Méthode de conception de logiciel système critique couplée à une démarche de vérification formelle

Amira Methni

▶ To cite this version:

Amira Methni. Méthode de conception de logiciel système critique couplée à une démarche de vérification formelle. Logique en informatique [cs.LO]. Conservatoire national des arts et metiers - CNAM, 2016. Français. NNT : 2016CNAM1057 . tel-01445983

HAL Id: tel-01445983
https://theses.hal.science/tel-01445983v1

Submitted on 25 Jan 2017

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

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

CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS | le cnam

École Doctorale Informatique, Télécommunications et Électronique de Paris

Centre d'Étude et de Recherche en Informatique et Communications

THÈSE DE DOCTORAT

présentée par : **Amira METHNI**

soutenue le : **7 juillet 2016**

pour obtenir le grade de : **Docteur du Conservatoire National des Arts et Métiers**

Discipline/Spécialité : **Informatique**

Méthode de conception de logiciel système critique couplée à une démarche de vérification formelle

THÈSE DIRIGÉE PAR

M. Barkaoui Kamel	*Professeur, CNAM, Cédric*, Co-directeur
M. Haddad Serge	*Professeur, ENS Cachan, LSV*, Co-directeur
M. Ben Hedia Belgacem	*Ingénieur chercheur, CEA-List*, Co-Encadrant
M. Lemerre Matthieu	*Ingénieur chercheur, CEA-List*, Co-Encadrant

RAPPORTEURS

Mme. Petrucci Laure	*Professeur, Université Paris 13, LIPN*
M. Pommereau Franck	*Professeur, Université d'Évry, IBISC*

EXAMINATEURS

M. Kordon Fabrice	*Professeur, Université Paris 6, LIP6*
M. Ben Salem Saddek	*Professeur, Université Joseph Fourier, VERIMAG*

Remerciements

Ce travail de thèse a été mené conjointement au CEA-LIST, Laboratoire des composants logiciels pour la Sécurité et la Sûreté des Systèmes (L3S) et au Centre d'Études et de Recherche en Informatique et Communications (Cédric) du CNAM.

C'est avec un grand plaisir que je réserve cette page, en signe de gratitude et de reconnaissance à tous ceux qui ont contribué de près ou de loin à la réalisation de ce travail de thèse.

Je tiens à remercier Madame Laure PETRUCCI et Monsieur Franck POMMEREAU pour avoir aimablement accepté la lourde tâche de rapporter cette thèse. Je remercie également Monsieur Fabrice KORDON pour avoir accepté d'examiner et juger mon travail.

Je remercie vivement mes directeurs de thèse Serge HADDAD et Kamel BARKAOUI pour la patience dont ils ont fait preuve à mon égard au cours de ces quelques années, leurs relectures scrupuleuses du manuscrit et leurs suggestions toujours avisées.

Je remercie mes encadrants de thèse Matthieu LEMERRE et Belgacem BEN HEDIA pour leurs conseils, leurs critiques toujours pertinentes, leur patience, leurs encouragements et pour l'intérêt constant qu'ils m'ont manifesté tout au long de ma thèse.

Merci au CEA de m'avoir permis de travailler dans de bonnes conditions. Merci à toutes les personnes que j'ai côtoyées tout au long de cette thèse, notamment les membres du L3S du DACLE et l'équipe Vespa du Cnam.

Je n'oublie pas mes amis proches dont les encouragements soutenus m'ont permis de finaliser cette recherche. Ne voulant omettre aucun d'entre eux, je préfère ne citer personne. Ils se reconnaîtront.

Un grand merci à mes parents. C'est à vous que je dois tout ce que je suis.

Résumé

Avec l'évolution des technologies, la complexité des systèmes informatiques ne cesse de s'accroître. Parmi ces systèmes, on retrouve les logiciels critiques qui doivent offrir une garantie de sûreté de fonctionnement qui s'avère cruciale et pour lesquels un dysfonctionnement peut avoir des conséquences graves. Les méthodes formelles fournissent des outils permettant de garantir mathématiquement l'absence de certaines erreurs. Ces méthodes sont indispensables pour assurer les plus hauts niveaux de sûreté. Mais l'application de ces méthodes sur un code système bas niveau se heurte à des difficultés d'ordre pratique et théorique. Les principales difficultés concernent la prise en compte des aspects bas niveau, comme les pointeurs et les interactions avec le matériel spécifique. De plus, le fait que ces systèmes soient concurrents conduit à une augmentation exponentielle du nombre de comportements possibles, ce qui rend plus difficile leur vérification.

Dans cette thèse, nous proposons une méthodologie pour la spécification et la vérification par model-checking de ce type de systèmes, en particulier, ceux implémentés en C. Cette méthodologie est basée sur la traduction de la sémantique de C en TLA+, un langage de spécification formel adapté à la modélisation de systèmes concurrents. Nous avons proposé un modèle de mémoire et d'exécution d'un programme C séquentiel en TLA+. En se basant sur ce modèle, nous avons proposé un ensemble de règles de traduction d'un code C en TLA+ que nous avons implémenté dans un outil, appelé C2TLA+. Nous avons montré comment ce modèle peut s'étendre pour modéliser les programmes C concurrents et gérer la synchronisation entre plusieurs processus ainsi que leur ordonnancement. Pour réduire la complexité du model-checking, nous avons proposé une technique permettant de réduire significativement la complexité de la vérification. Cette réduction consiste pour un code C à agglomérer une suite d'instructions lors de la génération du code TLA+, sous réserve d'un ensemble de conditions.
Nous avons appliqué la méthodologie proposée dans cette thèse sur un cas d'étude réel issu de l'implémentation d'un micronoyau industriel sur lequel nous avons vérifié un ensemble de propriétés fonctionnelles. L'application de la réduction a permis de réduire considérablement le temps de la vérification, ce qui la rend utilisable en pratique. Les résultats ont permis d'étudier le comportement du système, de vérifier certaines propriétés et de trouver des bugs indétectables par des simples tests.

Mots clés : vérification formelle, programmes C, logique temporelle des actions (TLA), vérification de modèles, réduction.

Abstract

Software systems are critical and complex. In order to guarantee their correctness, the use of formal methods is important. These methods can be defined as mathematically based techniques, languages and tools for specifying and reasoning about systems. But, the application of formal methods to software systems, implemented in C, is challenging due to the presence of pointers, pointer arithmetic and interaction with hardware. Moreover, software systems are often concurrent, making the verification process infeasible.

This work provides a methodology to specify and verify C software systems using model-checking technique. The proposed methodology is based on translating the semantics of C into TLA+, a formal specification language for reasoning about concurrent and reactive systems. We define a memory and execution model for a sequential program and a set of translation rules from C to TLA+ that we developed in a tool called C2TLA+. Based on this model, we show that it can be extended to support concurrency, synchronization primitives and process scheduling. Although model-checking is an efficient and automatic technique, it faces the state explosion problem when the system becomes large. To overcome this problem, we propose a state-space reduction technique. The latter is based on agglomerating a set of C instructions during the generation phase of the TLA+ specification. This methodology has been applied to a concrete case study, a microkernel of an industrial real-time operating system, on which a set of functional properties has been verified. The application of the agglomeration technique to the case study shows the usefulness of the proposed technique in reducing the complexity of verification. The obtained results allow us to study the behavior of the system and to find errors undetectable using traditional testing techniques.

Keywords : formal verification, C programs, Temporal Logic of Actions (TLA), model-checking, state-space reduction.

Table des matières

Introduction **1**

I État de l'art **9**

1 État de l'art sur la vérification formelle **11**
1.1 Les méthodes formelles 12
1.1.1 Spécification formelle 12
1.1.2 Expression des propriétés 14
1.1.3 Vérification 16
1.2 Vérification formelle des programmes C 20
1.2.1 Analyse statique par interprétation abstraite 21
1.2.2 Techniques implémentant l'approche CEGAR 22
1.2.3 Techniques basées sur le *model-checking* borné 24
1.2.4 Vérification par *model-checking* symbolique ou explicite 26
1.2.5 Autres approches 27
1.3 Résumé et discussion 29
1.4 Conclusion 31

2 La logique TLA et le langage TLA+ **33**
2.1 La logique TLA 33
2.1.1 Notion de variable et d'état 34
2.1.2 Notion de fonction d'état et de prédicat 34
2.1.3 Notion d'action 34
2.1.4 Les formules temporelles 35
2.1.5 Quantification en TLA 38
2.1.6 Spécification TLA d'un système 38
2.1.7 Expression des propriétés dans TLA 41
2.2 Le langage TLA+ 42
2.2.1 Structure d'un module TLA+ 42
2.2.2 Définition des fonctions 44
2.2.3 Les enregistrements 44
2.2.4 Les séquences 45
2.2.5 Les opérateurs 45
2.2.6 Constructeurs conditionnels 46

2.2.7 Constructeur LET/IN 47
2.2.8 Les chaînes de caractères 47
2.2.9 Les expressions λ 47
2.3 Le *model-checker* TLC 47
2.3.1 Le sous-ensemble de TLA+ géré par TLC 48
2.3.2 Évaluation des expressions 49
2.3.3 Fonctionnement de TLC 51
2.3.4 Résultats fournis par TLC 52
2.4 Travaux autour de TLA+ 54
2.5 Conclusion 55

II Approche 57

3 Traduction de C vers TLA+ 59
3.1 Introduction 59
3.2 La bibliothèque CIL 60
3.2.1 Les transformations CIL 60
3.2.2 Sous-ensemble considéré 62
3.3 Principe de la traduction 64
3.3.1 Notations 65
3.3.2 Modèle mémoire pour C2TLA+ 66
3.3.3 Traduction d'un programme 70
3.3.4 Expressions, lvalues et offsets 72
3.3.5 *Lvalue*s et offsets 77
3.3.6 Flot de contrôle intra-procédural 79
3.3.7 Contrôle de flot inter-procédural 88
3.3.8 Définition de fonction et génération de la spécification 90
3.4 Conclusion 93

4 Traduction des programmes C concurrents vers TLA+ 95
4.1 Modèle d'exécution 96
4.1.1 Modèle d'exécution de base 96
4.1.2 Modèle d'exécution étendu 101
4.2 Modélisation des solutions de synchronisation 104
4.2.1 Section critique 105
4.2.2 Solution de synchronisation purement logicielle 105
4.2.3 Les solutions de synchronisation purement matérielles 106
4.2.4 Solution logicielle basée sur un mécanisme matériel 110
4.3 Temps-réel et ordonnancement 116
4.3.1 Évolution du temps 117
4.3.2 Modèle avec contraintes temporelles 117
4.3.3 Modèle d'ordonnancement temps-réel 118
4.4 Autres alternatives de traduction proposées 122
4.4.1 Première proposition 122

4.4.2 Deuxième proposition . . . 123
4.4.3 Troisième proposition . . . 123
4.5 Conclusion . . . 125

5 Une technique de réduction de l'espace d'états 127
5.1 État de l'art des techniques de réduction de l'espace d'états . . . 128
5.1.1 Technique par abstraction . . . 128
5.1.2 Techniques structurelles . . . 128
5.1.3 Techniques intermédiaires . . . 130
5.1.4 Techniques basées sur la représentation des espaces états . . . 131
5.1.5 Autres techniques . . . 133
5.1.6 Discussion . . . 133
5.2 Approche de réduction proposée . . . 134
5.2.1 Schéma général . . . 134
5.2.2 Principe de réduction . . . 134
5.2.3 Application de la réduction . . . 136
5.2.4 Préservation des propriétés . . . 147
5.3 Expérimentations et résultats . . . 148
5.4 Alternatives d'implémentation de la réduction . . . 150
5.4.1 Alternative erronée . . . 150
5.4.2 Optimisation de la réduction basée sur une analyse sémantique . . . 151
5.5 Conclusion . . . 154

III Expérimentation 155

6 Spécification et vérification de l'ordonnanceur du micronoyau PharOS 157
6.1 Travaux sur la vérification formelle des micronoyaux . . . 158
6.2 Description du micronoyau PharOS . . . 158
6.2.1 Structure de PharOS . . . 159
6.2.2 Description et modèle des tâches PharOs . . . 160
6.2.3 Gestion des interruptions dans PharOS . . . 163
6.3 Modélisation TLA+ du micronoyau mono-cœur . . . 169
6.3.1 Modèle d'exécution . . . 169
6.3.2 Spécification TLA+ du comportement du micronoyau . . . 171
6.3.3 Spécification des propriétés du micronoyau mono-cœur . . . 178
6.3.4 Vérification des propriétés . . . 182
6.4 Spécification et vérification de la version multi-cœur de PharOS . . . 185
6.4.1 Modèle d'exécution . . . 185
6.4.2 Spécification du comportement du micronoyau multi-cœur . . . 186
6.4.3 Spécification des propriétés du micronoyau multi-cœur . . . 191
6.4.4 Vérification des propriétés . . . 194
6.5 Conclusion . . . 196

Conclusion 199

Bibliographie **203**

Annexes **217**

Annexe A Traduction TLA+ d'un exemple de code C **219**

Annexe B Traduction des primitives `spinlock_acquire()` **et** `spinlock_release()` **223**

Table des figures

1 Méthodologie proposée de la thèse 3

1.1 Illustration des opérateurs LTL 15
1.2 Illustration des opérateurs CTL 15
1.3 Principe du *model-checking* 18
1.4 Le processus de vérification dans l'approche CEGAR 22

2.1 Exemple de deux séquences d'états équivalentes par bégaiement 36
2.2 Exemple de producteurs/consommateurs partageant un tampon borné . . 38
2.3 STE correspondant à la spécification *PCSpec* de l'exemple des producteurs/consommateurs 40
2.4 Initialisation de TLC pour une instance de l'exemple des producteurs/consommateurs 49
2.5 Résultats du *model-checking* de la spécification *PCSpec* 52

3.1 Organisation de la mémoire d'un exemple de code 66

4.1 Architecture simplifiée du modèle mémoire considéré 97
4.2 Organisation de la mémoire d'un exemple de code C concurrent 98

5.1 Exemple d'application d'une pré-agglomération d'un réseau de Petri . . . 129
5.2 Exemple d'application d'une post-agglomération d'un réseau de Petri . . 129
5.3 Processus de génération d'une spécification TLA+ réduite 134
5.4 Application d'une pré et post-agglomération 135

6.1 Structure générale de PharOS 159
6.2 Exécution d'une tâche PharOS 160
6.3 Spécification d'une tâche dans PharOS 161
6.4 Listes des tâches PharOS 162
6.5 Mise à jour d'un contexte de tâche suite à un appel système 164
6.6 Interruptions inter-processeurs dans PharOS (version multi-cœur) 165
6.7 Exemple de scénario d'exécution dans PharOS version monoprocesseur . 167
6.8 Exemple de scénario d'exécution dans PharOS version multi-cœur 168
6.9 Scénario ne satisfaisant pas la propriété *overrunDetection* 184

Liste des tableaux

1.1 Vue d'ensemble sur les outils de vérification formelle de programmes C . 29

2.1 Résumé des opérateurs TLA+ 43

5.1 Résultats expérimentaux du *model-checking* avant et après application de la réduction (temps en secondes) 149

6.1 Résultats du *model-checking* de la spécification TLA+ du micronoyau (version mono-coeur) 184

6.2 Résultats du *model-checking* de la spécification TLA+ du micronoyau (version multi-cœur) 195

Introduction

Contexte de la thèse et Objectifs

Les logiciels sont devenus omniprésents dans toutes les réalisations technologiques. Ils sont qualifiés de réactifs quand ils interagissent en permanence avec leur environnement. De nos jours, ces logiciels gèrent nos voitures, les métros et les avions. Ils se retrouvent dans tout ce qui nous entoure. La complexité de ces logiciels s'accentue de plus en plus. Leur lien étroit avec les objets du quotidien met en question la confiance qu'on peut leur accorder. Ces systèmes sont dits critiques quand leur dysfonctionnement met en péril la vie de l'être humain ou la mission pour laquelle ils sont développés. On peut imaginer une indication erronée de la quantité de kérosène dans un avion pouvant entraîner une catastrophe. Malheureusement, ces dysfonctionnements sont nombreux et l'histoire a connu des accidents graves qui ont touché plusieurs secteurs, à savoir l'aéronautique, l'automobile, la banque et la médecine. Citons notamment le lancement de la fusée Ariane 5 qui, en 1996, a abouti à un échec. La fusée a explosé quelques secondes après son décollage, à 3700 mètres d'altitude. Cette perte exorbitante, estimée à 8,5 milliards de dollars [Jea96], a été causée par une erreur de conception dans le logiciel de calcul de la position de la fusée pendant le décollage. Cette erreur est entraînée par une variable qui stocke la valeur de la vitesse horizontale de l'appareil, qui a subi un dépassement de sa capacité maximale provoquant ainsi des valeurs fausses de la vitesse. Récemment en novembre 2015, une panne informatique à l'aéroport d'Orly a provoqué l'interruption du trafic durant une demi-heure. La panne a été provoquée par une erreur dans le système de *Diffusion des données d'Environnement Contrôle d'Orly et de Roissy* (DECOR) qui délivre les informations métrologiques aux pilotes.

Un dysfonctionnement dans un système peut être causé par le non-respect de la spécification (le comportement souhaitable du système) ou par un comportement inattendu non défini dans sa spécification. Face au rôle crucial que peuvent jouer les logiciels dans notre vie et dans l'économie, un niveau de sûreté doit être garanti par ces systèmes. Des outils de validation comme la revue du code ou les tests permettent d'éviter de nombreuses erreurs et sont utilisés durant plusieurs phases de conception de logiciel. Ces techniques impliquent un coût faible, mais elles ne fournissent pas une

information complète du comportement du logiciel. Pour assurer une meilleure maîtrise de ces systèmes critiques, il est intéressant d'utiliser des outils basés sur les méthodes formelles. Ces méthodes sont basées sur des concepts mathématiques permettant à la fois de spécifier de manière non-ambiguë le comportement d'un système et de vérifier de manière rigoureuse son bon fonctionnement.

Parmi ces systèmes, nous nous intéressons aux logiciels systèmes qui sont des programmes servant d'intermédiaire entre les logiciels applicatifs et le matériel informatique. Pour ce type de systèmes, l'application des méthodes formelles soulève plusieurs enjeux d'ordre théorique et pratique. L'enjeu théorique concerne le formalisme dans lequel est exprimé le modèle formel. Un tel formalisme doit être assez expressif pour capturer les aspects d'un langage de programmation bas niveau et pour modéliser l'environnement du système. Il est aussi important que le formalisme de modélisation permette d'exprimer des propriétés dynamiques de ces systèmes et soit doté d'outils de vérification.

L'effort fourni pour la modélisation peut varier si le passage au modèle se fait automatiquement ou manuellement. D'un côté, il est clair que la modélisation manuelle demande un temps considérable et peut être sujette à des erreurs. Une modélisation automatique permet alors de simplifier la tâche du vérificateur et ainsi réduire le temps consacré à la modélisation et la vérification. D'un autre côté, les logiciels sont souvent parallèles ou concurrents, ils sont composés de plusieurs entités communiquant entre elles et avec l'environnement. Cette concurrence introduit l'indéterminisme et constitue une difficulté d'ordre pratique, car elle limite les capacités de vérification. Le choix de l'outil de vérification est crucial puisqu'il influence directement l'effort consacré à la modélisation ainsi que l'efficacité de vérification en terme de détection d'erreurs. Un autre point concerne l'extensibilité du système, à savoir la possibilité d'étendre ou d'adapter le comportement du système dans des contextes différents.

Pour ces raisons, il faut disposer de méthodes et d'outils nécessaires à une utilisation pratique de méthodes formelles pour la spécification et la vérification de ce type de système. Nous nous intéressons, en particulier, dans cette thèse aux systèmes logiciels implémentés en C. Le but de la thèse est de proposer une méthodologie qui permette de répondre aux questions suivantes :

- Comment être assez expressif pour prendre en compte la nature de ces logiciels systèmes, qui sont réactifs, concurrents et implémentés en C ?
- Quel formalisme utiliser afin d'exprimer les propriétés du système d'une part et afin de rendre le modèle extensible pour gérer l'évolution du système d'autre part ?
- Comment réduire l'effort de la modélisation formelle ainsi que de la vérification ?

La méthodologie doit être validée en étudiant et en vérifiant formellement un cas d'étude réel issu d'un code système développé au sein du laboratoire et qui a été transféré à l'industrie.

CONTRIBUTIONS DE LA THÈSE

La figure 1 illustre la méthodologie que nous proposons ainsi que la démarche de la thèse. Nous proposons dans ce travail une méthodologie basée sur la traduction de la sémantique du langage C en une sémantique opérationnelle exprimée dans une logique formelle. Notre choix s'est porté sur la logique TLA (« *Temporal Logic of Actions* ») et son langage associé TLA+ [Lam94] dans lequel la traduction du code C est exprimée. TLA+ gère la concurrence et possède une boîte à outils complète incluant le *model-checker* TLC (« *Temporal Logic Checker* »).

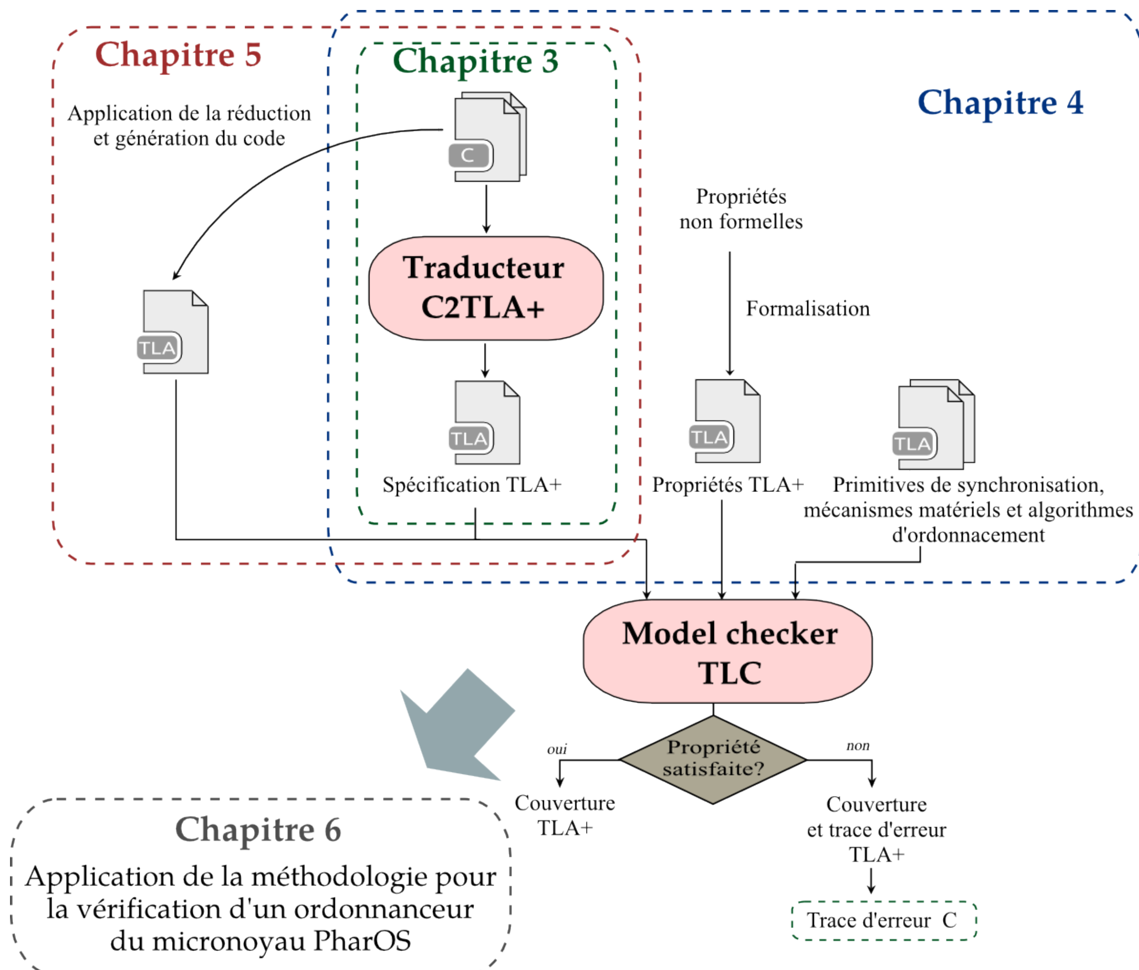


FIGURE 1 – Méthodologie proposée de la thèse

Dans un premier temps, nous avons défini un modèle mémoire et d'exécution permettant de modéliser les programmes C séquentiels. En se basant sur ce modèle, nous avons proposé un ensemble de règles de traduction d'un code C en TLA+ que nous avons implémenté dans un outil, appelé C2TLA+. Ce dernier prend en entrée un ou plusieurs fichiers C et les traduit en une spécification TLA+ vérifiable par les outils de vérification.

Nous avons dans un second temps étendu ce modèle d'exécution afin de gérer les programmes concurrents et les aspects liés à la synchronisation et la gestion des processus et leur ordonnancement. Pour cela, nous avons proposé une bibliothèque TLA+, facilement extensible, pour la gestion de la synchronisation entre les processus. Cette bibliothèque contient les définitions des primitives de synchronisation matérielles et logicielles. Nous avons proposé par la suite un modèle étendu permettant la spécification des processus avec des contraintes temps-réel. Pour cela, nous avons proposé un modèle permettant la spécification du temps et son évolution. Nous avons montré à travers des exemples comment C2TLA+ peut être utilisé pour spécifier des processus temps-réel avec différentes politiques d'ordonnancement.

Une fois le modèle formel établi, une phase de vérification est conduite. Mais l'utilisation du *model-checking* se heurte au problème de l'explosion combinatoire de l'espace d'états. Ce problème est dû à la taille des systèmes à vérifier, surtout quand ils sont concurrents. Face à ce constat, nous proposons une technique permettant de réduire significativement la complexité de la vérification. La réduction que nous proposons consiste pour un code C à agglomérer une suite d'instructions lors de la génération du code TLA+, sous réserve d'un ensemble de conditions. Ces conditions caractérisant ces agglomérations définissent ce que l'on appelle un *prédicat d'agglomération*. Ce prédicat détermine si une instruction C est agglomérable ou pas. En se basant sur le résultat retourné par ce dernier, C2TLA+ génère une spécification TLA+ réduite. Nous avons montré sur un ensemble d'expérimentations que la technique de réduction permet de réduire le nombre d'états d'une manière exponentielle.

Nous avons validé notre approche en l'appliquant sur un cas d'étude réel issu de l'implémentation du micronoyau PharOS [LOC+11], développé au sein du CEA-List et transféré en industrie. PharOS est un micronoyau temps-réel qui dispose de deux implémentations, une version monoprocesseur et une deuxième version s'exécutant sur un processeur multi-cœur. Nous proposons dans cette thèse une modélisation TLA+ de ces deux implémentations en utilisant C2TLA+ pour traduire les programmes C en TLA+. Les spécifications TLA+ générées par C2TLA+ sont, par la suite, intégrées avec des modules TLA+ spécifiés manuellement, qui modélisent les aspects matériels comme la gestion des interruptions et les timers. Nous avons ensuite appliqué la technique d'agglomération d'actions afin de réduire l'espace d'états de la spécification TLA+ du micronoyau. Nous avons comparé le nombre d'états et le temps de vérification de la spécification avant et après réduction. Les résultats obtenus montrent que l'application de la réduction a permis de réduire significativement la complexité de la vérification. En se basant sur cette modélisation TLA+ du micronoyau, nous avons spécifié et vérifié un ensemble de propriétés fonctionnelles sur le micronoyau. Il est à noter que la vérification des propriétés temps-réels n'est pas l'objectif de notre étude. Les propriétés que nous vérifions sur ce cas d'étude portent sur le fonctionnement de l'implémentation du micronoyau et non pas sur le respect des contraintes temps-réel des tâches de l'application. Néanmoins, l'adaptation du modèle proposé dans ce chapitre pour prendre en considération la vérification des propriétés dites, temps-réel, est possible et pourrait faire l'objet d'un travail complémentaire.

PLAN DE LA THÈSE

Ce manuscrit s'organise en trois parties. Nous exposons dans ce qui suit le plan de chacune de ces parties.

La partie I dresse un état de l'art du domaine de recherche connexe à la thèse. Elle est composée de deux chapitres.
Le premier chapitre est intitulé **Spécification et vérification formelle**. Il décrit dans sa première section, un panorama des formalismes utilisés dans la phase de spécification et de vérification formelle de manière générale. Sa deuxième section est consacrée à la présentation des travaux connexes dans la vérification des programmes C. Enfin, le chapitre se termine par une discussion relative au travail proposé dans cette thèse.

Le chapitre 2, intitulé **la logique TLA et le langage TLA+**, expose le cadre logique sur lequel est basé notre approche. Dans ce chapitre, nous exposons la syntaxe et la sémantique de cette logique. Nous présentons son langage associé TLA+ et nous introduisons les constructeurs que nous utilisons dans les prochains chapitres. La dernière section de ce chapitre décrit le *model-checker* TLC, qui fait l'objet de notre méthodologie. Nous détaillons le principe de son fonctionnement et ses spécificités.

Il est à noter qu'un état de l'art de la réduction de l'espace d'états ainsi que sur la vérification des logiciels systèmes sont placés respectivement au début des chapitres 5 et 6.

La partie II présente la méthodologie proposée. Elle est composée de trois chapitres.
Dans le chapitre 3, **Traduction de C vers TLA+**, nous commençons par introduire le sous-ensemble de C considéré. Nous définissons par la suite un modèle d'exécution pour un programme C séquentiel. En se basant sur ce modèle, nous traduisons la sémantique de ce sous-ensemble en TLA+ à travers un ensemble des règles de traduction. Le modèle généré peut être ensuite vérifié automatiquement par TLC.

Le chapitre 4, qui s'intitule **Traduction des programmes C concurrents vers TLA+**, est un prolongement du chapitre précédent. Il aborde comment le modèle d'exécution proposé dans le chapitre 3 peut s'étendre pour modéliser les programmes C concurrents. Nous décrivons ensuite comment modéliser et gérer la synchronisation entre plusieurs processus ainsi que leur ordonnancement.

Dans le chapitre 5, **Une technique de réduction de l'espace d'états**, nous proposons une technique pour réduire la complexité de la phase de vérification. Pour commencer, nous retraçons un état de l'art sur les techniques de réduction existantes. Nous présentons ensuite une technique de réduction qui permet de générer une spécification TLA+ réduite. En dernier lieu, nous décrivons les résultats obtenus sur un ensemble de benchmarks.

La partie III comprend le chapitre 6, **Spécification et vérification de l'ordonnanceur du micronoyau PharOS**. Ce chapitre porte sur la spécification et la vérification d'un cas d'étude réel issu de l'implémentation d'un micronoyau, appelé PharOS [LOC^{+}11], qui assure l'ordonnancement de tâches temps-réel. Dans ce chapitre, nous montrons comment adapter le modèle de vérification que nous avons proposé dans les chapitres précédents afin de spécifier le comportement de l'ordonnanceur ainsi que son environnement, à savoir les mécanismes matériels et la gestion des interruptions. En ce qui concerne la vérification, nous nous intéressons en particulier aux propriétés fonctionnelles de ce dernier. Enfin, nous discutons les résultats de vérification obtenus avant et après l'application de la technique de réduction proposée dans le chapitre 5.

Nous achevons la thèse par une conclusion et des perspectives. Dans cette partie nous établissons un bilan général sur la méthodologie proposée et nous proposons un ensemble de perspectives, donnant des pistes de recherches qui visent à enrichir et améliorer le présent travail.

LISTE DES PUBLICATIONS

Notre travail a fait l'objet d'un ensemble de communications. Elles sont classées par catégories.

Conférences internationales avec actes

[MLH+14c] Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Serge Haddad, and Kamel Barakaoui. An approach for verifying concurrent C programs. pages 33–36, Versailles, France, Août 2014. 8th Junior Researcher Workshop on Real-Time Computing.

[MLBH+15] Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Serge Haddad, and Kamel Barkaoui. Specifying and Verifying Concurrent C Programs with TLA+. In Cyrille Artho and Peter Csaba Ölveczky, editors, Formal Techniques for Safety-Critical Systems, volume 476 of Communications in Computer and Information Science, pages 206–222. Springer International Publishing, 2015.

[AMB+15] Methni Amira, Lemerre Matthieu, Ben Hedia Belgacem, Haddad Serge, and Barkaoui Kamel. State Space Reduction Strategie for Model Checking Concurrent C Programs. In Belgacem Ben Hedia and Florin Popentiu Vladicescu, editors, Proceedings of the 9th Workshop on Verification and Evaluation of Computer and Communication Systems(VECoS'15), volume 1431 of CEUR Workshop Proceedings, pages 65–76, Bucharest, Romania, September 2015. RWTH Aachen, Germany.

[MLH+ 15b] Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Serge Haddad,

and Kamel Barkaoui. Verifying and Constructing Abstract TLA Specifications : Application to the Verification of C programs. pages 32–39, Barcelone, Spain, 2015. 10th International Conference on Software Engineering Advances (ICSEA).

Communications sans actes

[MLH+14a] Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Kamel Barakaoui, and Serge Haddad. A translator from C to TLA+. Toulouse, Juin 2014. TLA workshop.

[MLH+14b] Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Kamel Barakaoui, and Serge Haddad. Spécification et vérification des programmes C concurrents. Groupe de travail GT-Verif, Juin 2014.

[MLH+15a] Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Kamel Barakaoui, and Serge Haddad. Méthode de conception de logiciel système critique couplée à une démarche de vérification formelle. ACtion Temps-Réel : Infrastructures et Services Systèmes (ACTRISS), 2015.

Première partie

État de l'art

1

État de l'art sur la vérification formelle

Avec l'essor des nouvelles technologies, les systèmes informatiques sont présents partout et ne cessent de s'accroître. Ils deviennent aussi de plus en plus complexes. Face à la taille et à la complexité de ces systèmes, il convient alors de mettre en œuvre des techniques afin de garantir un système sûr. On parle alors de deux concepts : la vérification et la validation. La vérification est la méthode qui permet d'assurer la conformité du système par rapport à sa spécification. Autrement dit, elle permet de répondre à la question : *est-ce que le système fonctionne correctement ?* De l'autre côté, la validation permet de s'assurer que le système répond aux besoins de l'utilisateur. Elle consiste à répondre à la question *est-ce que nous avons bien construit le système ?*

Le test logiciel est une méthode réputée dans le milieu industriel pour examiner un logiciel afin d'y trouver des bugs. Elle constitue une activité à part entière dans le cycle de développement du logiciel. Néanmoins, avec cette approche, seules les erreurs couvertes par les scénarios considérés sont détectées. Il s'avère alors impossible de tester tout le comportement du système. Cette insuffisance a été énoncée par l'informaticien Edsger W. Dijkstra dans sa fameuse citation [Dij70] : « Les tests peuvent servir à montrer la présence d'erreurs, mais jamais à garantir leur absence. »

Une approche beaucoup plus ambitieuse est celle des méthodes formelles. Elle est apparue durant les années 60 et depuis elle a considérablement évolué. De nos jours, elle s'impose de plus en plus durant le développement des composants critiques dans des domaines bien particuliers, par exemple dans l'industrie du transport de passagers.

Ce chapitre est consacré à l'étude bibliographique des méthodes formelles pour la spécification et la vérification des systèmes d'une part et des travaux de vérification des programmes C d'autre part. Pour cela, un panorama des formalismes utilisés dans la phase de spécification et de vérification formelle d'une manière générale est exposé dans la section 1.1. Nous consacrons la section 1.2 à la présentation des travaux connexes de la vérification des programmes C. Ce chapitre est conclu par une discussion dans la section 1.3 relative au travail proposé dans cette thèse.

1.1 LES MÉTHODES FORMELLES

Les fondements théoriques de l'utilisation des méthodes formelles dans l'informatique remontent à l'année 1949, où Turing a introduit l'idée de raisonner mathématiquement sur les programmes. Depuis, plusieurs chercheurs comme Floyd [Flo67], Hoare [Hoa69] et Owicki [Owi75] l'ont suivi, apportant ainsi d'importantes contributions à la science et plus particulièrement dans le domaine des méthodes formelles. Ces dernières sont définies dans [CW96] ainsi :

> « Les méthodes formelles constituent l'ensemble de langages, techniques et outils mathématiques permettant la spécification et la vérification des systèmes. »

Les méthodes formelles sont utilisées dans les différentes étapes du cycle de vie du logiciel : la phase d'analyse, de conception et de validation. Elles permettent de repérer et de corriger des erreurs assez tôt durant le processus de développement. Ces méthodes permettent d'accroître la qualité des logiciels en respectant les exigences du système et en évitant les erreurs possibles.

L'utilisation des méthodes formelles implique une phase de spécification du système qui consiste à décrire ce dernier avec un modèle formel ainsi que l'ensemble de propriétés qu'il doit vérifier. Cette phase est suivie d'une phase de vérification qui détermine si le système satisfait bien les propriétés exigées par sa spécification.

1.1.1 Spécification formelle

La phase de spécification repose sur un langage formel basé sur une syntaxe claire et précise avec une sémantique bien définie. Le choix de ce langage dépend du type de système à spécifier (fini, infini, séquentiel, ou concurrent) et aussi de la technique de vérification à appliquer. Étant donné le nombre de formalismes existants, nous décrivons dans ce qui suit, de façon non exhaustive, les plus communément utilisés.

- **Les automates** (ou les machines à états), sont des outils fondamentaux de l'informatique théorique. Un automate est un ensemble d'états où chaque état est représenté par un nœud. Les nœuds sont reliés entre eux par des transitions. Le type

d'automate le plus couramment répandu est l'automate à états finis. Pour étendre l'expressivité des machines à états finis, de nombreuses variantes ont été proposées, comme les automates temporisés [AD94].

- **Les algèbres des processus** sont un formalisme mathématique pour décrire et analyser des processus communicants. Ils utilisent des opérateurs algébriques comme la composition séquentielle, le choix non-déterministe et la composition parallèle pour décrire le comportement d'un système. Les algèbres de processus les plus anciennes pour les systèmes concurrents (ou asynchrones) sont les CSP (« *Communicating Sequential Processes* ») [Hoa78] et les CSS (« *Calculus of Communicating Systems* ») [Mil89].

- **Les réseaux de Petri** ont été introduits par Carl Adam Petri [Pet62]. Il s'agit d'une notation graphique basée sur des concepts mathématiques afin de modéliser les systèmes à événements discrets. Un graphe comporte des *places*, représentées par des cercles et des *transitions*, représentées par des rectangles. Les *arcs* relient les places et les transitions. Un arc à l'entrée d'une transition désigne une *pré-condition* et un arc en sortie d'une transition désigne une *post-condition*. Une pré ou une post-condition signifie une condition qui doit être vérifiée pour franchir la transition en question. Chaque place contient un nombre positif ou nul de *jetons*. Un marquage consiste à associer à chaque place un nombre de jetons. Ce marquage définit un état du réseau. Le comportement d'un réseau de Petri est analysé en construisant le *graphe d'états*.
  Par leur représentation graphique compacte, les réseaux de Petri constituent un modèle facile à comprendre, à manipuler et donc à analyser. Plusieurs variantes de réseaux de Petri ont été proposées. Les principales extensions sont les réseaux de Petri temporels [Mer74], temporisés [Ram74] et colorés [Jen81].

- **Les logiques temporelles** sont des extensions de la logique classique et qui permettent de spécifier l'évolution temporelle du comportement d'un système. Le premier à avoir proposé une logique temporelle est Pnueli [Pnu77] en 1977. Dans les années 80, des chercheurs comme Hailpern [Hai80] et Lamport [Lam83] ont intégré la logique temporelle pour spécifier et vérifier des systèmes. Ces logiques ont été aussi utilisées pour spécifier les propriétés du système. Un large éventail de logiques temporelles existent dans la littérature. Nous décrivons dans la section 1.1.2 les principales logiques temporelles existantes.

- D'autres formalismes bien connus existent, citons à titre d'exemple le langage B [Abr05] qui est fondé sur la théorie des ensembles.

