* *as counter*

instr ::= *inc oprm*
| *dec oprm*
| *not oprm*
| *add oprm* , *oprmi*
| *sub oprm* , *oprmi*
| *xchg oprm* , *oprm*
| *xadd oprm* , *opr*
| *cmp oprm* , *oprmi*
| *mov oprm* , *oprmi*
| *jmp ident*
| *jCC ident*
| *nop*
| *mfence*
| *lock instr*

opr ::= *reg*
oprm ::= *reg* | *mem*
oprmi ::= *reg* | *mem* | *imm*
mem ::= [*ident* (+ *thread*)?]
imm ::= *integer* | *ident*

Bibliographie

[1] *A formal specification of Intel Itanium processor family memory ordering*. Intel. 2002.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig et Ngo Tuan Phong. « The Best of Both Worlds : Trading Efficiency and Optimality in Fence Insertion for TSO ». In : *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, p. 308–332.

[3] Parosh Aziz Abdulla, Giorgio Delzanno et Ahmed Rezine. « Parameterized Verification of Infinite-State Processes with Global Conditions ». In : *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. 2007, p. 145–157.

[4] Parosh Aziz Abdulla et Bengt Jonsson. « Verifying Programs with Unreliable Channels ». In : *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*. 1993, p. 160–170.

[5] Parosh Aziz Abdulla et al. « A Load-Buffer Semantics for Total Store Ordering ». In : *Logical Methods in Computer Science* Volume 14, Issue 1 (jan. 2018).

[6] Parosh Aziz Abdulla et al. « Automatic Fence Insertion in Integer Programs via Predicate Abstraction ». In : *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*. 2012, p. 164–180.

[7] Parosh Aziz Abdulla et al. « Context-Bounded Analysis for POWER ». In : *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. 2017, p. 56–74.

[8] Parosh Aziz Abdulla et al. « Counter-Example Guided Fence Insertion under TSO ». In : *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, p. 204–219.

[9] Parosh Aziz Abdulla et al. « General Decidability Theorems for Infinite-State Systems ». In : *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. 1996, p. 313–321.

[10] Parosh Aziz Abdulla et al. « Memorax, a Precise and Sound Tool for Automatic Fence Insertion under TSO ». In : *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, p. 530–536.

[11] Parosh Aziz Abdulla et al. « Precise and Sound Automatic Fence Insertion Procedure under PSO ». In : *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*. 2015, p. 32–47.

[12] Parosh Aziz Abdulla et al. « Stateless Model Checking for TSO and PSO ». In : *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, p. 353–367.

[13] Parosh Aziz Abdulla et al. « The Benefits of Duality in Verifying Concurrent Programs under TSO ». In : *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*. 2016, 5 :1–5 :15.

[14] Allon Adir, Hagit Attiya et Gil Shurek. « Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture ». In : *IEEE Trans. Parallel Distrib. Syst.* 14.5 (2003), p. 502–515.

[15] Sarita V. Adve et Kourosh Gharachorloo. « Shared Memory Consistency Models : A Tutorial ». In : *IEEE Computer* 29.12 (1996), p. 66–76.

[16] Sarita V. Adve et Mark D. Hill. « Weak Ordering - A New Definition ». In : *Proceedings of the 17th Annual International Symposium on Computer Architecture. Seattle, WA, June 1990*. 1990, p. 2–14.

[17] Mustaque Ahamad et al. « Causal Memory : Definitions, Implementation, and Programming ». In : *Distributed Computing* 9.1 (1995), p. 37–49.

[18] Mustaque Ahamad et al. « The Power of Processor Consistency ». In : *SPAA*. 1993, p. 251–260.

[19] Francesco Alberti et al. « Automated Support for the Design and Validation of Fault Tolerant Parameterized Systems : a case study ». In : *ECEASST* 35 (2010).

[20] Jade Alglave. « A Shared Memory Poetics ». Thèse de doct. University of Paris 7 - Denis Diderot, Paris, France, 2010.

[21] Jade Alglave. « Simulation and Invariance for Weak Consistency ». In : *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. 2016, p. 3–22.

[22] Jade Alglave et Patrick Cousot. « Ogre and Pythia : an invariance proof method for weak consistency models ». In : *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 2017, p. 3–18.

[23] Jade Alglave, Daniel Kroening et Michael Tautschnig. « Partial Orders for Efficient Bounded Model Checking of Concurrent Software ». In : *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 2013, p. 141–157.

[24] Jade Alglave et Luc Maranget. « Stability in Weak Memory Models ». In : *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 2011, p. 50–66.

[25] Jade Alglave, Luc Maranget et Michael Tautschnig. « Herding Cats : Modelling, Simulation, Testing, and Data Mining for Weak Memory ». In : *ACM Trans. Program. Lang. Syst.* 36.2 (2014), 7 :1–7 :74.

[26] Jade Alglave et al. « 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*. 2014, p. 508–524.

[27] Jade Alglave et al. « Don't Sit on the Fence : A Static Analysis Approach to Automatic Fence Insertion ». In : *ACM Trans. Program. Lang. Syst.* 39.2 (2017), 6 :1–6 :38.

[28] Jade Alglave et al. « Fences in weak memory models (extended version) ». In : *Formal Methods in System Design* 40.2 (2012), p. 170–205.

[29] Jade Alglave et al. « Software Verification for Weak Memory via Program Transformation ». 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*. 2013, p. 512–532.

[30] Jade Alglave et al. « The semantics of power and ARM multiprocessor machine code ». In : *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009*. 2009, p. 13–24.

[31] Krzysztof R. Apt et Dexter Kozen. « Limits for Automatic Verification of Finite-State Concurrent Systems ». In : *Inf. Process. Lett.* 22.6 (1986), p. 307–309.

[32] Mohamed Faouzi Atig, Ahmed Bouajjani et Gennaro Parlato. « Context-Bounded Analysis of TSO Systems ». In : *From Programs to Systems. The Systems perspective in Computing - ETAPS Workshop, FPS 2014, in Honor of Joseph Sifakis, Grenoble, France, April 6, 2014. Proceedings*. 2014, p. 21–38.

[33] Mohamed Faouzi Atig, Ahmed Bouajjani et Gennaro Parlato. « Getting Rid of Store-Buffers in TSO Analysis ». In : *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 2011, p. 99–115.

[34] Mohamed Faouzi Atig et al. « On the verification problem for weak memory models ». In : *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 2010, p. 7–18.

[35] Mohamed Faouzi Atig et al. « What's Decidable about Weak Memory Models ? » In : *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, p. 26–46.

[36] Hagit Attiya et Roy Friedman. « Programming DEC-Alpha Based Multiprocessors the Easy Way (Extended Abstract) ». In : *SPAA*. 1994, p. 157–166.

[37] Christopher J. Banks et al. « Verification of a lazy cache coherence protocol against a weak memory model ». In : *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. 2017, p. 60–67.

[38] Mark Batty, Mike Dodds et Alexey Gotsman. « Library abstraction for C/C++ concurrency ». In : *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. 2013, p. 235–248.

[39] Mark Batty et al. « Clarifying and compiling C/C++ concurrency : from C++11 to POWER ». In : *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 2012, p. 509–520.

[40] Mark Batty et al. « 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*. 2011, p. 55–66.

[41] Giovanni Bernardi et Alexey Gotsman. « Robustness against Consistency Models with Atomic Visibility ». In : *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*. 2016, 7 :1–7 :15.

[42] Jasmin Christian Blanchette et al. « Nitpicking C++ concurrency ». In : *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*. 2011, p. 113–124.

[43] Hans-Juergen Boehm et Sarita V. Adve. « Foundations of the C++ concurrency memory model ». In : *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 2008, p. 68–78.

[44] Ahmed Bouajjani, Egor Derevenetc et Roland Meyer. « Checking and Enforcing Robustness against TSO ». 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*. 2013, p. 533–553.