Pour la spécification et l'analyse d'applications concurrentes et réparties, les modèles sémantiques du parallélisme sont utilisés afin d'exprimer la sémantique du parallélisme des modèles de spécification. Dans la théorie, plusieurs modèles du parallélisme ont été étudiés. Ces modèles peuvent être classés en deux catégories : les modèles d'entrelacement et les modèles de non-entrelacement.

- Le modèle d'entrelacement (comme les systèmes de transitions [Kel76] et les arbres de synchronisations [Mil82]) considère que les actions du système s'exécutent séquentiellement. Par exemple, l'exécution parallèle de deux actions a et b est représentée par un choix non-déterministe qui consiste à considérer deux scénarios : a s'exécute en premier et ensuite b, ou bien b s'exécute en premier et ensuite a.

- La sémantique de non-entrelacement, dite aussi de *vraie concurrence* (comme les systèmes de transitions étiquetés causaux [CDC93] et les arbres causaux [DD89]) considère que les actions du système sont non atomiques. C'est-à-dire que l'exécution parallèle de a et b est représentée au même moment. Le parallélisme est décrit en représentant les relations de dépendance entre les actions.

1.1.2 Expression des propriétés

Pour vérifier des propriétés sur un système, il convient de les spécifier dans un formalisme dédié. Dans la littérature, les propriétés peuvent s'exprimer dans une logique temporelle ou à travers des modèles.

(a) Logiques temporelles. Les formalismes basés sur la logique consistent à exprimer les propriétés du système avec des formules dans une logique temporelle. Parmi les deux principales variantes de logique temporelle, on trouve celles qui s'expriment sur les séquences d'exécution du modèle comme la logique temporelle linéaire LTL (« *Linear Temporal Logic* ») [Pnu77] et celles qui s'expriment sur les arbres d'exécution du modèle comme la logique temporelle arborescente CTL (« *Computation Tree Logic* ») [CES86] et CTL* [EH86]. Des extensions de ces logiques ont été proposées. Par exemple, la logique TCTL (« *Timed Computation Tree Logic* ») [ACD93] qui est une extension de CTL afin d'exprimer des propriétés temporisées et la logique TLA (« *Temporal Logic of Actions* ») [Lam94] qui fait intervenir la notion d'action. Une brève description des logiques LTL et CTL est donnée dans la suite de ce paragraphe.

- La logique LTL [Pnu77] est une logique pour laquelle le temps se déroule linéairement, c'est-à-dire chaque instant dans le temps a un unique futur possible. L'interprétation d'une formule est relative à un chemin d'exécution. Un chemin d'exécution est une séquence infinie d'états du système. Chaque état est étiqueté par un ensemble de *propositions atomiques* et possède un seul successeur.
Les formules LTL sont composées d'un ensemble de propositions atomiques, utilisant les opérateurs booléens usuels ($\wedge$, $\neg$, $\Leftrightarrow$) et des opérateurs temporels. La figure 1.1 illustre des exemples de formules utilisant les opérateurs LTL. L'opérateur **F** (pour « *finally* ») ou $\Diamond$ signifie *fatalement dans le futur*, **G** (pour « *globally* ») ou $\Box$ signifie *toujours dans le futur*, **X** (pour « *next* ») ou $\circ$ indique *à l'état suivant* et **U** (pour « *until* ») signifie *jusqu'à ce que*.

- La logique temporelle arborescente CTL [CE82] permet d'exprimer des propriétés sur l'arbre d'exécution du programme. Elle considère plusieurs futurs possibles à partir d'un état du système plutôt que d'avoir une vue linéaire du système

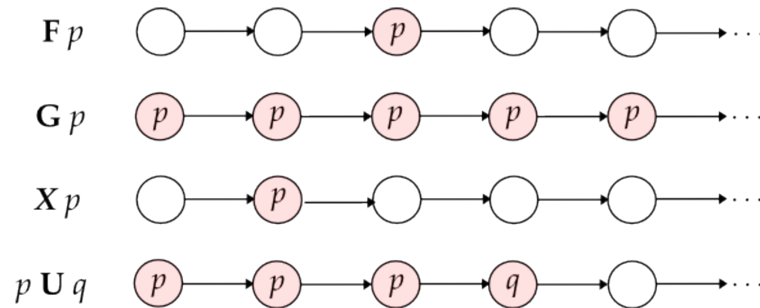


FIGURE 1.1 – Illustration des opérateurs LTL

considéré.

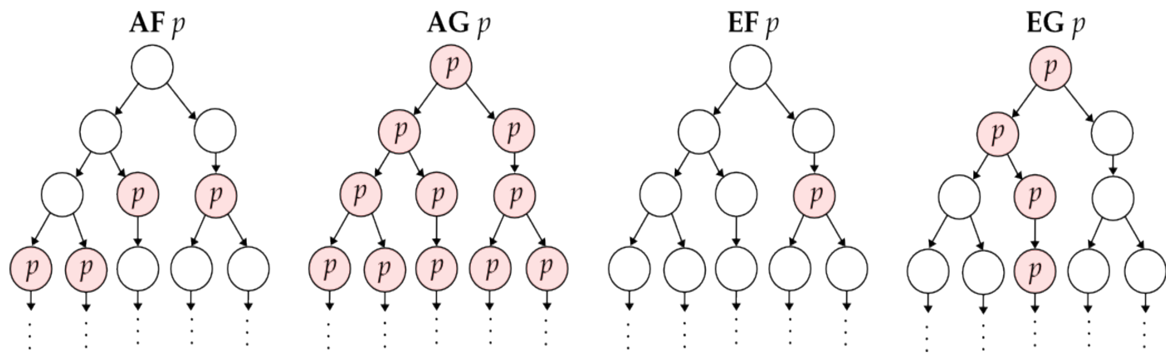


FIGURE 1.2 – Illustration des opérateurs CTL

Cette logique introduit des quantificateurs de chemin par rapport à LTL. Les opérateurs temporels (comme **F**, **G** et **X**) doivent être directement précédés par les quantificateurs **E** qui signifie *il existe un chemin d'exécution* ou **A** qui signifie *pour tous les chemins d'exécution*. Nous exposons dans la figure 1.2 des exemples de propriétés exprimées avec les opérateurs CTL. Par exemple, la propriété **EF**p se traduit par *il existe un chemin où* p *est vraie*.

Les propriétés à vérifier sur un système peuvent être classées de plusieurs manières. Une classification générique considère deux catégories de propriétés : sûreté et vivacité.

Une propriété de sûreté (« *safety* ») énonce que *quelque chose de mal ne doit jamais arriver*. Elle est évaluée sur une séquence finie d'états. Dans cette classe de propriétés, on trouve les invariants du programme. Les propriétés suivantes sont des exemples de propriétés de sûreté : *la valeur de x est toujours positive durant l'exécution du programme* et *au maximum un seul processus peut accéder à la section critique*.

La propriété de vivacité (« *liveness* ») exprime que *quelque chose de bon doit finir par arriver, sous certaines conditions*. La satisfaction de telle propriété se fait sur une séquence infinie d'états. Par exemple, les propriétés suivantes sont des propriétés de vivacité : *le programme doit toujours renvoyer un résultat* et *si un processus envoie une demande*

pour accéder à la section critique, il y entrera fatalement.

(b) Spécification des propriétés par des modèles. Dans ce formalisme, le comportement du système et sa spécification sont décrits à travers des automates. Des relations d'équivalence ou de préordre entre ces automates permettent de décider si le système vérifie bien la spécification établie. Une classification de ces différentes relations est donnée dans [vG90].

1.1.3 Vérification

La phase de vérification permet de s'assurer qu'un système vérifie les propriétés attendues. Elle permet de garantir que le système est développé correctement. Nous classons les techniques de vérification en trois grandes approches : la preuve, le *model-checking* et l'interprétation abstraite.

1.1.3.1 La preuve

Dans cette approche, le système est modélisé à travers des formules mathématiques. Des règles de déduction sont appliquées sur ce modèle afin de prouver la propriété à vérifier. Parmi les premiers travaux dans cette approche, citons notamment la logique de Floyd-Hoare [Flo67, Hoa69], dont le but est de formaliser la preuve de correction d'un programme. Dans cette logique, un programme S est annoté avec des formules logiques P et Q, représentant des prédicats sur les variables du programme. La formule P représente la *précondition*, c'est-à-dire le prédicat qui doit être vrai lors de l'appel à S. La formule Q est le prédicat qui doit être vrai après l'exécution de S. Un ensemble de règles est inféré afin de prouver la correction du programme S vis-à-vis des prédicats P et Q.

Plusieurs travaux ont succédé à ceux de Floyd et Hoare. On trouve le concept de la *conception par contrats* (« *Design by Contracts* » en anglais), qui est apparu avec le langage Eiffel [Mey92], dans le cadre de la programmation orientée objet. Des préconditions, des postconditions, des invariants et des assertions permettent de décrire des contrats.

Dans le même concept, on trouve aussi ACSL (« *ANSI/ISO C Specification Language* ») [BFM+10] qui est un langage de contrats pour C. Il permet de spécifier des propriétés sur le comportement d'un programme C à travers des annotations. Ces propriétés sont exprimées dans la logique du premier ordre. Des outils permettent de raisonner sur ces annotations, notamment les outils intégrés dans la plateforme Frama-C [CKK+12] que nous décrivons dans la section 1.2.1.2.

Les preuves sur un programme peuvent être déduites à travers un outil semi-automatique, appelé un assistant de preuve ou un outil automatique comme le démonstrateur de théorèmes.

- Assistant de preuve : est un logiciel interactif qui permet de vérifier des preuves en se basant sur des formules mathématiques données par l'utilisateur. Pour pouvoir

utiliser un assistant de preuve, l'utilisateur doit formaliser son programme dans le langage de celui-ci. L'utilisateur doit ensuite détailler toutes les étapes de son raisonnement, même les plus évidentes. Ce qui constitue dans la majorité cas une tâche fastidieuse. Parmi les assistants de preuve, citons par exemple Coq [BC10] et Isabelle/HOL [NPW02].

- Démonstration automatique : outre les assistants de preuve, la démonstration d'un théorème peut être accomplie par un outil entièrement automatique appelé démonstrateur automatique de théorèmes comme l'approche des tableaux sémantiques [Fit90]. Dans cette catégorie, on trouve aussi les solveurs « *Boolean SATisfiability* » (SAT)/« *Satisfiability Modulo Theories* » (SMT). Ces derniers fournissent des procédures de décision permettant de déterminer si une formule est satisfiable par rapport à une ou des théories données.

  – Un solveur SAT permet un raisonnement inductif automatique sur des formules de la logique propositionnelle données en entrée.

  – Un solveur SMT permet de résoudre des problèmes de décision dans lesquels les formules sont exprimées dans une logique du premier ordre, en se basant sur une théorie donnée. Un problème SMT peut être vu comme une généralisation d'un problème SAT, dans lequel les variables sont remplacées par des prédicats. Parmi les solveurs SMT on peut citer Alt-Ergo [BCCL08] et Z3 [DMB08].

Actuellement, on trouve des outils intégrant des assistants de preuves ainsi que des démonstrateurs de théorèmes. Citons notamment la plateforme Why [BFMP11], qui permet de générer des *obligations de preuves* assurant la correction d'un programme à partir d'un programme annoté. Les annotations sont exprimées dans le langage Why, un langage fonctionnel proche de Caml. Les obligations générées par Why sont ensuite utilisées par des outils en aval comme l'assistant de preuve Coq [BC10] ou le prouveur Simplify [DNS05].

La technique de preuve a été utilisée en milieu industriel, notamment dans le projet Météor (acronyme de « MÉTro Est Ouest Rapide ») [BBFM99] qui a donné naissance à la ligne 14 du métro parisien fonctionnant en automatisme intégral sans conducteur. Les composants critiques de cette ligne ont été prouvés avec la méthode B [Abr05]. Cette méthode permet la spécification et la conception incrémentale des logiciels. Les modèles dans la méthode B sont écrits dans un langage basé sur la théorie des ensembles. Un modèle peut être raffiné de manière successive jusqu'à avoir un modèle concret à partir duquel une implémentation peut être générée. La relation de raffinement peut-être prouvée mathématiquement par des outils associés à cette méthode.

L'avantage de ces techniques déductives est qu'elles sont indépendantes de la taille du système. Elles permettent de mieux comprendre le comportement du programme. Toutefois, l'inconvénient des assistants de preuve est qu'ils ne sont pas entièrement automatiques. Même avec un démonstrateur automatique de théorèmes, il est nécessaire

que le processus de vérification soit guidé par l'utilisateur, plus particulièrement par un spécialiste, car leur usage demande une bonne expertise et un temps considérable pour spécifier les détails du raisonnement.

1.1.3.2 Model-checking

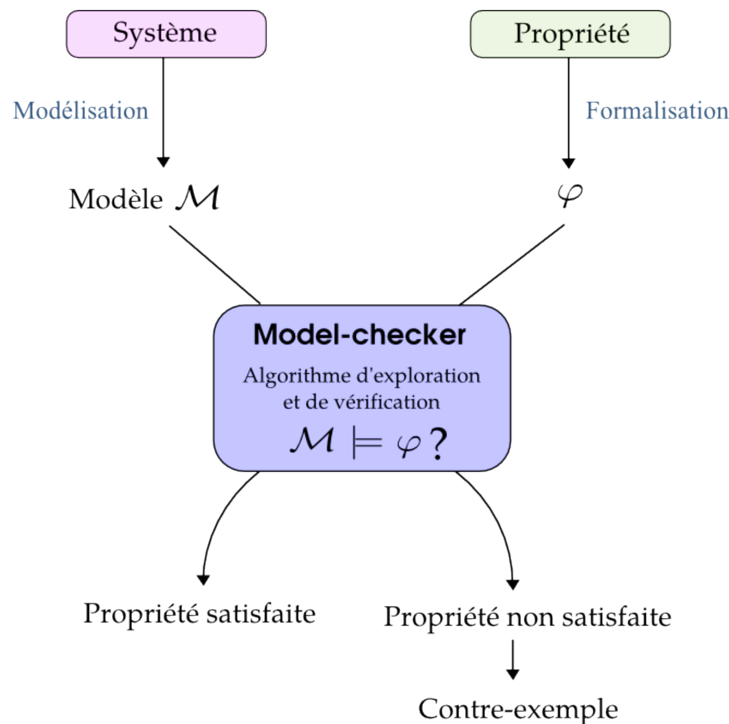


FIGURE 1.3 – Principe du *model-checking*

Le *model-checking* [CES86] est une technique de vérification formelle qui consiste à déterminer si un modèle $\mathcal{M}$ du système satisfait une propriété donnée φ, exprimée dans une logique temporelle. Le modèle $\mathcal{M}$ est souvent décrit par un automate, ou un *système de transitions*. La vérification par *model-checking* repose sur l'exploration exhaustive de l'*espace d'états*, un système de transitions qui encode le comportement du système. Le principe du *model-checking* comporte trois phases (voir figure 1.3) :

- la construction d'un modèle $\mathcal{M}$ du système dans un formalisme formel (voir section 1.1.1) ;

- la spécification des propriétés à vérifier (voir section 1.1.2) ;

- la vérification des propriétés sur le modèle du système en utilisant un algorithme bien défini qui dépend de la nature du modèle du système ainsi que de la propriété.

Formellement, étant donné un modèle $\mathcal{M}$ et une propriété φ, le *model-checking* revient à vérifier si $\mathcal{M}$ satisfait φ, noté $\mathcal{M} \models \varphi$. Si la propriété n'est pas vérifiée sur le modèle du

système, un contre-exemple est produit indiquant comment le modèle a atteint l'erreur. Ce contre-exemple décrit un comportement du système qui ne vérifie pas la propriété attendue.

Les techniques de *model-checking* peuvent être classées selon la manière avec laquelle les états du système à vérifier sont représentés. Selon ce critère, nous distinguons deux approches de *model-checking* : *explicite* et *symbolique*. Nous décrivons dans ce qui suit ces deux classes en présentant des exemples de *model-checker*. Nous nous intéressons plus particulièrement au formalisme utilisé dans chaque *model-checker*, qui peut engendrer dans certains cas une limitation lors de la spécification du système.

(a) Model-checking explicite. Il consiste à représenter chaque état explicitement dans la mémoire. L'espace d'états est représenté sous la forme d'un graphe et son exploration consiste à parcourir celui-ci. Parmi les *model-checker* explicites, on peut citer notamment SPIN [Hol97] qui a été développé aux laboratoires Bell. Il permet la vérification et la simulation des systèmes concurrents finis. SPIN accepte des modèles écrits en langage Promela (« *PROcess MEta Language* ») et les propriétés à vérifier sont exprimées à travers la logique temporelle LTL. Promela est un langage inspiré du C. Il ne gère ni les pointeurs ni l'allocation dynamique de mémoire. Il offre des outils de communication comme des canaux FIFO. Il ne fait pas de différence entre une condition et une instruction. Si une instruction n'est pas exécutable, elle sera bloquée en attente d'exécution. De ce fait, Promela est plus adapté pour la modélisation des systèmes communicant par rendez-vous ainsi que pour les protocoles de communication.

(b) Model-checking symbolique. Dans cette approche les états sont représentés par des équations [McM92]. L'exploration d'états se fait en manipulant un ensemble d'états au lieu de les considérer un par un comme dans le *model-checking* explicite. Le premier *model-checker* symbolique est SMV (« *Symbolic Model Checker* ») [McM93]. Une réimplémentation de celui-ci est NuSMV [CCGR99], qui est open source, à la différence de SMV qui ne l'est pas. Le modèle dans NuSMV est décrit dans le langage d'entrée de SMV. Un système est décomposé en plusieurs modules. Chaque module définit une machine à états finis. Les données supportées par le langage sont les booléens, les scalaires et les tableaux de taille fixe. Le système peut être ensuite modélisé à travers la composition synchrone ou asynchrone de ces modules. NuSMV permet de vérifier des propriétés CTL et LTL.

La technique de *model-checking* est entièrement automatique, ne requiert aucun effort de la part de l'utilisateur et donc plus rapide par rapport à d'autres méthodes comme la preuve, dont l'utilisation peut nécessiter des mois de travail en mode interactif [Cla08]. L'autre atout du *model-checking* est qu'il produit des contre-exemples quand la vérification d'une propriété échoue. Cette trace d'erreur est un outil utile permettant de corriger les bugs du système. Le *model-checking* n'est applicable que sur des systèmes finis. Cependant lorsque la taille du système à vérifier devient grande, la vérification devient inapplicable. Ce problème est connu sous le nom de *l'explosion combinatoire*. Des techniques de réduction ont été proposées afin de réduire cette explosion d'états.

1.1.3.3 L'interprétation abstraite

L'interprétation abstraite est une théorie de l'approximation des sémantiques des langages informatiques (de programmation ou de spécification), qui a été introduite par Patrick et Radhia Cousot en 1970 [CC77]. Elle consiste à exécuter un programme sur un domaine abstrait en construisant une sur-approximation de la sémantique du programme, appelée une *sémantique abstraite*. Cette dernière est suffisante pour prouver que le programme est correct par rapport à sa spécification. Un analyseur statique basé l'interprétation abstraite consiste d'abord à construire une sémantique concrète d'un programme P. Un domaine abstrait est ensuite déterminé permettant de représenter l'ensemble d'états de P. Ce domaine ne contiendra que les informations pertinentes par rapport à la propriété à vérifier. Un exemple de domaine abstrait de l'ensemble des entiers relatifs consiste à considérer l'ensemble des entiers positifs, l'entier nul et l'ensemble des entiers négatifs. Si on veut prouver que P retourne un résultat positif, il suffit de prouver que les résultats sont dans l'ensemble des entiers positifs, sans pour autant connaître le résultat exact envoyé par le programme.

Plusieurs outils implémentent cette technique. Citons par exemple Astrée [BCC+03] qui a été utilisé pour prouver l'absence d'erreurs à l'exécution dans le logiciel de commande de vol d'Airbus A340 et A380. On trouve aussi Polyspace [Mat16], outil utilisé dans le secteur de l'aéronautique et de l'automobile et qui permet de vérifier du code C, C++ et Ada. Enfin, citons le plugin *value analysis* de la plateforme Frama-C [CKK+12] pour la vérification formelle du code C.

La technique d'interprétation abstraite permet de traiter des systèmes infinis en considérant une abstraction finie de celui-ci. Mais comme il s'agit d'une approximation, cette technique est incomplète. Elle peut alors générer des *fausses alarmes* (ou *faux positifs*). Une fausse alarme correspond à une erreur détectée dans la sémantique abstraite mais qui ne correspond pas à une erreur effective dans la sémantique concrète.

1.2 Vérification formelle des programmes C

Les techniques de vérification formelles décrites dans la section précédente ont été utilisées pour la vérification des circuits matériels, des systèmes de contrôle-commande ainsi que pour les programmes. Le domaine de la vérification des programmes est très actif et cela s'explique par le fait que les langages de programmation sont complexes et que ces programmes sont souvent concurrents. Il existe dans la littérature un large spectre d'outils de vérification formelle des programmes qui relèvent des domaines industriels et académiques différents. Étant donné que nous nous sommes intéressés, dans cette thèse, à la vérification des programmes écrit avec le langage C, nous décrivons dans cette section les principaux travaux proposés pour la vérification des programmes C. Ces travaux combinent plusieurs techniques de vérification, nous avons fait le choix de les classer en quatre catégories : ceux basés sur l'analyse statique par interprétation abstraite, les outils implémentant l'approche CEGAR (voir section 1.2.2), ceux basés sur le *model-checking*

borné et les outils basés sur le *model-checking* symbolique ou explicite (voir section 1.1.3.2).

1.2.1 Analyse statique par interprétation abstraite

Les méthodes d'interprétation abstraite sont quasiment automatiques et permettent de gérer des logiciels de grandes tailles. L'interprétation abstraite a été adoptée par plusieurs outils d'analyse statique. Dans ce qui suit, nous présentons les analyseurs statiques C les plus connus.

1.2.1.1 Astrée

Astrée (« *Analyseur statique de logiciels temps-réel embarqués* ») [BCC+03] est un outil d'analyse statique des programmes embarqués temps-réel écrits en C, développé en 2001 à l'École Normale Supérieure. Il utilise la technique d'interprétation abstraite pour détecter et prouver l'absence des erreurs d'exécution (« *runtime errors* ») telles que la division par zéro et le débordement de mémoire. Astrée a été utilisé dans le milieu industriel et a permis de vérifier les composants critiques d'Airbus.

Astrée supporte un large sous-ensemble de C mais il ne traite ni l'allocation dynamique de la mémoire ni la récursivité. Il est spécialisé pour analyser les programmes embarqués, plus particulièrement les applications de contrôle/commande avioniques et spatiales. Astrée ne supporte pas les programmes C concurrents. Son successeur AstréeA [CCF+07] (« *Analyseur statique de logiciels temps-réel asynchrones embarqués* ») permet de gérer la concurrence.

1.2.1.2 Frama-C

Frama-C (« *Framework for modular analysis of C* ») [CKK+12] est une plateforme de vérification des programmes C regroupant plusieurs techniques d'analyse statique, développée par le CEA-List et INRIA à Saclay. Elle est composée de plusieurs greffons (*plugins*) destinés à effectuer différents types d'analyse du code. Son architecture modulaire permet de développer de nouveaux greffons (*plugins*) en utilisant les greffons existants. Parmi ces greffons, citons le greffon *value* [CCM09] qui est basé sur l'interprétation abstraite et qui permet l'analyse des valeurs dans le code C. Le greffon *slicing* [CEA16] permet de réduire un programme initial en un programme équivalent simplifié par rapport à des critères donnés par l'utilisateur. Le greffon Jessie [MM15] permet la vérification déductive de programmes C. Dans ce cas, le programme C à analyser est annoté en utilisant le langage de spécification ACSL avec les propriétés qu'il doit vérifier. Le programme annoté est traduit en une représentation intermédiaire, qui est ensuite passée à des assistants de preuve comme Coq [BC10] et Isabelle-HOL [NPW02] ou bien à des prouveurs automatiques comme Simplify [DNS05] et Yices [Dut14]. Le plugin Aoraï [CL15a] permet d'annoter automatiquement un programme C à partir d'une formule LTL afin de prouver que le programme vérifie cette formule. Ce plugin ne gère que les propriétés de sûreté et ne supporte pas les propriétés de vivacité.

1.2.2 Techniques implémentant l'approche CEGAR

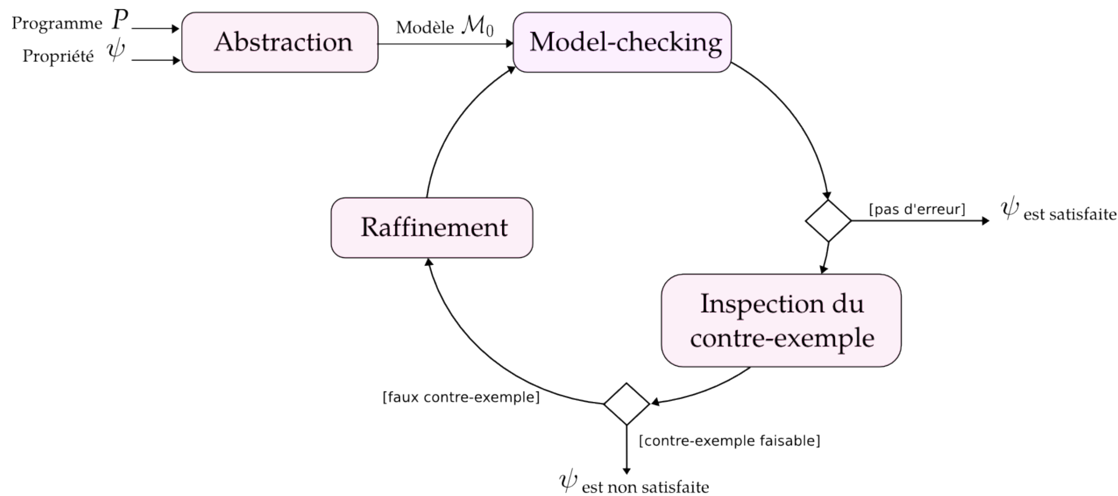


FIGURE 1.4 – Le processus de vérification dans l'approche CEGAR

CEGAR (« *Counter-Example Guided Abstraction Refinement* ») est une approche de vérification qui se base un mécanisme de raffinements successifs guidés par les contre-exemples [CGJ+00]. Le processus de vérification d'une propriété φ sur un programme P se fait à travers un processus itératif d'analyse (voir figure 1.4), composé des étapes suivantes :

(i) générer un modèle abstrait $\mathcal{M}_0$ (le plus abstrait possible) du programme, exprimé dans un langage accepté par le *model-checker* ;

(ii) vérifier la satisfaisabilité de la propriété φ sur le modèle $\mathcal{M}_0$:

- si la propriété est vérifiée sur $\mathcal{M}_0$, alors la propriété est satisfaite sur le programme et le processus peut alors s'arrêter,
- sinon, le *model-checker* fournit un contre-exemple ;

(iii) vérifier que le contre-exemple est effectif sur le programme :

- si oui, alors la propriété n'est pas vérifiée sur le modèle $\mathcal{M}_0$ et le processus est arrêté,
- sinon, il s'agit d'un faux contre-exemple. Dans ce cas, le modèle $\mathcal{M}_0$ est raffiné en un modèle $\mathcal{M}_1$ en éliminant le contre-exemple et le processus de vérification continue en exécutant l'opération (ii) sur le modèle raffiné $\mathcal{M}_1$.

Le processus de vérification s'arrête quand l'abstraction est assez précise pour vérifier la propriété φ ou pour reproduire le contre-exemple sur le programme P initial.

Cette approche a eu beaucoup de succès. Elle a été implémentée dans des outils comme SLAM (« *Software (Specifications), Languages, Analysis, and Model checking* ») [BR01] et BLAST (« *Berkeley Lazy Abstraction Software Verification Tool* ») [HJMS03] que nous détaillons dans les paragraphes suivants.

1.2.2.1 SLAM

SLAM [BR01] est un outil conçu par Microsoft pour la vérification des propriétés de sûreté sur des programmes C séquentiels. Il a été initialement utilisé pour la vérification des drivers sous Windows XP. La phase de construction d'une abstraction dans SLAM est basée sur le mécanisme d'*abstraction par prédicats*, qui consiste à abstraire automatiquement un programme en un programme booléen. Ce programme booléen garde le flot de contrôle du programme initial et dans lequel toutes les variables sont de type booléen. Les prédicats sont définis dans un langage de spécification appelé SLIC (acronyme de « *Specification Language for Interface Checking* »). La phase de vérification se fait ensuite par un *model-checker* symbolique appelé BEBOP, dédié aux programmes booléens. Le raffinement consiste à ajouter de nouveaux prédicats.

SLAM a été initialement développé pour gérer les programmes séquentiels. Même si des extensions ont été ensuite proposées pour gérer les programmes concurrents, son support pour la concurrence reste limité.

1.2.2.2 BLAST

BLAST (« *Berkeley Lazy Abstraction Software Verification Tool* ») [HJMS03] est un outil, similaire à SLAM, développé en 2002 par l'université de Californie à Berkeley pour la vérification des propriétés de sûreté des programmes C séquentiels. Il combine du *model-checking* et la preuve de théorèmes pour implémenter la boucle CEGAR.

La différence par rapport à SLAM c'est qu'il implémente la technique d'*abstraction paresseuse* (« *lazy abstraction* » [HJMS02]) dans la phase d'abstraction. Cette technique consiste à ne raffiner que l'espace d'états de l'abstraction ne contenant pas la trace de l'erreur. Ce qui permet de rendre la vérification moins coûteuse. À la différence de SLAM où l'abstraction est construite avant la phase de vérification, BLAST génère l'abstraction à la volée. Une extension a été implémentée dans BLAST afin de vérifier les programmes C multithreadés. Elle repose sur l'approche *assume-guarantee* [HJMQ03] dans laquelle la vérification se fait en vérifiant chaque thread séparément.

BLAST supporte un large sous-ensemble du C, mais il ne gère ni les pointeurs de fonction ni les appels récursifs. BLAST ne supporte pas la conversion explicite des types de données (opération `cast`). Il suppose que l'utilisation de l'arithmétique des pointeurs pour manipuler les éléments d'un tableau respecte la taille de celui-ci. Les premières versions de BLAST utilisent le prouveur de théorèmes Simplify [DNS05].

1.2.2.3 SMACK

SMACK [HCE+15] est un outil qui permet la vérification modulaire des programmes C. Il utilise en aval l'outil CORRAL [LQL12]. L'outil SMACK [RE14] consiste à traduire un programme C en une représentation LLVM IR (pour « *Low Level Virtual Machine Itermediate Representation* »). Il s'agit d'un langage bas niveau et multiplateforme utilisant la représentation SSA (« *Static Single Assignment* »), dans laquelle chaque variable n'est affectée qu'une seule fois. Des optimisations sont réalisées sur le code LLVM IR, à savoir la propagation de constante et l'élimination de code mort. À partir de la représentation LLVM IR optimisée, SMACK génère un code Boogie IVL (pour « *Boogie Intermediate Verification Language* »).

CORRAL implémente l'approche CEGAR. Il prend en entrée le code Boogie IVL, qui est converti en une formule du premier ordre. Cette formule est ensuite transmise au solveur Z3 [DMB08].

L'intérêt de l'utilisation de l'abstraction par prédicats réside sur le fait que le problème d'atteignabilité et de la terminaison des programmes booléens est décidable. D'un autre côté, l'abstraction par prédicat préserve la correction. En effet, le programme abstrait contient plus de comportement que le programme initial, s'il est correct alors le programme initial l'est aussi. Si une erreur est détectée, le programme initial ne contient pas nécessairement une erreur, car elle peut correspondre à un faux contre-exemple.

Comme pour SLAM et BLAST, la difficulté d'une technique basée sur l'abstraction par prédicats est l'identification des prédicats qui permet d'avoir une abstraction assez précise pour vérifier une propriété donnée et qui génère moins de faux contre-exemples. Quand les prédicats sont insuffisants pour montrer la correction d'un programme, l'assistance de l'utilisateur est requise.

1.2.3 Techniques basées sur le *model-checking* borné

Le *model-checking* borné (ou BMC pour « Bounded Model Checking » en anglais) [BCCZ99] est une technique introduite en 1999 pour exploiter les solveurs SAT. Initialement, cette technique a été proposée pour les systèmes de transitions avec des propriétés temporelles exprimées en LTL. L'idée est de réduire le problème du *model-checking* à un problème de satisfaisabilité d'une formule propositionnelle et ceci en un temps polynomial. Le mot *borné* vient du fait que les états sont explorés jusqu'à une borne k.

Pour un système $\mathcal{M}$, le *model-checking* borné consiste à vérifier s'il existe un chemin d'exécution de longueur k violant une propriété φ et qui constitue la trace d'erreur pour $\mathcal{M}$. Si la formule φ n'est pas satisfiable, il faut dans ce cas augmenter la valeur de k et reprendre la procédure de vérification.

Cette technique a connu un certain succès dans la vérification des circuits matériels. Ce succès est dû principalement au fait que cette technique a tiré profit des solveurs SAT.

1.2.3.1 CBMC

La première implémentation de la technique BMC pour les programmes C est CBMC [CKL04], développé à CMU (l'université Carnegie-Mellon à Pittsburgh). CBMC permet de vérifier les erreurs d'exécution dans les programmes C et C++, comme l'accès d'un tableau en dehors des bornes, le débordement de mémoire, des assertions définies par l'utilisateur et le déréférencement de pointeur null. Il permet aussi de vérifier la cohérence d'un programme C avec une implémentation en Verilog. Dans CBMC, le programme C est d'abord transformé en un programme équivalent, dans lequel les boucles `for` et `do while` sont remplacées par des instructions équivalentes utilisant `while` et les expressions à effets de bord sont remplacées par des expressions sans effet de bord. Le programme est ensuite dépilé, à savoir les boucles `while` sont transformées en des instructions `if/then/else` en ajoutant une assertion qui vérifie que le dépliage ne dépasse pas la borne k. Une méthode similaire s'applique à l'appel aux fonctions récursives et les instructions `goto`. Le programme déplié est par la suite traduit en une représentation SSA. Cette représentation constitue la formule booléenne du programme, noté P. Enfin, pour vérifier une propriété φ, il suffit de construire la formule $P \wedge \neg\varphi$ et la passer au solveur SAT. Parmi les solveurs supportés par CBMC, citons par exemple Z3 [DMB08] et Yices [Dut14].

1.2.3.2 Plateforme CSeq

CSeq [INT+16] est une plateforme permettant la vérification de programme C concurrent utilisant des threads de l'API POSIX. Elle est basée sur la technique de *séquentialisation* [QW04] (« *sequentialization* » en anglais). La technique de séquentialisation permet de traduire un programme C multithreadé en un programme séquentiel non déterministe qui simule le comportement du programme initial. L'idée est donc de transformer l'indéterminisme d'exécution, induit par la concurrence, en un indéterminisme sur les données. La plate-forme inclut des outils de transformation de code. Parmi ces outils, citons Lazy-CSeq [ITF+14] et MU-CSeq [TIF+14]. Le programme traduit peut-être ensuite vérifié par des outils de vérification de programmes C séquentiels.

- Lazy-CSeq [ITF+14] implémente une technique d'analyse d'accessibilité avec un changement de contexte borné (« *bounded-context switch* » en anglais), qui consiste à borner le nombre de changements de contexte entre les threads. Cette technique part de la constatation suivante : plusieurs erreurs peuvent être détectées dès les premiers changements de contexte, il convient alors de borner ce nombre. Lazy-CSeq intègre des outils de vérification en aval. Il supporte plusieurs *model-checker* bornés comme CBMC [CKL04].

- Une autre extension est l'outil MU-Cseq [TIF+14], qui implémente une technique appelée « *memory unwinding* » ou MU. Cette technique consiste à borner le nombre d'opérations d'écriture dans la mémoire partagée. L'idée de la technique vient de la constatation que plusieurs erreurs se manifestent dans un ordre partiel d'exécution

de threads et jusqu'à un nombre faible d'accès mémoire.
Dans cette technique, une séquence d'opérations d'écriture de taille n est définie de manière indéterministe. Cette séquence définit la MU du programme. Les opérations d'écriture et de lecture dans la mémoire partagée sont remplacées par des opérations sur la MU. Les threads sont ensuite simulés par des fonctions C dans un ordre d'exécution compatible à la séquence d'opérations définie dans la MU. L'objectif est de trouver une séquence d'écriture qui mène à l'échec d'une assertion. En aval, la vérification est assurée par le *model-checker* CBMC.

Lazy-CSeq supporte l'allocation dynamique à la différence de MU-CSeq. Les deux outils ont obtenu de bons résultats dans les compétitions SV-COMP (« *Competition on Software Verification* ») organisées durant les conférences TACAS. Dans la catégorie *concurrence* de la compétition, Lazy-CSeq a été classé premier en 2014 et 2015 face à CBMC, MU-CSeq, et BLAST. De l'autre coté, MU-CSeq a été classé deuxième en 2014 et 2015, face à Lazy-CSeq et CBMC et premier en 2016 devant Lazy-CSeq.

En ce qui concerne la trace d'erreur, le contre-exemple fourni par les outils de CSeq correspond au code transformé et non pas au code C initial. Comme le programme séquentiel traduit contient plusieurs lignes de code injectées par la phase de traduction, il n'est pas aisé pour un utilisateur de traduire manuellement cette trace en une trace du code initial multithreadé.

1.2.4 Vérification par *model-checking* symbolique ou explicite

1.2.4.1 Verisoft

Verisoft [God97] est un *model-checker* conçu par les laboratoires Bell en 1997. Il permet la vérification à la volée des programmes concurrents écrits en C ou C++. Il s'agit du premier *model-checker* à avoir implémenté la technique d'ordre partiel qui permet de vérifier un programme sans l'exploration de tout l'espace d'états. Ce qui permet alors de réduire l'espace d'états et favorise une utilisation efficace de la mémoire. L'exploration de l'espace d'états se fait en un parcours en profondeur jusqu'à une borne spécifiée par l'utilisateur. Verifsoft est dit *sans état* (« *stateless* » en anglais), c'est-à-dire que les états ne sont pas stockés en mémoire.

Les propriétés de sûreté vérifiées par Verifsoft sont les assertions spécifiées par l'utilisateur et les interblocages. L'outil a été utilisé pour vérifier des protocoles de communication. Néanmoins, le projet de Verisoft a été abandonné depuis 2007.

1.2.4.2 Modex et C2Promela

Modex (« *Model Extractor* ») [Hol00] est un outil qui permet l'extraction automatique d'un modèle Promela à partir d'un code C. Promela est un langage de spécification, inspiré du C dans lequel les propriétés à vérifier sont exprimées à travers la logique temporelle LTL. Le modèle Promela généré est ensuite vérifié avec l'outil SPIN, un

model-checker explicite développé par Gerard J. Holzmann [Hol97].

Dans Modex, l'utilisateur peut générer un modèle Promela en se basant sur une table de règles fournie par défaut. Cette table peut être enrichie par l'utilisateur afin de générer un modèle plus précis.

C2Promela [KJ09] est un autre outil d'extraction de modèle Promela à partir du C. L'outil gère les aspects manquants de Modex, notamment les structures de données, les pointeurs et l'appel de fonction. L'outil supporte aussi l'allocation dynamique de la mémoire et la récursivité mais ne supporte pas les opérations arithmétiques sur les pointeurs.

1.2.4.3 Cubicle

Cubicle est un *model-checker* symbolique, qui utilise le solveur SMT Alt-Ergo [BCCL08]. Il permet de vérifier des propriétés de sûreté sur des systèmes de transitions paramétrés [GNRZ08], dont les états sont représentés par un ensemble fini de tableaux. Le langage de spécification de Cubicle est une version typée du langage Murϕ. Les travaux de [CDMM14] proposent une approche pour vérifier automatiquement la sûreté des barrières de synchronisation écrites en C. Leur approche consiste à traduire les programmes C en un système de transition paramétré à tableaux, qui sera ensuite vérifié à l'aide du *model-checker* Cubicle [CGK+12]. L'outil supporte un sous-ensemble restreint du langage C. Il ne supporte que les types `int`, `void` et les structures non récursives. L'utilisation des pointeurs est restreinte qu'au passage de paramètres par référence et au retour de fonction de type `void *`. Cubicle a été utilisé pour la vérification d'un protocole de cohérence de cache.