[45] Ahmed Bouajjani, Egor Derevenetc et Roland Meyer. « Robustness against Relaxed Memory Models ». In : *Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25. Februar - 28. Februar 2014, Kiel, Deutschland*. 2014, p. 85–86.

[46] Ahmed Bouajjani, Roland Meyer et Eike Möhlmann. « Deciding Robustness against Total Store Ordering ». In : *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*. 2011, p. 428–440.

[47] Ahmed Bouajjani et al. « Lazy TSO Reachability ». In : *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, p. 267–282.

[48] Gérard Boudol et Gustavo Petri. « Relaxed memory models : an operational approach ». In : *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 2009, p. 392–403.

[49] Sebastian Burckhardt, Rajeev Alur et Milo M. K. Martin. « Bounded Model Checking of Concurrent Data Types on Relaxed Memory Models : A Case Study ». In : *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. 2006, p. 489–502.

[50] Sebastian Burckhardt, Rajeev Alur et Milo M. K. Martin. « CheckFence : checking consistency of concurrent data types on relaxed memory models ». In : *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 2007, p. 12–21.

[51] Sebastian Burckhardt et Madanlal Musuvathi. « Effective Program Verification for Relaxed Memory Models ». In : *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*. 2008, p. 107–120.

[52] Sebastian Burckhardt, Madanlal Musuvathi et Vasu Singh. « Verifying Local Transformations on Relaxed Memory Models ». In : *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. 2010, p. 104–123.

[53] Sebastian Burckhardt et al. « Concurrent Library Correctness on the TSO Memory Model ». In : *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, p. 87–107.

[54] Jacob Burnim, Koushik Sen et Christos Stergiou. « Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models ». In : *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. 2011, p. 11–25.

[55] Pietro Cenciarelli, Alexander Knapp et Eleonora Sibilio. « The Java Memory Model : Operationally, Denotationally, Axiomatically ». In : *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*. 2007, p. 331–346.

[56] Prosenjit Chatterjee et Ganesh Gopalakrishnan. « Towards A formal Model of Shared Memory Consistency for Intel Itanium$^{\text{TM}}$ ». In : *19th International Conference on Computer Design (ICCD 2001), VLSI in Computers and Processors, 23-26 September 2001, Austin, TX, USA, Proceedings*. 2001, p. 515–518.

[57] Florent Chevrou, Aurélie Hurault et Philippe Quéinnec. « On the diversity of asynchronous communication ». In : *Formal Asp. Comput.* 28.5 (2016), p. 847–879. doi : 10.1007/s00165-016-0379-x. url : https://doi.org/10.1007/s00165-016-0379-x.

[58] Nathan Chong et Samin Ishtiaq. « Reasoning about the ARM weakly consistent memory model ». In : *Proceedings of the 2008 ACM SIGPLAN workshop on Memory Systems Performance and Correctness : held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08), Seattle, Washington, USA, March 2, 2008*. 2008, p. 16–19.

[59] Edmund M. Clarke, Orna Grumberg et Michael C. Browne. « Reasoning About Networks With Many Identical Finite-State Processes ». In : *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing, Calgary, Alberta, Canada, August 11-13, 1986*. 1986, p. 240–248.

[60] Sylvain Conchon, David Declerck et Fatiha Zaïdi. « Compiling Parameterized X86-TSO Concurrent Programs to Cubicle-$\mathcal{W}$ ». In : *Formal Methods and Software Engineering - 19th International Conference on Formal Engineering Methods, ICFEM 2017, Xi'an, China, November 13-17, 2017, Proceedings*. 2017, p. 88–104.

[61] Sylvain Conchon, David Declerck et Fatiha Zaïdi. « Cubicle-$\mathcal{W}$: Parameterized Model Checking on Weak Memory ». In : *System Descriptions - 9th International Joint Conference, IJCAR 2018, Oxford, United Kingdom, July 14 - 17, 2018, Proceedings*. 2018.

[62] Sylvain Conchon, Alain Mebsout et Fatiha Zaïdi. « Certificates for Parameterized Model Checking ». In : *FM 2015 : Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*. 2015, p. 126–142.

[63] Sylvain Conchon, Alain Mebsout et Fatiha Zaïdi. « Vérification de systèmes paramétrés avec Cubicle ». In : *JFLA*. Aussois, France, fév. 2013.

[64] Sylvain Conchon et al. « 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*. 2012, p. 718–724.

[65] Sylvain Conchon et al. « Invariants for finite instances and beyond ». In : *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 2013, p. 61–68.

[66] *Cubicle-$\mathcal{W}$*. `https://www.lri.fr/~declerck/cubiclew/`.

[67] Andrei Marian Dan et al. « Effective Abstractions for Verification under Relaxed Memory Models ». In : *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*. 2015, p. 449–466.

[68] Egor Derevenetc. « Robustness against Relaxed Memory Models ». Thèse de doct. University of Kaiserslautern, 2015.

[69] Egor Derevenetc et Roland Meyer. « Robustness against Power is PSpace-complete ». In : *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*. 2014, p. 158–170.

[70] John Derrick et al. « A Proof Method for Linearizability on TSO Architectures ». In : *Provably Correct Systems*. 2017, p. 61–91.

[71] Michel Dubois, Christoph Scheurich et Faye A. Briggs. « Memory Access Buffering in Multiprocessors ». In : *Proceedings of the 13th Annual Symposium on Computer Architecture, Tokyo, Japan, June 1986*. 1986, p. 434–442.

[72] J. M. Stone F. Corella et C. M. Barton. *A formal specification of the PowerPC shared memory architecture. Technical Report RC18638*. IBM. 1993.

[73] Xing Fang, Jaejin Lee et Samuel P. Midkiff. « Automatic fence insertion for shared memory multiprocessing ». In : *Proceedings of the 17th Annual International Conference on Supercomputing, ICS 2003, San Francisco, CA, USA, June 23-26, 2003*. 2003, p. 285–294.

[74] Shaked Flur et al. « Mixed-size concurrency : ARM, POWER, C/C++11, and SC ». In : *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 2017.

[75] Shaked Flur et al. « Modelling the ARMv8 architecture, operationally : concurrency and ISA ». In : *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, p. 608–621.

[76] Florian Furbach et al. « Memory Model-Aware Testing - A Unified Complexity Analysis ». In : *14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, June 23-27, 2014*. 2014, p. 92–101.

[77] Steven M. German et A. Prasad Sistla. « Reasoning about Systems with Many Processes ». In : *J. ACM* 39.3 (1992), p. 675–735.

[78] Kourosh Gharachorloo et al. « Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors ». In : *Proceedings of the 17th Annual International Symposium on Computer Architecture. Seattle, WA, June 1990*. 1990, p. 15–26.

[79] Silvio Ghilardi et Silvio Ranise. « MCMT : A Model Checker Modulo Theories ». In : *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*. 2010, p. 22–29.

[80] Silvio Ghilardi et al. « Towards SMT Model Checking of Array-Based Systems ». In : *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*. 2008, p. 67–82.

[81] H. J. M. Goeman. « The arbiter : an active system component for implementing synchronizing primitives ». In : *Fundam. Inform.* (1981).

[82] James R. Goodman. *Cache consistency and sequential consistency*. Rapp. tech. Technical Report 61. University of Wisconsin-Madison, mar. 1989.

[83] Ganesh Gopalakrishnan, Yue Yang et Hemanthkumar Sivaraj. « QB or Not QB : An Efficient Execution Verification Tool for Memory Orderings ». In : *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. 2004, p. 401–413.

[84] Alexey Gotsman et Sebastian Burckhardt. « Consistency Models with Global Operation Sequencing and their Composition ». In : *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*. 2017, 23 :1–23 :16.