1.2.4.4 CMC

CMC (« *C Model Checker* ») [MPC+02] est un *model-checker* qui permet la vérification de propriétés de sûreté de programmes concurrents écrits en C ou C++. Les propriétés à vérifier sont définies dans le code C à travers des assertions ou des fonctions booléennes. Parmi ces propriétés, citons la mauvaise utilisation de pointeur et l'accès à la mémoire après libération avec l'opération `free()`. L'utilisateur ensuite doit simuler l'environnement d'exécution à travers des appels de fonction. La vérification se fait par la suite à la volée en générant l'espace d'états directement à partir du C. CMC a été utilisé pour la vérification des protocoles de communication.

1.2.5 Autres approches

1.2.5.1 VCC

VCC (« *Verifier for Concurrent C* ») [CDH+09] est une plateforme de vérification développée par Microsoft en 2008. Elle permet de vérifier des propriétés fonctionnelles sur des programmes C concurrents. La vérification dans VCC est modulaire, c'est-à-dire

chaque fonction est vérifiée en isolation. Chaque fonction peut être annotée par quatre types de clauses : des préconditions (constituant les conditions à respecter pour pouvoir appeler la fonction), des postconditions (l'ensemble de conditions à respecter après le retour de la fonction), des prédicats qui devront être vrais durant l'exécution de la fonction, des prédicats portant sur les parties mémoires que la fonction est autorisée à modifier. D'autres contrats sont utilisés dans VCC comme les invariants de type. Il est possible d'introduire des variables *fantômes* (« *ghosts* » en anglais). Ces variables permettent de maintenir une copie cachée sur une partie de la mémoire. Par exemple, elles sont utilisées par l'analyseur afin de représenter une liste par un ensemble et d'exprimer des propriétés sur cet ensemble.

VCC prend en entrée le programme annoté et le traduit en une représentation intermédiaire sous la forme d'un programme Boogie. Ce programme traduit est ensuite passé au prouveur de théorèmes Z3 [DMB08] qui vérifie si les annotations sont satisfaites sur le programme initial. Dans le cas contraire, VCC fournit un contre-exemple.

1.2.5.2 Extraction de LTS

Dans les travaux de [GJMS12], les auteurs décrivent une méthode d'extraction d'un LTS à partir d'un code C. Le LTS produit est ensuite analysé à travers la boîte à outils CADP [GLMS13] qui fournit une panoplie d'outils dédiés à l'analyse des LTS. Dans ce dernier travail, le sous-ensemble considéré inclut les pointeurs, l'arithmétique des pointeurs, les références et l'allocation dynamique de mémoire. Mais il ne supporte pas l'appel à des fonctions récursives.

1.2.5.3 K-maude

Maude [CDE+02] est un environnement de programmation basé sur la logique de réécriture [Mes92] (« *rewriting logic* » en anglais). Il permet grâce à un langage appelé Maude de définir des types de données algébriquement et de spécifier la dynamique d'un système à travers des règles appelées des *règles de réécriture*. Maude offre un ensemble d'outils comme un *model-checker* explicite pour vérifier des propriétés LTL et un prouveur de théorèmes.

Une extension de Maude est l'environnement $\mathbb{K}$-maude (appelé aussi $\mathbb{K}$) [RŞ10], spécialisée pour la spécification de la sémantique des langages de programmation. Les travaux de [ER12] proposent une méthode de spécification et de vérification d'un sous-ensemble C utilisant $\mathbb{K}$-maude. Les propriétés vérifiées sur les programmes sont des propriétés LTL. Mais, ces travaux se limitent à des programmes séquentiels.

L'avantage de $\mathbb{K}$-maude est l'expressivité due à la logique de réécriture d'une part et la disponibilité d'outils de vérification d'autre part. Néanmoins, la syntaxe de Maude n'est pas intuitive et sa sémantique requiert une expertise dans la logique de réécriture. Par exemple, dans les travaux cités plus haut, la spécification de la sémantique d'un sous-ensemble C a nécessité la définition de 1163 règles, ce qui est considérable.

1.3 RÉSUMÉ ET DISCUSSION

Catégorie	Outil	Concurrence	Sûreté	Vivacité	Méthode
CEGAR	BLAST [HJMS03]	✓	✓		- abstraction par prédicats - preuve de théorèmes - *model-checking* symbolique
	SLAM [BR01]	✓	✓		- abstraction par prédicats - abstraction paresseuse - *model-checking* symbolique - preuve de théorèmes
	SMACK [HCE+15]	✓	✓		- traduction en LLVM IR - solveur SMT
Interprétation abstraite	Astrée [BCC+03]		✓		- preuve de théorèmes
	Frama-C [CKK+12]	✓	✓		- annotation du code - preuve de théorèmes
Model-checking borné	CBMC [CKL04]		✓		- *model-checking* symbolique - solveur SAT
	CSeq [INT+16]	✓	✓		- séquentialisation - analyse accessibilité avec changements de contexte borné
Model-checking explicite ou symbolique	Modex [KJ09] & C2Promela [HS99]	✓ ✓	✓	✓	- traduction vers Promela utilisant des règles de traduction
	Cubicle	✓	✓		- traduction en un système de transitions paramétrés - démonstrateur SMT

TABLE 1.1 – Vue d'ensemble sur les outils de vérification formelle de programmes C

Nous résumons dans la table 1.1 les outils de vérification de programmes C. Pour chaque outil, nous indiquons s'il supporte la concurrence, les propriétés de sûreté et de vivacité et enfin les méthodes et techniques utilisées pour la vérification formelle.

Les outils basés sur l'interprétation abstraite sont quasiment automatiques et ils permettent de vérifier des logiciels de grande taille (plusieurs centaines de milliers de lignes de code). Un avantage de Frama-C est qu'il dispose d'une architecture modulaire permettant d'intégrer de nouveaux modules d'analyse. Cependant, les outils d'analyse

statique par interprétation abstraite ne permettent d'analyser que des propriétés génériques, à savoir les erreurs d'exécution comme la division par zéro et le débordement de tableau. En ce qui concerne AstréeA, son objectif pour la vérification des programmes concurrents se limite à la détection des accès concurrents aux sections critiques et à la détection des mauvaises utilisations des verrous. D'un autre côté, comme ces techniques intègrent des assistants de preuves, le choix des invariants de preuve est crucial. Il faut choisir un invariant qui ne soit ni trop faible pour mener à bien la preuve et ni trop fort en vue de son maintien.

Par ailleurs, puisque les techniques d'analyse statique sont basées sur une sur-approximation du comportement du programme, des fausses alarmes peuvent être déclenchées et il s'avère difficile de les éliminer. Dans Astrée, il est possible de paramétrer le niveau de précision de la sur-approximation. Mais cela implique un coût, à savoir une vitesse d'analyse très lente.

SLAM et BLAST utilisent des preuves de théorèmes pour générer des abstractions et des BDD pour le *model-checking*. Mais la difficulté de ces techniques consiste dans la phase d'identification des prédicats pour construire l'abstraction.

Pour les techniques utilisant des *model-checker* explicites, on trouve Modex et C2Promela qui profitent de l'outil Spin car celui-ci intègre plusieurs techniques de réduction comme la réduction d'ordre partiel et la compression d'états. Le modèle construit par Modex est une abstraction construite en se basant sur une table de règles de traduction définie par l'utilisateur. Néanmoins, Modex et C2Promela supportent un sous-ensemble très restreint de C. Ces outils ne gèrent pas l'arithmétique des pointeurs qui constitue un aspect important dans un code bas niveau.

Par rapport à l'interprétation abstraite et les techniques utilisant l'approche CEGAR, l'erreur détectée par le *model-checking* correspond à un cas d'erreur réel. De plus, l'utilisateur du *model-checker* n'a pas besoin de construire une preuve de correction, ce qui est souvent une tâche ardue.

En ce qui concerne Cubicle, il est limité à la vérification de la sûreté d'implémentation des barrières de synchronisation.

À la différence du *model-checking* explicite et symbolique, le *model-checking* borné est une technique qui ne souffre pas du problème d'explosion combinatoire. Il peut ainsi traiter des variables avec des milliers d'états, et ceci revient à l'utilisation des solveurs SAT. Néanmoins, le *model-checking* borné ne permet pas de prouver l'absence d'erreurs, il permet seulement de détecter les erreurs. Les deux outils MU-CSeq et Lazy-CSeq n'utilisent que des model checker bornés, ils assurent alors une correction partielle du programme vu qu'ils ne vérifient l'absence d'erreurs que pour une borne donnée.

Plusieurs outils que nous avons présentés dans la section précédente (à savoir Frama-C, CBMC et Astrée) ne permettent pas la vérification des propriétés de vivacité, une classe de propriétés importantes des systèmes concurrents et réactifs. On trouve l'outil SPIN et son langage Promela, qui permettent d'exprimer de telles propriétés. Mais

Promela est un langage non formel proche d'un langage de programmation. Il ne fournit pas des structures de données abstraites comme les listes ou les ensembles.

Dans cette thèse nous proposons une méthodologie de vérification par *model-checking* en se basant sur la traduction de la sémantique de C en une logique. Cette logique doit permettre d'exprimer la concurrence d'une part et des propriétés de sûreté et de vivacité d'autre part. Elle doit être accessible à un utilisateur standard et fournir des outils de vérification. Notre choix s'est porté sur la logique TLA.

Donner une sémantique pour le langage C est un sujet qui a été abordé dans la littérature. Ainsi, le projet CompCert [Ler15] est une chaîne de compilation pour le C certifiée formellement dans le domaine de l'embarqué critique, initiée en 2005 et entrepris à INRIA Rocquencourt. Le processus de compilation de CompCert utilise de multiples transformations du code afin de simplifier le code initial, comme l'élimination des expressions à effets de bord et la simplification des boucles itératives. Cette normalisation du code est opérée par CIL [NMRW02], un analyseur C qui fournit un AST du programme. Le code normalisé par CIL subit ensuite plusieurs transformations successives, donnant naissance à une représentation intermédiaire, écrite en Cminor. C'est à partir de ce langage que se fait la traduction en un code assembleur destiné à des processeurs PowerPC, ARM et x86.

Il est à noter que l'autre objectif du projet CompCert est de s'assurer que les optimisations opérées par un compilateur ne produisent pas des erreurs et que cette optimisation préserve la sémantique du code initial. Le modèle mémoire employé dans CompCert est un modèle adapté pour faciliter l'écriture de la preuve de préservation de la sémantique lors de la compilation du programme. Ce modèle est décrit dans [LB08].

Dans le contexte de la vérification par TLA, Lamport a proposé un langage appelé PlusCal [Lam09] pour exprimer des algorithmes parallèles et distribués. Le code PlusCal est traduit en TLA+ pour être vérifié ensuite par le *model-checker* TLC. PlusCAL dispose d'une syntaxe proche du C. Il souffre de certaines limitations. En effet, les variables dans PlusCAL peuvent être des variables globales, locales ou des paramètres. Mais le générateur de PlusCAL ne fait pas la distinction entre les types de variables dans la traduction TLA+. Une amélioration de PlusCal, nommée PlusCal-2 [AMQ10] a été proposée. Cette dernière propose de nouveaux constructeurs comme la spécification du grain d'atomicité des instructions et la possibilité de modéliser des processus imbriqués.

Dans cette thèse, nous avons envisagé une traduction de C en PlusCAL. Mais PlusCal ne supporte ni l'appel de fonction ni la représentation des pointeurs. En plus il n'est plus maintenu depuis 2011.

1.4 Conclusion

Dans ce chapitre, nous avons d'abord présenté les méthodes de spécification et vérification formelle. Nous avons classé ces méthodes en trois catégories : *model-checking*,

interprétation abstraite et vérification déductive. Puis, nous nous sommes intéressés aux techniques de vérification des programmes C. Ce thème constitue un domaine actif de recherche et de nombreux outils combinant plusieurs méthodes de vérification ont été proposés. Enfin, nous avons décrit les outils les plus connus du domaine et nous avons ensuite discuté leurs avantages et inconvénients.

Après cette étude bibliographique, nous consacrons le chapitre suivant à une présentation détaillée du cadre formel qui fait l'objet de notre travail, à savoir la logique TLA.

2

La logique TLA et le langage TLA+

Dans ce chapitre, nous présentons le cadre logique sur lequel s'appuie notre travail, à savoir la logique TLA (« *Temporal Logic of Actions* ») [Lam94] proposée par Leslie Lamport. Cette logique a été créée afin de raisonner sur les systèmes réactifs et concurrents en utilisant un formalisme simple pour les spécifier et formuler leurs propriétés.

Nous commençons par décrire dans la section 2.1 la logique TLA ainsi que sa sémantique. Nous présentons ensuite, dans la section 2.2, la syntaxe du langage TLA+. La section 2.3 décrit le *model-checker* de TLA+, appelé TLC (« *Temporal Logic Checker* »). Enfin, nous présentons dans la section 2.4 un aperçu des travaux réalisés autour de TLA+.

2.1 La logique TLA

TLA est une logique proposée par Leslie Lamport [Lam94] pour décrire les systèmes réactifs. Elle est une extension de la logique temporelle linéaire en une logique des actions. La syntaxe de TLA est résumée dans le listing 2.1. Les formules TLA sont formées à partir d'opérateurs booléens de la logique des prédicats, d'opérateurs arithmétiques, d'opérateurs ensemblistes et d'opérateurs de la logique temporelle linéaire LTL [SC85].

$\langle formule \rangle$	$\triangleq$ $\langle prédicat \rangle \mid \Box[\langle action \rangle]_{\langle fonction\ d'état \rangle} \mid \neg\langle formule \rangle$ $\mid \langle formule \rangle \wedge \langle formule \rangle \mid \Box\langle formule \rangle$
$\langle action \rangle$	$\triangleq$ expression à valeur booléenne comprenant des constantes et des variables primées, et non primées
$\langle prédicat \rangle$	$\triangleq$ $\langle formule \rangle$ sans variables primées | ENABLED $\langle action \rangle$
$\langle fonction\ d'état \rangle$	$\triangleq$ expression générale contenant des constantes et des variables

Listing 2.1 – Syntaxe de TLA [Lam02]

2.1.1 Notion de variable et d'état

L'ensemble des valeurs est noté **Val**. Il comprend les nombres comme -1 et 14, les chaînes de caractères comme "abc" et des ensembles, comme l'ensemble Nat des entiers naturels (notation TLA). L'ensemble des variables est noté par **Var**.

La sémantique de la logique d'actions de TLA est basée sur la notion d'état. Un état correspond à une valuation des variables. Il s'agit d'une fonction de l'ensemble de variables **Var** vers l'ensemble de valeurs **Val**. L'ensemble de tous les états possibles est l'ensemble de toutes les valuations possibles des variables, noté **St**.

Lamport note par $[\![F]\!]$ l'interprétation sémantique d'un objet syntaxique F (qui peut être un prédicat, une action ou une formule). Par exemple, pour une variable x, l'interprétation sémantique $[\![x]\!]$ est une fonction qui affecte à la variable x une valeur dans **Val**. La valeur de x dans l'état s est notée par $s[\![x]\!]$ (notation postfixée utilisée par Lamport).

2.1.2 Notion de fonction d'état et de prédicat

Une fonction d'état (« *state function* ») est une expression générale, formée de variables et de constantes. L'interprétation sémantique $[\![f]\!]$ d'une fonction d'état f est une fonction d'un ensemble d'états **St** vers l'ensemble de valeurs **Val**. La définition de $s[\![f]\!]$ est donnée comme suit :

$$s[\![f]\!] \triangleq f(\forall\ v : s[\![v]\!]/v)$$

La fonction $f(\forall\ v : s[\![v]\!]/v)$ substitue chaque occurrence de la variable v par $s[\![v]\!]$. Par exemple, $s[\![x + y^2]\!]$ associe à chaque état s la valeur retournée par $s[\![x]\!] + (s[\![y]\!])^2$.

Un prédicat d'état (ou prédicat) P est une expression à valeur booléenne formée de variables et constantes. Il s'agit d'une fonction d'état telle que $s[\![P]\!]$ assigne la valeur vrai ou faux à l'état s.

2.1.3 Notion d'action

Une action est une expression à valeur booléenne contenant des variables primées et/ou non primées et des constantes. Elle décrit la transition entre l'état courant et l'état

suivant. Une variable non primée représente la valeur d'une variable à l'état courant alors qu'une variable primée représente la valeur d'une variable à l'état suivant. Un exemple d'action est la formule $x' = x - y$. Cette action spécifie que la valeur de la variable x à l'état suivant sera décrémentée de la valeur de la variable y à l'état courant.

Formellement, l'interprétation sémantique $[\![\mathcal{A}]\!]$ d'une action $\mathcal{A}$ est une fonction qui associe à une paire d'états (s, t) un booléen noté par $s[\![\mathcal{A}]\!]t$, avec s et t représentent respectivement l'état courant et l'état suivant. Le booléen $s[\![\mathcal{A}]\!]t$ est défini comme suit :

$$s[\![\mathcal{A}]\!]t \triangleq \mathcal{A}(\forall v : s[\![v]\!]/v, t[\![v]\!]/v')$$

$s[\![\mathcal{A}]\!]t$ consiste à remplacer toutes les occurrences d'une variable non primée v par $s[\![v]\!]$ et toutes les occurrences d'une variable primée v' par $t[\![v]\!]$. Par exemple, $s[\![x' = x^2 + y]\!]t$ est égale au booléen $t[\![x]\!] = (s[\![x]\!])^2 + s[\![y]\!]$.

Pour toute action $\mathcal{A}$, où $vars$ est le tuple de variables apparaissant dans $\mathcal{A}$, on a :

- ENABLED $\mathcal{A}$ est le prédicat qui est vrai dans un état s s'il est possible d'atteindre un état t à partir de s. Par définition :

  $$s[\![\text{ENABLED } \mathcal{A}]\!] \triangleq \exists t \in \mathbf{St}\text{:} \; s[\![\mathcal{A}]\!]t$$

  Une paire d'états (s, t) est une $\mathcal{A}$ *transition* (« $\mathcal{A}$ *step* ») si l'exécution de l'action $\mathcal{A}$ permet de passer de l'état s à l'état t.

- Le prédicat UNCHANGED $vars$, avec $vars$ est un tuple de variables, est l'abréviation de $vars' = vars$. Il exprime que les variables $vars$ restent inchangées dans l'état suivant. Ce prédicat permet d'exprimer le *bégaiement*, qui correspond à une suite d'états où les valeurs des variables restent inchangées.

- La formule $[\mathcal{A}]_{vars}$ est l'abréviation de $\mathcal{A} \vee (vars' = vars)$. Cette formule est vraie si l'action $\mathcal{A}$ est exécutable ou si les valeurs des variables $vars$ sont inchangées.

- La formule $\langle\mathcal{A}\rangle_{vars}$ est l'abréviation de $\mathcal{A} \wedge (vars' \neq vars)$. Elle exprime que les variables $vars$ changeront de valeur en exécutant l'action $\mathcal{A}$.

2.1.4 Les formules temporelles

Une séquence infinie σ est une suite d'états, notée $\sigma = \langle s_0, s_1, s_2, \ldots\rangle$. Pour une formule temporelle φ, l'interprétation sémantique de $\sigma[\![\varphi]\!]$, notée aussi par $\langle s_0, s_1, s_2, \ldots\rangle[\![\varphi]\!]$, est la valeur booléenne que la formule φ assigne à σ. La séquence σ satisfait φ si et seulement si $\sigma[\![\varphi]\!]$ est vraie, notée aussi $\sigma \models \varphi$.

Une formule temporelle TLA ne peut être formée que par la combinaison de trois formes de formules temporelles suivantes :

- un prédicat d'état P est vrai pour un comportement σ si et seulement si P est vrai pour le premier état de σ.

- une formule $\Box P$, avec P est un prédicat, est vraie pour un comportement σ si P est vrai à chaque état de σ.

- une formule $\Box[\mathcal{A}]_{vars}$, avec $\mathcal{A}$ une action et $vars$ un tuple de variables apparaissant dans $\mathcal{A}$, est vraie pour un comportement σ si et seulement si pour chaque paire d'états successifs (s, t), l'action $[\mathcal{A}]_{vars}$ permet de passer de l'état s à l'état t.

Définition 2.1.1 (Équivalence par bégaiement (« *stuttering invariance* »)). *Soit $\sharp\sigma$ la séquence d'états obtenue à partir de σ en supprimant tous les états de bégaiement. Deux comportements σ_1 et σ_2 sont invariants par bégaiement si et seulement si $\sharp\sigma_1 = \sharp\sigma_2$.*

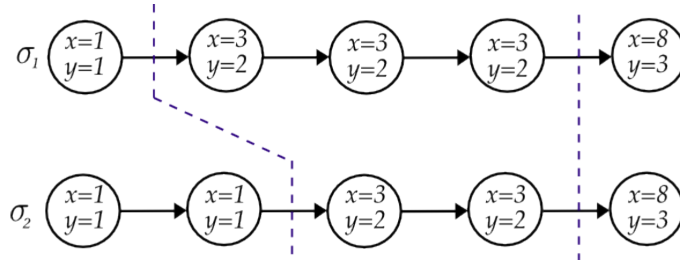


FIGURE 2.1 – Exemple de deux séquences d'états équivalentes par bégaiement

Exemple 2.1.1. *Nous illustrons dans la figure 2.1 un exemple de deux comportements σ_1 et σ_2 équivalents par bégaiement.*

Définition 2.1.2 (Invariance au bégaiement (« *stutter invariance* »)). *Une formule φ est dite invariante au bégaiement si pour tous les comportements σ_1 et σ_2 tels que $\sharp\sigma_1 = \sharp\sigma_2$, $\sigma_1 \models \varphi$ si et seulement si $\sigma_2 \models \varphi$.*

À partir de la définition 2.1.2, il est clair que les trois formes de formules temporelles TLA définies ci-dessus sont insensibles au bégaiement. Comme l'invariance au bégaiement est préservée par les opérateurs booléens et l'opérateur $\Box$, alors si on a φ et ψ deux formules temporelles invariantes au bégaiement, alors $\Box\varphi$, $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$ sont aussi invariantes au bégaiement.

Cette invariance est dûe au fait que l'opérateur « $\prime$ » (prime) de TLA n'est utilisé que pour des formules ayant une forme bien déterminée. Par exemple, la formule $\Box(x' = x + y)$ est une formule invalide dans TLA. Lamport a imposé cette restriction syntaxique afin que toutes les formules TLA soient insensibles au bégaiement. Ceci est essentiel afin d'exprimer dans TLA la relation de raffinement et de composition (pour plus de détails, se reporter à [Mer03]).

Théorème 2.1.1. *Toutes les formules temporelles TLA sont insensibles au bégaiement ([Lam02] page 90).*

Soient deux formules temporelles φ et ψ, $\sigma[\![\varphi \wedge \psi]\!]$ et $\sigma[\![\neg\varphi]\!]$ sont définies ainsi :

$$\sigma[\![\varphi \wedge \psi]\!] \triangleq \sigma[\![\varphi]\!] \wedge \sigma[\![\psi]\!]$$
$$\sigma[\![\neg\varphi]\!] \triangleq \neg\sigma[\![\varphi]\!]$$

Nous décrivons dans la suite la sémantique des opérateurs définis à partir des opérateurs booléens et de l'opérateur $\Box$.

Soit φ une formule d'états. Pour l'opérateur $\Diamond$ (se lit *inévitablement* ou *finalement*), la formule $\Diamond\varphi$ est définie comme suit :

$$\Diamond\varphi \triangleq \neg\Box\neg\varphi$$

Cette formule spécifie que φ n'est pas toujours fausse, c'est-à-dire φ est finalement vraie. Comme $\neg\forall\neg$ est égal à $\exists$, l'interprétation sémantique de $\langle s_0, s_1, s_2, \ldots\rangle[\![\Diamond\varphi]\!]$ est donnée par la formule suivante :

$$\langle s_0, s_1, s_2, \ldots\rangle[\![\Diamond\varphi]\!] \triangleq \exists n \in Nat\ :\ \langle s_n, s_{n+1}, s_{n+2}, \ldots\rangle[\![\varphi]\!]$$

Pour une formule d'états φ, une séquence $\langle s_0, s_1, s_2, \ldots\rangle$ satisfait $\Diamond\varphi$ si et seulement si φ est vraie dans un état de la séquence la séquence $\langle s_0, s_1, s_2, \ldots\rangle$.

À partir des opérateurs $\Box$ et $\Diamond$, il est possible d'exprimer qu'une formule φ est infiniment souvent (notée $\Box\Diamond\varphi$) ou finalement toujours vraie (notée $\Diamond\Box\varphi$). Ainsi, la formule $\Box\Diamond\varphi$ est vraie pour une séquence infinie $\sigma = \langle s_0, s_1, s_2, \ldots\rangle$ si et seulement si $\Diamond\varphi$ est vraie à tout instant n durant la séquence σ. Son interprétation sémantique est donnée ainsi :

$$\langle s_0, s_1, s_2, \ldots\rangle[\![\Box\Diamond\varphi]\!] \triangleq \forall n \in Nat\ :\ \exists m \in Nat : \langle s_{n+m}, s_{n+m+1}, s_{n+m+2}, \ldots\rangle[\![\varphi]\!]$$

De la même manière, la formule $\Diamond\Box\varphi$ spécifie que φ est finalement toujours vraie. Son interprétation sémantique est donnée comme suit :

$$\langle s_0, s_1, s_2, \ldots\rangle[\![\Diamond\Box\varphi]\!] \triangleq \exists n \in Nat\ :\ \forall m \in Nat : \langle s_{n+m}, s_{n+m+1}, s_{n+m+2}, \ldots\rangle[\![\varphi]\!]$$

TLA définit l'opérateur, noté $\leadsto$ (se lit « *leads to* »), tel que :

$$\varphi \leadsto \psi \triangleq \Box(\varphi \Rightarrow \Diamond\psi)$$

avec φ et ψ deux formules TLA. Cette formule spécifie que lorsque φ est vraie, alors la formule ψ est inévitablement vraie dans le futur, sans pour autant exiger que φ demeure vraie, contrairement à l'opérateur **U** (« *until* ») de la logique LTL [Pnu77].

Une formule φ est valide, notée $\models \varphi$, si et seulement si φ est satisfaite par toutes les séquences infinies d'états. Par définition :

$$\models \varphi \triangleq \forall\sigma \in \mathbf{St}^{\infty} : \sigma[\![\varphi]\!]$$

avec $\mathbf{St}^{\infty}$ est l'ensemble de toutes les séquences infinies d'états.

2.1.5 Quantification en TLA

TLA dispose d'opérateurs de quantification existentielle et universelle de la logique des prédicats. Pour une formule temporelle φ, un ensemble fini de constantes S, la formule $\forall x \in S : \varphi$ est vraie si et seulement si φ est vraie pour toutes les valeurs de x dans S. La formule $\exists x \in S : \varphi$ est vraie si et seulement si φ est vraie pour au moins un élément x dans S.

TLA définit des quantificateurs non bornés (« *unbounded quantifier* »), comme $\boldsymbol{\exists}$ et $\boldsymbol{\forall}$. Comme TLC ne supporte pas ce type de quantificateurs et vu que nous ne les utilisons pas, nous invitons le lecteur à consulter le chapitre 10 de [Lam02] pour plus de détails sur la sémantique de ces opérateurs.

2.1.6 Spécification TLA d'un système

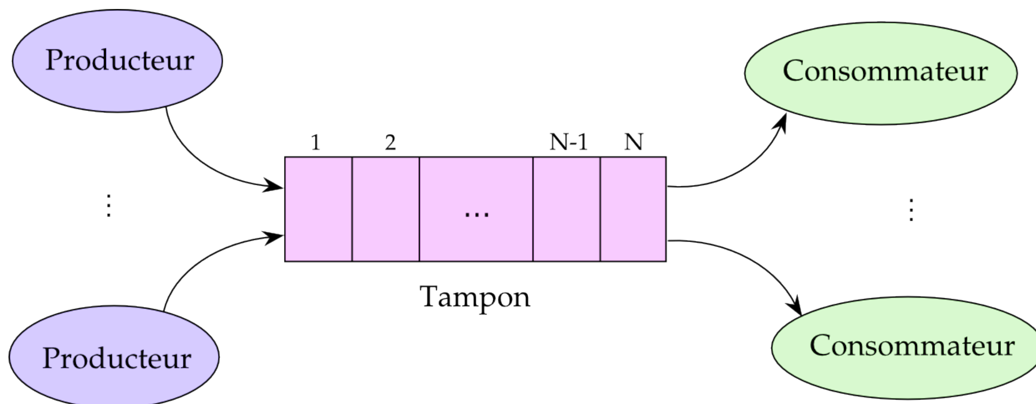


FIGURE 2.2 – Exemple de producteurs/consommateurs partageant un tampon borné

Pour illustrer comment spécifier un système en TLA, nous considérons un exemple de producteurs/consommateurs partageant un tampon borné de taille N (voir figure 2.2). Les producteurs remplissent le tampon avec des messages et les consommateurs utilisent ces derniers et les retirent. L'exemple doit satisfaire la spécification suivante : un producteur ne peut pas déposer un message lorsque le tampon est plein et un consommateur ne peut pas retirer un message quand le tampon est vide.

$$Put(m) \triangleq (Len(buffer) < Capacity) \land (buffer' = buffer \circ \langle m \rangle)$$

$$Get \triangleq (Len(buffer) \neq 0) \land (buffer' = Tail(buffer))$$

Listing 2.2 – Définition des actions $Put()$ et Get de l'exemple des producteurs/consommateurs

Pour spécifier cet exemple en TLA, nous considérons une variable $buffer$ qui modélise le tampon borné et une constante $Capacity$ qui représente la taille maximale du tampon.

L'action d'un producteur consiste en l'action $Put()$ et celle d'un consommateur en l'action Get. Nous définissons ces deux actions dans le listing 2.2. La variable $buffer$ est modélisée par une séquence (ou tuple). L'action $Put()$ prend en paramètre un message m et permet de l'insérer dans $buffer$ (l'insertion se fait en utilisant l'opérateur $\circ$ que nous décrivons dans la section 2.2.4). Cette action n'est exécutable que lorsque la taille du tampon donnée par $Len(buffer)$ est inférieure à la taille maximale $Capacity$.

L'action Get permet de retirer le premier message contenu dans le tampon. Le retrait d'un élément d'une séquence se fait par l'opérateur $Tail()$ (voir section 2.2.4). Cette action n'est exécutable que si la taille du tampon est différente de 0.

Maintenant, nous spécifions les transitions possibles du système, à savoir comment le système composé de la variable $buffer$ et des actions $Put()$ et Get peuvent changer l'état de la variable $buffer$.

$$\begin{aligned} bufferNext \triangleq\ & \exists\, p \in Producers : \exists\, m \in Data : Put(m) \\ & \vee \exists\, c \in Consumers : Get \end{aligned}$$

$$bufferInit \triangleq buffer = \langle\rangle$$

$$PCSpec \triangleq bufferInit \wedge \Box[bufferNext]_{buffer}$$

Listing 2.3 – Définition de l'action $bufferNext$ et $PCSpec$ des producteurs/consommateurs

Nous définissons dans le listing 2.3 le comportement des producteurs/consommateurs par la formule $PCSpec$. Nous utilisons deux ensembles (constants) $Producers$ et $Consumers$, qui désignent respectivement l'ensemble des producteurs et des consommateurs. La constante $Data$ désigne l'ensemble des messages à produire. L'action $bufferNext$ est exprimée par une disjonction de deux actions. Cette dernière est vraie soit lorsqu'il existe un producteur p parmi l'ensemble $Producers$ qui peut produire un message m, soit lorsqu'il existe un consommateur c qui peut exécuter l'action Get.

Une fois les transitions possibles du système sont spécifiées, nous modélisons l'exemple des producteurs/consommateurs à travers la formule temporelle $PCSpec$ qui exprime une conjonction de deux formules :

- un prédicat $bufferInit$ qui spécifie l'initialisation de la (ou les) variable(s) du système. Dans notre cas, le tampon $buffer$ est initialement vide ;

- la formule qui consiste à toujours exécuter l'action $bufferNext$ qui modélise les actions possibles du système.

La formule $PCSpec$ constitue une spécification TLA qui décrit l'ensemble des séquences d'états possibles du système. Cette spécification correspond à un *système de transitions étiqueté (STE)* défini ci-dessous.

Définition 2.1.3. *Un système de transitions étiqueté (STE) est un tuple* $\mathcal{T} = (\mathcal{Q}, \mathcal{I}, \mathcal{A}, \delta)$*, avec :*

- *$\mathcal{Q}$ est un ensemble fini d'états,*
- *$\mathcal{I} \subseteq \mathcal{Q}$ est l'ensemble d'états initiaux,*
- *$\mathcal{A}$ est l'ensemble d'actions,*
- *$\delta \subseteq \mathcal{Q} \times \mathcal{A} \times \mathcal{Q}$ est une relation de transition entre les états.*

Soit un système modélisé par une spécification TLA $\mathcal{S}$, de la forme $\mathcal{S} \triangleq Init \wedge \Box[Next]_{vars}$, où *Init* est le prédicat qui initialise toutes les variables du système, $Next$ est l'action qui spécifie les transitions possibles du système et $vars$ le tuple de variables que fait apparaître l'action $Next$. Le STE de la spécification TLA $\mathcal{S}$ noté par $\mathcal{T_S}$ est défini par $\mathcal{T_S} = (\mathcal{Q_S}, \mathcal{I_S}, \mathcal{A_S}, \delta_\mathcal{S})$, tel que :

- chaque état $s \in \mathcal{Q_S}$ correspond à une valuation du tuple des variables $vars$;
- les états initiaux $\mathcal{I_S}$ sont spécifiés par le prédicat *Init* ;
- une action $a \in \mathcal{A}$ est exécutable dans l'état s si et seulement si $(s, a, s') \in \delta_\mathcal{S}$ avec $s, s' \in \mathcal{Q_S}$,
- chaque transition $t \in \delta_\mathcal{S}$ correspond à la satisfaction de l'action $[Next]_{vars}$, qui est vraie si l'action $Next$ est exécutable ou si le tuple $vars$ reste inchangé. Ce dernier cas est appelé *pas de bégaiement* (« *stuttering step* » en anglais).

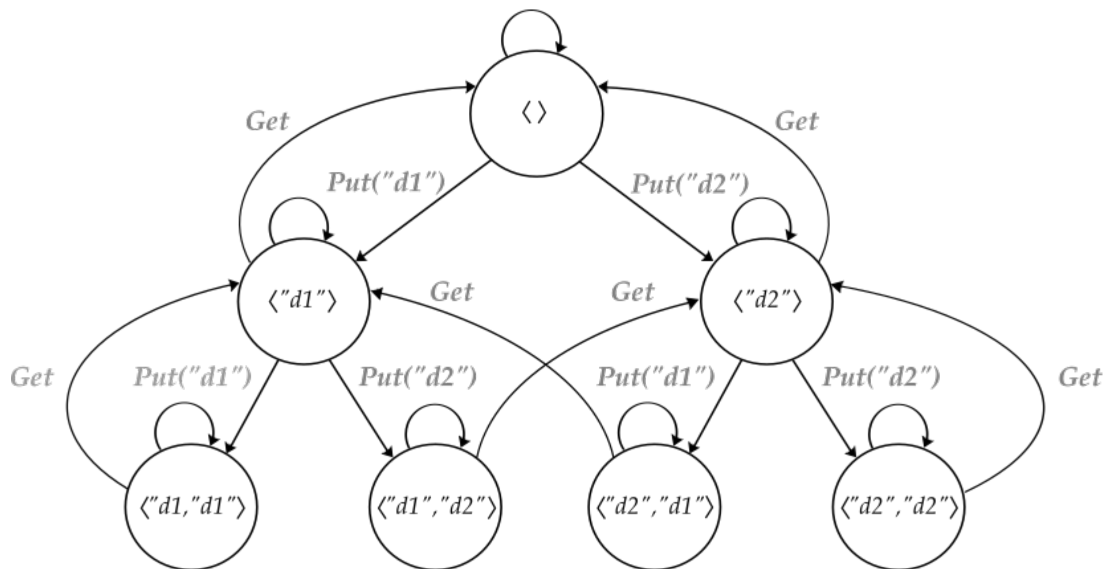


FIGURE 2.3 – STE correspondant à la spécification $PCSpec$ de l'exemple des producteurs/consommateurs

Reprenons l'exemple des producteurs/consommateurs (figure 2.2). Pour le cas d'un seul producteur et d'un seul consommateur, avec un tampon de capacité égale à 2 et un ensemble de messages $Data \triangleq \{\text{“d1”}, \text{“d2”}\}$, le STE de la spécification $PCSpec$ est représenté par la figure 2.3. Chaque état de ce STE est une valuation de la variable $buffer$. L'état initial correspond à l'état où la variable $buffer$ est vide et chaque transition entre deux états différents correspond à l'exécution de l'une des actions $Put()$ ou Get. Chaque

boucle sur un état correspond au cas où la variable $buffer$ reste inchangée.

2.1.7 Expression des propriétés dans TLA

Dans TLA, on peut exprimer des propriétés de sûreté, de vivacité et d'équité.

2.1.7.1 Propriété de sûreté

Une propriété de sûreté exprime que quelque chose de mauvais n'arrivera jamais. L'invariance est une propriété de sûreté qui s'exprime par la formule $\Box\varphi$ avec φ étant un prédicat. Étant donné une spécification $\mathcal{S}$, vérifier que $\Box\varphi$ est satisfaite revient à prouver que l'implication $\mathcal{S} \Rightarrow \Box\varphi$ est vraie. Par exemple, la formule $\Box(Len(buffer) \leq Capacity)$ est une propriété qui est vraie pour tous les états de la spécification $PCSpec$.

2.1.7.2 L'équité

L'équité permet de spécifier si une action est exécutable alors elle sera exécutée dans le futur. Lamport distingue deux types d'équité, l'équité faible et l'équité forte.

- L'équité faible exprime le fait que si une action $\mathcal{A}$ est autorisée indéfiniment alors elle sera exécutée infiniment. L'équité faible est exprimée par la formule $WF_{vars}(\mathcal{A}) \triangleq \Diamond\Box\ \text{ENABLED}\ \langle\mathcal{A}\rangle_{vars} \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_{vars}$.
- L'équité forte est exprimée par $SF_{vars}(\mathcal{A}) \triangleq \Box\Diamond\ \text{ENABLED}\ \langle\mathcal{A}\rangle_{vars} \Rightarrow \Box\Diamond\langle\mathcal{A}\rangle_{vars}$. Elle spécifie que si une action $\mathcal{A}$ est autorisée infiniment souvent alors elle sera exécutée infiniment souvent.

Une propriété d'équité forte implique la propriété d'équité faible correspondante. Elle est donc moins facile à obtenir que la propriété d'équité faible.

2.1.7.3 Propriété de vivacité

Une propriété de vivacité exprime que quelque chose de bon finira par avoir lieu. Pour l'exemple des producteurs/consommateurs, une propriété de vivacité $Liveness$ consiste à exprimer que lorsque le tampon est non vide, un consommateur doit être capable de retirer un message. Cette propriété s'exprime en TLA par $buffer \neq \langle\rangle \Rightarrow \exists c \in Consumers : \Box\Diamond\langle Get\rangle_{buffer}$.

Comme la vivacité s'exprime sur des comportements infinis, la propriété $Liveness$ n'est pas vérifiée sur la spécification $PCSpec$. Elle n'est satisfaite qu'en spécifiant une condition d'équité. La propriété $Liveness$ est vraie pour la spécification $PCSpec \triangleq bufferInit \land \Box[bufferNext]_{buffer} \land WF_{buffer}(bufferNext)$.

2.2 LE LANGAGE TLA+

Le langage TLA+ est un langage formel de spécification qui étend la logique TLA par une structuration en modules.

Dans cette section, nous décrivons la structure d'un module TLA+. Nous détaillons ensuite la syntaxe de TLA+ en décrivant les constructions et les opérateurs standard du langage que nous utilisons dans notre travail. La table 2.1 illustre un résumé des opérateurs TLA+.

2.2.1 Structure d'un module TLA+

TLA+ introduit la notion de modules pour structurer les spécifications. La structure d'un module TLA+ de l'exemple des producteurs/consommateurs est illustrée par le listing 2.4.

———— MODULE *ProducersConsumers* ————

EXTENDS *Naturals*, *Sequences*

CONSTANTS *Capacity*, *Data*, *Producers*, *Consumers*

VARIABLES *buffer*

$$Put(m) \triangleq \land\ Len(buffer) < Capacity$$
$$\qquad \land\ buffer' = buffer \circ \langle m \rangle$$

$$Get \triangleq \land\ Len(buffer) \neq 0$$
$$\qquad \land\ buffer' = Tail(buffer)$$

$$bufferNext \triangleq \lor\ \exists\, p \in Producers : \exists\, m \in Data : Put(m)$$
$$\qquad \lor\ \exists\, c \in Consumers : Get$$

$$bufferInit \triangleq buffer = \langle\rangle$$

$$PCSpec \triangleq bufferInit \land \Box[bufferNext]_{buffer} \land WF_{buffer}(bufferNext)$$

$$Invariant \triangleq Len(buffer) \leq Capacity$$

$$Liveness \triangleq buffer \neq \langle\rangle \Rightarrow \exists c \in Consumers : \Box\Diamond\langle Get\rangle_{buffer}$$

————————————————

Listing 2.4 – Module TLA+ de l'exemple des producteurs/consommateurs

Le début du module est marqué par une ligne horizontale contenant le nom du module, à savoir *ProducersConsumers* et la fin du module est marquée aussi par une ligne horizontale. Ces derniers sont nécessaires et n'ont pas de signification particulière.

Le mot clé EXTENDS exprime que le module courant étend les modules *Naturals* et *Sequences*. Étendre un module permet d'accéder aux variables, constantes et prédicats définis dans les modules étendus.

Un module TLA+ est composé d'une déclaration de paramètres, à savoir les constantes

Logique

TRUE FALSE $\wedge$ $\vee$ $\neg$ $\Rightarrow$ $\equiv$

Ensembles

$\neq$ $\in$ $\notin$ $\cap$ $\cup$ $\subset$ $\setminus$	[Opérateurs ensemblistes]
$\{e_1, \ldots, e_n\}$	[Ensemble consistant en les éléments e_i]
$x \in S : p$	[Ensemble des éléments x dans S satisfaisant p]
$e : x \in S$	[Ensemble des éléments e tels que x est dans S]
SUBSET S	[Ensemble des sous-ensembles de S]
UNION S	[Union de tous les éléments de S]

Fonctions

$f[e]$	[Application de la fonction f à un élément e]
DOMAIN f	[Domaine de la fonction f]
$[x \in S \mapsto e]$	[Fonction f telle que $f[x] = e$ pour $x \in S$]
$[S \rightarrow T]$	[Ensemble des fonctions f avec $f[x] \in T$ pour tout $x \in S$]
$[f$ EXCEPT $![e_1] = e_2]$	[Fonction $\hat{f}$ égale à f telle que $\hat{f}[e_1] = e_2$]
$\{[f$ EXCEPT $![e] \in S]\}$	[Ensemble des fonctions $\hat{f}$ égales à f telles que $\hat{f}[e] \in S$]

Enregistrements

$e.h$	[Le champ h de l'enregistrement e]
$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$	[L'enregistrement dont le champ h_i est égal à e_i]
$\{[h_1 : S_1, \ldots, h_n : S_n]\}$	[Ensemble des tous les enregistrements tels que chaque champ h_i est dans l'ensemble S_i]
$[r$ EXCEPT $!.h = e]$	[Enregistrement $\hat{r}$ égal à r tel que $\hat{r}.h = e$]
$\{[r$ EXCEPT $!.h] \in S]\}$	[Ensemble des enregistrements r égaux à r tels que $r.h \in S$]

Séquences

$e[i]$	[Le $i^{ième}$ élément de la séquence e]
$\langle e_1, \ldots, e_n \rangle$	[Le n-uplet dont le $i^{ième}$ élément est e_i]
$S1 \times \ldots \times S_n$	[Ensemble de tous les n-uplets tels que le $i^{ième}$ élément est dans l'ensemble S_i]

Chaînes

"$\mathtt{c}_1 \ldots \mathtt{c}_n$"	[Une chaîne littérale de n caractères]
STRING	[L'ensemble de toutes les chaînes]

Divers

IF pTHEN e_1 ELSE e_2	[Égal à e_1 si p est vrai, sinon e_2]
CASE $p_1 \rightarrow e_1 \Box \ldots \Box\ p_n \rightarrow e_n$	[Égal à e_i si p_i est vrai]
LET $d_1 \triangleq e_1 \ldots d_n \triangleq e_n$ IN e	[Égal à e dans le contexte des définitions $d_1, \ldots, d_n$]

$\wedge$	p_1	[Conjonction de $p1 \wedge \ldots \wedge p_n$]	$\vee$	p_1	[Disjonction de $p1 \vee \ldots \vee p_n$]
$\vdots$			$\vdots$		
$\wedge$	p_n		$\vee$	p_n	

TABLE 2.1 – Résumé des opérateurs TLA+

(spécifiées par le mot clé CONSTANTS) et les variables (spécifiées par le mot clé VARIABLES). Le module définit ensuite les prédicats *bufferInit* et l'action *bufferNext*. Le module spécifie les formules temporelles, à savoir la formule *PCSpec* qui définit les transitions du système et les propriétés à vérifier.

2.2.2 Définition des fonctions

Une fonction f possède un domaine de définition, noté DOMAIN f. L'expression $f[v]$ spécifie la valeur de la fonction f pour l'élément v si v est un élément du DOMAIN f. Le constructeur d'une fonction f s'écrit comme suit :

$$[x \in S \mapsto e]$$

Le domaine de f est S et $f[v]$ égale la valeur obtenue en substituant x par v dans l'expression e. Par exemple,

$$[n \in Nat \mapsto n + 1]$$

est la fonction telle que $f[0] = 1, f[1] = 2$, ...

TLA+ définit l'opérateur EXCEPT qui s'utilise comme suit :

$$[f \text{ EXCEPT } ![u] = a, ![v] = b]$$

Cette expression renvoie une fonction $\hat{f}$ ayant les mêmes arguments et les mêmes valeurs de retour que la fonction f, à l'exception des arguments u et v pour lesquels on a $\hat{f}[u] = a$ et $\hat{f}[v] = b$.

Le module standard TLC définit les opérateurs :> et @@ qui pourront être utilisés pour représenter les fonctions. Ils sont définis comme suit :

$$d :> e \triangleq [x \in \{d\} \mapsto e]$$
$$f @@ g \triangleq [x \in (\text{DOMAIN } f) \cup (\text{DOMAIN } g) \mapsto \text{IF } x \in \text{DOMAIN } f \text{ THEN } f[x] \text{ ELSE } g[x]]$$

Utilisant :> et @@, l'expression

$$d_1 :> e_1 @@ d_2 :> e_2 @@ \ldots @@ d_{n-1} :> e_{n-1} @@ d_n :> e_n$$

est une fonction f ayant comme domaine $\{d_1, d_2, \ldots, d_{n-1}, d_n\}$ avec $f[d_i] = e_i$ pour $i \in 1..n$. Par exemple, la séquence $\langle$"a", "b"$\rangle$ est une fonction ayant le domaine $\{1, 2\}$ et qui peut s'écrire de la forme $\langle 1 :>$ "a" @@ $2 :>$ "b"$\rangle$. La définition TLA+ du module standard TLC est donnée dans le livre [Lam02] à la page 248.

2.2.3 Les enregistrements

Un enregistrement r s'écrit sous la forme :

$$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$$

dont chaque champ (ou composante) h_i est égal à e_i. L'accès à un champ se fait par l'expression $r.h_i$. Dans TLA+, un enregistrement est une fonction dont le domaine

définition est un ensemble fini de chaînes de caractères. Pour cela, la valeur du champ $r.h_i$ correspond à la valeur de $r[\text{"h}_i\text{"}]$.

Le constructeur EXCEPT, défini dans la section 2.2.2 peut être utilisé sur les enregistrements. Par exemple,

$$[r \text{ EXCEPT } !.a = e]$$

est l'enregistrement $\hat{r}$ ayant les mêmes champs et les mêmes valeurs de retour que r, à l'exception du champ a pour lequel $\hat{r}.a = e$.
Ainsi, les deux expressions suivantes sont équivalentes :

$$[un \rightarrow 1, deux \rightarrow 2] \qquad [i \in \{\text{"un"}, \text{"deux"}\} \rightarrow \text{IF } i = \text{"un"} \text{THEN } 1 \text{ ELSE } 2]$$

2.2.4 Les séquences

Une séquence (ou un tuple) est une fonction TLA+ qui s'écrit sous la forme $\langle e_1, \ldots, e_n \rangle$ et dont le domaine de définition est l'ensemble $\{1, \ldots, n\}$, avec $\langle e_1, \ldots, e_n \rangle[i] = e_i$. Les index des séquences commencent à 1. Le 0-uplet est la séquence vide, notée $\langle\rangle$.

Plusieurs opérateurs sur les séquences sont définis dans le module standard *Sequences*. Le listing 2.5 présente les principaux opérateurs sur les séquences.

$Seq(S)$	:	l'ensemble de toutes les séquences construites à partir des éléments de S
$Len(S)$	:	la longueur de la séquence S
$S \circ T$	:	la séquence résultante de la concaténation de S et T
$Head(S)$	:	le premier élément de S
$Tail(S)$	:	la séquence S en omettant le premier élément $S[1]$
$Append(S, e)$	:	la séquence résultante de l'ajout de l'élément e à la séquence S
$SubSeq(S, m, n)$	:	la séquence $\langle S[m], S[m+1], \ldots, S[n] \rangle$

Listing 2.5 – Les principaux opérateurs définis dans le module *Sequences*

2.2.5 Les opérateurs

Les opérateurs dans TLA+ s'écrivent sous les formes suivantes :

$$Op \triangleq expr \qquad\qquad Op(p_1, \ldots, p_n) \triangleq expr$$

avec Op un identificateur, $expr$ une expression et p_i un identificateur ou un opérateur de la forme $Op(_, \ldots, _)$.

Par exemple, ci-dessous est la définition de l'opérateur $Tail()$:

$$Tail(s) \triangleq [i \in 1..(Len(s) - 1) \mapsto s[i+1]]$$

A la différence d'une fonction, un opérateur n'a pas un domaine alors qu'une fonction doit avoir un domaine qui est un ensemble.

2.2.5.1 Les opérateurs booléens

TLA+ dispose d'opérateurs booléens de la logique propositionnelle :

$$\wedge \quad \vee \quad :\neg \quad \Rightarrow \quad \text{TRUE} \quad \text{FALSE}$$

La conjonction et la disjonction de formules peuvent s'écrire sous la forme de listes alignées comme suit :

$$\begin{array}{l} \wedge\, p_1 \\ \vdots \\ \wedge\, p_n \end{array} \quad \triangleq \quad p_1 \wedge \ldots \wedge p_n \qquad\qquad \begin{array}{l} \vee\, p_1 \\ \vdots \\ \vee\, p_n \end{array} \quad \triangleq \quad p_1 \vee \ldots \vee p_n$$

2.2.5.2 Les opérateurs ensemblistes

TLA+ dispose d'opérateurs de la théorie des ensembles. Les opérateurs basiques les plus utilisés sont les suivants :

$$\in \quad \notin \quad \cup \quad \cap \quad \subset \quad \setminus \quad \text{UNION} \quad \text{SUBSET}$$

TLA+ dispose aussi des constructeurs ensemblistes suivants :

- $\{e_1, e_2, \ldots, e_n\}$: définit un ensemble formé des éléments $e_1, e_2, \ldots, e_n$,
- $\{x \in S : p\}$: définit l'ensemble des éléments x de S satisfaisant la propriété p.

2.2.5.3 L'opérateur CHOOSE

La générale de l'opérateur CHOOSE s'écrit comme suit :

$$\text{CHOOSE } x \in S : p$$

Cette expression est égale à une valeur v de l'ensemble S, s'il existe une valeur v pour laquelle le prédicat p est vrai en substituant x par v. Par exemple, l'expression suivante :

$$min(S) \triangleq \text{CHOOSE } i \in S : \forall j \in S : i \leq j$$

retourne la première valeur minimale de l'ensemble des valeurs contenues dans S.

2.2.6 Constructeurs conditionnels

Inspiré des structures conditionnelles des langages de programmation, le langage TLA+ dispose de deux constructeurs conditionnels : IF/THEN/ELSE et CASE. La forme générale du constructeur IF/THEN/ELSE est :

$$\text{IF } p \text{THEN } e_1 \text{ ELSE } e_2$$

Cette expression est égale à e_1 si le prédicat p est vrai et à e_2 quand p est faux. Dans le cas où on dispose de plusieurs constructeurs IF/THEN/ELSE imbriqués, il est plus simple d'utiliser le constructeur CASE. Ce dernier possède deux formes syntaxiques :

$$\text{CASE } p_1 \rightarrow e_1 \Box \ldots \Box p_n \rightarrow e_n$$
$$\text{CASE } p_1 \rightarrow e_1 \Box \ldots \Box p_n \rightarrow e_n \Box \text{ OTHER } \rightarrow e$$

La sémantique de ces deux constructeurs est détaillée dans [Lam02] page 298.

2.2.7 Constructeur LET/IN

Le constructeur LET/IN sert à définir une expression locale. La forme générale de ce constructeur est la suivante :

$$\text{LET } d_1 \ldots d_n \text{ IN } e$$

avec d_i présente une définition TLA+. Cette expression est égale à e dans le contexte des définitions d_i. Par exemple

$$\text{LET } mult(i, j) \triangleq (i * j) \text{ IN } mult(1, 2) + mult(2, 4)$$

est égale à $(1 * 2) + (2 * 4)$, à savoir 10.

2.2.8 Les chaînes de caractères

Une chaîne de caractères est une séquence de caractères. Ainsi, la chaîne "hello" est équivalente à la séquence suivante : $\langle$"h", "e", "l", "l", "o"$\rangle$.

2.2.9 Les expressions λ

Une expression λ dans TLA+ est une expression qui permet de définir un ou des opérateurs locaux qui peuvent être passés comme arguments ou qui peuvent retourner des valeurs. La forme générale d'une expression λ est comme suit :

$$\lambda p_1, \ldots, p_n : expr$$

avec $p_1, \ldots, p_n$ sont des paramètres λ. Quand $n = 0$, l'expression $\lambda : expr$ n'est autre que l'expression $expr$.

Soit un opérateur Op défini par une expression λ tel que $Op \triangleq \lambda p_1, \ldots, p_n : expr$. $Op(e_1, \ldots, e_n)$ consiste à remplacer chaque paramètre λ p_i dans l'expression $expr$ par e_i, pour tout $i \in 1..n$.

Nous ne détaillons pas dans cette section la sémantique des expressions λ ainsi que les règles de substitution. Cette partie est détaillée dans [Lam02] page 325.

2.3 Le *model-checker* TLC

Le langage TLA+ est supporté par un ensemble d'outils, à savoir l'analyseur syntaxique SANY, un assistant de preuve TLAPS et un *model-checker* TLC. Ces outils sont

intégrés dans une boîte à outils TLA+ Toolbox, disponible en téléchargement sur le site web [Cor10].

TLC est un *model-checker* explicite, implémenté en Java. Il a été conçu par Yuan Yu et l'équipe de Lamport [YML99] pour la vérification d'une spécification TLA+. Il dispose de deux modes : mode simulation et mode *model-checking*. Dans le mode simulation, l'utilisateur fixe le profondeur de recherche dans l'espace d'états, à savoir le nombre maximal d'états composant un comportement. Dans ce cas, TLC permet de vérifier des comportements choisis de manière aléatoire.

Dans la suite de ce travail, nous nous intéressons qu'au mode *model-checking* de TLC. Ce dernier prend en entrée une spécification TLA+ sous la forme suivante :

$$Spec \triangleq Init \land \Box[Next]_{\langle v_1,\ldots,v_n\rangle} \land Fairness \tag{2.1}$$

La formule $Spec$ est composée d'un prédicat $Init$ qui définit l'état initial du système. L'action $Next$ définit les actions à exécuter sur les variables du système et $\langle v_1, \ldots, v_n\rangle$ spécifie le tuple contenant les variables impliquées dans la définition de l'action $Next$. Finalement, $Fairness$ est une formule optionnelle qui exprime des contraintes d'équité sur l'exécution des actions.

Pour vérifier la formule $Spec$, TLC prend en entrée le nom de la spécification à vérifier, les valeurs des constantes s'il y en a et les propriétés à vérifier sur la spécification. Reprenons l'exemple de producteurs/consommateurs, dont la définition TLA+ est donnée par le listing 2.4. Pour vérifier les propriétés $Invariant$ et $Liveness$ sur la spécification $PCSpec$, il faut spécifier dans TLA+ Toolbox le nom de la spécification, à savoir $PCSpec$, les valeurs des constantes $Capacity$, $Data$, $Producers$ et $Consumers$ ainsi que les propriétés. Nous illustrons dans la figure 2.4 une capture d'écran de l'interface homme machine de TLA+ Toolbox de cette initialisation.

2.3.1 Le sous-ensemble de TLA+ géré par TLC

TLC peut évaluer toute formule temporelle formée par des formules élémentaires dont chacune appartient à l'une des classes suivantes :

- un prédicat ;
- une formule de la forme $\Box[\mathcal{A}]_v$, avec $\mathcal{A}$ une action ;
- un invariant de la forme $\Box P$, avec P un prédicat ;
- une formule temporelle simple définie en utilisant les opérateurs booléens et les opérateurs temporels $\Box$, $\Diamond$ et $\leadsto$;
- une formule sur une action $\mathcal{A}$, de la forme $\mathrm{WF}_v(\mathcal{A})$, $\mathrm{SF}_v(\mathcal{A})$, $\Box\Diamond\langle\mathcal{A}\rangle_v$ et $\Diamond\Box[\mathcal{A}]_v$.

TLA+ Toolbox fournit un module appelé TLC contenant des opérateurs permettant le débogage des spécifications TLA+, comme l'opérateur spécial $Assert(P, out)$ qui permet

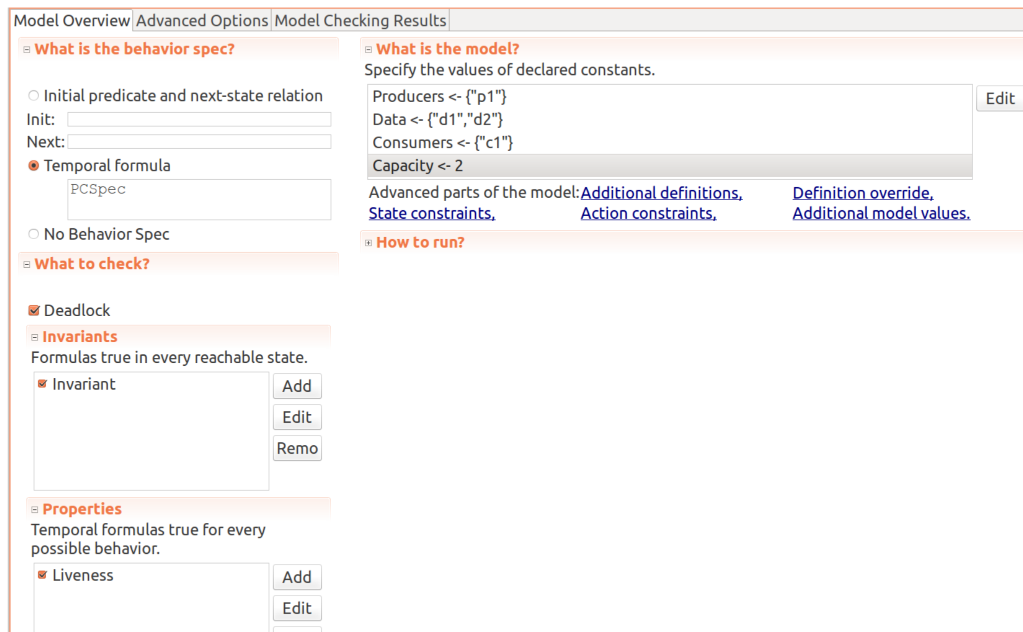


FIGURE 2.4 – Initialisation de TLC pour une instance de l'exemple des producteurs/consommateurs

de vérifier la validité d'un prédicat P. Si P est vrai alors $Assert()$ retourne vrai. Sinon, l'exploration de l'espace d'états s'interrompt et TLC reporte le message d'erreur out. Cet opérateur nous permet de traduire les fonctions C `assert()` contenues souvent dans un code C.

2.3.2 Évaluation des expressions

Dans TLC, l'ordre d'évaluation des expressions logiques se fait de *gauche à droite* :

- il évalue l'expression $p \wedge q$ en commençant par p. Si p est vraie, il évalue q.

- il évalue l'expression $p \vee q$ en commençant par p. Si p est fausse, il évalue q. L'évaluation de $p \Rightarrow q$ revient à l'évaluation de $\neg p \vee q$. Cette évaluation ne s'applique pas dans le cas où les expressions p et q contiennent des actions. L'évaluation d'une disjonction d'actions se fait différemment. Nous la décrivons dans la suite de ce paragraphe.

- il évalue IF pTHEN e_1 ELSE e_2 en commençant par évaluer p. Puis, évaluer e_1 ou e_2.

Pour les formules avec quantificateurs, TLC ne peut évaluer que des ensembles bornés. L'évaluation de la formule $\exists x \in S : p$ consiste à énumérer les éléments $s_1, \ldots, s_n$ de S et à évaluer ensuite p, en substituant x par s_i de manière successive pour tout i de 1 à n. Le principe d'évaluation est le même pour la formule $\forall x \in S : p$.

TLC peut évaluer des expressions qui contiennent des sous-expressions non gérées par TLC. Par exemple, il peut évaluer l'expression $[n \in Nat \mapsto n * (n+2)][4]$ qui retourne la valeur 24 mais il ne peut pas évaluer l'expression $[n \in Nat \mapsto n * (n+2)]$ qui correspond à une fonction dont le domaine est l'ensemble non borné Nat.

Pour les fonctions TLA+ définies de la forme $f[x \in S] \triangleq e$, la valeur $f[c]$ est évaluée en commençant par évaluer la partie droite de la formule, à savoir l'expression e.

L'évaluation d'une action de la forme $A_1 \vee \ldots \vee A_n$ consiste en n évaluations séparées. Prenons un exemple de spécification TLA+ avec deux variables v et u ayant respectivement, à un état donné, les valeurs 0 et $\langle 1, 2 \rangle$. Soit l'action suivante :

$$\begin{aligned} &\vee \wedge v' = v + 1 \\ &\quad \wedge u' = Append(u, v) \\ &\vee \wedge v' = 1 \\ &\quad \wedge u' = Append(Tail(u), v') \end{aligned} \tag{2.2}$$

L'évaluation cette action consiste à évaluer les deux actions suivantes séparément :

$$\begin{aligned} &\wedge v' = v + 1 \\ &\wedge u' = Append(u, v) \end{aligned} \tag{2.3}$$

$$\begin{aligned} &\wedge v' = 1 \\ &\wedge u' = Append(Tail(u), v') \end{aligned} \tag{2.4}$$

L'évaluation de l'action 2.3 retourne vrai en affectant à v la valeur 1 ensuite la séquence $\langle 1, 2, 0 \rangle$ à u et un nouveau état est créé. De même, l'évaluation de l'action 2.4 retourne vrai et crée un nouveau état tel que $v = 1$ et $u = \langle 1, 1 \rangle$. Il est à noter que l'ordre est important dans l'évaluation d'une action s'écrivant sous la forme d'une conjonction de formule. Par exemple, TLC ne peut pas évaluer l'action si elle était écrite sous cette forme :

$$\begin{aligned} &\wedge u' = Append(Tail(u), v') \\ &\wedge v' = 1 \end{aligned} \tag{2.5}$$

car TLC commence par évaluer la première formule qui contient l'expression $Append(Tail(u), v')$ alors qu'il n'a pas encore évalué la valeur de v'. Dans ce cas, TLC rapporte une erreur.

L'évaluation de l'expression UNCHANGED $\langle e_1, \ldots, e_n \rangle$ consiste à évaluer la conjonction $(e'_1 = e_1) \wedge \ldots \wedge (e'_n = e_n)$. Pour une spécification s'écrivant sous la forme 2.1, dans le cas où l'évaluation de l'action $Next$ retourne vrai et il existe une ou plusieurs variables qui n'ont pas été assignées, TLC reporte une erreur. Par conséquent, l'action $Next$ doit spécifier la nouvelle valeur de chacune des variables du tuple $\langle v_1, \ldots, v_n \rangle$. Si une variable v_i ne change pas de valeur, il faut le spécifier explicitement par UNCHANGED v_i.

2.3.3 Fonctionnement de TLC

TLC est un *model-checker* explicite. Il utilise des *hashs* (appelés aussi *fingerprints*) de 64-bits pour représenter les états d'une spécification TLA+. Un hash est une séquence de chiffres et lettres, générée par une fonction de *hachage*.

Soit une spécification $\mathcal{S}$ de la forme 2.1. Soit φ la conjonction de propriétés à vérifier sur $\mathcal{S}$. TLC décompose cette formule en l'écrivant comme une conjonction de la forme suivante : $\varphi \triangleq Invariants \wedge Liveness$, avec $Invariants$ et $Liveness$ désignent respectivement les invariants et les propriétés de vivacité à vérifier sur la spécification $\mathcal{S}$. Pour explorer l'espace d'états, TLC maintient deux structures de données [YML99] :

- une structure appelée $\mathcal{G}$ qui représente le graphe d'états. Elle stocke des nœuds et des arcs. Chaque nœud constituant le graphe $\mathcal{G}$ représente le hash d'un état exploré s, noté $h(s)$. Un arc correspond à une transition entre les hashs de deux états s et s', noté $h(s) \rightarrow h(s')$;
- une file appelée $\mathcal{U}$ qui contient les éléments de $\mathcal{G}$ dont le ou les successeurs ne sont pas encore explorés.

Initialement, $\mathcal{G}$ et $\mathcal{U}$ sont vides. Le principe d'exploration d'états par TLC suit les étapes suivantes :

1. générer les états initiaux en évaluant le prédicat $Init$. Pour chaque état initial s :
   (a) vérifier si s satisfait la formule $Invariants$. Sinon, TLC reporte une erreur et arrête son exploration ;
   (b) empiler l'état s dans $\mathcal{U}$ et insérer ensuite $h(s)$ et une transition $h(s) \rightarrow h(s)$ dans $\mathcal{G}$;
2. tant que $\mathcal{U}$ est non vide, exécuter les étapes suivantes :
   (a) retirer le premier élément s de la file $\mathcal{U}$;
   (b) générer l'ensemble T de tous les successeurs de s en évaluant l'action $Next$ (en suivant le principe d'évaluation des actions énoncé dans la section 2.3.2) ;
   (c) si T est vide, une erreur d'*interblocage* est reportée. Cette erreur correspond au cas où s n'a pas de successeur. Dans ce cas, l'exploration d'états s'arrête ;
   (d) sinon, pour chaque état t dans T, exécuter les étapes suivantes :
      (i) si t satisfait tous les invariants, alors
         - si $h(t)$ n'est pas dans $\mathcal{G}$, alors $h(t)$ est inséré dans la file $\mathcal{U}$. Il est ensuite inséré dans $\mathcal{G}$ avec la transition $h(t) \rightarrow h(t)$;

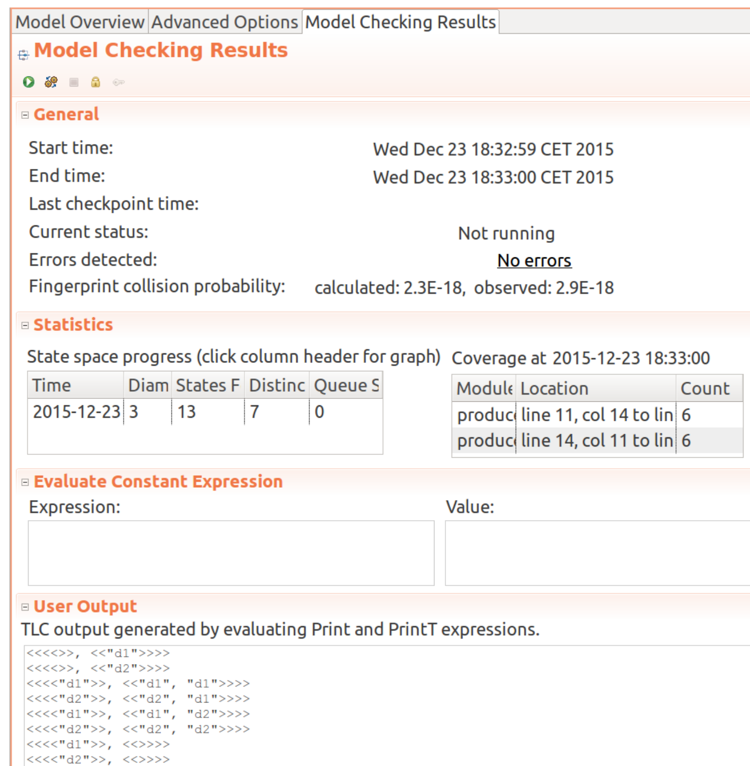


FIGURE 2.5 – Résultats du *model-checking* de la spécification $PCSpec$

- insérer la transition $h(s) \rightarrow h(t)$ dans $\mathcal{G}$;

(ii) sinon, une trace d'erreur est générée.

TLC vérifie à la volée si la propriété $Liveness$ est satisfaite. Ceci en vérifiant si toute séquence d'états infinie π, formée à partir du graphe $\mathcal{G}$, satisfait la formule $Fairness \Rightarrow Liveness$.

2.3.4 Résultats fournis par TLC

Nous illustrons dans la figure 2.5 l'interface graphique de TLA+ Toolbox montrant les résultats de vérification fournis par TLC pour l'exemple des producteurs/consommateurs. Cette interface est divisée en plusieurs *sections*. Nous présentons dans ce qui suit seulement les sections les plus importantes.

La section « *statistics* » est composée de deux tables. La première table indique des informations, générées périodiquement, sur l'espace d'états. Chaque ligne de cette table contient les informations suivantes :

- « *Time* » : le temps courant ;

- « *Diameter* » : le diamètre du graphe d'états, qui désigne la longueur de la séquence d'états la plus longue ;

- « *States Found* » : le nombre total d'états calculés jusqu'à présent. Ce nombre inclut les états initiaux et les successeurs des états générés. Il correspond à la taille de l'espace d'états vu comme un arbre ;

- « *Distinct States* » : le nombre d'états différents générés ;

- « *Queue Size* » : le nombre d'états (distincts) dont le ou les successeurs ne sont pas encore examinés.

La deuxième table contient des informations sur la couverture du code TLA+. Chaque ligne contient un lien vers une sous action de l'action $bufferNext$ ainsi que le nombre de fois qu'elle a été exécutée. Cette information est très utile pour détecter les portions du code qui ne sont jamais exécutées.

Dans la section « *general* », TLC indique :

- le moment auquel TLC a été lancé et le temps où la vérification est finie ;

- l'état de TLC (en exécution, pas en exécution, stoppé) ;

- les erreurs détectées (s'il y en a) ;

- la probabilité de collision entre deux états différents. Nous expliquons cette valeur dans la suite de ce paragraphe.

Quand TLC termine son exécution, deux valeurs de probabilité sont fournies :

- La première est la valeur *calculée* (« *calculated* »). Cette valeur est basée sur la supposition que la probabilité que deux états différents aient le même hash est égale à 2^{-64}. Dans ce cas, si TLC génère n états (valeur donnée par « *states found* ») et m états distincts (valeur donnée par « *distinct states* »), alors la probabilité de collision est égale à $m * (n - m) * 2^{-64}$.

- La deuxième probabilité est la valeur *observée* (« *observed* »). Elle est calculée en supposant que la fonction de hachage ne garantisse pas forcément que la probabilité de collision est de 2^{-64}. Cette estimation de probabilité est égale à la valeur maximale de $1/|h(s_1) - h(s_2)|$ pour toute paire $(h(s_1), h(s_2))$.

Pour l'exemple des producteurs consommateurs, la probabilité calculée est égale à $13 * (13 - 7) * 2^{-64}$, à savoir $2.37 * 10^{-18}$. En ce qui concerne la probabilité observée, elle est égale à $1.07 * 10^{-17}$.

Enfin, la section « *User Ouput* » affiche les messages produits par l'opérateur $PrintT()$. Cet opérateur est prédéfini dans le module standard TLC.

Si aucune propriété n'est définie dans TLA+ Toolbox, TLC permet de détecter deux

types d'erreurs : erreurs sémantiques (par exemple, $3 + \langle 2, 3 \rangle$) ainsi que l'absence d'interblocage. L'interblocage dans TLC correspond au cas où un état n'a pas de successeur. Pour une spécification $\mathcal{S}$ (de la forme 2.1), l'absence d'interblocage est exprimée par la formule $\Box(\text{ENABLED } Next)$.

Quand une propriété n'est pas vérifiée, TLC indique la trace menant vers l'état qui ne satisfait la propriété concernée.

2.4 Travaux autour de TLA+

Le langage TLA+ a été utilisé dans plusieurs contextes différents, comme la spécification de protocoles de communication et leur vérification et le développement de logiciel système. Nous décrivons dans ce qui suit un aperçu non exhaustif des travaux impliquant TLA+.

La première expérience industrielle de TLA+ a été menée par Compaq pour la spécification et la vérification d'un protocole de cohérence de cache pour le multiprocesseur Alpha version EV6, appelé « *Wildfire* ». À cette époque TLC n'était pas disponible et l'équipe de Compaq utilisait des preuves manuelles pour vérifier des propriétés sur la spécification TLA+. C'est en 1998 que fut la première utilisation de TLC afin de vérifier des propriétés de sûreté sur une spécification TLA+ du protocole de cohérence de cache de la version EV7 du processeur Alpha [JLM+03]. Cette spécification a généré 12 millions d'états et TLC a pu détecter 67 erreurs.

Après les expériences acquises dans la spécification de EV7 et sa vérification avec TLC, Intel a adopté TLA+ afin de vérifier les spécifications haut-niveau des protocoles de cohérence de cache de ses processeurs Itanium. Dans ce projet, TLC a été utilisé pour vérifier des optimisations sur le protocole qui ont été par la suite intégrées dans le protocole.

En ce qui concerne les logiciels systèmes, on trouve OpenComRTOS [VdJ07], un RTOS développé formellement. L'interaction des différentes entités du système a été développée d'une manière incrémentale en utilisant TLA+. TLC a été utilisé pour vérifier si un modèle raffine un modèle plus abstrait. Les propriétés temps-réel de ce RTOS ont été vérifiées par le *model-checker* Uppaal [BDL04].

Dans [NRZ+15], les auteurs proposent TLA+ afin de détecter des erreurs dans les services web de Amazon (« *Amazon Web Services* »). La première expérience de Amazon avec TLA+ date de l'année 2012. Cette expérience a visé le service DynamoDB [AWS16a], un service de base de données NoSQL (base de données non relationnelle) qui permet à des clients la gestion et le dimensionnement de leurs bases de données distribuées. Dans le cadre du service DynamoDB, TLA+ a été utilisé par les ingénieurs de Amazon afin de vérifier les algorithmes implémentés dans le mécanisme de tolérance aux pannes par réplication. La vérification a porté sur des propriétés de sûreté et a été menée avec TLC. Ce qui a permis de détecter 3 erreurs subtiles. TLA+ et TLC ont été aussi utilisés dans

d'autres services comme le service EBS (« *Elastic Block Store* ») [AWS16b] qui fournit des volumes de stockage dans le cloud AWS.

Les travaux de [MLW11] proposent une spécification formelle TLA+ de Pastry, un algorithme de routage pour les réseaux pair à pair, proposé par Microsoft. Dans cet algorithme, le routage est basé sur une structure en anneau des différents nœuds. Deux principaux protocoles de cet algorithme ont été vérifiés : le protocole d'arrivée d'un nouveau nœud dans le réseau, appelé « *join* » et le protocole de routage de messages vers un nœud spécifique, appelé « *lookup* ». La vérification de Pastry a été menée avec TLC qui a permis de détecter une erreur dans le protocole « *lookup* ». Le modèle a été rectifié et les auteurs ont proposé par la suite une preuve de correction de cette spécification TLA+ en utilisant TLAPS.

Dans [GL03], les auteurs proposent une spécification formelle en TLA+ de Disk Paxos, un algorithme qui garantit la tolérance aux pannes dans un réseau distribué, composé de processeurs et de disques. Cette spécification a été ensuite vérifiée par TLC. En se basant sur cette spécification TLA+, les auteurs de [JM05] proposent une preuve de correction de cet algorithme en traduisant TLA+ en Isabelle/HOL.

D'autres travaux sont réalisés autour de TLA+ et que nous ne décrivons pas dans cette section. Comme la spécification d'un sous-ensemble de fonctions de la bibliothèque MPI (« *Message Passing Interface* ») [LPD+11].

2.5 CONCLUSION

Nous avons présenté tout au long de ce chapitre les notions de base de la logique TLA et le langage TLA+ qui constituent le cadre sur lequel repose notre étude. TLA+ est un langage fonctionnel avec des prédicats et des actions. Il dispose de plusieurs mécanismes permettant de spécifier la sémantique d'un langage de programmation comme le C. De plus, il permet de modéliser les systèmes concurrents d'une part et d'exprimer des propriétés de sûreté et de vivacité d'autre part.

Dans le chapitre suivant, nous décrivons comment utiliser les mécanismes fournis par TLA+ pour traduire la sémantique de C en une spécification TLA+ vérifiable par TLC.

Deuxième partie

Approche

3

Traduction de C vers TLA+

3.1 Introduction

La vérification formelle d'un programme passe nécesssairement par une phase de modélisation. Durant cette phase, un modèle formel décrivant le comportement du programme en question est construit. Cette phase de construction de modèle formel peut se faire manuellement ou d'une manière automatique. La construction manuelle d'un modèle peut engendrer des erreurs et s'avère une tâche pénible. Une solution à ce problème consiste à générer automatiquement le modèle formel.

L'objectif de ce chapitre est de modéliser des programmes C séquentiels en TLA+. Pour cela, ce chapitre décrit le principe de traduction de C vers TLA+. Nous considérons dans ce travail un sous-ensemble du langage C résultant d'une phase de prétraitement opérée en amont. Cette étape de prétraitement du code C nous permet d'éliminer plusieurs éléments de la syntaxe originale du langage C pour avoir un programme plus simple à analyser. Nous décrivons dans la section 3.2 cette étape de transformation du code et nous présentons le sous-ensemble du langage C que nous considérons dans ce travail. Dans la section 3.3, nous présentons le modèle mémoire considéré d'un programme C séquentiel ainsi que les opérations d'accès à cette dernière. Nous introduisons les notations utilisées pour représenter les règles de traduction et nous décrivons par la suite les règles de traduction des éléments du langage C que nous considérons en détaillant chaque règle et

en donnant pour chaque règle un exemple de traduction.

3.2 La bibliothèque CIL

CIL (« *C Intermediate Language* ») [NMRW02] est une bibliothèque de fonctions permettant l'analyse syntaxique et la transformation d'un programme C. Elle fournit un AST du sous-ensemble simplifié du C et permet de transformer un programme C en un programme équivalent en préservant la même sémantique.

CIL est implémentée en OCaml et elle a été développée par George C. Necula, Scott Mcpeak, S. P. Rahul et Westley Weimer de l'université de Californie à Berkeley. Elle a été adoptée notamment par la plateforme Frama-C [CKK+12], une plateforme d'analyse statique de code C, mais aussi dans le cadre du projet CompCert [Ler09], un compilateur certifié pour le C.