[85] Alexey Gotsman, Madanlal Musuvathi et Hongseok Yang. « Show No Weakness : Sequentially Consistent Specifications of TSO Libraries ». In : *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*. 2012, p. 31–45.

[86] Kathryn E. Gray et al. « An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors ». In : *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. 2015, p. 635–646.

[87] *herd7 tutorial*. `http://diy.inria.fr/doc/herd.html`.

[88] Maurice Herlihy et Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[89] Lisa Higham, LillAnne Jackson et Jalal Kawash. « Programmer-Centric Conditions for Itanium Memory Consistency ». In : *Distributed Computing and Networking, 8th International Conference, ICDCN 2006, Guwahati, India, December 27-30, 2006.* 2006, p. 58–69.

[90] Mark D. Hill. « Multiprocessors Should Support Simple Memory-Consistency Models ». In : *IEEE Computer* 31.8 (1998), p. 28–34.

[91] Alex Horn et Daniel Kroening. « On Partial Order Semantics for SAT/SMT-Based Symbolic Encodings of Weak Memory Concurrency ». In : *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*. 2015, p. 19–34.

[92] Phillip W. Hutto et Mustaque Ahamad. « Slow Memory : Weakening Consistency to Enhance Concurrency in Distributed Shared Memories ». In : *10th International Conference on Distributed Computing Systems (ICDCS 1990), May 28 - June 1, 1990, Paris, France*. 1990, p. 302–309.

[93] Thuan Quang Huynh et Abhik Roychoudhury. « A Memory Model Sensitive Checker for C# ». In : *FM 2006 : Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*. 2006, p. 476–491.

[94] IEEE et The Open Group. *POSIX Standard*. IEEE Std 1003.1-2017. 2017.

[95] *Intel 64 and IA-32 Architectures SDM*. Intel Corporation. Déc. 2016.

[96] ISO. *C++11 Standard*. ISO/IEC 14882 :2011. 2011.

[97] ISO. *C11 Standard*. ISO/IEC 9899 :2011. 2011.

[98] Huafeng Jin, Tuba Yavuz-Kahveci et Beverly A. Sanders. « Java Memory Model-Aware Model Checking ». In : *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, p. 220–236.

[99] Michalis Kokologiannakis et al. « Effective stateless model checking for C/C++ concurrency ». In : *PACMPL* 2.POPL (2018), 17 :1–17 :32.

[100] Michael Kuperstein, Martin T. Vechev et Eran Yahav. « Automatic inference of memory fences ». In : *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*. 2010, p. 111–119.

[101] Michael Kuperstein, Martin T. Vechev et Eran Yahav. « Partial-coherence abstractions for relaxed memory models ». In : *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 2011, p. 187–198.

[102] Leslie Lamport. « How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs ». In : *IEEE Trans. Computers* 28.9 (1979), p. 690–691.

[103] Jaejin Lee et David A. Padua. « Hiding Relaxed Memory Consistency with Compilers ». In : *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00), Philadelphia, Pennsylvania, USA, October 15-19, 2000*. 2000, p. 111–122.

[104] Alexander Linden et Pierre Wolper. « A Verification-Based Approach to Memory Fence Insertion in PSO Memory Systems ». In : *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, p. 339–353.

[105] Alexander Linden et Pierre Wolper. « A Verification-Based Approach to Memory Fence Insertion in Relaxed Memory Systems ». In : *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*. 2011, p. 144–160.

[106] Alexander Linden et Pierre Wolper. « An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models ». In : *Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings*. 2010, p. 212–226.

[107] Richard J. Lipton et Jonathan Sandberg. *PRAM : A Scalable Shared Memory*. Rapp. tech. CS-TR-180-88. Princetown University, sept. 1988.

[108] Feng Liu et al. « Dynamic synthesis for relaxed memory models ». In : *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 2012, p. 429–440.

[109] Sela Mador-Haim et al. « An Axiomatic Memory Model for POWER Multiprocessors ». In : *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 2012, p. 495–512.

[110] Jeremy Manson, William Pugh et Sarita V. Adve. « The Java memory model ». 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, p. 378–391.

[111] Alain Mebsout. « Inférence d'invariants pour le model checking de systèmes paramétrés. (Invariants inference for model checking of parameterized systems) ». Thèse de doct. University of Paris-Sud, Orsay, France, 2014.

[112] Robin Morisset et Francesco Zappa Nardelli. « Partially redundant fence elimination for x86, ARM, and power processors ». In : *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017*. 2017, p. 1–10.

[113] Madanlal Musuvathi et Shaz Qadeer. « Iterative context bounding for systematic testing of multithreaded programs ». In : *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 2007, p. 446–455.

[114] Kyndylan Nienhuis, Kayvan Memarian et Peter Sewell. « An operational semantics for C/C++11 concurrency ». In : *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 2016, p. 111–128.

[115] Scott Owens. « Reasoning about the Implementation of Concurrency Abstractions on x86-TSO ». In : *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*. 2010, p. 478–503.

[116] Scott Owens, Susmit Sarkar et Peter Sewell. « A Better x86 Memory Model : x86-TSO ». In : *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. 2009, p. 391–407.

[117] Seungjoon Park et David L. Dill. « An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order) ». In : *SPAA*. 1995, p. 34–41.

[118] *PMCx86*. `https://www.lri.fr/~declerck/pmcx86/`.

[119] Christopher Pulte et al. « Simplifying ARM concurrency : multicopy-atomic axiomatic and operational models for ARMv8 ». In : *PACMPL* 2.POPL (2018), 19 :1–19 :29.

[120] Susmit Sarkar et al. « Synchronising C/C++ and POWER ». In : *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 2012, p. 311–322.

[121] Susmit Sarkar et al. « The semantics of x86-CC multiprocessor machine code ». In : *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 2009, p. 379–391.

[122] Susmit Sarkar et al. « 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*. 2011, p. 175–186.

[123] Philippe Schnoebelen. « Verifying lossy channel systems has nonprimitive recursive complexity ». In : *Inf. Process. Lett.* 83.5 (2002), p. 251–261.

[124] Jaroslav Sevcík et al. « CompCertTSO : A Verified Compiler for Relaxed-Memory Concurrency ». In : *J. ACM* 60.3 (2013), 22 :1–22 :50.

[125] Jaroslav Sevcík et al. « Relaxed-memory concurrency and verified compilation ». In : *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 2011, p. 43–54.

[126] Peter Sewell et al. « x86-TSO : a rigorous and usable programmer's model for x86 multiprocessors ». In : *Commun. ACM* 53.7 (2010), p. 89–97.

[127] Dennis E. Shasha et Marc Snir. « Efficient and Correct Execution of Parallel Programs that Share Memory ». In : *ACM Trans. Program. Lang. Syst.* 10.2 (1988), p. 282–312.

[128] Zehra Sura et al. « Compiler techniques for high performance sequentially consistent java programs ». In : *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*. 2005, p. 2–13.

[129] Ermenegildo Tomasco et al. « Lazy sequentialization for TSO and PSO via shared memory abstractions ». In : *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*. 2016, p. 193–200.

[130] Ermenegildo Tomasco et al. « Using Shared Memory Abstractions to Design Eager Sequentializations for Weak Memory Models ». In : *Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings*. 2017, p. 185–202.

[131] Oleg Travkin et Heike Wehrheim. « Verification of Concurrent Programs on Weak Memory Models ». In : *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings*. 2016, p. 3–24.

[132] Viktor Vafeiadis. « Formal Reasoning about the C11 Weak Memory Model ». In : *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*. 2015, p. 1–2.

[133] Viktor Vafeiadis. « Program Verification Under Weak Memory Consistency Using Separation Logic ». In : *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. 2017, p. 30–46.

[134] Viktor Vafeiadis et Chinmay 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*. 2013, p. 867–884.