3.2.1 Les transformations CIL

Les transformations opérées par CIL permettent de normaliser un code C en un code équivalent sémantiquement. Ces transformations facilitent l'analyse syntaxique du code et ainsi la traduction de C vers TLA+. Parmi les principales transformations effectuées par CIL on trouve : la transformation des initialisations implicites, le renommage des variables locales, l'élimination des expressions à effet de bord et la transformation des structures itératives (`for`, `while` et `do-while`) en une structure itérative utilisant `while(1)`. Toutes ces transformations sont utilisées dans la traduction C vers TLA+. Nous décrivons dans ce qui suit seulement les principales transformations qui affectent le plus le programme C initial. La liste complète de toutes les transformations CIL est détaillée dans [Nec15].

3.2.1.1 Transformation des initialisations

Les initialisations implicites de tableaux sont normalisées en des initialisations explicites. Par exemple, les éléments du tableau `t1` du listing 3.1a sont initialisés dans le code normalisé du listing 3.1b. Quand l'initialisation du tableau `t2` omet la longueur de celui-ci (ligne 2 du listing 3.1a), CIL calcule la longueur du tableau sur la base des initialisations de ses éléments (ligne 2 du listing 3.1b).

```
1  int t1[3] = {1};
2  int t2[] = {1,2,3};
```

(a) Code C initial

```
1  int t1[3] = {1, 0, 0};
2  int t2[3] = {1, 2, 3};
```

(b) Code C normalisé

Listing 3.1 – Exemple de transformation des initialisations

3.2.1.2 Renommage des variables locales

Les variables déclarées à l'intérieur d'un bloc d'instructions peuvent avoir les mêmes noms que les variables locales de la fonction ou les variables globales (déclarées à l'extérieur de toute fonction). Pour éliminer les conflits potentiels entre les noms de variables, CIL renomme toutes les variables qui sont concernées par ces conflits.

```
1  int x = 5;
2  void f()
3  {
4    int y;
5    {
6      int x = 0;
7      y = x-2;
8      int y;
9      y = 0;
10   }
11   x = y +1;
12 }
```

(a) Code C initial

```
1  int x = 5;
2  void f(void)
3  {
4    int y;
5    {
6      int x_0;
7      int y_0;
8      x_0 = 0;
9      y = x_0 - 2;
10     y_0 = 0;
11   }
12   x = y + 1;
13   return;
14 }
```

(b) Code normalisé

Listing 3.2 – Exemple de transformation des déclarations de variables locales

Dans l'exemple donné par le listing 3.2, les variables x et y, déclarées à l'intérieur du bloc d'instructions (lignes 6 et 8 du listing 3.2a) sont renommées dans le code normalisé en x_0 et y_0 (lignes 6 et 7 du listing 3.2b) et leurs déclarations sont remontées à l'entrée du bloc.

3.2.1.3 Élimination des expressions à effets de bord

```
1  int f(int x, int y)
2  {
3   return
4     (y++ * incrementer(x));
5  }
```

(a) Code C initial

```
1  int f(int x, int y)
2  { int __retres;
3    int tmp;
4    int tmp_0;
5    { tmp = y;
6      y ++;
7      tmp_0 = incrementer(x);
8    }
9    __retres = tmp * tmp_0;
10   return (__retres);
11 }
```

(b) Code C normalisé

Listing 3.3 – Exemple de transformation d'une instruction à effet de bord

CIL permet d'isoler les expressions à effet de bord en ajoutant de nouvelles variables et instructions. Dans l'exemple du listing 3.3, l'expression de la ligne 4 du listing 3.3a fait appel à deux expressions à effet de bord : y++ et incrementer(x). Cette expression est alors normalisée en introduisant de nouvelles variables locales : __retres, tmp et tmp_0

(lignes 2, 3 et 4 du listing 3.3b) et l'instruction à effet de bord est décomposée en plusieurs instructions (lignes 5, 6, 7 et 9 du listing 3.3b) contenant chacune une expression sans effet de bord.

3.2.1.4 Transformation des structures itératives

```
1  int x = 0;
2  void f()
3  {
4    int i;
5    for (i = 0;i ++;i <10)
6    {
7      x ++;
8    }
9  }
```

(a) Code C initial

```
1  int x = 0;
2  void f(void)
3  {
4    int i;
5    i = 0;
6    while (1) {
7      int tmp;
8      {tmp = i;
9       i ++; }
10     if (! tmp < 10)
11       { goto while_0_break; }
12     x ++;
13   }
14   while_0_break: /* internal */ ;
15   return;
16 }
```

(b) Code C normalisé

Listing 3.4 – Exemple de transformation d'une structure itérative

Toutes les structures itératives (`while`, `for`, `do`) sont transformées par CIL en une structure unique, représentée par un `while(1)`, une structure `if` et des instructions `goto`. Le listing 3.4 illustre un exemple de transformation d'un code C contenant une instruction `for`. L'instruction itérative `for` du listing 3.4a est transformée en une structure itérative infinie `while(1)` avec une structure `if` et une instruction `goto` (ligne 6 jusqu'à la ligne 13 du listing 3.4b). Dans le code transformé, quand la condition `(! tmp < 10)` est vraie, le contrôle est passé à l'instruction `x++`. Sinon, on sort de la boucle.

3.2.2 Sous-ensemble considéré

Grâce aux transformations opérées par CIL, vues précédemment, les constructions C deviennent considérablement simplifiées.

Dans la suite de ce travail, nous considérons un sous-ensemble de C, tel que normalisé par CIL. Ce sous-ensemble comprend le sous-ensemble avec lequel notre étude de cas a été implémentée. Par ailleurs, C2TLA+ peut être étendu pour supporter les aspects du C non considérés actuellement.

La syntaxe du sous-ensemble considéré est décrite par la grammaire en forme BNF (notation *backus-naur form*) donnée par le listing 3.5 et dont les règles de grammaire respectent la convention suivante :

- Les éléments non-terminaux sont les noms des catégories du langage et sont écrits

entre chevrons, par exemple une expression est un élément non-terminal désigné par <expr>.

- Les éléments terminaux sont dénotés en gras. Ils peuvent être des mots clés du langage, par exemple `int`, des identifiants comme le nom d'une variable **VAR_ID** ou d'une fonction **FUN_ID** ou bien des entiers littéraux, par exemple l'entier `27`.

- Le méta-symbole « | » exprime le choix entre plusieurs terminaux et/ou non-terminaux.

- Les parenthèses « (» et «) » suivies d'un astérisque « * » entourent les éléments à répéter un nombre indéfini de fois (zéro ou plusieurs) et le symbole « ? » entoure les éléments optionnels.

Des instructions destinées au préprocesseur peuvent exister dans un code C mais on suppose que le code C a déjà été traité par le préprocesseur. Le sous-ensemble considéré comprend les types de données de base (`int` et `struct`). D'après la grammaire donnée par le listing 3.5, nous considérons que C2TLA+ gère les opérations sur les entiers, les tableaux, les pointeurs, l'arithmétique des pointeurs, tous les types de flot de contrôle (intra et inter-procédural) ainsi que la récursivité. Toutefois, C2TLA+ ne supporte pas le type `float`, la conversion explicite entre différents types de données (opération `cast`), l'affectation des structures de données et l'allocation dynamique de la mémoire. Dans C2TLA+, une fonction ne retourne qu'un résultat de type pointeur, de type `int` ou bien de type `void`.

Comme le montre la grammaire du listing 3.5, un programme C <prg> est formé d'une suite de déclarations <decls> et de définitions de fonctions (<fun_def>)*. Le programme peut aussi inclure des attributs <attrs>, qui permettent d'associer des informations au programme C. Ces attributs seront utilisés dans le chapitre 4. Une définition de fonction <fun_def> comprend un entête et un corps. L'entête contient le type de retour <type> de la fonction <fun_def>, le nom de la fonction **FUN_ID** et la liste des déclarations des paramètres <params> si elle en a.

Une instruction <stmt> peut être une instruction d'affectation (<lval> = <expr>`;`), une instruction conditionnelle (`if` <expr> <stmt> (`else` <stmt>) ?), une instruction étiquetée (**LABEL_ID**`:` <stmt>`;`), une instruction de saut (`goto` **LABEL_ID**, **break**, **continue**, `return` <expr> ?), une structure itérative (`while(1)` <stmt>), un appel de fonction (<lval> = **FUN_ID** `(` (<expr>`,`)* `)`), ou enfin un bloc d'instructions (<stmt>*).

Une expression <expr> peut être une expression simple, s'écrivant sous forme d'une *lvalue* <lval>, ou sous forme d'une référence à une lvalue, ou bien sous forme d'une constante **CONSTANT**. Comme elle peut être composée d'une combinaison récursive de sous-expressions. Une expression composée se décline soit en une expression binaire, soit en une expression unaire. Les symboles $+_{pi}$ et $-_{pi}$ désignent respectivement l'opérateur d'addition et de soustraction entre une expression de type pointeur et une expression de type entier. Le symbole $-_{pp}$ désigne l'opérateur de soustraction entre deux expressions de type pointeur.

```
<prg>       ::=  (<attrs>) ?<decls> (<fun_def>)*
<attrs>     ::=  <attr> | <attr><attrs>
<attr>      ::=  <type> FUN_ID (<params>) __attribute__
                 ((CONSTANT,alias("FUN_ID")))
<decls>     ::=  <decl> | <decl><decls>
<decl>      ::=  <type> VAR_ID; | typedef <type> VAR_ID;
                 | <type>* VAR_ID;
<params>    ::=  <param> | , <params>
<param>     ::=  <type> VAR_ID
<fun_def>   ::=  <type> FUN_ID (<params>) {<decls><stmt>}
<type>      ::=  void | int | struct STRUCT_ID { (<type> FIELD_ID ; )* } ;
<stmt>      ::=  <lval> = <expr> ;
                 | if (<expr>) <stmt> (else <stmt>) ?
                 | LABEL_ID: <stmt> | goto LABEL_ID;
                 | break; | continue;
                 | { <decls> (<stmt>)* }
                 | while(1) {<decls> (<s:stmt>)* }
                 | <lval> = FUN_ID ( (<expr>, )* ) ;
                 | return (<expr>) ?;
                 | ε_stmt ; \* Skip instruction *\
<expr>      ::=  CONSTANT
                 | <expr> <bin_op><expr> | <un_op><expr>
                 | <lval> | &<expr>
                 | <expr>+_pi<expr> | <expr>−_pi<expr>
                 | <expr>−_pp<expr>
<lval>      ::=  VAR_ID<offs> | (*<expr>) <offs>
<offs>      ::=  .FIELD_ID<offs> | [<expr>] <offs> | ε_offs
<bin_op>    ::=  * | + | − | % | / | > | >= | < | <= | == | != | & | |
<un_op>     ::=  !
{VAR, FUN, LABEL, STRUCT,FIELD}_ID ::=[a-zA-Z][0-9a-zA-Z_]*
CONSTANT ::=[1-9]([0-9])*
```

Listing 3.5 – Grammaire en forme de BNF du sous-ensemble CIL de C

3.3 Principe de la traduction

C2TLA+ est un greffon « *plugin* » de la plateforme Frama-C, implémenté en OCaml. Il traduit un code CIL vers une spécification TLA+ vérifiable par des outils de vérification, comme TLC. Le choix de développer C2TLA+ sous Frama-C a été motivé par le fait que la plateforme dispose d'outils permettant de naviguer dans le flot de contrôle du programme C ainsi que de parcourir ou modifier l'arbre de syntaxe abstraite ou l'AST (pour « *Abstract Syntax Tree* » en anglais) du programme grâce au mécanisme des visiteurs. Un visiteur est un objet disposant d'un ensemble de méthodes pour visiter tous les nœuds de l'arbre et éventuellement appliquer une fonction sur ces nœuds. Ainsi, avec

le principe des visiteurs, il est facile d'explorer tous les nœuds de l'AST du programme.

En plus, Frama-C possède une architecture modulaire composée de greffons permettant la collaboration de plusieurs techniques d'analyse dont nous pourrons bénéficier. Une liste détaillée des greffons et leur fonctionnalité peut être trouvée sur le site de Frama-C [CL13]. Il est possible sous Frama-C de développer en OCaml son propre greffon en suivant le guide de développement disponible sur le site de Frama-C [SCLP09].

C2TLA+ prend en entrée un ou plusieurs fichiers C ainsi que le nom de la fonction principale du programme C. CIL normalise le code C et fournit l'AST du programme. C2TLA+ utilise les visiteurs fournis par Frama-C pour parcourir et manipuler cet AST et applique pour chaque élément de l'arbre une règle de traduction.

Comme décrit dans le chapitre 2, TLA+ est un langage logique fonctionnel avec des actions. Il offre un éventail de mécanismes, comme les prédicats, les opérateurs, et les structures conditionnelles. Nous détaillons dans cette section comment nous avons utilisé ces mécanismes afin de spécifier formellement la sémantique de C. Nous introduisons dans un premier temps les notations utilisées dans la traduction. Nous décrivons ensuite le modèle mémoire considéré dans cette étude et nous détaillons par la suite les règles de traduction du sous-ensemble C considéré.

3.3.1 Notations

Pour une meilleure lisibilité de la traduction, nous adoptons une notation particulière pour exprimer les règles de traduction. Nous définissons une règle de traduction pour chaque élément terminal et non terminal du sous-ensemble CIL de C considéré (listing 3.5).

Une règle de traduction s'écrit sous la forme $[\![$ <elemt> $]\!]_{type}$ où $type$ détermine le type de la règle et <elemt> désigne l'élément de la grammaire sur lequel on applique la règle. Chaque règle peut générer un code TLA+ et/ou appeler d'autres règles de traduction. Le code TLA+ est écrit en gris afin de distinguer entre le code final généré et le code C non encore traduit.

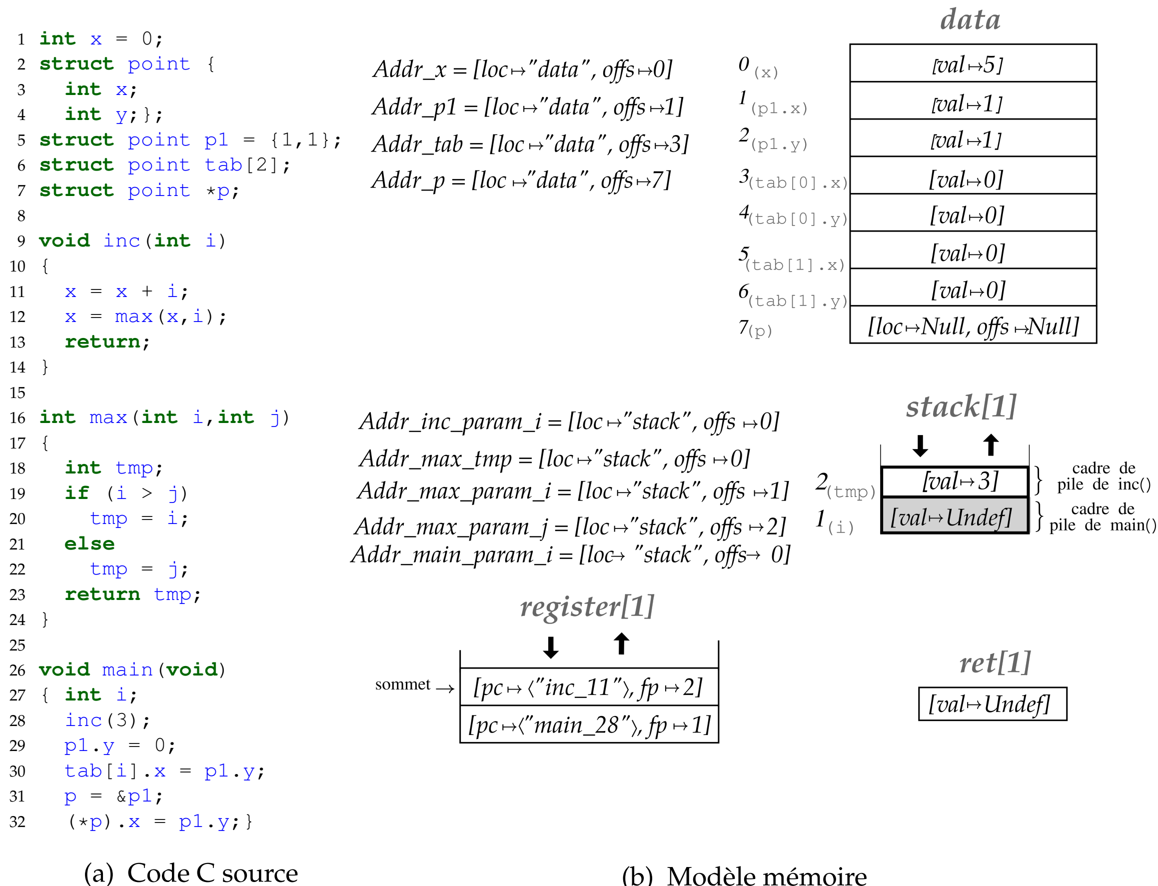
La notation <e_1 : elemt> désigne un élément de la grammaire nommé e_1 et de type <elemt>. Par exemple, la règle $[\![$<e_1 : expr> $+_{pi}$<e_2 : expr>$]\!]_{expr}$ désigne la règle de traduction d'une expression dont la syntaxe de la forme <e_1 : expr> $+_{pi}$<e_2 : expr>, avec <e_1 : expr> et <e_2 : expr> deux éléments de la grammaire de type expression. Cette règle appelle récursivement une règle pour chaque élément composant l'expression. La traduction se termine quand tous les éléments terminaux sont traduits.

Les règles de traduction utilisent plusieurs fonctions OCaml, fournies par la bibliothèque CIL. Par exemple, la fonction **`type(`**<e : expr>**`)`** prend en argument une expression <e : expr> et retourne son type.

Dans les exemples de traduction décrits dans ce qui suit, nous utilisons la notation $\underset{R.i}{\longrightarrow}$

pour désigner un appel à la règle i. Le code à droite de cette notation désigne le code résultant après application de la règle i.

3.3.2 Modèle mémoire pour C2TLA+



(a) Code C source (b) Modèle mémoire

FIGURE 3.1 – Organisation de la mémoire d'un exemple de code

Nous proposons d'organiser la mémoire d'un programme C séquentiel en quatre zones mémoires. La figure 3.1a illustre un exemple de code C contenant les fonctions `inc()`, `max()` et `main()` ainsi qu'une représentation de l'état mémoire du programme au moment où `main()` fait un appel à la fonction `inc()` et exécute l'instruction de la ligne 11. La représentation mémoire d'un programme C dans C2TLA+ est répartie de la façon suivante :

1. Une zone mémoire appelée *data* stocke les données globales du programme. Cette mémoire est modélisée en TLA+ par un tableau indexé par l'adresse de la variable en question (figure 3.1b). Ainsi, la variable globale `x` est stockée dans *data*[0].
   La taille de cette zone mémoire reste inchangée durant l'exécution du programme C. Nous supposons qu'un entier ou un pointeur est stocké dans une cellule

mémoire, c'est-à-dire que la taille d'un type entier et d'un type pointeur est égale à 1.

2. Une zone mémoire appelée $stack$ sert à stocker les paramètres et les variables locales de la fonction que le programme exécute. Nous modélisons cette zone mémoire à travers un tableau de séquences. Ce tableau contient un seul élément. Cet élément se comporte comme une pile. Les données sont regroupées dans cette pile dans ce qu'on appelle des cadres de pile (« *stack frame* »), fonctionnant selon le principe LIFO (*Last In First Out*) : on empile d'abord puis on libère en commençant par la dernière zone empilée.
   Quand le programme C appelle une fonction, un cadre contenant les variables locales et les arguments est créé, puis empilé dans $stack[1]$.
   Ainsi, dans l'exemple de la figure 3.1, quand la fonction `main()` appelle la fonction `inc()`, son cadre de pile contenant la valeur 3 est empilé dans $stack[1]$. Lorsque la fonction `inc()` finit son exécution, son cadre est dépilé.

3. Une zone mémoire appelée $register$ sert à stocker l'état d'exécution du programme. Cette zone est considérée comme une pile fonctionnant aussi selon le principe LIFO. Elle est modélisée dans TLA+ par un tableau de séquences d'enregistrements. De la même manière, nous supposons que ce tableau est à un élément.
   Chaque enregistrement est composé de deux champs :

   - Un champ pc, le pointeur d'instruction (« *program counter* ») : le programme gère ce champ de manière à ce qu'il contienne en permanence l'identificateur de l'instruction en cours d'exécution.

   - Un champ fp, le pointeur de cadre (« *frame pointer* ») : il mémorise l'adresse de base du cadre de pile de la fonction en cours d'exécution.

   La tête de pile, désignée par l'expression $Head(register[1])$, contient en permanence les informations sur l'instruction en cours d'exécution. Dans l'exemple de la figure 3.1, la tête de pile contient un enregistrement dont le champ pc contient l'identificateur "inc_11" et dont le champ fp est égal à l'adresse de base du cadre de pile de la fonction `inc()`, à savoir 2. Quand la fonction `inc()` se termine, son cadre de pile est dépilé et le champ fp est mis à jour pour contenir l'adresse de base de la fonction `main()`, à savoir la valeur 1. Nous rappelons que les index des séquences commencent à 1.

4. Une mémoire appelée ret stocke la valeur de retour de la fonction exécutée. Cette mémoire est modélisée par un tableau de valeurs. Ce tableau contient un seul élément.

Modéliser $stack$, $register$ et ret comme des tableaux nous permet de modéliser des programmes C concurrents (voir chapitre 4) sans changer le modèle mémoire décrit dans cette section. Ainsi, dans la suite de ce chapitre, certains opérateurs TLA+ générés par C2TLA+ sont paramétrés par un identificateur id. Comme nous modélisons un

programme séquentiel (un seul processus), nous considérons que cet identificateur vaut 1. Le paramétrage des opérateurs TLA+ avec l'identificateur id nous permet de réutiliser les règles de traduction définies dans ce chapitre pour la modélisation de programmes concurrents. C'est pour cette raison que les tableaux $stack$, $register$ et ret définis plus haut contiennent chacun un seul élément. Aussi, le choix de la traduction proposée dans ce chapitre permet d'appliquer une technique de réduction de l'espace d'états d'une spécification TLA+ générée à partir d'un code C. Cette partie fait l'objet du chapitre 5.

3.3.2.1 Déclaration de variables et de paramètres

C2TLA+ associe à chaque déclaration de variable globale, locale ou paramètre de fonction une constante dans TLA+ décrivant son adresse. La traduction de déclaration de variables et de paramètres est donnée par la règle 3.1.

$$.1.\ [\![\text{<type>}\textbf{FUN_ID}\,(\text{<params>})\,\{\text{<decls><s:stmt>}\}]\!]_{decls} \rightarrow [\![\text{<decls>}]\!]_{decls}[\![\text{<params>}]\!]_{params}$$
$$.2.\ [\![\text{<decl><decls>}]\!]_{decls} \rightarrow [\![\text{<decl>}]\!]_{decl}\ [\![\text{<decls>}]\!]_{decls}$$
$$.3.\ [\![\text{<param><params>}]\!]_{params} \rightarrow [\![\text{<param>}]\!]_{param}[\![\text{<params>}]\!]_{params}$$
$$.4.\ [\![\text{<type>}\textbf{VAR_ID}]\!]_{decl} \rightarrow Addr_[\![\textbf{VAR_ID}]\!]_{vid} \triangleq [loc \mapsto \texttt{loc}(\textbf{VAR_ID}),\ offs \mapsto \texttt{offs}(\textbf{VAR_ID})]$$
$$.5.\ [\![\text{<type>}\textbf{VAR_ID}]\!]_{param} \rightarrow Addr_[\![\textbf{VAR_ID}]\!]_{vid} \triangleq [loc \mapsto \texttt{loc}(\textbf{VAR_ID}),\ offs \mapsto \texttt{offs}(\textbf{VAR_ID})]$$
$$.6.\ [\![\textbf{VAR_ID}]\!]_{id} \rightarrow \texttt{id}(\textbf{VAR_ID})$$

Règle 3.1 – Traduction de déclarations de variables et de paramètres

La règle 3.1.1 est la règle qui pour chaque définition de fonction appelle la règle de traduction de déclaration de variables locales $[\![\text{<decls>}]\!]_{decls}$ et la règle de traduction de déclaration de paramètres $[\![\text{<params>}]\!]_{params}$.

Les règles 3.1.2 et 3.1.3 sont respectivement les règles de traduction d'une liste de déclarations de variables locales et d'une liste de paramètres.

Les règles 3.1.4 et 3.1.5 génèrent, respectivement pour chaque variable locale et paramètre, une adresse TLA+. La règle 3.1.6 permet de générer un identificateur TLA+ pour la variable **VAR_ID**, en appelant la fonction OCaml **`id()`**. Par exemple, pour la variable `x` de la figure 3.1, **`id(x)`** retourne x. Pour une variable locale ou un paramètre, la fonction **`id()`** retourne un identificateur contenant le nom de la fonction dans laquelle la variable est déclarée et spécifie s'il s'agit d'un paramètre de fonction. Par exemple, pour le paramètre `i` (déclaré dans la fonction `max()`), l'identificateur généré par la fonction **`id()`** est max_param_i. Pour la variable `tmp`, l'identificateur généré est max_tmp.

Les règles 3.1.4 et 3.1.5 génèrent une constante TLA+. Cette constante est modélisée par un enregistrement composé de deux champs :

- Un champ nommé loc (« *location* ») : il spécifie la zone mémoire où sera stockée la variable en question ($data$ ou $stack$). La valeur de ce dernier est donnée par la fonction **`loc(VAR_ID)`** qui rend la valeur "data" si **VAR_ID** est une variable globale sinon "stack".

- Un champ nommé $offs$ (« *offset* ») : cette valeur est calculée par la fonction **`offs()`**.

> Cette fonction calcule le déplacement requis dans la zone mémoire loc pour accéder à la valeur de la variable **VAR_ID**. Cette valeur est égale à 0 pour la première variable globale et pour le premier paramètre (ou variable locale si la fonction est sans paramètre) de chaque fonction. Le déplacement dans $data$ se fait donc par rapport à la première cellule mémoire, tandis que le déplacement dans la pile $stack[1]$ se fait par rapport au début de la pile de la fonction en cours d'exécution, qui est donné par l'expression $Head(register[1]).fp$.

Par exemple, pour la variable `x`, l'adresse TLA+ $Addr_x$ contient le champ loc qui vaut "data" et le champ $offs$ qui vaut 0. Pour la constante $Addr_max_param_i$, le champ loc est égal à "stack" et le déplacement est égal à la taille de la variable `tmp`, à savoir 1.

3.3.2.2 Lecture en mémoire

La lecture en mémoire globale ou locale se fait par l'opérateur TLA+ $load()$ que nous définissons dans le listing 3.6.

$$\begin{array}{l} load(id, mem, ptr) \triangleq \\ \quad \text{IF } ptr.loc = \text{“data”} \text{ THEN } mem.data[ptr.offs] \\ \quad \text{ELSE } mem.stack[id][Head(mem.register[1]).fp + ptr.offs] \end{array}$$

Listing 3.6 – Définition de l'opérateur $load()$

L'opérateur $load()$ prend en paramètre un identificateur id, l'état de la mémoire mem et une adresse TLA+ ptr. La mémoire globale est accessible par $mem.data$ et la mémoire locale par $mem.stack$.

Quand l'adresse ptr pointe vers une variable globale, c'est-à-dire $ptr.loc$ vaut "data", $load()$ retourne la valeur contenue dans cette adresse en tenant compte du déplacement contenu dans $ptr.offs$. Quand l'adresse ptr pointe vers une variable locale, le déplacement est calculé par rapport à l'adresse de base du cadre de pile de la fonction en cours d'exécution, qui est donnée par l'expression $Head(mem.register[1]).fp$.

3.3.2.3 Écriture en mémoire

$$\begin{array}{l} store(id, mem, ptr, value) \triangleq \\ \quad \text{IF } ptr.loc = \text{“data”} \\ \quad \text{THEN } [data \mapsto [mem.data \text{ EXCEPT } ![ptr.offs] = value], \\ \qquad\quad stack \mapsto mem.stack] \\ \quad \text{ELSE } [data \mapsto mem.data, \\ \qquad\quad stack \mapsto [stack \text{ EXCEPT } ![id][Head(mem.register[id]).fp + ptr.offs] = value]] \end{array}$$

Listing 3.7 – Définition de l'opérateur $store()$

L'écriture en mémoire globale ou locale se fait par l'opérateur TLA+ $store()$ que nous définissons dans le listing 3.7. Cet opérateur prend en argument l'identificateur id, la mémoire mem, une adresse ptr et une valeur $value$. Il met à jour la mémoire locale ou

globale en fonction de la valeur contenue dans $ptr.loc$ et en utilisant le constructeur EXCEPT (dont la sémantique est définie dans la section 2.2.2).

Lorsque $ptr.loc$ vaut "data", l'écriture d'une valeur $value$ dans la mémoire $data$ se fait à l'adresse $ptr.offs$. Sinon, l'écriture dans $stack$ se fait à l'adresse donnée par $Head(mem.register[id]).fp + ptr.offs$, c'est-à-dire en calculant le déplacement par rapport à l'adresse de base du cadre de pile de la fonction en cours d'exécution.

Enfin, $store()$ retourne un enregistrement contenant la mémoire globale $data$ et la mémoire locale $stack$.

3.3.3 Traduction d'un programme

C2TLA+ prend en entrée un programme C s'écrivant sous la forme d'une suite de déclarations <d : decls> et de définitions de fonctions $\texttt{<}f_1 : \texttt{fun_def>} \ldots \texttt{<}f_k : \texttt{fun_def>}$. Dans TLA+, le flot de contrôle se fait par des appels à des opérateurs qui modélisent les instructions du programme ainsi que l'utilisation des structures conditionnelles de TLA+. L'exécution d'une instruction du C est modélisée par l'exécution d'une action en TLA+. La traduction d'un programme C suit alors la règle 3.2.

En appliquant cette règle, C2TLA+ génère en sortie deux modules TLA+ : un module appelé $Parameters$ qui contient la déclaration des variables et la définition des constantes et un module nommé $Program$ contenant la traduction des définitions de fonctions C, la définition de l'opérateur $dispatch()$, le prédicat $Init$, l'action $Next$ et enfin la formule $Spec$ qui modélise le comportement du programme C. Nous décrivons chacun de ces modules dans les paragraphes suivants.

(a) Module $Parameters$. Comme le montre la règle 3.2, le module $Parameters$ importe le module $Runtime$ qui contient la définition des opérateurs TLA+ arithmétiques, logiques et relationnels utilisés par la traduction. Le contenu de ce dernier est détaillé dans le paragraphe (b). Le module $Parameters$ appelle ensuite la règle $[\![\texttt{<decls>}]\!]_{decls}$ qui génère pour toutes les déclarations de variables globales des adresses TLA+. De la même manière, les règles $[\![\texttt{<}f_1 : \texttt{fun_def>}]\!]_{decls} \ldots [\![\texttt{<}f_k : \texttt{fun_def>}]\!]_{decls}$ génèrent pour toutes les variables locales de chaque fonction $\texttt{<}f_i : \texttt{fun_def>}$ des adresses TLA+. Ce principe est détaillé dans le paragraphe suivant.

Le module $Parameters$ déclare aussi la constante $size_of_int$ qui spécifie la taille d'un type entier. Puisque nous supposons qu'un entier est stocké dans une cellule mémoire, la taille de cette constante est égale à 1. Le module appelle par la suite la règle $[\![\texttt{<d : decls>}]\!]_{field_offs}$, qui génère les valeurs des déplacements des champs de structures de données pour toutes les déclarations de type `struct`. En effet, C2TLA+ associe à chaque champ d'une structure de données une constante TLA+. Cette constante représente le déplacement requis par rapport à l'adresse de base de la structure de données pour accéder à la valeur de ce champ. Cette génération de constantes suit la règle 3.3. Pour appliquer cette règle, nous définissons la règle 3.4 qui génère un identificateur pour un champ **FIELD_ID**.

```
⟦<d:decls><f1:fun_def>...<fk:fun_def>⟧prg  →

---------------------------- MODULE Parameters ----------------------------
EXTENDS Runtime
⟦<d:decls>⟧decls
⟦<f1:fun_def>⟧decls ... ⟦<fk:fun_def>⟧decls
size_of_int ≜ 1
⟦<d:decls>⟧field_offs
===========================================================================

----------------------------- MODULE Program -----------------------------
EXTENDS Parameters
⟦ <f1:fun_def> ::= <type> FUN_ID (<params>) { <decls><s1:stmt>...<sn:stmt> } ⟧fun_def
...
⟦ <fk:fun_def> ::=<type> FUN_ID (<params>) { <decls><sh:stmt>...<sm:stmt> } ⟧fun_def

Init ≜ memory = [data ↦ ⟦<d:decls>⟧init,
                 stack ↦ (1 :> init_stack()),
                 register ↦ (1 :> ⟨[pc ↦ init_pc(), fp ↦ 1]⟩),
                 ret ↦ (1 :> [val ↦ Undef])]

dispatch(id, mem) ≜
  CASE mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<f1:fun_def>⟧fid_lbl(<s1:stmt>)"
           → stmt_⟦<s1:stmt>⟧sid(id, mem)
     ...
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<f1:fun_def>⟧fid_lbl(<sn:stmt>)"
           → stmt_⟦<sn:stmt>⟧sid(id, mem)
     ...
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<fk:fun_def>⟧fid_lbl(<sh:stmt>)"
           → stmt_⟦<sh:stmt>⟧sid(id, mem)
     ...
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<fk:fun_def>⟧fidlbl(<sm:stmt>)"
           → stmt_⟦<sm:stmt>⟧sid(id, mem)
     □ mem.register[id] = ⟨⟩ → mem
     □ OTHER → "error"

Next ≜ ∨ ∧ memory.register[1] ≠ ⟨⟩
         ∧ memory' = dispatch(1, memory)
       ∨ ∧ memory.register[1] = ⟨⟩
         ∧ UNCHANGED memory

Spec ≜ Init ∧ □[Next]memory ∧ WFmemory(Next)
===========================================================================
```

Règle 3.2 – Traduction d'un programme C

```
⟦struct STRUCT_ID {<type> FIELD_ID1;...<type> FIELD_IDn; }; ⟧field_offs  →
    Offset_⟦FIELD_ID1⟧oid ≜ 0
    ...
    Offset_⟦FIELD_IDn⟧oid ≜ Σ_{i=1}^{n−1} size(type(FIELD_IDi))
```

Règle 3.3 – Génération de l'offset d'un champ d'une structure de données

$[\![\textbf{FIELD_ID}]\!]_{oid} \rightarrow$ **struct(FIELD_ID)_fid(FIELD_ID)**

Règle 3.4 – Génération d'un identificateur pour un champ d'une structure de données

La règle 3.4 consiste à générer un identificateur pour un champ **FIELD_ID**. Cet identificateur contient le nom de la structure dans laquelle le champ est défini, afin de différencier plusieurs champs issus de différentes structures de données et ayant les mêmes identificateurs. Pour l'exemple de la figure 3.1a, la traduction $[\![\texttt{.x}]\!]_{oid}$ retourne l'expression $point_x$.

La constante générée par $Offset_[\![\textbf{FIELD_ID}_i]\!]_{oid}$ contient le déplacement requis par rapport à l'adresse de base de la structure de données pour accéder au champ **FIELD_ID**$_i$. Ceci en tenant compte de la taille de tous les champs le précédant.

Exemple 3.3.1. *Pour la structure* `point` *de l'exemple 3.1a, la génération des offsets pour les champs la composant se fait comme suit :*

$[\![\texttt{struct point \{ int x; int y;\};}]\!]_{field_offs} \xrightarrow[R.3.3]{}$
$Offset_[\![\texttt{x}]\!]_{oid} \triangleq 0$
$Offset_[\![\texttt{y}]\!]_{oid} \triangleq \textbf{size(type(}\texttt{x}\textbf{))}$

En appliquant la règle 3.4 de génération d'identificateur de champ, la définition des constantes est la suivante :

$Offset_point_x \triangleq 0$
$Offset_point_y \triangleq 1$

(b) Module $Runtime$**.** C2TLA+ utilise le module $Runtime$ dont le contenu est statique. Ce module est donné par le listing 3.8. Il importe les modules standard fournis par TLA+, à savoir le module $Naturals$ et $Sequences$. Nous déclarons ensuite la variable $memory$ qui modélise la mémoire du programme C. Elle est modélisée par un enregistrement ayant 4 champs, et contenant respectivement, la mémoire globale $data$, la mémoire locale $stack$, la pile $register$ du programme et la mémoire ret. Le module $Runtime$ déclare ensuite les constantes $Undef$ et $Null$. Ces constantes ont des valeurs indéfinies, appelées des « *model values* ». Dans C2TLA+, $Undef$ et $Null$ désignent respectivement *une valeur indéfinie* et *une valeur nulle*. Le module contient la définition des opérateurs conditionnels et arithmétiques ainsi que la définition des opérateurs de lecture et d'écriture en mémoire.

(c) Module $Program$**.** Ce module contient la traduction de toutes les définitions de fonctions contenues dans le programme C donné en entrée. La règle de traduction d'une définition de fonction <f$_i$: fun_def> est notée par $[\![\text{<f}_i : \text{fun_def>}]\!]_{fun_def}$. Cette règle est décrite dans la section 3.3.8. et elle fait appel aux règles de traduction de flot de contrôle et celles des expressions. Le reste du module comprend la définition de l'opérateur $dispatch()$ qui gère le flot d'exécution des instructions, le prédicat $Init$, l'action $Next$ et la formule $Spec$.

```
------------------------------ MODULE Runtime ------------------------------
EXTENDS Naturals, Sequences
CONSTANTS ProcSet, Undef, Null
VARIABLES memory

lt(expr1, expr2) ≜ IF (expr1.val < expr2.val)THEN [val ↦ 1] ELSE [val ↦ 0]
gt(expr1, expr2) ≜ IF (expr1.val > expr2.val)THEN [val ↦ 1] ELSE [val ↦ 0]
le(expr1, expr2) ≜ IF (expr1.val ≤ expr2.val)THEN [val ↦ 1] ELSE [val ↦ 0]
ge(expr1, expr2) ≜ IF (expr1.val ≥ expr2.val)THEN [val ↦ 1] ELSE [val ↦ 0]
eq(expr1, expr2) ≜ IF (expr1 = expr2)THEN [val ↦ 1] ELSE [val ↦ 0]
ne(expr1, expr2) ≜ IF (expr1 ≠ expr2)THEN [val ↦ 1] ELSE [val ↦ 0]
bAnd(expr1, expr2) ≜ IF (expr1.val = 1) ∧ (expr2.val = 1)THEN [val ↦ 1] ELSE [val ↦ 0]
bOr(expr1, expr2) ≜ IF (expr1.val = 1) ∨ (expr2.val = 1)THEN [val ↦ 1] ELSE [val ↦ 0]
not(expr) ≜ IF (expr.val ≠ 0)THEN [val ↦ 1] ELSE [val ↦ 0]
not_zero(expr) ≜ IF (expr ≠ [val ↦ 0])THEN TRUE ELSE FALSE
not_null(expr) ≜ IF (expr ≠ [loc ↦ Null, offs ↦ Null])THEN TRUE ELSE FALSE
store(id, mem, ptr, value) ≜
        IF ptr.loc = "data"
       THEN [data ↦ [mem.data EXCEPT ![ptr.offs] = value],
              stack ↦ mem.stack]
        ELSE [data ↦ mem.data,
              stack ↦ [stack EXCEPT ![id][Head(mem.register[id]).fp + ptr.offs] = value]]
load(id, mem, ptr) ≜
        IF ptr.loc = "data"THEN mem.data[ptr.offs]
         ELSE mem.stack[id][Head(mem.register[id]).fp + ptr.offs]
plusPI(ptr1, exp, size_of_ptr1) ≜
              [loc ↦ ptr1.loc, offs ↦ plus([val ↦ ptr1.offs], mult(exp, size_of_ptr1))]
minusPI(ptr1, exp, size_of_ptr1) ≜
              [loc ↦ ptr1.loc, offs ↦ minus([val ↦ ptr1.offs], mult(exp, size_of_ptr1)))]
plus(expr1, expr2) ≜ [val ↦ (expr1.val + expr2.val)]
minus(expr1, expr2) ≜ [val ↦ (expr1.val − expr2.val)]
mult(expr1, expr2) ≜ [val ↦ (expr1.val ∗ expr2.val)]
div(expr1, expr2) ≜ [val ↦ (expr1.val ÷ expr2.val)]
mod(expr1, expr2) ≜ [val ↦ (expr1.val%expr2.val)]
============================================================================
```

Listing 3.8 – Définition du module *Runtime*

3.3.4 Expressions, lvalues et offsets

Une expression C peut être soit une *lvalue*, soit une *rvalue*. Une *lvalue* (« *left value* ») possède un type, une valeur et un emplacement dans la mémoire (ou une adresse). La valeur de celle-ci est définie comme le contenu de son emplacement. Ce qui implique que seule une *lvalue* peut figurer à gauche de l'opérateur d'affectation « = ». Toutes les variables sont des *lvalue*s. Une *rvalue* (« *right values* ») correspond à une valeur temporaire qui ne possède pas un emplacement mémoire. Ce qui fait qu'une *rvalue* ne peut apparaître qu'à droite de l'opérateur d'affectation « = ».

Soit e la traduction TLA+ d'une *lvalue*. Comme une *lvalue* désigne un emplacement mémoire (ou une adresse), la valeur retournée par e est telle que :

$$e \in [loc : \{\text{“data”}, \text{“stack”}\}, offs : Nat]$$

cette valeur désigne une adresse dont le champ loc est égal soit à "data" si la *lvalue* est une variable globale, soit à "stack" si la *lvalue* est un paramètre ou une variable locale.

Soit v la traduction d'une *rvalue* <rval>. Cette *rvalue* peut désigner une valeur entière ou une adresse. La valeur retournée par v est telle que :

$v \in [loc : \{\text{"data"}, \text{"stack"}\},\ offs : Nat]$, si <rval> est de type pointeur (non nul)
$v \in [val : Nat]$, si <rval> est de type entier.

Nous représentons un pointeur nul par l'enregistrement $[loc \mapsto Null,\ offs \mapsto Null]$ et une valeur nulle par l'enregistrement $[val \mapsto Null]$.

Dans ce qui suit, nous décrivons la traduction des expressions simples et composées, les lvalues ainsi que les offsets. La traduction utilise les fonctions OCaml suivantes :

- La fonction **`type(<e:expr>)`** : elle prend en argument une expression e et retourne son type.

- La fonction **`size()`** : elle prend en argument un type et calcule sa taille. Nous rappelons que la taille d'un pointeur est 1, puisqu'une adresse TLA+ est stockée dans une cellule mémoire. La taille d'un entier est égale à 1. La taille d'un tableau est la taille de tous ses éléments. Par exemple, pour un tableau d'entiers `tab[2]`, la fonction **`size(type(tab))`** retourne 2.

3.3.4.1 Expressions constantes

Les expressions constantes C que nous considérons sont les constantes numériques entières. Dans C2TLA+, une constante entière est modélisée par un enregistrement contenant sa valeur. Par exemple, la constante 2 est stockée dans la mémoire comme $[val \mapsto 2]$. Ce choix de représentation vient du fait que TLC explore les états du système en comparant à chaque fois l'état courant à tous les états explorés et ceci, en comparant les valuations de l'ensemble de variables. Et comme nous avons défini nos propres types TLA+ (les *model values*) $Undef$ et $Null$, TLC ne peut pas comparer un type entier avec un type $Undef$ et génère dans ce cas une erreur. Il est donc nécessaire que les valeurs des variables TLA+ soient des enregistrements.

3.3.4.2 Expressions conditionnelles et arithmétiques

Une expression composée est la combinaison récursive de différents types d'expressions simples s'achevant par l'ensemble de terminaux. La traduction des expressions conditionnelles et arithmétiques est donnée par la règle 3.5. Elle utilise des fonctions TLA+ que nous définissons dans le listing 3.9.

Les fonctions arithmétiques binaires ($mult$, $plus$, $minus$, div et mod) (définies dans le listing 3.9) prennent comme arguments deux expressions $expr1$ et $expr2$ (désignant respectivement la traduction de deux expressions C) et retournent le résultat de l'opération arithmétique dans un enregistrement. Les fonctions conditionnelles (gt, ge,

.1. $[\![<e_1: expr> > <e_2: expr>]\!]_{expr} \rightarrow gt([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.2. $[\![<e_1: expr> >= <e_2: expr>]\!]_{expr} \rightarrow ge([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.3. $[\![<e_1: expr> < <e_2: expr>]\!]_{expr} \rightarrow lt([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.4. $[\![<e_1: expr> <= <e_2: expr>]\!]_{expr} \rightarrow le([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.5. $[\![<e_1: expr> == <e_2: expr>]\!]_{expr} \rightarrow eq([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.6. $[\![<e_1: expr> != <e_2: expr>]\!]_{expr} \rightarrow ne([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.7. $[\![<e_1: expr> \&\& <e_2: expr>]\!]_{expr} \rightarrow bAnd([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.8. $[\![<e_1: expr> || <e_2: expr>]\!]_{expr} \rightarrow bOr([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.9. $[\![<e_1: expr> * <e_2: expr>]\!]_{expr} \rightarrow mult([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.10. $[\![<e_1: expr> + <e_2: expr>]\!]_{expr} \rightarrow plus([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.11. $[\![<e_1: expr> - <e_2: expr>]\!]_{expr} \rightarrow minus([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.12. $[\![<e_1: expr> / <e_2: expr>]\!]_{expr} \rightarrow div([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.13. $[\![<e_1: expr> \% <e_2: expr>]\!]_{expr} \rightarrow mod([\![<e_1: expr>]\!], [\![<e_2: expr>]\!])$
.14. $[\![!<expr>]\!]_{expr} \rightarrow not([\![<expr>]\!])$

Règle 3.5 – Traduction des expressions conditionnelles et arithmétiques

$$
\begin{aligned}
>(expr1, expr2) \triangleq \text{IF } (expr1.val > expr2.val) \text{THEN } [val \mapsto 1] \text{ ELSE } [val \mapsto 0]\\
&ge(expr1, expr2) \triangleq \text{IF } (expr1.val \geq expr2.val) \text{THEN } [val \mapsto 1] \text{ ELSE } [val \mapsto 0]\\
<(expr1, expr2) \triangleq \text{IF } (expr1.val < expr2.val) \text{THEN } [val \mapsto 1] \text{ ELSE } [val \mapsto 0]\\
&le(expr1, expr2) \triangleq \text{IF } (expr1.val \leq expr2.val) \text{THEN } [val \mapsto 1] \text{ ELSE } [val \mapsto 0]\\
&eq(expr1, expr2) \triangleq \text{IF } (expr1 = expr2) \text{THEN } [val \mapsto 1] \text{ ELSE } [val \mapsto 0]\\
&ne(expr1, expr2) \triangleq \text{IF } (expr1 \neq expr2) \text{THEN } [val \mapsto 1] \text{ ELSE } [val \mapsto 0]\\
&bAnd(expr1, expr2) \triangleq \text{IF } (expr1.val = 1) \land (expr2.val = 1) \text{THEN } [val \mapsto 1] \text{ ELSE } [val \mapsto 0]\\
&bOr(expr1, expr2) \triangleq \text{IF } (expr1.val = 1) \lor (expr2.val = 1) \text{THEN } [val \mapsto 1] \text{ ELSE } [val \mapsto 0]\\
&mult(expr1, expr2) \triangleq [val \mapsto (expr1.val * expr2.val)]\\
&plus(expr1, expr2) \triangleq [val \mapsto (expr1.val + expr2.val)]\\
&minus(expr1, expr2) \triangleq [val \mapsto (expr1.val - expr2.val)]\\
&div(expr1, expr2) \triangleq [val \mapsto (expr1.val \div expr2.val)]\\
&mod(expr1, expr2) \triangleq [val \mapsto (expr1.val \% expr2.val)]\\
¬(expr) \triangleq \text{IF } (expr.val \neq 0) \text{THEN } [val \mapsto 1] \text{ ELSE } [val \mapsto 0]
\end{aligned}
$$

Listing 3.9 – Définition des fonctions TLA+ arithmétiques et conditionnelles

lt, le, eq, ne, $bAnd$ et bOr) prennent en arguments deux expressions TLA+ $expr1$ et $expr2$ et retournent soit $[val \mapsto 1]$ soit $[val \mapsto 0]$.

3.3.4.3 Évaluation d'une *lvalue* comme une expression

Les identificateurs de variables sont notamment des *lvalue*s. Toute *lvalue* peut être évaluée comme une expression représentant le contenu de la lvalue. C'est le cas pour les *lvalue*s figurant à droite de l'opérateur d'affectation « = ». Dans ce cas, elle est désignée par le terme *rvalue* (« *right value* »). Sa traduction est donnée par la règle 3.6.

$$[\![<lval>]\!]_{expr} \rightarrow load(id, mem, [\![<lval>]\!]_{lval})$$

Règle 3.6 – Traduction de l'évaluation d'une lvalue comme une expression

Cette traduction consiste à charger la valeur contenue dans la *lvalue* <lval> en utilisant l'opérateur $load()$ défini dans le listing 3.6.

Exemple 3.3.2. *L'expression* x *(ligne 11 du listing 3.1a) est traduite comme suit :*

$[\![\text{x}]\!]_{expr} \underset{R.3.6}{\longrightarrow} load(id, mem, [\![\text{x}]\!]_{lval}) \underset{R.3.10}{\longrightarrow} load(id, mem, Addr_x)$

3.3.4.4 Les références

Le référencement d'une variable se fait en C avec l'opérateur `&`. La règle de traduction d'une référence sur une *lvalue* est donnée par la règle 3.7.

$$[\![\texttt{\&}\text{<lval>}]\!]_{expr} \rightarrow [\![\text{<lval>}]\!]_{lval}$$

Règle 3.7 – Référencement d'une lvalue

Comme expliqué dans la section précédente, chaque variable globale ou locale possède une adresse constante TLA+. La traduction d'une référence à une *lvalue* est l'emplacement mémoire de cette *lvalue*.

Exemple 3.3.3. *L'expression* `&x` *se traduit ainsi :*

$[\![\texttt{\&x}]\!]_{expr} \underset{R.3.7}{\longrightarrow} [\![\text{x}]\!]_{lval}$

Et comme $[\![\text{x}]\!]_{lval}$ *se traduit en une adresse, la traduction de l'expression* `&x` *est* $Addr_x$ *telle que :*

$Addr_x \triangleq [loc \mapsto \text{“data”}, offs \mapsto 0]$

3.3.4.5 Arithmétique des pointeurs

Les seules opérations arithmétiques valides sur les pointeurs sont les opérations d'addition/soustraction des entiers et la soustraction de pointeurs. Ces opérations tiennent compte du type pointé et de sa taille mémoire.

.1. $[\![\text{<e}_1\text{: expr>} +_{pi} \text{<e}_2\text{: expr>}]\!]_{expr} \rightarrow plusPI([\![\text{<e}_1\text{: expr>}]\!]_{expr}, [\![\text{<e}_2\text{: expr>}]\!]_{expr})$
.2. $[\![\text{<e}_1\text{: expr>} -_{pi} \text{<e}_2\text{: expr>}]\!]_{expr} \rightarrow minusPI([\![\text{<e}_1\text{: expr>}]\!]_{expr}, [\![\text{<e}_2\text{: expr>}]\!]_{expr}, [val \mapsto \texttt{size(type(e}_1\texttt{))}])$

Règle 3.8 – Addition et soustraction d'un pointeur avec un entier

La traduction d'une opération d'addition $+_{pi}$ ou d'une soustraction $-_{pi}$ d'une expression de type pointeur <e$_1$: expr> et d'une expression entière <e$_2$: expr> est donnée par la règle 3.8. Cette traduction consiste en un déplacement dans la mémoire. Elle utilise les opérateurs $minusPI$ et $plusPI$ que nous définissons dans le listing 3.10.

$$plusPI(ptr, expr, size_of_ptr) \triangleq [loc \mapsto ptr.loc, offs \mapsto ptr.offs + mult(expr, size_of_ptr)]$$

$$minusPI(ptr, expr, size_of_ptr) \triangleq [loc \mapsto ptr.loc, offs \mapsto ptr.offs + mult(expr, size_of_ptr)]$$

Listing 3.10 – Définition des opérateurs $plusPI()$ et $minusPI()$

L'opérateur $plusPI$ prend en argument une adresse ptr, une expression $expr$ de type entier et la taille du type pointé par ptr. Il retourne une adresse TLA+ dont la valeur du champ $offs$ est incrémentée par le résultat de la multiplication de la valeur retournée par $expr$ avec la taille du type pointé $size_of_ptr$.

Le principe est le même pour la soustraction d'un entier par un pointeur. La soustraction de pointeurs de même type est traduite selon la règle 3.9. Cette soustraction fournit le nombre d'éléments du type en question situés entre les deux pointeurs (cette valeur est exprimée par un nombre entier).

$$[\![\text{<e}_1\text{: expr>} -_{pp} \text{<e}_2\text{: expr>}]\!]_{expr} \rightarrow ([\![\text{<e}_1\text{: expr>}]\!]_{expr}.offs - [\![\text{<e}_2\text{: expr>}]\!]_{expr}.offs) \div \texttt{size(type(}e_1\texttt{))}$$

Règle 3.9 – Traduction d'une soustraction de pointeurs

Exemple 3.3.4. *Soient* `p` *et* `q` *deux variables de type* `struct point*`*. En appliquant la règle 3.8.2, l'expression* `p - q` *se traduit ainsi :*

$$[\![p \text{ - } q]\!]_{expr} \underset{R.3.8.2}{\longrightarrow} ([\![p]\!]_{expr}.offs - [\![q]\!]_{expr}.offs) \div \texttt{size(type(p))}$$

Comme la taille du type `struct point*` *est donnée par* $size_of_point$*, en appliquant la règle 3.6 sur les expressions* `p` *et* `q`*, ensuite, la règle 3.10 sur les lvalues* `p` *et* `q`*, le reste de la traduction se fait comme suit :*

$$\underset{R.3.1}{\longrightarrow} ([\![p]\!]_{expr}.offs - [\![q]\!]_{expr}.offs) \div size_of_point$$

$$\underset{R.3.6}{\longrightarrow} (load(id, mem, [\![p]\!]_{lval}).offs - load(id, [\![q]\!]_{lval}).offs) \div size_of_point$$

$$\underset{R.3.10}{\longrightarrow} (load(id, mem, Addr_p).offs - load(id, mem, Addr_q).offs) \div size_of_point$$

Nous rappelons que la constante TLA+ $size_of_point$ est générée automatiquement par C2TLA+ dans le module $Parameters$. Cette constante vaut 1 vu qu'un pointeur est stocké dans une cellule mémoire.

3.3.5 *Lvalue*s et offsets

Comme expliqué précédemment, une *lvalue* détermine un emplacement dans la mémoire. Une *lvalue* peut s'écrire de la forme **VAR_ID**<offs>, avec **VAR_ID** étant un identificateur de variable (locale ou globale) ou de la forme `*(`<expr>`)` <offs>.

Une *lvalue* peut contenir un offset <offs> qui peut être vide (désigné par ϵ_{offs}) ou qui spécifie une séquence de champs de structure et/ou d'index de tableaux.

3.3.5.1 *Lvalue*s

La règle de traduction d'une *lvalue* de la forme **VAR_ID**<offs> est donnée par la règle 3.10.

$$[\![\textbf{VAR_ID}\text{<offs>}]\!]_{lval} \rightarrow [loc \mapsto Addr_[\![\textbf{VAR_ID}]\!]_{vid}.loc, offs \mapsto Addr_[\![\textbf{VAR_ID}]\!]_{vid}.offs[\![\text{<offs>}]\!]_{offs}]$$

Règle 3.10 – Traduction d'une *lvalue* d'une variable

La traduction consiste à générer une adresse TLA+ telle que le champ offs est incrémenté par la valeur générée par la règle $[\![$<offs>$]\!]_{\mathit{offs}}$ que nous expliquons dans la section 3.3.5.2. Ainsi, pour les lvalues sans offset, par exemple la variable `x` de l'exemple de la figure 3.1a, la traduction n'est autre que l'adresse TLA+ $Addr_x$ générée par la règle 3.1.4.

La traduction d'une *lvalue* de la forme `*(`<expr>`)` suit la règle 3.11. L'expression <expr> désigne un pointeur dont l'évaluation est une adresse mémoire.

$$[\![*(\text{<expr>})\text{<offs>}]\!]_{lval} \rightarrow \text{LET } ptr \triangleq [\![\text{<expr>}]\!]_{expr} \quad \text{IN } [loc \mapsto ptr.loc, offs \mapsto ptr.offs\ [\![\text{<offs>}]\!]_{offs}\ ]$$

Règle 3.11 – Traduction d'une *lvalue* de type pointeur

La traduction consiste à définir une variable temporaire ptr qui contient cette adresse mémoire et qui retourne l'adresse de la variable pointée par le pointeur. Ceci, en tenant compte de l'offset <offs>.

Exemple 3.3.5. *La traduction de la lvalue* `*(p).x` *(ligne 32 de l'exemple de la figure 3.1a) se fait comme suit :*

$$[\![*(\mathtt{p}).\mathtt{x}]\!]_{lval} \underset{R.3.11}{\longrightarrow} \text{LET } ptr \triangleq [\![\mathtt{p}]\!]_{expr} \quad \text{IN } [loc \mapsto ptr.loc, offs \mapsto ptr.offs[\![.\mathtt{x}]\!]_{offs}] \qquad (1)$$

En remplaçant $[\![.\mathtt{x}]\!]_{offs}$ *par le résultat vu précédemment, cela donne :*

$$(1) \underset{R.3.13}{\longrightarrow} \text{LET } ptr \triangleq [\![\mathtt{p}]\!]_{expr} \quad \text{IN } [loc \mapsto ptr.loc, offs \mapsto ptr.offs + Offset_point_x] \qquad (2)$$

En appliquant la règle 3.6 sur l'expression `p` *ensuite la règle 3.10 sur la lvalue* `p`*, le reste de la traduction se fait comme suit :*

$$(2) \underset{R.3.6}{\longrightarrow} \text{LET } ptr \triangleq load(id, data, [\![\mathtt{p}]\!]_{lval}) \quad \text{IN } [loc \mapsto ptr.loc, offs \mapsto ptr.offs + Offset_point_x] \qquad (3)$$

$$(3) \underset{R.3.10}{\longrightarrow} \text{LET } ptr \triangleq load(id, data, Addr_p) \quad \text{IN } [loc \mapsto ptr.loc, offs \mapsto ptr.offs + Offset_point_x]$$

3.3.5.2 Les offsets

(a) L'offset vide. Un offset vide se traduit (règle 3.12) par un code vide , désigné par $\llcorner\!\lrcorner$ et qui correspond à un déplacement qui vaut 0.

$$[\![\varepsilon_{offs}]\!]_{offs} \rightarrow \llcorner\!\lrcorner$$

Règle 3.12 – Traduction d'un offset vide

(b) Les structures de données. Le langage C permet au programmeur de construire ses propres types de données agrégées. Une variable de type structure est une lvalue. Un champ de cette structure est un offset.

La traduction d'un offset dont la syntaxe est de la forme **.FIELD_ID**<offs>, est donnée alors par la règle 3.13.

$$[\![\texttt{.}\mathbf{FIELD_ID}\text{<offs>}]\!]_{offs} \rightarrow + \mathit{Offset_}[\![\mathbf{FIELD_ID}]\!]_{oid}[\![\text{<offs>}]\!]_{offs}$$

Règle 3.13 – Traduction d'un *offset* d'une structure de données

Exemple 3.3.6. *La traduction de l'offset* `.y` *(ligne 32 de l'exemple 3.1a) se fait comme suit :*

$$[\![\texttt{.y}]\!]_{offs} \underset{R.3.13}{\longrightarrow} + \mathit{Offset_}[\![\texttt{y}]\!]_{oid}[\![\varepsilon_{offs}]\!]_{offs} \underset{R.3.13}{\longrightarrow} + \mathit{Offset_point_y}$$

Étant donné que le champ `y` *est un entier, la constante* $\mathit{Offset_point_y}$ *vaut* 1. *Nous rappelons que cette constante est générée dans le module* $\mathit{Parameters}$ *selon la règle 3.3.*

(c) Les tableaux. L'accès à un élément d'un tableau en C se fait par la manipulation de son adresse de base, de son type et de l'indice de l'élément. L'indice d'un tableau est une expression entière. Un déplacement s'écrivant de la forme `[`<expr>`]`<offs> se traduit alors selon la règle 3.14.

$$[\![\texttt{[}\text{<e:expr>}\texttt{]}\text{<offs>}]\!]_{offs} \rightarrow + ([\![\text{<expr>}]\!]_{expr} * \texttt{\textbf{size(type(}}\text{<e:expr>}\texttt{\textbf{)))}})\ [\![\text{<offs>}]\!]_{offs}$$

Règle 3.14 – Traduction d'un *offset* d'un tableau

Cette règle consiste à traduire l'expression entière <e:expr> contenue dans l'indice du tableau ainsi que l'offset <offs>, en tenant compte de la taille de l'expression <e : expr>. Cette taille est déterminée par **`size(type(`**<e:expr>**`))`**.

Exemple 3.3.7. *Pour l'expression* `tab[i].x` *de l'exemple de la figure 3.1a, la traduction du déplacement* `[i].x` *se fait comme suit :*

$$[\![\texttt{[i].x}]\!]_{offs} \underset{R.3.14}{\longrightarrow} + ([\![\texttt{i}]\!]_{expr} * \texttt{\textbf{size(type(}}\texttt{i}\texttt{\textbf{))}})[\![\texttt{.x}]\!]_{offs} \qquad (1)$$

Comme l'expression `i` *est une variable entière, sa taille est égale à* 1. *En remplaçant la règle* $[\![\texttt{.x}]\!]_{offs}$ *par le résultat de la traduction vu précédemment, cela donne :*

$$(1) \underset{R.3.13}{\longrightarrow} + ([\![\texttt{i}]\!]_{expr} * \mathit{size_of_int}) + \mathit{Offset_point_x} \qquad (2)$$

En appliquant la règle 3.6 sur l'expression `i` *puis la règle 3.10 sur la lvalue* `i`, *le reste de la traduction se fait comme suit :*

$$(2) \underset{R.3.6}{\longrightarrow} + (\mathit{load}(\mathit{id}, \mathit{mem}, [\![\texttt{i}]\!]_{lval}) * \mathit{size_of_int}) + \mathit{Offset_point_x} \qquad (3)$$

$$(3) \underset{R.3.10}{\longrightarrow} + (\mathit{load}(\mathit{id}, \mathit{mem}, \mathit{Addr_main_i}) * \mathit{size_of_int}) + \mathit{Offset_point_x}$$

L'expression générée correspond au déplacement requis, par rapport à l'adresse de base de la variable `tab`, *pour accéder au champ* `x`.

3.3.6 Flot de contrôle intra-procédural

Pour déterminer le flot de contrôle dans chaque fonction du programme, CIL maintient un graphe de flot de contrôle GFC (« *Control Flow Graph* » en anglais). Il s'agit d'une représentation sous forme de graphe constitué d'un ensemble de nœuds et d'arcs. Chaque nœud du graphe constitue une instruction et comporte un arc le reliant avec le nœud de l'instruction suivante, à l'exception d'un nœud appelé nœud final et qui n'a pas de successeur. CIL affecte à chaque nœud un identifiant unique et détient pour chaque nœud un ensemble d'informations comme le type de l'instruction (bloc, structure conditionnelle ou structure itérative) et la liste de ses successeurs et de ses prédécesseurs.

Dans C2TLA+, chaque instruction C correspond à un opérateur TLA+ qui décrit l'opération effectuée par celle-ci. Le flot d'exécution du programme est assuré par le constructeur conditionnel TLA+ CASE ainsi que par les appels aux opérateurs TLA+ générés à partir des instructions. Nous détaillons dans cette section comment les instructions de flot de contrôle intra-procédural sont traduites dans TLA+.

3.3.6.1 Règles et notations préliminaires

Dans la suite de cette section, nous utilisons les fonctions OCaml suivantes :

- La fonction **`succ()`** prend en argument une instruction C <s : stmt> et rend l'instruction C suivante en se basant sur la GFC du programme C.
- La fonction **`fst()`** prend en argument un identificateur de fonction C **FUN_ID** et retourne la première instruction C <s:stmt> de son bloc d'instructions.
- La fonction **`func()`** prend en argument une instruction C <s:stmt> et retourne le nom de la fonction dans laquelle l'instruction <s:stmt> est définie.
- La fonction **`lbl()`** prend en argument une instruction C <s : stmt> et retourne l'identifiant (numérique) de l'instruction, attribué par CIL.

CIL attribue pour chaque instruction C un identifiant numérique, appelé *étiquette*. C2TLA+ génère (selon la règle 3.15), pour une instruction <s : stmt>, un identificateur unique, sous forme d'une chaîne de caractères. Cette chaîne spécifie le nom de la fonction dans laquelle l'instruction est définie et l'étiquette attribuée par CIL.

$$[\![\text{<s:stmt>}]\!]_{sid} \rightarrow \text{“}\mathtt{func(}\text{<s:stmt>}\mathtt{)_lbl(}\text{<s:stmt>}\mathtt{)}\text{”}$$

Règle 3.15 – Génération d'un identificateur d'instruction

Exemple 3.3.8. *L'instruction* `inc(3)` *(ligne 28 de la figure 3.1a) a une étiquette (attribuée par CIL) égale à 28 et elle est définie dans la fonction* `main()`*. L'identificateur généré par CIL est alors “main_28”.*

Dans C2TLA+, chaque instruction C est traduite en un opérateur qui prend en paramètre l'état de la mémoire mem contenant les valeurs de $data$, $stack$, $register$ et

ret. Il met à jour la mémoire globale $data$ ou locale $stack$ s'il s'agit d'une instruction d'affectation. Il met ensuite à jour la pile $register$ pour passer le flot de contrôle à l'instruction suivante. Enfin, il met à jour le champ ret s'il s'agit d'une instruction `return`.

Dans la suite de ce chapitre, tout opérateur TLA+ modélisant une instruction C est paramétré par un identificateur id. Puisque nous modélisons un programme séquentiel (un seul processus), nous considérons que cet identificateur vaut 1. Le paramétrage des opérateurs TLA+ avec l'identificateur id nous permet de réutiliser les règles de traduction définies dans ce chapitre pour la modélisation de programmes concurrents (que nous décrivons dans le chapitre 4.

La mise à jour de la pile $register$ se fait en utilisant l'opérateur $update_reg()$ défini dans le listing 3.11.

$$update_reg(reg, nxt) \triangleq [\langle [pc \mapsto nxt, fp \mapsto Head(reg).fp] \rangle \circ Tail(reg)]$$

Listing 3.11 – Définition de l'opérateur $update_reg()$

L'opérateur $update_reg()$ prend comme argument la pile des registres reg du programme à mettre à jour et l'identificateur de l'instruction suivante nxt. Il retourne un nouvel enregistrement dont le champ pc contient l'identificateur de l'instruction suivante désignée par nxt et le champ fp contenant la valeur de l'adresse de base du cadre de pile de la fonction en exécution, désignée par l'expression $Head(reg).fp$.

3.3.6.2 Instruction d'affectation

Comme expliqué dans la section 3.3.5, la partie gauche de l'opérateur d'affectation « = » doit être une *lvalue*. La traduction d'une instruction d'affectation s de type <stmt>, dont la syntaxe de la forme $[\![$<l:lval> = <e:expr>$]\!]_{stmt}$ se fait en appliquant la règle 3.16.

$$\begin{array}{l} [\![\text{<s:stmt> ::=<l:lval> = <e:expr>;}]\!]_{stmt} \rightarrow \\ \quad stmt_[\![\text{<s:stmt>}]\!]_{sid}(id, mem) \triangleq \\ \qquad \text{LET } nmem \triangleq store(id, mem, [\![\text{<l:lval>}]\!]_{lval}, [\![\text{<e:expr>}]\!]_{expr}) \\ \qquad \text{IN } [mem \text{ EXCEPT } !.data = nmem.data, !.stack = nmem.stack, \\ \qquad\qquad !.register = update_reg(mem.register[id], \text{“}[\![\textbf{succ(s)}]\!]_{sid}\text{”})] \end{array}$$

Règle 3.16 – Traduction d'une instruction d'affectation

La règle consiste à définir une variable temporaire $nmem$ qui sauvegarde le nouvel état mémoire retourné par la fonction $store()$ (définie dans la section 3.3.3). Le champ $register$ est mis à jour afin de pointer vers l'instruction suivante. Cette mise à jour consiste à modifier le champ pc de la tête de pile $mem.register[id]$ par l'identificateur de l'instruction suivante, généré par la règle $[\![\textbf{succ(s)}]\!]_{sid}$.

Exemple 3.3.9. *Soit une instruction* `x = y + 1` *définie dans une fonction* `main()` *et ayant comme identificateur “main_1”. En appliquant la règle 3.16 sur le code* `x = y + 1`*, cela donne :*

$$
\begin{array}{l}
[\![\texttt{x = y + 1;}]\!]_{stmt} \rightarrow \\
\quad stmt_[\![\texttt{x = y + 1}]\!]_{sid}(id, mem) \triangleq \qquad\qquad \textit{(1)} \\
\qquad \text{LET } nmem \triangleq store(id, mem, [\![\texttt{x}]\!]_{lval}, [\![\texttt{y+1}]\!]_{expr}) \\
\qquad \text{IN } [mem \text{ EXCEPT } !.data = nmem.data, !.stack = nmem.stack, \\
\qquad\qquad !.register = update_reg(mem.register[id], \text{“}[\![\textbf{succ}(y = x + 1;)]\!]_{sid}\text{”})]
\end{array}
$$

Sachant que l'instruction suivante a comme identificateur $main_2$, la suite de la traduction consiste à appliquer la règle 3.15.

$$
\begin{array}{l}
\textit{(1)} \underset{R.3.15}{\longrightarrow} stmt_main_1(id, data) \triangleq \\
\qquad \text{LET } nmem \triangleq store(id, mem, [\![\texttt{x}]\!]_{lval}, [\![\texttt{y+1}]\!]_{expr}) \qquad \textit{(2)} \\
\qquad \text{IN } [mem \text{ EXCEPT } !.data = nmem.data, !.stack = nmem.stack, \\
\qquad\qquad !.register = update_reg(mem.register[id], \text{“main_2”})]
\end{array}
$$

Le reste de la traduction consiste à appliquer respectivement la règle 3.10 sur les lvalues et les règles 3.6, 3.14 et 3.5.10 sur les expressions.

$$
\begin{array}{l}
\textit{(2)} \underset{R.3.10}{\longrightarrow} stmt_main_1(id, data) \triangleq \\
\qquad \text{LET } nmem \triangleq store(id, mem, Addr_x, [\![\texttt{y+1}]\!]_{expr}) \qquad \textit{(3)} \\
\qquad \text{IN } [mem \text{ EXCEPT } !.data = nmem.data, !.stack = nmem.stack, \\
\qquad\qquad !.register = update_reg(mem.register[id], \text{“main_2”})]
\end{array}
$$

$$
\begin{array}{l}
\textit{(3)} \underset{R.3.5.10}{\longrightarrow} stmt_main_1(id, mem) \triangleq \\
\qquad \text{LET } nmem \triangleq store(id, mem, Addr_x, plus([\![\texttt{y}]\!]_{expr}, [\![\texttt{1}]\!]_{expr}) \qquad \textit{(4)} \\
\qquad \text{IN } [mem \text{ EXCEPT } !data = nmem.data, !.stack = nmem.stack, \\
\qquad\qquad !.register = update_reg(mem.register[id], \text{“main_2”})]
\end{array}
$$

$$
\begin{array}{l}
\textit{(4)} \underset{R.3.14}{\longrightarrow} stmt_main_1(id, mem) \triangleq \\
\qquad \text{LET } nmem \triangleq store(id, mem, Addr_x, plus([\![\texttt{y}]\!]_{expr}, [val \mapsto 1]) \qquad \textit{(5)} \\
\qquad \text{IN } [mem \text{ EXCEPT } !.data = nmem.data, !.stack = nmem.stack, \\
\qquad\qquad !.register = update_reg(mem.register[id], \text{“main_2”})]
\end{array}
$$

$$
\begin{array}{l}
\textit{(5)} \underset{R.3.6}{\longrightarrow} stmt_main_1(id, mem) \triangleq \qquad\qquad \textit{(6)} \\
\qquad \text{LET } nmem \triangleq store(id, mem, Addr_x, plus(load(id, mem\ [\![\texttt{y}]\!]_{lval}), [val \mapsto 1]) \\
\qquad \text{IN } [mem \text{ EXCEPT } !data = nmem.data, !.stack = nmem.stack, \\
\qquad\qquad !.registerupdate_reg(mem.register[id], \text{“main_2”})]
\end{array}
$$

Enfin, la traduction finale de l'instruction `x = y + 1;` *donne le code TLA+ suivant :*

$$
\begin{array}{l}
\textit{(6)} \underset{R.3.10}{\longrightarrow} stmt_main_1(id, mem) \triangleq \\
\qquad \text{LET } nmem \triangleq store(id, data, Addr_x, plus(load(id, mem, Addr_y), [val \mapsto 1]) \\
\qquad \text{IN } [mem \text{ EXCEPT } !data = nmem.data, !.stack = nmem.stack, \\
\qquad\qquad !.register = update_reg(mem.register[id], \text{“main_2”})]
\end{array}
$$

La traduction finale consiste dans un premier temps à mettre à jour les mémoires $data$ et $stack$ avec la nouvelle valeur de la variable x *et dans un second temps à modifier la pile $register[id]$ pour pointer vers l'instruction suivante.*

(a) Instruction conditionnelle. L'instruction `if` s'écrit sous la forme `if` <e : expr> <s_1: stmt> `else` <s_2: stmt>. L'expression <e : expr> peut être une condition booléenne ou une expression délivrant une valeur entière. Étant donné que le constructeur IF/THEN/ELSE

de TLA+ n'utilise que des conditions booléennes, l'expression <e : expr> est traduite en une condition booléenne selon la règle 3.17.

$$
\begin{array}{ll}
.1.\ [\![\text{<e:expr>}]\!]_{cond} & \rightarrow not_zero([\![\text{<e:expr>}]\!]_{expr}) \\
.2.\ [\![\text{<e:expr>}]\!]_{cond} & \rightarrow not_null([\![\text{<e:expr>}]\!]_{expr})
\end{array}
$$

Règle 3.17 – Traduction d'une condition

$$
\begin{array}{lcl}
not_zero(expr) & \triangleq & \text{IF } (expr \neq [val \mapsto 0])\text{THEN TRUE ELSE FALSE} \\
not_null(expr) & \triangleq & \text{IF } (expr \neq [loc \mapsto Null, offs \mapsto Null])\text{THEN TRUE ELSE FALSE}
\end{array}
$$

Listing 3.12 – Définition des opérateurs $not_zero()$ et $not_null()$

Si <e : expr> est une expression entière, la traduction $[\![\text{<e : expr>}]\!]_{cond}$ utilise la règle 3.17.1. Si <e : expr> est de type pointeur, la traduction $[\![\text{<e : expr>}]\!]_{cond}$ utilise la règle 3.17.2. Les opérateurs $not_zero()$ et $not_null()$ sont définis dans le listing 3.12. Ces deux opérateurs sont définis dans le module $Runtime$. Ils prennent en argument une expression TLA+ $expr$ et retournent TRUE si $expr$ est différente de $[val \mapsto 0]$ et $[loc \mapsto Null, offs \mapsto Null]$ respectivement et FALSE sinon.

Dans le GFC, l'instruction `if` dispose de deux successeurs : le premier est le bloc `then` et le deuxième est le bloc `else`. La règle de traduction d'une instruction conditionnelle est donnée alors par la règle 3.18.

$$
\begin{array}{l}
[\![\text{<s:stmt>::=}\texttt{if}\text{<e:expr><}s_1\text{:stmt> }\texttt{else}\text{ <}s_2\text{:stmt>}]\!]_{stmt} \rightarrow \\
\quad stmt_[\![\text{<s:stmt>}]\!]_{sid}(id, mem) \triangleq \\
\qquad \text{IF } ([\![\text{<e:exp>}]\!]_{cond}) \\
\qquad \text{THEN } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“}[\![\text{<}s_1\text{:stmt>}]\!]_{sid}\text{”})] \\
\qquad \text{ELSE } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“}[\![\text{<}s_2\text{:stmt>}]\!]_{sid}\text{”})] \\
\quad [\![\text{<}s_1\text{:stmt>}]\!]_{stmt} \\
\quad [\![\text{<}s_2\text{:stmt>}]\!]_{stmt}
\end{array}
$$

Règle 3.18 – Traduction d'une instruction conditionnelle

La règle 3.18 fonctionne ainsi : si la condition $[\![\text{<e : expr>}]\!]_{cond}$ est évaluée à vrai, alors on retourne un nouvel état de la mémoire dont le champ pc de la tête de pile $mem.register[id]$ est remplacé par un l'identificateur de l'instruction <s_1 : stmt>, sinon par celui de l'instruction <s_2 : stmt>. La traduction consiste ensuite à traduire les blocs d'instructions <s_1: stmt> et <s_2: stmt> selon la règle 3.21, que nous définissons plus tard. C2TLA+ utilise le GFC afin de déterminer les instructions <s_1 : stmt> et <s_2 : stmt>, qui sont les deux successeurs de l'instruction `if`.

Exemple 3.3.10. *Soit l'instruction* `if (i > j) tmp = i; else tmp = j;` *ayant un identificateur égal “max_13”. L'application de la règle 3.18 sur cette instruction conditionnelle donne le résultat suivant :*

$$
\begin{array}{l}
[\![\text{<s:stmt>::=}\texttt{if (i > j) tmp = i; else tmp = j;}]\!]_{stmt} \\
\underset{R.3.6}{\longrightarrow} \; stmt_[\![\texttt{if (i > j) tmp = i; else tmp = j;}]\!]_{sid}(id, mem) \triangleq \qquad \textit{(1)} \\
\quad \text{IF } ([\![\texttt{(i > j)}]\!]_{cond}) \\
\quad \text{THEN } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“}[\![\texttt{tmp = i;}]\!]_{sid}\text{”})] \\
\quad \text{ELSE } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“}[\![\texttt{tmp = j;}]\!]_{sid}\text{”})] \\
\quad [\![\texttt{tmp = i;}]\!]_{stmt} \\
\quad [\![\texttt{tmp = j;}]\!]_{stmt}
\end{array}
$$

Supposons que l'identificateur de l'instruction `tmp = i` *vaut "max_14" et celui de* `tmp = j` *vaut "max_15", en appliquant la règle 3.15, la traduction donne le code suivant :*

$$
\begin{array}{l}
\textit{(1)} \underset{3.15}{\longrightarrow} \; stmt_max_13(id, mem) \triangleq \qquad \textit{(2)} \\
\quad \text{IF } ([\![\texttt{(i > j)}]\!]_{expr} \neq [\![\text{<e:expr>}]\!]_{null_expr}) \\
\quad \text{THEN } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“max_14”})] \\
\quad \text{ELSE } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“max_15”})] \\
\quad [\![\texttt{tmp = i;}]\!]_{stmt} \\
\quad [\![\texttt{tmp = j;}]\!]_{stmt}
\end{array}
$$

Dans la suite de la traduction, nous ne nous intéressons pas à la traduction de $[\![\texttt{tmp = i;}]\!]_{stmt}$ *et* $[\![\texttt{tmp = j;}]\!]_{stmt}$*. En appliquant la règle 3.17.1 sur la condition* `(i > j)`*, le reste de la traduction se fait comme suit :*

$$
\begin{array}{l}
\textit{(2)} \underset{R.3.17.1}{\longrightarrow} \; stmt_max_13(id, mem) \triangleq \qquad \textit{(3)} \\
\quad \text{IF } (not_zero([\![\texttt{(i > j)}]\!]_{expr}) \\
\quad \text{THEN } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“max_14”})] \\
\quad \text{ELSE } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“max_15”})] \\
\quad [\![\texttt{tmp = i;}]\!]_{stmt} \\
\quad [\![\texttt{tmp = j;}]\!]_{stmt}
\end{array}
$$

En appliquant la règle 3.5.1 sur l'expression `(i < j)`*, ensuite la règle 3.6 sur les expressions* `i` *et* `j` *et enfin la règle 3.10 sur les lvalues* `i` *et* `j`*, le reste de la traduction se fait comme suit :*

$$
\begin{array}{l}
\textit{(3)} \underset{R.3.5.1}{\longrightarrow} \; stmt_max_13(id, mem) \triangleq \\
\quad \text{IF } (not_zero(gt([\![\texttt{ i }]\!]_{expr}, [\![\texttt{ j }]\!]_{expr}))) \qquad \textit{(4)} \\
\quad \text{THEN } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“max_14”})] \\
\quad \text{ELSE } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“max_15”})] \\
\quad [\![\texttt{tmp = i;}]\!]_{stmt} \\
\quad [\![\texttt{tmp = j;}]\!]_{stmt}
\end{array}
$$

$$
\begin{array}{l}
\textit{(4)} \underset{R.3.6}{\longrightarrow} \; stmt_max_13(id, mem) \triangleq \\
\quad \text{IF } (not_zero(gt(load(id, mem, [\![\texttt{ i }]\!]_{lval}), load(id, mem, [\![\texttt{ j }]\!]_{lval})))) \qquad \textit{(5)} \\
\quad \text{THEN } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“max_14”})] \\
\quad \text{ELSE } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“max_15”})] \\
\quad [\![\texttt{tmp = i;}]\!]_{stmt} \\
\quad [\![\texttt{tmp = j;}]\!]_{stmt}
\end{array}
$$

$$\textit{(5)} \underset{R.3.6}{\longrightarrow} stmt_max_13(id, mem) \triangleq$$
$$\text{IF } (not_zero(gt(load(id, mem, Addr_max_param_i), load(id, mem, Addr_max_param_i))))$$
$$\text{THEN } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“max_14”})]$$
$$\text{ELSE } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“max_15”})]$$
$$[\![\texttt{tmp = i;}]\!]_{stmt}$$
$$[\![\texttt{tmp = j;}]\!]_{stmt}$$

(b) Instructions de saut et instruction étiquetée. Les instructions `goto`, `break` et `continue` sont utilisées pour réaliser des sauts inconditionnels dans un programme. La traduction de ces instructions est donnée par la règle 3.19.