[135] Viktor Vafeiadis et Francesco Zappa Nardelli. « Verifying Fence Elimination Optimisations ». In : *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*. 2011, p. 146–162.

[136] Viktor Vafeiadis et al. « Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it ». In : *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 2015, p. 209–220.

[137] John Wickerson et al. « Automatically comparing memory consistency models ». In : *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 2017, p. 190–204.

[138] Yue Yang, Ganesh Gopalakrishnan et Gary Lindstrom. « UMM : an operational memory model specification framework with integrated model checking capability ». In : *Concurrency - Practice and Experience* 17.5-6 (2005), p. 465–487.

[139] Yue Yang et al. « Nemos : A Framework for Axiomatic and Executable Specifications of Memory Consistency Models ». In : *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*. 2004.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Vérification par Model Checking de Programmes Concurrents Paramétrés sur des Modèles Mémoires Faibles

Mots clefs : vérification, model checking, mémoire faible

Résumé : Les multiprocesseurs et microprocesseurs multicœurs modernes mettent en oeuvre des modèles mémoires dits *faibles* ou *relâchés*, dans lesquels l'ordre apparent des opérations mémoires ne suit pas la *cohérence séquentielle* (SC) proposée par Leslie Lamport. Tout programme concurrent s'exécutant sur une telle architecture et conçu avec un modèle SC en tête risque de montrer à l'exécution de nouveaux comportements, dont certains sont potentiellement des comportements incorrects. Par exemple, un algorithme d'exclusion mutuelle correct avec une sémantique par entrelacement pourrait ne plus garantir l'exclusion mutuelle lorsqu'il est mis en oeuvre sur une architecture plus relâchée.
Raisonner sur la sémantique de tels programmes s'avère très difficile. Par ailleurs, bon nombre d'algorithmes concurrents sont conçus pour fonctionner indépendamment du nombre de processus mis en oeuvre. On voudrait donc pouvoir s'assurer de la correction d'algorithmes concurrents, quel que soit le nombre de processus impliqués. Pour ce faire, on s'appuie sur le cadre du Model Checking Modulo Theories (MCMT), développé par *Ghilardi et Ranise*, qui permet la vérification de propriétés de sûreté de programmes concurrents *paramétrés*, c'est-à-dire mettant en oeuvre un nombre arbitraire de processus. On étend cette technologie avec une théorie permettant de raisonner sur des modèles mémoires faibles. Le résultat ce ces travaux est une extension du model checker Cubicle, appelée Cubicle-$\mathcal{W}$, permettant de vérifier des propriétés de systèmes de transitions paramétrés s'exécutant sur un modèle mémoire faible similaire à TSO.

Title : Verification via Model Checking of Parameterized Concurrent Programs on Weak Memory Models

Keywords : verification, model checking, weak memory

Abstract : Modern multiprocessors and microprocesseurs implement *weak* or *relaxed* memory models, in which the apparent order of memory operation does not follow the *sequential consistency* (SC) proposed by Leslie Lamport. Any concurrent program running on such architecture and designed with an SC model in mind may exhibit new behaviors during its execution, some of which may potentially be incorrect. For instance, a mutual exclusion algorithm, correct under an interleaving semantics, may no longer guarantee mutual exclusion when implemented on a weaker architecture.
Reasoning about the semantics of such programs is a difficult task. Moreover, most concurrent algorithms are designed for an arbitrary number of processus. We would like to ensure the correctness of concurrent algorithms, regardless of the number of processes involved. For this purpose, we rely on the Model Checking Modulo Theories (MCMT) framework, developed by *Ghilardi and Ranise*, which allows for the verification of safety properties of *parameterized* concurrent programs, that is to say, programs involving an arbitrary number of processes. We extend this technology with a theory for reasoning about weak memory models. The result of this work is an extension of the Cubicle model checker called Cubicle-$\mathcal{W}$, which allows the verification of safety properties of parameterized transition systems running under a weak memory model similar to TSO.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France