$[\![$<s:stmt>::=`goto` **LABEL_ID** `;` | `break;` | `continue;`$]\!]_{stmt} \rightarrow$
$stmt_[\![$<s:stmt>$]\!]_{sid}(id, mem) \triangleq [\, mem \text{ EXCEPT } !.register = update_reg(mem.register[id],$ “$[\![$**succ(s)**$]\!]_{sid}$”$)]$

Règle 3.19 – Traduction des instructions de saut

En C, l'instruction `break` termine l'exécution de l'instruction itérative englobante `do`, `while`, `for`, ou `switch` dans laquelle elle apparaît alors que l'instruction `continue` permet de transmettre le contrôle à la fin d'accolade fermante la plus interne de la boucle. Dans le GFC, le successeur d'une instruction de type `goto`, `break` ou `continue` est l'instruction vers laquelle le flot de contrôle est passé. Par exemple, le successeur de `break` est l'instruction qui suit l'instruction itérative englobante, cette instruction est retournée par **succ()**.

La traduction d'une instruction de saut consiste à mettre à jour la valeur de $mem.register[id]$, en appelant l'opérateur $update_reg()$.

Exemple 3.3.11. *Reprenons l'instruction* `goto` *de l'exemple du listing 3.4b (ligne 10). Sachant que cette instruction a comme identificateur “f_10”. L'application de la règle 3.19 donne le résultat suivant :*

$[\![$`goto` ***while_0_break*** `;` $]\!]_{stmt} \underset{R.3.19}{\longrightarrow}$
$stmt_f_10(id, mem) \triangleq$ *(1)*
$[\, mem \text{ EXCEPT } !.register = update_reg(mem.register[id],$ “$[\![$***succ(s)***$]\!]_{sid}$”$)]$

L'instruction `goto` *envoie l'exécution du programme à l'instruction repérée par l'étiquette* `while_0_break`*, qui est déterminée par la fonction* ***succ(s)****. Comme l'identificateur de l'instruction étiquetée vaut “f_13”. Le résultat de la traduction est le suivant :*

$$\textit{(1)} \underset{R.3.15}{\longrightarrow} stmt_f_10(id, mem) \triangleq [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“f_13”})]$$

La traduction d'une instruction étiquetée de la forme **LABEL_ID:**<stmt> est la traduction de l'instruction elle-même. Cette traduction est définie par la règle 3.20.

(c) Bloc d'instructions. Un bloc d'instructions est une suite séquentielle d'instructions éventuellement précédée par une liste de déclarations, sous la forme de <decls> <s_1 :

$$[\![\textbf{LABEL_ID}: <s_1: \text{stmt}>;\,]\!]_{stmt} \rightarrow [\![<s_1: \text{stmt}>]\!]_{stmt}$$

Règle 3.20 – Traduction d'une instruction étiquetée

stmt><s$_2$: stmt>. . .<s$_n$: stmt>. La règle de traduction d'un bloc d'instructions est donnée par la règle 3.21.

$$\begin{array}{l}[\![<\text{decls}>\ <s_1: \text{stmt}>\ <s_2: \text{stmt}>\ldots<s_n: \text{stmt}>]\!]_{block} \rightarrow \\ \quad [\![<s_1: \text{stmt}>]\!]_{stmt} \\ \quad [\![<s_2: \text{stmt}>]\!]_{stmt} \\ \quad \ldots \\ \quad [\![<s_n: \text{stmt}>]\!]_{stmt}\end{array}$$

Règle 3.21 – Traduction d'un bloc d'instructions

La traduction d'un bloc d'instructions revient à traduire chaque instruction le composant.

(d) Instruction itérative. Comme décrit dans la section 3.2.1.4, toutes les structures itératives (`for`, `do while`, et `while`) sont normalisées par CIL en une boucle `while(1)` avec des instructions `goto`. La règle de traduction d'une instruction itérative `while` est donnée par la règle 3.22.

$$\begin{array}{l}[\![\texttt{while(1)}\ \ \{<d: \text{decls}>\ <s_1: \text{stmt}>\ldots<s_n: \text{stmt}>\ \}\ (\text{as } <s: \text{stmt}>)]\!]_{stmt} \rightarrow \\ \quad stmt_[\![<s: \text{stmt}>]\!]_{sid}(id, mem) \triangleq \\ \qquad [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“}[\![<s_1: \text{stmt}>]\!]_{sid}\text{”})] \\ \quad [\![<s_1: \text{stmt}>]\!]_{stmt} \\ \quad \ldots \\ \quad [\![<s_n: \text{stmt}>]\!]_{stmt}\end{array}$$

Règle 3.22 – Traduction d'une instruction itérative

Dans GFC, le successeur de l'instruction `while(1)` est la première instruction contenue dans son bloc. La traduction donnée par la règle 3.22 consiste alors à modifier $mem.registre[id]$ pour pointer vers la première instruction du bloc `while`.

Exemple 3.3.12. *Reprenons l'exemple du listing 3.4b. Sachant que l'identificateur de l'instruction itérative (ligne 1) vaut “f_6”, en appliquant la règle 3.15 et 3.22 sur cette instruction, le résultat est le suivant :*

$$\begin{array}{l}[\![\texttt{while(1)\{ int tmp; \{tmp = i; i++;\} ... \}}]\!]_{stmt} \xrightarrow[R.3.15\&R.3.22]{} \\ \quad stmt_f_6(id, mem) \triangleq [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“}[\![\texttt{tmp = i;}]\!]_{sid}\text{”})] \qquad (1) \\ \quad [\![\texttt{\{ int tmp; \{tmp = i; i++\} ... \}}]\!]_{stmt}\end{array}$$

La traduction illustrée ci-dessus consiste à passer le contrôle d'exécution à la première instruction du bloc `while`*. Dans cet exemple, il s'agit de l'instruction* `tmp = i`*, dont l'identificateur vaut “f_8”. La traduction consiste ensuite à appliquer la règle 3.15 et appliquer la*

règle 3.21 sur le bloc d'instructions de la boucle `while(1)`.

(1) $\underset{R.3.15}{\longrightarrow}$ $stmt_f_6(id, mem) \triangleq [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“f_8”})]$
$[\![$`{ int tmp; {tmp = i; i++} ... }`$]\!]_{block}$

(e) Instruction `assert()`**.** L'instruction `assert(<expr>)` permet de tester une expression booléenne <expr>. Si l'expression est fausse, l'exécution du programme est arrêtée.

$$
\begin{array}{l}
[\![\text{<s:stmt> ::=assert (<expr>)}]\!]_{stmt} \rightarrow \\
\quad stmt_[\![\text{<s:stmt>}]\!]_{sid}(id, mem) \triangleq \\
\qquad \text{LET } expr \triangleq \text{ IF } [\![\text{<expr>}]\!]_{cond} \text{THEN TRUE ELSE FALSE} \\
\qquad \text{IN IF } Assert(expr, \text{“assert error”}) \\
\qquad\quad \text{THEN } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“}[\![\textbf{succ (s)}]\!]_{sid}\text{”})] \\
\qquad\quad \text{ELSE “assert error”}
\end{array}
$$

Règle 3.23 – Traduction d'une instruction `assert()`

La règle 3.23 illustre la traduction de cette instruction. Dans cette règle, nous définissons une variable temporaire $expr$ qui va contenir la valeur retournée par l'expression $[\![\text{<expr>}]\!]_{cond}$. Nous rappelons que $[\![\text{<expr>}]\!]_{cond}$ est la règle de traduction d'une condition (voir règle 3.12). Quand l'évaluation de $Assert(expr, \text{“assert error”})$ retourne vrai, le flot de contrôle est passé à l'instruction suivante. Sinon, TLC arrête l'exploration d'états et reporte le message d'erreur “assert error”.

Exemple 3.3.13. *Soit l'instruction* `assert(i > j )` *avec* `i` *et* `j` *des entiers. La traduction de cette instruction est donnée comme suit :*

$$
\begin{array}{ll}
[\![\texttt{assert(i > j)}]\!]_{stmt} \underset{R.3.23}{\longrightarrow} & \\
\quad stmt_[\![\texttt{assert(i > j)}]\!]_{sid}(id, mem) \triangleq & (1) \\
\qquad \text{LET } expr \triangleq \text{ IF } [\![\texttt{(i > j)}]\!]_{cond} \text{THEN TRUE ELSE FALSE} & \\
\qquad \text{IN IF } Assert(expr, \text{“assert error”}) & \\
\qquad\quad \text{THEN } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“}[\![\textbf{succ (s)}]\!]_{sid}\text{”})] & \\
\qquad\quad \text{ELSE “assert error”} &
\end{array}
$$

Supposons que cette instruction a un identifiant égal à “main_4” et l'instruction suivante a un identificateur égal à “main_5”. L'application de la règle 3.15 donne la traduction suivante :

$$
\begin{array}{ll}
(1) \underset{R.3.15}{\longrightarrow} & \\
\quad stmt_main_4(id, mem) \triangleq & (2) \\
\qquad \text{LET } expr \triangleq \text{ IF } [\![\texttt{i > j}]\!]_{cond} \text{ THEN TRUE ELSE FALSE} & \\
\qquad \text{IN IF } Assert(expr, \text{“assert error”}) & \\
\qquad\quad \text{THEN } [mem \text{ EXCEPT } !.register = update_reg(mem.register[id], \text{“main_5”})] & \\
\qquad\quad \text{ELSE “assert error”} &
\end{array}
$$

Le résultat de la traduction de $[\![\texttt{i > j}]\!]_{cond}$ *est donné dans l'exemple 3.3.10. La traduction complète est comme suit :*

(3) $\underset{R.3.3.10.1}{\longrightarrow}$

$stmt_main_4(id, mem) \triangleq$
$\quad$ LET $expr \triangleq$ IF $(not_zero(gt(load(id, mem, Addr_max_param_j),$
$\quad\quad load(id, mem, Addr_max_param_i))))$ THEN TRUE ELSE FALSE
$\quad$ IN IF $Assert(expr,$ *"assert error"*$)$
$\quad\quad$ THEN $[mem$ EXCEPT $!.register = update_reg(mem.register[id],$ *"main_5"*$)]$
$\quad\quad$ ELSE *"assert error"*

Dans cette traduction, si la valeur de l'entier `i` *donnée par l'expression TLA+* $load(id, mem, Addr_max_param_i)$ *est supérieure à la valeur de* `j` *donnée par l'expression* $load(id, mem, Addr_max_param_j)$*, le flot de contrôle est transmis à l'instruction ayant l'identificateur "main_5". Sinon, l'exécution de TLC s'interrompt avec le message d'erreur "assert error".*

3.3.7 Contrôle de flot inter-procédural

3.3.7.1 Appel de fonction

⟦<s:stmt>::=<lval> = **FUN_ID** (<e1:expr>...<en:expr>) $]\!]_{stmt} \rightarrow$

$stmt_$⟦<s:stmt>⟧$_{sid}$$_pre(id, mem) \triangleq$
$\quad [mem$ EXCEPT
$\quad\quad !.stack = [mem.stack[id] \circ \langle$ ⟦<e1:expr>⟧$_{expr}, \ldots,$ ⟦<en:expr>⟧$_{exp}\rangle],$
$\quad\quad !.register = [mem.register$ EXCEPT $![id] =$
$\quad\quad\quad \langle[pc \mapsto \langle$"⟦**FUN_ID**⟧$_{fid}$⟦ `fst` **(FUN_ID)** ⟧$_{sid}$"$\rangle, fp \mapsto Len(mem.stack[id]) + 1]\rangle$
$\quad\quad\quad \circ \langle[pc \mapsto \langle$" ⟦<s:stmt>⟧$_{sid}$_post "$\rangle, fp \mapsto Head(mem.register[id]).fp]\rangle . \circ Tail(mem.register[id])]]$

$stmt_$⟦<s:stmt>⟧$_{sid}$$_post(id, mem) \triangleq$
$\quad$ LET $nmem \triangleq store(id, mem,$ ⟦<lval>⟧$_{lval}, mem.ret[id])$
$\quad$ IN $[mem$ EXCEPT $!.data = nmem.data, !.stack = nmem.stack,$
$\quad\quad !register = update_reg(mem.register[id],$ "⟦`succ` (s)⟧$_{sid}$"$)]$

Règle 3.24 – Traduction d'un appel de fonction

L'instruction d'appel de fonction <s : stmt> est de la forme <lval> = **FUN_ID** (<e1 : expr>...<en : expr>). La traduction d'un appel de fonction est donnée par la règle 3.24. Cette traduction consiste à générer deux opérateurs :

- $stmt_$⟦<s : stmt>⟧$_{sid}$$_pre()$: cet opérateur permet d'empiler la pile $mem.stack[id]$ avec le cadre de pile de la fonction appelée et de mettre à jour la pile $mem.register[id]$ pour pointer vers la première instruction de la fonction appelée. Ceci, en sauvegardant l'identificateur de l'instruction suivant l'appel de fonction.

- $stmt_$⟦<s:stmt>⟧$_{sid}$$_post()$: cet opérateur permet de stocker dans la lvalue <lval> la valeur retournée par la fonction appelée et de mettre à jour $mem.register[id]$ pour pointer vers l'instruction suivante.

Dans le cas où l'instruction d'appel de fonction <s:stmt> est de la forme **FUN_ID** (<e1: expr>...<en:expr>), la traduction génère seulement l'opérateur $stmt_$⟦<s:stmt>⟧$_{sid}$$_pre()$.

Exemple 3.3.14. *La traduction de l'instruction d'appel à la fonction* `max()` *(ligne 12 de la figure 3.1a), dont l'identificateur vaut "inc_12" se fait comme suit :*

$$
\begin{array}{l}
[\![\texttt{x = max(x,i)}]\!]_{stmt} \underset{R.3.24}{\longrightarrow} \\
\quad stmt_inc_12_pre(id, mem) \triangleq \qquad\qquad (1) \\
\qquad [mem \text{ EXCEPT} \\
\qquad\quad !.stack = [mem.stack \text{ EXCEPT } ![id] = mem.stack[id] \circ \langle\, [\![\texttt{x}]\!]_{expr},\ [\![\texttt{i}]\!]_{exp}\,\rangle], \\
\qquad\quad !.register = [mem.register \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"max_18"} \rangle, fp \mapsto Len(mem.stack[id]) + 1] \rangle \\
\qquad\qquad \circ \langle [pc \mapsto \langle \text{"inc_12_post"} \rangle, fp \mapsto Head(mem.register[id]).fp] \rangle \circ Tail(mem.register[id])]] \\
\\
\quad stmt_inc_12_post(id, mem) \triangleq \\
\qquad \text{LET } nmem \triangleq store(id, mem, [\![\texttt{x}]\!]_{lval}, nmem.ret[id]) \\
\qquad \text{IN } [mem \text{ EXCEPT } !.data = nmem.data, \\
\qquad\quad !.stack = nmem.stack, \\
\qquad\quad !.register = update_reg(mem.register[id], \text{"inc_13"})]
\end{array}
$$

L'appel à la fonction `max()` *se traduit en deux opérateurs :*

- *Le premier opérateur, $stmt_inc_12_pre()$, empile le cadre de pile de la fonction* `max()` *dans la pile $memory.stack[id]$. Puis, un enregistrement contenant l'identificateur de l'instruction suivant l'appel à la fonction* `max()`*, est empilé dans $memory.register[id]$. Enfin, la tête de la pile est mise à jour. Ceci en empilant un enregistrement contenant un champ pc qui pointe vers la première instruction de la fonction $max()$ et un champ fp qui pointe vers l'adresse de base du cadre de pile de la fonction appelée.*

- *Le deuxième opérateur, $stmt_inc_12_post()$, charge la valeur retournée par la fonction* `max()` *dans le registre $ret[id]$ et met ensuite à jour le sommet de la pile afin de pointer vers l'instruction suivante, dont l'identificateur vaut "inc_13".*

Le reste de la traduction consiste à appliquer la règle 3.6 sur les arguments passés en paramètres et la règle 3.10 sur les lvalues `i` *et* `x`*.*

$$
\begin{array}{l}
(1) \underset{R.3.6}{\longrightarrow} \\
\quad stmt_inc_12_pre(id, mem) \triangleq \qquad\qquad (2) \\
\qquad [mem \text{ EXCEPT} \\
\qquad\quad !.stack = [mem.stack \text{ EXCEPT } ![id] = mem.stack[id] \\
\qquad\qquad \circ \langle load(id, mem,\ [\![\texttt{x}]\!]_{lval}), load(id, mem,\ [\![\texttt{i}]\!]_{exp}) \rangle], \\
\qquad\quad !.register = [mem.register \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"max_18"} \rangle, fp \mapsto Len(mem.stack[id]) + 1] \rangle \\
\qquad\qquad \circ \langle [pc \mapsto \langle \text{"inc_12_post"} \rangle \circ Tail(mem.register[id])]] \\
\\
\quad stmt_inc_12_post(id, mem) \triangleq \\
\qquad \text{LET } nmem \triangleq store(id, mem, [\![\texttt{x}]\!]_{lval}, new_mem.ret[id]) \\
\qquad \text{IN } [mem \text{ EXCEPT } !.data = nmem.data, !.stack = nmem.stack, \\
\qquad\qquad !.register = update_reg(mem.register[id], \text{"inc_13"})]
\end{array}
$$

Enfin, en appliquant la règle 3.10, la traduction finale résultante est la suivante :

$$\underset{R.3.10}{\text{(2)}\longrightarrow}$$

$$
\begin{array}{l}
stmt_inc_12_pre(id, mem) \triangleq \\
\quad [mem \text{ EXCEPT } !.stack = [mem.stack \text{ EXCEPT } ![id] = mem.stack[id] \\
\qquad\qquad \circ \langle load(id, mem, Addr_x), load(id, mem, Addr_param_inc_i)\rangle], \\
\qquad !.register = [mem.register \text{ EXCEPT } ![id] = \\
\qquad\qquad \langle [pc \mapsto \langle \text{“max_18”}\rangle, fp \mapsto Len(mem.stack[id]) + 1]\rangle \\
\qquad\qquad \circ \langle [pc \mapsto \langle \text{“inc_12_post”}, fp \mapsto Head(mem.register[id]).fp]\rangle \circ Tail(mem.register[id])]]
\end{array}
$$

$$
\begin{array}{l}
stmt_inc_12_post(id, mem) \triangleq \\
\quad \text{LET } nmem \triangleq store(id, mem, Addr_c, nmem.ret[id]) \\
\quad \text{IN } [mem \text{ EXCEPT } !.data = nmem.data, !.stack = nmem.stack, \\
\qquad\qquad !.register = update_reg(mem.register[id], \text{“inc_13”})]
\end{array}
$$

3.3.7.2 Retour de fonction

L’instruction `return` met fin à l’exécution de la fonction. Cette instruction ne dispose pas de successeur dans le GFC. Syntaxiquement, elle peut être suivie d’une expression <expr> qui sera la valeur de retour de la fonction. La traduction d’un retour de fonction est donnée par la règle 3.25.

.1. $[\![\text{<s:stmt>::=return;}]\!]_{stmt} \rightarrow$

$$
\begin{array}{l}
stmt_[\![\text{<s:stmt>}]\!]_{sid}(id, mem) \triangleq \\
\quad [mem \text{ EXCEPT} \\
\qquad !.stack = [mem.stack \text{ EXCEPT } ![id] = SubSeq(mem.stack[id], 1, Head(mem.register[id]).fp - 1)], \\
\qquad !.register = [register \text{ EXCEPT } ![id] = Tail(mem.register[id])]
\end{array}
$$

.2. $[\![\text{<s:stmt>::=return <e:expr>;}]\!]_{stmt} \rightarrow$

$$
\begin{array}{l}
stmt_[\![\text{<s:stmt>}]\!]_{sid}(id, mem) \triangleq \\
\quad [mem \text{ EXCEPT} \\
\qquad !.stack = [mem.stack \text{ EXCEPT } ![id] = SubSeq(mem.stack[id], 1, Head(mem.register[id]).fp - 1)], \\
\qquad !.register = [mem.register \text{ EXCEPT } ![id] = Tail(mem.register[id])], \\
\qquad !.ret = [mem.ret \text{ EXCEPT } ![id] = [\![\text{<e:expr>}]\!]_{exp}]]
\end{array}
$$

Règle 3.25 – Traduction d’un retour de fonction

Quand la fonction se termine son cadre de pile est dépilé de la pile $mem.stack[id]$. Cette opération est réalisée en utilisant l’opérateur $SubSeq()$ (dont la sémantique est définie dans la section 2.2.4). Le sommet de la pile d’exécution du processus $register[id]$ est ensuite dépilé en utilisant l’opérateur $Tail()$. Quand la fonction ne renvoie rien, le registre ret reste inchangé. Sinon, la valeur retournée par la fonction est stockée dans le registre $ret[id]$.

3.3.8 Définition de fonction et génération de la spécification

3.3.8.1 Définition de fonction

La règle de traduction d’une définition de fonction est illustrée par la règle 3.26. Cette traduction revient à traduire le bloc d’instructions contenue dans celle-ci.

$$[\![\text{<type>}\textbf{FUN_ID}\,(\text{<params>})\ \{\text{<decls><s}_1\text{:stmt>}\ldots\text{<s}_n\text{:stmt>}\}]\!]_{fun_def} \rightarrow [\![\text{<s}_1\text{:stmt>}\ldots\text{<s}_n\text{:stmt>}]\!]_{bloc}$$

Règle 3.26 – Traduction d'une définition de fonction

3.3.8.2 Génération de la spécification

Une fois toutes les définitions de fonctions traduites, C2TLA+ génère la spécification qui modélise le comportement du programme C. Comme expliqué dans la section 2.3, pour qu'une spécification TLA+ soit vérifiable par TLC, elle doit s'écrire de la forme :

$$Spec \triangleq Init \land \Box[Next]_{memory} \land Fairness \tag{3.1}$$

La formule $Spec$ contient un prédicat initial $Init$ qui définit l'état initial et une action $Next$ qui définit toutes les transitions possibles du système. La formule $Fairness$ spécifie des conditions d'équité sur l'exécution de l'action $Next$.

(a) Prédicat $Init$. La génération du prédicat $Init$ suit la règle 3.27. Nous expliquons cette règle à travers l'exemple qui suit.

$$\begin{aligned} Init \triangleq memory = [&data \mapsto [\![\text{<d:decls>}]\!]_{init},\\ &stack \mapsto (1 :> \ \texttt{init_stack()}),\\ ®ister \mapsto (1 :> \langle[pc \mapsto \texttt{init_pc()}, fp \mapsto 1]\rangle),\\ &ret \mapsto (1 :> [val \mapsto Undef])] \end{aligned}$$

Règle 3.27 – Génération du prédicat $Init$

Exemple 3.3.15. *Pour l'exemple de la figure 3.1a, la traduction de l'initialisation est donnée comme suit :*

$$\begin{aligned} Init \triangleq memory = [&mem \mapsto (0 :> [val \mapsto 5] \mathbin{@@} 1 :> [val \mapsto 1] \mathbin{@@} 2 :> [val \mapsto 1]\\ &\quad \mathbin{@@} 3 :> [val \mapsto 0] \mathbin{@@} 4 :> [val \mapsto 0] \mathbin{@@} 5 :> [val \mapsto 0]\\ &\quad \mathbin{@@} 6 :> [val \mapsto 0] \mathbin{@@} 7 :> [loc \mapsto Null, offs \mapsto Null])\\ &stack \mapsto (1 :> \langle\rangle),\\ ®ister \mapsto (1 :> \langle[pc \mapsto \text{“main_28”}, fp \mapsto 1\rangle),\\ &ret \mapsto (1 :> [val \mapsto Undef])] \end{aligned}$$

La règle 3.27 consiste à initialiser la variable $memory$, à savoir les champs $data$, $stack$, $register$ et ret, dont le principe est le suivant :

- *La règle $[\![\text{<d : decls>}]\!]_{init}$ parcourt toutes les déclarations globales C et traduit chaque déclaration initialisée. Pour les déclarations non initialisées, C2LTA+ affecte à un entier non initialisé l'enregistrement $[val \mapsto 0]$ et à un type pointeur l'enregistrement $[loc \mapsto Null, offs \mapsto Null]$ qui modélise le pointeur nul en C. L'index du tableau $data$ commence par 0 et s'incrémente en fonction de la taille de la variable à stocker.*

- *La pile $register$ est un tableau à un élément. L'élément $memory.register[1]$ contient une séquence composée d'un enregistrement dont les champs pc et fp sont initialement définis comme suit :*

– *Le champ* pc *contient l'identificateur de l'instruction présentant le point d'entrée du programme. Cette valeur est générée par la fonction OCaml* **`init_pc()`**. *Ainsi, dans cet exemple,* **`init_pc()`** *génère l'identificateur "main_28".*

– *Le champ* fp *est égal à l'adresse de base du cadre de pile de la fonction* `main()`. *Comme les index des séquences commencent à* 1*, la valeur initiale de* fp *vaut* 1.

- *La mémoire* $stack$ *est un tableau à un élément. Dans cet exemple, l'élément* $memory.stack[1]$ *contient initialement une séquence vide, retournée par la fonction OCaml* **`init_stack()`**. *La fonction* **`init_stack()`** *retourne une séquence qui correspond au cadre de pile de la fonction C appelée par le programme C, à savoir la fonction* `main()`. *Si la fonction* `main()` *n'utilise pas des variables locales, la fonction* **`init_stack()`** *retourne la séquence vide.*

- *Le champ* ret *est un tableau à un élément. L'élément* $memory.ret[1]$ *est une séquence contenant l'enregistrement* $[val \mapsto Undef]$. *Ce dernier désigne une valeur indéfinie.*

(b) Action $Next$. La génération de l'action $Next$ est définie par la règle 3.28. Sa définition consiste en la disjonction de deux actions :

- Une action qui exprime que le programme peut mettre à jour la valeur de la variable $memory$ tant que sa pile d'exécution $memory.register[1]$ n'est pas vide. Cette mise à jour est assurée par l'appel à l'opérateur $dispatch()$ décrit dans le paragraphe suivant.

- Une action qui modélise la fin du programme C, c'est-à-dire que la pile $register[1]$ est vide et la variable $memory$ reste inchangée. Cette formule permet à TLC de ne pas reporter une erreur d'interblocage due à l'absence d'un état successeur (voir section 2.3.4).

$$\begin{array}{lll} Next \triangleq & \lor & \\ & & \land\ memory.register[1] \neq \langle\rangle \\ & & \land\ memory' = dispatch(1, memory) \\ & \lor & \\ & & \land\ memory.register[1] = \langle\rangle \\ & & \land\ \text{UNCHANGED}\ memory \end{array}$$

Règle 3.28 – Définition de l'action $Next$

(c) L'opérateur $dispatch()$. L'opérateur $dispatch()$ est généré selon la règle 3.29.

L'opérateur $dispatch()$ généré par la règle 3.29 est défini à travers le constructeur CASE (dont la sémantique est donnée dans la section 2.2.6). Il teste les valeurs possibles que peut prendre le champ pc contenu dans le sommet de la pile du programme. En fonction de cette valeur, il appelle l'opérateur de l'instruction C correspondante. Cet opérateur permet de gérer le flot de contrôle du programme C. Ainsi, quand la pile $memory.register[1]$ est non vide, chaque exécution de l'action $Next$ fait appel à l'opérateur $dispatch()$, qui correspond à l'exécution d'une instruction dans le programme C.

$$
\begin{array}{l}
dispatch(id, mem) \triangleq \\
\quad \text{CASE } mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“}[\![\text{<f}_1\text{: fun_def>}]\!]_{fid}\, \mathtt{lbl\ (}\text{<s}_1\text{: stmt>}\mathtt{)}\text{”} \\
\qquad\qquad \rightarrow stmt_[\![\text{<s}_1\text{: stmt>}]\!]_{sid}(id, mem) \\
\qquad \ldots \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“}[\![\text{<f}_1\text{: fun_def>}]\!]_{fid}\, \mathtt{lbl\ (}\text{<s}_n\text{: stmt>}\mathtt{)}\text{”} \\
\qquad\qquad \rightarrow stmt_[\![\text{<s}_n\text{: stmt>}]\!]_{sid}(id, mem) \\
\qquad \ldots \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“}[\![\text{<f}_k\text{: fun_def>}]\!]_{fid}\, \mathtt{lbl\ (}\text{<s}_h\text{: stmt>}\mathtt{)}\text{”} \\
\qquad\qquad \rightarrow stmt_[\![\text{<s}_h\text{: stmt>}]\!]_{sid}(id, mem) \\
\qquad \ldots \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“}[\![\text{<f}_k\text{: fun_def>}]\!]_{fid}\, \mathtt{lbl\ (}\text{<s}_m\text{: stmt>}\mathtt{)}\text{”} \\
\qquad\qquad \rightarrow stmt_[\![\text{<s}_m\text{: stmt>}]\!]_{sid}(id, mem) \\
\qquad \Box\ mem.register[id] = \langle\rangle \rightarrow mem \\
\qquad \Box\ \text{OTHER} \rightarrow \text{“error”}
\end{array}
$$

Règle 3.29 – Génération de l'opérateur $dispatch()$

Exemple 3.3.16. *L'opérateur $dispatch()$ associé au programme C, donné par la figure 3.1a, est généré comme suit :*

$$
\begin{array}{l}
dispatch(id, mem) \triangleq \\
\quad \text{CASE } mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“inc_11”} \rightarrow stmt_inc_11(id, mem) \\
\quad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“inc_12”} \rightarrow stmt_inc_12(id, mem) \\
\quad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“inc_13”} \rightarrow stmt_inc_13(id, mem) \\
\quad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“max_19”} \rightarrow stmt_max_19(id, mem) \\
\quad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“max_20”} \rightarrow stmt_max_20(id, mem) \\
\quad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“max_22”} \rightarrow stmt_max_22(id, mem) \\
\quad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“max_23”} \rightarrow stmt_max_23(id, mem) \\
\quad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“main_28”} \rightarrow stmt_main_28(id, mem) \\
\quad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“main_29”} \rightarrow stmt_main_29(id, mem) \\
\quad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“main_30”} \rightarrow stmt_main_30(id, mem) \\
\quad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“main_31”} \rightarrow stmt_main_31(id, mem) \\
\quad \Box\ mem.register[id] \neq \langle\rangle \wedge Head(mem.register[id]).pc = \text{“main_32”} \rightarrow stmt_main_32(id, mem) \\
\quad \Box\ mem.register[id] = \langle\rangle \rightarrow mem \\
\quad \Box\ \text{OTHER} \rightarrow mem
\end{array}
$$

3.4 Conclusion

Dans ce chapitre, nous avons présenté le principe de traduction de C vers TLA+. Pour cela, nous avons illustré l'étape de transformation utilisant la bibliothèque CIL. Cette étape permet de rendre le code C plus simple à analyser syntaxiquement. En se basant sur les transformations CIL, nous avons présenté le sous-ensemble CIL de C que nous considérons dans ce présent travail. Nous avons ensuite décrit un modèle mémoire pour un programme C séquentiel. En se basant sur ce modèle, nous avons expliqué les règles de traduction de la sémantique de C vers TLA+ que nous avons implémenté dans le traducteur automatique C2TLA+.

Le modèle mémoire et les règles de traduction définies dans ce chapitre sont définies dans l'objectif de permettre de gérer les programmes C concurrents. Cette partie est le sujet du chapitre suivant.

4

Traduction des programmes C concurrents vers TLA+

Avec le développement des architectures multiprocesseurs et multi-cœurs, des architectures parallèles, des technologies web et des langages de programmation, la programmation concurrente joue un rôle important pour répondre aux besoins croissants de performance. Elle permet de réaliser des applications ou des tâches complexes en divisant le travail en simples tâches. Parmi ces applications, on peut citer les systèmes pair-à-pair (« *peer-to-peer* » en anglais), les systèmes d'exploitation, et les systèmes clients/serveurs.

Toutefois, si la programmation concurrente forme un outil puissant pour augmenter les performances des systèmes, elle fait néanmoins face à plusieurs difficultés. En effet, dans un système séquentiel, l'ordre d'exécution est complètement déterminé. On parle alors d'un ordre total. Pour un système concurrent, l'exécution forme un ordre partiel, c'est-à-dire un ordre d'exécution inconnu. C'est ce qu'on appelle l'*indéterminisme* d'exécution. Ce dernier impose alors de mettre en place des mécanismes de synchronisation adéquats entre les différentes entités du système. Et pour s'assurer du bon fonctionnement du système ainsi que de la bonne gestion des primitives de synchronisation, une phase de vérification est essentielle. Cette étape requiert un modèle du système qui exprime son comportement dans une représentation non-ambiguë et précise.

L'objectif de ce chapitre est de modéliser des systèmes concurrents implémentés en C avec TLA+ en vue de les vérifier formellement. Pour cela, nous commençons dans la section 4.1 par introduire un modèle d'exécution de base, à savoir le modèle mémoire et le modèle de concurrence que nous considérons. À partir de ce modèle, nous présentons comment ce dernier peut s'étendre afin de gérer la synchronisation entre processus et leur ordonnancement. La section 4.2 est consacrée à la définition TLA+ d'exemples de primitives de synchronisation. Dans la section 4.3, nous étendons notre modèle d'exécution pour spécifier des programmes temps-réel afin de les vérifier par la suite avec TLC. Pour cela, nous présentons dans un premier temps une modélisation des contraintes temporelles. Puis dans un deuxième temps, nous donnons des exemples d'ordonnancement temps-réel en TLA+. Enfin, pour justifier le choix du modèle mémoire proposé ainsi que la méthode de traduction des programmes C adoptée dans cette étude, nous décrivons dans la section 4.4 les premières propositions que nous avons étudiées au début de ce travail ainsi que leurs limitations.

4.1 MODÈLE D'EXÉCUTION

Dans le cadre de la programmation concurrente, une application ou un programme concurrent est composé de plusieurs flots d'exécution. En C, on parle de la programmation *multi-threads*, qui consiste à exécuter une application en faisant appel à plusieurs *threads* ou *processus légers* au sein d'un même processus.

Nous nous intéressons dans cette section à la définition du modèle d'exécution de base d'un programme concurrent, à savoir le modèle mémoire et de concurrence. Nous décrivons ensuite un modèle étendu dans lequel on peut spécifier plusieurs aspects liés à la synchronisation et la gestion des processus.

4.1.1 Modèle d'exécution de base

Dans cette section, nous introduisons le modèle mémoire d'un programme concurrent ainsi que le modèle de concurrence. Nous décrivons ensuite, comment ces deux modèles sont implémentés dans le traducteur C2TLA+.

4.1.1.1 Modèle mémoire

La figure 4.1 présente le modèle mémoire d'un programme concurrent tel que considéré dans notre étude. Chaque processus P_i exécute une suite d'instructions séquentiellement et dispose d'un espace mémoire local M_i. La communication entre les processus se fait à travers une mémoire partagée, qui est unique et accessible dans sa totalité à tous les processus. Si un processus P_i écrit dans la mémoire partagée, la modification est immédiatement perçue par les autres processus.

Ce modèle correspond à un modèle mémoire d'un programme C multithreadé. La mémoire globale est une mémoire partagée par tous les threads et chaque thread dispose

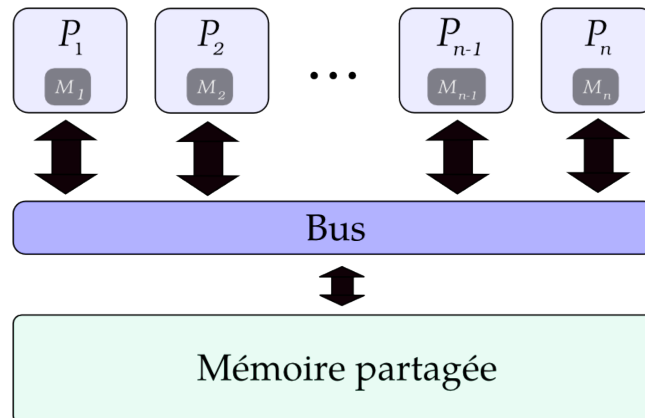


FIGURE 4.1 – Architecture simplifiée du modèle mémoire considéré

de sa propre pile contenant les paramètres passés aux fonctions appelées, les valeurs retournées par ces fonctions et les variables locales. Ce modèle de base ne dispose pas d'un tas (*heap* en anglais) dédié à l'allocation dynamique de blocs de mémoire. Cet aspect constitue une des perspectives de la thèse.

Dans la suite de ce manuscrit, nous employons le mot *processus* pour désigner un thread.

4.1.1.2 Modèle de concurrence

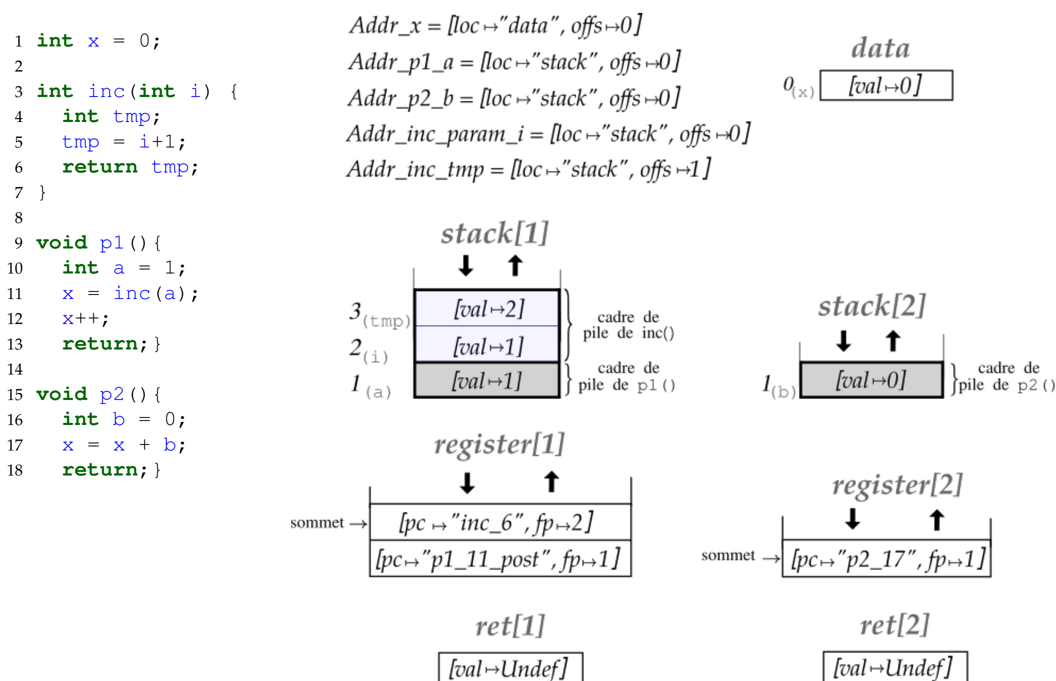
Le modèle de concurrence que nous considérons dans cette étude est celui de l'entrelacement dans lequel l'exécution de plusieurs processus se fait en choisissant d'une manière indéterministe une instruction de l'un des processus, puis d'un autre processus ou éventuellement du même processus. Une autre sémantique consisterait à définir la concurrence d'une manière explicite en spécifiant un ordre partiel des actions. Une telle sémantique est dite *vraie concurrence* (« *true concurrency* » en anglais) [CDC93] (voir section 1.1.1). Le choix s'est porté sur une sémantique par entrelacement en raison de la difficulté à représenter un ordre partiel en TLA+.

4.1.1.3 Traduction en TLA+ d'un modèle d'exécution de base

(a) Modèle mémoire. La modélisation TLA+ de la mémoire pour un programme concurrent se base sur le modèle mémoire d'un programme séquentiel, décrit dans le chapitre 3. Nous rappelons que la modélisation TLA+ de la mémoire d'un programme séquentiel consiste en une mémoire globale appelée $data$ (modélisée avec un tableau) ainsi que de plusieurs mémoires locales : $register$, $stack$ et ret. Chacune de ces mémoires locales est modélisée avec un tableau avec un unique élément.

Pour un programme concurrent à n processus, chaque mémoire locale devient un tableau à n éléments. Par conséquent, chaque processus :

- dispose d'un identifiant id unique ;
- possède son propre espace mémoire, à savoir une pile d'exécution $stack[id]$, une pile de registres $register[id]$ et un registre $ret[id]$;
- interagit avec l'ensemble des processus par le biais de la mémoire statique partagée, à savoir la mémoire $data$.



(a) Code C source (b) Modèle mémoire

FIGURE 4.2 – Organisation de la mémoire d'un exemple de code C concurrent

Exemple 4.1.1. *La figure 4.2 illustre un exemple de code C concurrent et la représentation mémoire telle que définie par C2TLA+. L'exemple du code C (donné par la figure 4.2a) contient une variable globale* `x` *partagée entre deux processus ayant respectivement les identifiants* 1 *et* 2, *et trois fonctions dont* `p1()` *et* `p2()` *sont exécutées chacune par un processus.*

La figure 4.2b présente l'état de la mémoire en considérant le scénario suivant : le processus 1 exécute l'appel à la fonction `inc()`, *à savoir l'instruction de la ligne 6 et le processus 2 est en cours d'exécution de l'instruction de la ligne 17. La variable* `x` *étant partagée entre les deux processus, elle est alors stockée dans la mémoire* $data$. *Les piles de registres des processus 1 et 2 sont représentées respectivement par* $register[1]$ *et* $register[2]$. *Le sommet de chacune de ces piles indique l'instruction en cours d'exécution par le processus (contenue dans le champ* pc*) et l'adresse de base du cadre de pile de la fonction en exécution (contenue dans le champ* fp*). Les piles d'exécution des processus sont représentées respectivement par* $stack[1]$ *et* $stack[2]$. $stack[1]$ *contient le cadre de pile des fonctions* `p1()` *et* `inc()`, *quant à* $stack[2]$ *contient le cadre de pile de* `p2()`. *Étant donné que les fonctions* `inc()` *et* `p2()` *n'ont pas encore terminé leur exécution, les registres* $ret[1]$ *et* $ret[2]$ *contiennent la valeur* $[val \mapsto Undef]$.

(b) Génération de la spécification TLA+ d'un programme concurrent. En se basant sur le modèle mémoire et de concurrence décrits ci-dessus, la traduction d'un programme concurrent ne présente pas en soi de difficulté particulière puisqu'elle repose sur le même principe que celui d'un programme séquentiel. Par conséquent, nous réutilisons les règles de traduction décrites dans la section 3.3. Seule la règle 3.2 présente quelques ajouts et modifications que nous décrivons dans la suite de ce paragraphe.

Dans C2TLA+, l'utilisateur fournit le programme C à traduire en spécifiant le nombre de processus ainsi que la fonction C exécutée par chacun des processus. La traduction d'un programme C concurrent est donnée par la règle 4.1. Comme pour la traduction d'un programme séquentiel, C2LTA+ génère en sortie deux modules : $Parameters.tla$ et $Program.tla$. Nous rappelons que le module $Parameters.tla$ contient la déclaration des variables et la définition des constantes (les adresses des variables C). Le module $Program.tla$ contient la traduction des définitions de fonctions C, la définition du prédicat $Init$, de l'action $Next$ et de la formule $Spec$ qui modélise le comportement du programme C.

Dans la règle 3.2, une constante appelée $ProcSet$, est déclarée dans le module $Runtime$. Elle définit l'ensemble des identifiants de processus. Par exemple, pour deux processus, $ProcSet$ peut-être définie comme $ProcSet \triangleq \{1, 2\}$ ou $ProcSet \triangleq \{\text{“p1”}, \text{“p2”}\}$. La fonction $procsExec()$ définit l'ensemble des processus activables. Par défaut, tous les processus dans $ProcSet$ sont prêts à s'exécuter dès l'état initial et à chaque état un processus est choisi de manière indéterministe pour exécuter une action.

Comme TLA+ est basée sur la notion d'actions, le comportement d'un programme concurrent se ramène alors à une exécution séquentielle arbitraire d'actions. La spécification TLA+ d'un programme C concurrent est donnée par la formule $Spec \triangleq Init \land \Box[Next]_{memory} \land WF_{memory}(Next)$. L'espace d'états généré par la formule $Spec$ correspond à tous les entrelacements possibles d'actions entre les processus.

(c) Définition du prédicat $Init$. Le prédicat $Init$ initialise les champs $data$, $stack$, $register$ et ret de la variable $memory$.

- L'initialisation du champ $data$ revient à traduire les initialisations des déclarations globales C. Elle repose sur le même principe que celui décrit dans la traduction séquentielle (voir section 3.3.8.2).

- La fonction OCaml **`init_pc_procs()`** permet d'initialiser la pile $register$ de tous les processus. Elle génère un tableau TLA+ qui associe à chaque identifiant de processus un enregistrement composé d'un champ pc contenant l'identifiant de la première instruction C de la fonction appelée par le processus, et un champ fp contenant l'adresse de base de la pile $stack[id]$, qui vaut 1 par défaut.

- La fonction OCaml **`init_stack_procs()`** permet d'initialiser le champ $stack$. Cette fonction génère un tableau qui pour chaque processus associe une séquence contenant la cadre de pile de la fonction appelée.

```
⟦<d : decls><f1: fun_def>. . .<fk: fun_def>⟧prg →

┌──────────────────── MODULE Parameters ────────────────────┐
EXTENDS Runtime
⟦<d : decls>⟧decls
⟦<f1: fun_def>⟧decls . . . ⟦<fk: fun_def>⟧decls
size_of_int ≜ 1
⟦<d : decls>⟧field_offs
└───────────────────────────────────────────────────────────┘
┌──────────────────── MODULE Program ───────────────────────┐
EXTENDS Parameters
⟦ <f1: fun_def> : := <type> FUN_ID (<params>) { <decls><s1: stmt>. . .<sn: stmt> } ⟧fun_def
...
⟦ <fk: fun_def> : :=<type> FUN_ID (<params>) { <decls><sh: stmt>. . .<sm: stmt> } ⟧fun_def

Init ≜ memory = [data ↦ ⟦<decls>⟧init,
                 register ↦ init_pc_procs(),
                 stack ↦ init_stack_procs(),
                 ret ↦ [i ∈ ProcSet ↦ [val ↦ Undef]]

dispatch(id, mem) ≜
  CASE mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<f1: fun_def>⟧fid_lbl(<s1: stmt>)"
          → stmt_⟦<s1: stmt>⟧sid(id, mem)
    . . .
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<f1: fun_def>⟧fid_lbl(<sn: stmt>)"
          → stmt_⟦<sn: stmt>⟧sid(id, mem)
    . . .
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<fk: fun_def>⟧fid_lbl(<sh: stmt>)"
          → stmt_⟦<sh: stmt>⟧sid(id, mem)
    . . .
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<fk: fun_def>⟧fid_lbl(<sm: stmt>)"
          → stmt_⟦<sm: stmt>⟧sid(id, mem)
    □ mem.register[id] = ⟨⟩ → mem
    □ OTHER → "error"

procsExec(mem) ≜ ProcSet

Next ≜ ∨ ∃ id ∈ procsExec(memory) :
           ∧ memory.register[id] ≠ ⟨⟩
           ∧ memory′ = dispatch(id, memory)
       ∨ ∀ id ∈ procsExec(memory) :
           ∧ memory.register[id] = ⟨⟩
           ∧ UNCHANGED ⟨memory⟩

Spec ≜ Init ∧ □[Next]⟨memory⟩ ∧ WF⟨memory⟩(Next)
└───────────────────────────────────────────────────────────┘
```

Règle 4.1 – Traduction d’un programme C concurrent

- Le champ ret est aussi un tableau dont chaque élément contient la valeur $[val \mapsto Undef]$.

(d) Définition de l'action $Next$. La définition de l'action $Next$ consiste en la disjonction de deux actions :

- une action qui exprime qu'un processus id est choisi de manière indéterministe pour exécuter une action tant que sa pile $register[id]$ n'est pas vide. Cette action met à jour la variable $memory$, en appelant l'opérateur $dispatch()$ que nous avons défini dans le chapitre 3 (voir section 3.3.8.2) ;

- un prédicat qui exprime la fin d'exécution de tous les processus. Ce prédicat correspond à l'état où la pile $register[id]$ de chaque processus est vide et la variable $memory$ TLA+ reste inchangée.

(e) Les conditions d'équité. Par défaut, la spécification $Spec$ générée par C2TLA+ considère les conditions d'équité faible sur l'exécution de l'action $Next$. Cette condition est exprimée par la formule : $WF_{\langle memory \rangle} Next$.

Dans certains cas, cette condition est insuffisante pour vérifier des propriétés de vivacité. L'utilisateur doit alors compléter la spécification avec les bonnes conditions d'équité pour vérifier les propriétés voulues.

La traduction TLA+ complète de l'exemple de code C de la figure 4.2a est donnée dans l'annexe A.

4.1.2 Modèle d'exécution étendu

Le modèle d'exécution étendu enrichi le modèle d'exécution de base en ajoutant des champs complémentaires dans la mémoire $memory$. Dans cette section, nous décrivons alors comment nous étendons le modèle de base afin de définir un choix sur les processus à exécuter et aussi pour gérer des primitives de synchronisation entre les processus.

4.1.2.1 Choix du processus à exécuter

Le modèle d'exécution de base considère que tous les processus sont toujours prêts à s'exécuter. La fonction $procsExec()$ retourne dans ce cas l'ensemble des processus $ProcSet$ et l'action $Next$ choisit de manière indéterministe un processus à exécuter parmi cet ensemble.

Dans un modèle d'exécution étendu, la définition de $procsExec()$ dépend de la façon dont l'utilisateur veut arbitrer l'exécution des processus, c'est-à-dire les ordonnancer. Il est possible par exemple de définir $procsExec()$ afin de n'exécuter que le processus ayant la plus haute priorité ou celui qui n'est pas bloqué sur une ressource. Dans ce cas, le modèle d'exécution de base est alors extensible pour intégrer ces nouveaux aspects. Ceci se fait en rajoutant des nouveaux champs à la variable $memory$.

4.1.2.2 Gestion des primitives de synchronisation

Le modèle d'exécution de base peut aussi s'étendre pour gérer des primitives de synchronisation. Dans cette section, nous expliquons comment ces primitives sont définies dans le code C à traduire. Nous expliquons ensuite comment nous traduisons l'appel à ces primitives dans TLA+. C'est dans la section 4.2 que nous décrivons comment ces primitives sont spécifiées formellement en TLA+ et comment elles peuvent changer l'état de la mémoire.

(a) Déclaration C des primitives de synchronisation. L'utilisateur déclare ces primitives dans le code C via le mécanisme des attributs (`__attribute__`). Ce mécanisme permet d'associer des informations au programme C à traduire, qui sont ensuite traitées par le traducteur C2TLA+. La syntaxe d'un attribut tel qu'utilisé dans C2TLA+ est donnée comme suit :

<a : attr> ::=<type> **FUN_ID** (<params>) **__attribute__ ((MODULE_ID,alias("FUN_ID")))**

avec **MODULE_ID** est le nom du module TLA+ dans lequel la primitive de synchronisation désignée par **FUN_ID** est spécifiée.

Exemple 4.1.2. *Reprenons l'exemple de code C de la figure 4.2a dans lequel nous avons ajouté la déclaration et des appels à des primitives de synchronisation.*

```
1  int lock = 1;
2  int x = 0;
3  void spinlock_acquire(int *l) __attribute__((Sync_primitives, alias("spinlock_acquire")));
4  void spinlock_release(int *l) __attribute__((Sync_primitives, alias("spinlock_release")));
5
6  int inc(int i) {
7    int tmp;
8    tmp = i+1;
9    return tmp;
10 }
11 void p1() {
12   int a = 1;
13   spinlock_acquire(&lock);
14   x = inc(a);
15   x++;
16   spinlock_release(&lock);
17   return; }
18 void p2(){
19   int b = 0;
20   spinlock_acquire(&lock);
21   x = x + b;
22   spinlock_release(&lock);
23   return;
24 }
```

Listing 4.1 – Exemple d'un code C utilisant des primitives de synchronisation

La synchronisation entre les deux processus est assurée via l'appel aux fonctions `spinlock_acquire()` *et* `spinlock_release()`. *Vu que ces deux fonctions ne sont pas implémentées directement en C, nous déclarons ces primitives via le mécanisme d'attribut (lignes 3 et 4 du*

listing 4.1). Ce mécanisme permet d'associer des informations au code C. Dans notre cas, nous indiquons dans chaque attribut le module TLA+ où chaque primitive est spécifiée en TLA+. Dans cet exemple, ces primitives sont spécifiées dans un module appelé $Sync_primitives$.

(b) Appels aux primitives de synchronisation. Nous rappelons que le flot de contrôle d'un programme C est modélisé et géré par l'opérateur $dispatch()$. Par rapport à la définition de l'opérateur $dispatch()$ donnée dans le chapitre 3 (voir règle 3.29) la nouvelle définition de $dispatch()$ pour un programme concurrent avec des primitives de synchronisation est générée selon la règle 4.2.

$$
\begin{array}{l}
dispatch(id, mem) \triangleq \\
\quad \text{CASE } mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“}[\![\texttt{<f}_1\texttt{: fun_def>}]\!]_{fid}_\mathtt{lbl(<s_1: stmt>)}\text{”} \\
\qquad\qquad \rightarrow stmt_[\![\texttt{<s}_1\texttt{: stmt>}]\!]_{sid}(id, mem) \\
\qquad \ldots \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“}[\![\texttt{<f}_1\texttt{: fun_def>}]\!]_{fid}_\mathtt{lbl(<s_n: stmt>)}\text{”} \\
\qquad\qquad \rightarrow stmt_[\![\texttt{<s}_n\texttt{: stmt>}]\!]_{sid}(id, mem) \\
\qquad \ldots \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“}[\![\texttt{<f}_k\texttt{: fun_def>}]\!]_{fid}_\mathtt{lbl(<s_h: stmt>)}\text{”} \\
\qquad\qquad \rightarrow stmt_[\![\texttt{<s}_h\texttt{: stmt>}]\!]_{sid}(id, mem) \\
\qquad \ldots \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“}[\![\texttt{<f}_k\texttt{: fun_def>}]\!]_{fid}_\mathtt{lbl(<s_m: stmt>)}\text{”} \\
\qquad\qquad \rightarrow stmt_[\![\texttt{<s}_m\texttt{: stmt>}]\!]_{sid}(id, mem) \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“}[\![\texttt{<a}_1\texttt{: attr>}]\!]_{fid}_init\text{”} \\
\qquad\qquad \rightarrow [\![\texttt{<a}_1\texttt{: attr>}]\!]_{fid}(id, mem) \\
\qquad \ldots \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“}[\![\texttt{<a}_p\texttt{: attr>}]\!]_{fid}_init\text{”} \\
\qquad\qquad \rightarrow [\![\texttt{<a}_p\texttt{: attr>}]\!]_{fid}(id, mem) \\
\qquad \Box\ mem.register[id] = \langle\rangle \rightarrow mem \\
\qquad \Box\ \text{OTHER} \rightarrow \text{“error”}
\end{array}
$$

Règle 4.2 – Opérateur $dispatch()$ pour un programme concurrent avec des primitives de synchronisation

L'opérateur $dispatch()$ teste les valeurs possibles que peut prendre le champ pc contenu dans le sommet de la pile du processus id. Quand celui-ci contient l'identificateur d'une primitive de synchronisation, l'opérateur $dispatch()$ appelle l'opérateur TLA+ $[\![\texttt{<a}_i\texttt{: attr>}]\!]_{fid}(id, mem)$ qui modélise la primitive de synchronisation. Il est à noter que la règle $[\![\texttt{<a}_i\texttt{: attr>}]\!]_{fid}$ renvoie l'identificateur de fonction **FUN_ID** contenu dans l'attribut $\texttt{<a}_i\texttt{: attr>}$ (voir la syntaxe d'un attribut dans le paragraphe précédent).

Par défaut, l'identifiant de la première instruction de toute primitive de synchronisation déclarée dans un attribut $\texttt{<a}_i\texttt{: attr>}$ s'écrit sous la forme suivante :

$$\text{“}[\![\mathbf{FUN_ID}]\!]_{fid}_init\text{”}$$

Exemple 4.1.3. *Reprenons l'exemple donné par le listing 4.1. En appliquant la règle 4.2, l'opérateur $dispatch()$ est défini comme suit :*

$dispatch(id, mem) \triangleq$
$\text{CASE } mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"inc_7"} \rightarrow inc_7(id, mem)$
$\ldots$
$\Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"inc_9"} \rightarrow inc_9(id, mem)$
$\Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p1_12"} \rightarrow p1_12(id, mem)$
$\ldots$
$\Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p1_17"} \rightarrow p1_17(id, mem)$
$\Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p2_19"} \rightarrow p2_19(id, mem)$
$\ldots$
$\Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p2_23"} \rightarrow p2_23(id, mem)$
$\Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"spinlock_acquire_init"}$
$\quad \rightarrow spinlock_acquire(id, mem)$
$\Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"spinlock_release_init"}$
$\quad \rightarrow spinlock_release(id, mem)$
$\Box\ mem.register[id] = \langle\rangle \rightarrow mem$
$\Box\ \text{OTHER} \rightarrow mem$

Quand le champ pc du sommet de la pile du processus id contient la valeur "spinlock_acquire_init", l'opérateur $dispatch()$ appelle $spinlock_acquire()$ dont la définition est donnée dans le module $Sync_primitives$.

La traduction d'un appel à une primitive de synchronisation C consiste à appliquer la règle 3.24, définie dans le chapitre 3.

Exemple 4.1.4. *En appliquant la règle 3.24 sur l'instruction de la ligne 13 du code C donné par le listing 4.1, la traduction TLA+ est la suivante :*

$[\![\texttt{spinlock_acquire(\&lock);}]\!]_{stmt} \underset{R.3.24}{\longrightarrow}$
$stmt_p1_13(id, mem) \triangleq [mem \text{ EXCEPT}$
$\quad !.stack = [mem.stack \text{ EXCEPT } ![id] = mem.stack[id] \circ \langle Addr_lock\rangle],$
$\quad !.register \mapsto [mem.register \text{ EXCEPT } ![id] = \langle[pc \mapsto \langle\text{"spinlock_acquire_init"}\rangle, fp \mapsto Len(mem.stack[id]) + 1]\rangle$
$\qquad \circ \langle[pc \mapsto \langle\text{"p1_14_post"}\rangle, fp \mapsto Head(mem.register[id]).fp]\rangle \circ Tail(mem.register[id])]]$

La traduction d'un appel à la primitive `spinlock_acquire()` *consiste à empiler la pile $stack[id]$ avec les paramètres de la primitive appelée, à mettre à jour le sommet de la pile $register[id]$ avec l'identifiant "spinlock_acquire_init" et enfin à sauvegarder l'identifiant de l'instruction suivante, à savoir l'identifiant $p1_14_post$.*

Nous avons vu dans cette section comment une primitive de synchronisation est déclarée dans le code C. Nous avons ensuite décrit comment un appel à cette dernière est traduit dans TLA+. Nous consacrons la section suivante à la modélisation TLA+ de différentes primitives de synchronisation.

4.2 Modélisation des solutions de synchronisation

La synchronisation entre processus a suscité un intérêt particulier chez les chercheurs et pour cela, plusieurs solutions ont été proposées dans la littérature.

Après avoir décrit le modèle d'exécution dans la section 4.1, nous commençons cette section par introduire une section critique. Nous présentons ensuite des exemples de solutions de synchronisation, que nous divisons en trois catégories : solutions purement logicielles, solutions purement matérielles et solutions logicielles basées sur des mécanismes matériels. Nous donnons une spécification formelle en TLA+ de chaque exemple de solution en nous basant sur le modèle d'exécution proposé dans la section 4.1.

4.2.1 Section critique

En programmation concurrente, la partie du code dans laquelle se fait des accès concurrents à la mémoire partagée est appelée *section critique* (abrégée SC). Les processus doivent alors vérifier la propriété qu'à tout instant, au plus un processus parmi eux est en train d'exécuter une instruction de cette section critique. Cette section critique est exécutée en *exclusion mutuelle*.

```
// entreeSectionCritique()
// section_critique
// sortieSectionCritique()
```

Listing 4.2 – Structure d'un processus exécutant une section critique

Dans la plupart des cas, les solutions implémentant une section critique comprennent deux parties : un *protocole d'entrée* et un *protocole de sortie*. L'appel à ces deux protocoles suit le schéma d'exécution illustré par le listing 4.2. Le protocole d'entrée est une suite d'instructions permettant de vérifier si la section critique est exécutée par un autre processus. Si la section critique est occupée, le processus est mis en attente. Sinon, il y accède. À la sortie de de la section critique, le processus exécute un protocole de sortie permettant d'avertir les processus en attente de la section critique. L'attente générée par le protocole d'entrée peut-être :

- active : le processus boucle jusqu'à ce que la section critique devienne libre. Généralement dans une telle approche, un processus vérifie de façon itérative si une condition est vraie ;
- passive : le processus est mis dans l'état *bloqué* dès lors qu'un autre processus y est déjà. Il sera réveillé quand la section critique deviendra libre.

4.2.2 Solution de synchronisation purement logicielle

Plusieurs algorithmes ont été proposés pour résoudre le problème de l'exclusion mutuelle dans le cadre de machines multiprocesseurs à mémoire partagée. Citons, parmi les solutions purement logicielles, celle proposée par Lamport [Lam74] en 1974, connue sous le nom de l'algorithme de la boulangerie (« *bakery algorithm* » en anglais). Cette solution est donnée par l'algorithme 4.1.

ALGORITHME 4.1 : Pseudo code de l'algorithme de la boulangerie

Variables :
$choosing[1..n] \leftarrow \{false, \ldots, false\}$
$number[1..n] \leftarrow \{0, \ldots, 0\}$

Function : ENTREESECTIONCRITIQUE(i)
$choosing[i] \leftarrow true$
$number[i] = 1 + max(number[1], \ldots, number[N])$
$choosing[i] = false$
for $j \leftarrow 1$ **to** N
  while $(choosing[j])$ **do skip**
  while $(number[j] \neq 0 \wedge ((number[j], j) < (number[i], i)))$ **do skip**

Function : SORTIESECTIONCRITIQUE(i)
$number[i] = 0$

L'auteur s'est inspiré d'un protocole de gestion d'une file d'attente utilisé dans la vie quotidienne, où le dernier arrivé s'attribue un numéro d'ordre derrière les arrivants précédents.

Dans cet algorithme, les processus partagent deux variables : $choosing$ et $number$. La variable $choosing$ est un tableau de booléens dont tous les éléments sont initialisés à $false$. Ce tableau permet d'indiquer aux autres processus qu'une ou plusieurs demandes de numéro sont en cours. La variable $number$ est un tableau d'entiers dont tous les éléments sont initialisés à 0. Ce tableau permet d'ordonner les accès à la SC. Chaque processus choisit son propre numéro en fonction des numéros pris par les autres processus. Si deux processus ont choisi le même numéro, alors celui dont l'indice est le plus petit passe devant. En sortant de la section critique, le processus met $number[i]$ à 0.

La particularité de cette solution est qu'elle ne repose pas sur l'hypothèse de l'atomicité des accès mémoires à la différence par exemple de la solution de Peterson [Pet81] ou celle de Dekker [Dij68]. Par contre, son inconvénient est que les valeurs des compteurs $number[i]$ ne sont pas bornées et elles peuvent évoluer indéfiniment.

La spécification TLA+ de la solution de la boulangerie consiste à traduire directement l'implémentation C de l'algorithme 4.1 par C2TLA+.

4.2.3 Les solutions de synchronisation purement matérielles

Ce type de solution est basé sur un dispositif matériel. On trouve dans cette catégorie le mécanisme de désarmement des interruptions et les instructions matérielles. Dans la suite de cette section, nous présentons des exemples de solutions matérielles et nous décrivons leurs spécifications TLA+.

4.2.3.1 Désarmement des interruptions

Cette solution s'applique sur les systèmes monoprocesseurs. Elle consiste à masquer les interruptions quand un processus demande une section critique, et à les démasquer à la sortie de la section critique. Cette action permet de ne pas allouer le processeur à un autre processus tant qu'il existe un processus exécutant la section critique.

```
void mask_interrupt() __attribute__ ((Sync_primitives,alias ("__mask_interrupt")));
void unmask_interrupt() __attribute__ ((Sync_primitives,alias ("__unmask_interrupt")));

void proc1(){
  ...
  mask_interrupt()
  x = max(x,y)
  unmask_interrupt()
  ...}
```

Listing 4.3 – Exemple d'utilisation des fonctions de masquage/démasquage d'interruption

Nous considérons que l'appel aux primitives de masquage et de démasquage d'interruption dans le code C se fait respectivement par les fonctions `mask_interrupt()` et `unmask_interrupt()`. Le listing 4.3 illustre un extrait d'un exemple de code C dans lequel une section critique est délimitée par l'appel à ces deux primitives. Ces dernières sont déclarées dans le code C à traduire via les attributs. Dans ce qui suit, nous donnons la spécification TLA+ de ces primitives.

(a) Spécification TLA+ des opérations de masquage/démasquage d'interruption. Pour modéliser le masquage/démasquage d'interruption, nous étendons le modèle d'exécution en rajoutant dans la variable $memory$ un champ appelé $bitmask$. Ce champ peut prendre deux valeurs : un identificateur de processus ou la valeur $Undef$.

$$
\begin{aligned}
&mask_interrupt(id, mem) \triangleq [mem \text{ EXCEPT} \\
&\quad !.bitmask = id, !.register = [mem.register \text{ EXCEPT } ![id] = Tail(mem.register[id])]]
\end{aligned}
$$

$$
\begin{aligned}
&unmask_interrupt(id, mem) \triangleq [mem \text{ EXCEPT} \\
&\quad !.bitmask = Null, !.register = [mem.register \text{ EXCEPT } !![id] = Tail(mem.register[id])]]
\end{aligned}
$$

Listing 4.4 – Définition TLA+ des opérations $mask_interrupt()$ et $unmask_interrupt()$

Le listing 4.4 montre la définition TLA+ des opérations de masquage et de démasquage d'interruption. L'opérateur $mask_interrupt()$ assigne au champ $bitmask$ l'identifiant id du processus courant et dépile la pile des registres $mem.register[id]$. L'opération $unmask_interrupt$ met à $Null$ le champ $bitmask$ et dépile la pile $register[id]$.

Une fois que les deux opérateurs TLA+ $mask_interrupt()$ et $unmask_interrupt()$ sont définis, il suffit de réécrire la définition TLA+ de la fonction $procsExec()$ comme suit :

$$
\begin{aligned}
procsExec() \triangleq\ & \text{IF } mem.bitmask = Null \\
& \text{THEN } ProcSet \text{ ELSE } \{mem.bitmask\}
\end{aligned}
$$

La fonction $procsExec()$ spécifie l'ensemble des processus prêts à s'exécuter en tenant compte de la valeur du champ $bitmask$. Si $bitmask$ vaut $Null$, alors $procsExec()$ retourne l'ensemble $ProcSet$. Dans ce cas, tous les processus sont prêts à s'exécuter. Sinon, $procsExec()$ retourne un ensemble contenant l'identificateur du processus qui a appelé la primitive de masquage d'interruption, qui est l'unique processus autorisé à s'exécuter.

4.2.3.2 Les instructions matérielles

Les systèmes multiprocesseurs disposent d'instructions de lecture/écriture atomiques s'appuyant sur un dispositif matériel. Une instruction est dite *atomique* si son exécution est indivisible, c'est-à-dire qu'elle ne peut pas être interrompue par un autre processus. L'atomicité est une propriété fondamentale dans la programmation concurrente (ou parallèle) afin d'assurer la cohérence des données entre les différents processus. Les instructions matérielles sont des instructions bas niveau permettant d'échanger atomiquement une valeur contenue dans un registre dans une case mémoire de façon indivisible.

Étant donné que la sémantique de concurrence, que nous considérons dans ce travail, est basée sur l'entrelacement des actions et comme les instructions atomiques sont non-interruptibles, nous modélisons ces dernières en TLA+ de façon à ce que leur exécution se fasse en une seule action.

Il existe plusieurs instructions de synchronisation matérielles. Dans la suite de la section, nous décrivons deux exemples d'instructions : « *Test-And-Set* » et « *Compare-and-Swap* » et nous donnons ensuite leurs spécifications TLA+.

(a) Instruction « *Test-and-Set* ». L'instruction « *Test-and-Set* », abrégée TAS, est une instruction permettant de lire et d'écrire dans la mémoire d'une manière atomique. Une implémentation possible de cette instruction, exécutée atomiquement, est définie par le pseudo-code (séquentiel) donné par le listing 4.5.

```
int Test-and-Set (int *lock)
{
  if (*lock == 0)
    *lock = 1;
  return *lock;
}
```

Listing 4.5 – Pseudo code de l'instruction TAS

L'instruction TAS fait référence à une variable partagée, appelée `lock`. Cette instruction, sans être interrompue, lit la valeur pointée par `lock`, vérifie si elle est égale à 0, la met à 1 si c'est le cas, et enfin la retourne. Ainsi, deux processus ne peuvent pas passer simultanément la valeur pointée par `lock` de 0 à 1. Si un processus parvient à le faire, il a alors le droit d'accéder à la section critique.

$$Addr_param_TAS_lock \triangleq [loc \mapsto \text{“stack”}, offs \mapsto 0]$$

$$
\begin{array}{l}
TAS(id, mem) \triangleq \\
\quad \text{IF } load(id, mem, load(id, mem, Addr_param_TAS_lock)).val = 0 \\
\quad \text{THEN } [mem \text{ EXCEPT} \\
\qquad !.data = store(id, mem, load(id, mem, Addr_param_TAS_lock), [val \mapsto 1]).data, \\
\qquad !.stack = [mem.stack \text{ EXCEPT } ![id] = SubSeq(mem.stack[id], 1, Head(mem.register[id]).fp - 1)], \\
\qquad !.register = [mem.register \text{ EXCEPT } ![id] = Tail(mem.register[id])], \\
\qquad !.ret = [mem.ret \text{ EXCEPT } ![id] = load(id, mem, Addr_param_TAS_lock)]] \\
\quad \text{ELSE } [mem \text{ EXCEPT} \\
\qquad !.stack = [mem.stack \text{ EXCEPT } ![id] = SubSeq(mem.stack[id], 1, Head(mem.register[id]).fp - 1)], \\
\qquad !.register = [mem.register \text{ EXCEPT } ![id] = Tail(mem.register[id])], \\
\qquad !.ret = [mem.ret \text{ EXCEPT } ![id] = load(id, mem, Addr_param_TAS_lock)]]
\end{array}
$$

Listing 4.6 – Définition TLA+ de l'opérateur $TAS()$

La spécification TLA+ de l'instruction TAS est illustrée par le listing 4.6. Le pseudo-code C de TAS (illustré par le listing 4.5) prend en argument le paramètre `lock`. Nous définissons alors l'adresse TLA+ de ce paramètre par la constante $Addr_param_TAS_lock$. L'opérateur TLA+ $TAS()$, qui modélise l'opération réalisée par l'instruction TAS, prend en paramètre l'identificateur id de processus et la mémoire mem et retourne le nouvel état mémoire. Si la valeur donnée par l'expression $load(id, mem, load(id, mem, Addr_param_TAS_lock))$ (qui correspond à l'expression conditionnelle `(*lock)== 0`) vaut 0, alors la valeur 1 est chargée dans $data$ à l'adresse donnée par $load(id, mem, Addr_param_TAS_lock)$. Les piles $register$ et $stack$ sont ensuite dépilées. Enfin, la valeur donnée par $load(id, mem, Addr_param_TAS_lock)$, (qui correspond à la valeur C donnée par `*lock`) est mise dans le registre $ret[id]$.

(b) Instruction « *CAS* ». Un pseudo-code (séquentiel) de l'instruction atomique « *Compare-and-swap* », abrégée CAS, est donné par le listing 4.7.

```
int CAS (int* addr, int oldVal, int newVal)
  {
     if (*addr == oldVal)
       {
          *addr = newVal;
          return 1;
       }
       else return 0;
   }
```

Listing 4.7 – Pseudo code de l'instruction CAS

L'instruction CAS prend 3 arguments : une adresse mémoire `addr`, la valeur `oldVal` contenue dans cette adresse et la nouvelle valeur `newVal` à écrire dans `addr`. Si la valeur contenue dans l'adresse `addr` est égale à la valeur `oldVal`, l'instruction CAS charge la nouvelle valeur `newVal` dans cette adresse et retourne la valeur 1. Sinon, elle retourne la valeur 0, c'est-à-dire la valeur contenue dans `addr` a été modifiée entre-temps par un autre processus.

Nous proposons dans le listing 4.8 une spécification TLA+ de l'instruction CAS.

$$
\begin{array}{l}
Addr_CAS_param_addr \triangleq [loc \mapsto \text{“stack”}, offs \mapsto 0] \\
Addr_CAS_param_old \triangleq [loc \mapsto \text{“stack”}, offs \mapsto 1] \\
Addr_CAS_param_new \triangleq [loc \mapsto \text{“stack”}, offs \mapsto 2] \\
\\
CAS(id, mem) \triangleq \\
\quad \text{IF } (load(id, mem, load(id, mem, Addr_CAS_param_addr)) = load(id, mem, Addr_CAS_param_old)) \\
\quad \text{THEN } [mem \text{ EXCEPT} \\
\qquad\qquad !.data = [mem.data \text{ EXCEPT } ![load(id, mem, Addr_CAS_param_addr).offs] = \\
\qquad\qquad\qquad load(id, mem, Addr_CAS_param_new)], \\
\qquad\qquad !.stack = [mem.stack \text{ EXCEPT } ![id] = SubSeq(mem.stack[id], 1, Head(mem.register[id]).fp - 1)], \\
\qquad\qquad !.register = [mem.register \text{ EXCEPT } ![id] = Tail(mem.register[id])], \\
\qquad\qquad !.ret = [mem.ret \text{ EXCEPT } ![id] = [val \mapsto 1]]] \\
\quad \text{ELSE } [mem \text{ EXCEPT} \\
\qquad\qquad !.data = [mem.stack \text{ EXCEPT } ![id] = SubSeq(mem.stack[id], 1, Head(mem.register[id]).fp - 1)], \\
\qquad\qquad !.register = [mem.register \text{ EXCEPT } ![id] = Tail(mem.register[id])], \\
\qquad\qquad !.ret = [mem.ret \text{ EXCEPT } ![id] = [val \mapsto 0]]]
\end{array}
$$

Listing 4.8 – Définition TLA+ de l'instruction $CAS()$

Dans cette définition, nous déclarons les constantes $Addr_CAS_param_addr$, $Addr_CAS_param_old$ et $Addr_CAS_param_new$ modélisant respectivement les adresses mémoires des paramètres C `addr`, `oldVal` et `newVal`. Comme pour l'instruction TAS, la définition TLA+ de CAS est donnée par l'opérateur $CAS()$ qui prend en argument l'identificateur de processus id et la mémoire mem, et retourne le nouvel état mémoire. Dans cette définition, la valeur donnée par $load(id, mem, load(id, mem, Addr_CAS_param_addr)$ (qui correspond à l'expression C `*addr`) est comparée à la valeur de l'expression $load(id, mem, Addr_CAS_param_old)$. Si les deux valeurs sont égales, alors on charge dans la mémoire $data$ la valeur donnée par l'expression $load(id, mem, Addr_CAS_param_new)$ et la valeur $[val \mapsto 1]$ dans le registre $ret[id]$. Sinon, on charge la valeur $[val \mapsto 0]$ dans $ret[id]$. Enfin, on dépile les files $register[id]$ et $stack[id]$.

Nous présentons dans la section 4.2.4.1 suivante une solution de synchronisation utilisant l'instruction CAS.

4.2.4 Solution logicielle basée sur un mécanisme matériel

Nous décrivons dans cette partie les solutions de synchronisation logicielles s'appuyant sur un mécanisme matériel. Nous prenons comme exemples le verrou tournant et le mécanisme de sémaphore.

4.2.4.1 Verrou tournant ou *spinlock*

Un *spinlock* ou *verrou tournant* est un mécanisme de synchronisation basé sur une attente active. Le listing 4.9 illustre un exemple d'implémentation C d'un *spinlock* contenant deux fonctions : une fonction `spinlock_acquire()` permettant d'entrer en

SC et une `spinlock_release()` permettant de libérer la SC. Ces deux fonctions font référence à une variable partagée désignée par `l`.

```
void spinlock_acquire(spinlock *lock){
  while (CAS(&lock,1,0));}

void spinlock_release(spinlock *lock){
*lock = 1;}
```

Listing 4.9 – Exemple d'implémentation d'un *spinlock*

Les opérations de lecture/écriture de la valeur pointée par `lock` se font en appelant l'instruction atomique CAS. La fonction `spinlock_acquire()` permet de boucler sur l'instruction CAS jusqu'à ce que la valeur pointée par `lock` devienne égale à 0. La fonction `spinlock_release()` relâche le verrou en le mettant à 1.

(a) Spécification TLA+ d'un *spinlock*. Pour l'implémentation donnée par le listing 4.9, l'instruction atomique utilisée est une instruction `CAS()`. En utilisant C2TLA+, la traduction de cette implémentation est donnée dans l'annexe B. Dans cette traduction, chaque instruction C est traduite en un opérateur TLA+ en se basant sur le principe de traduction décrit dans le chapitre 3.

Si on veut se focaliser sur la vérification des sections critiques et on ne s'intéresse pas à l'implémentation du *spinlock*, il serait utile que la spécification de cette dernière génère moins d'entrelacements, c'est-à-dire que la spécification contient moins d'actions. Nous proposons alors une autre définition TLA+ des primitives `spinlock_acquire()` et `spinlock_release()` que nous illustrons dans le listing 4.10. La particularité de cette définition est que les opérations d'acquisition et de libération du verrou sont considérées atomiques et sont donc traduites en un seul opérateur TLA+.

Dans la définition donnée par le listing 4.10, les constantes $Addr_param_spinlock_acquire_lock$ et $Addr_param_spinlock_release_lock$ modélisent respectivement les paramètres des fonctions `spinlock_acquire()` et `spinlock_release()`.

L'opérateur TLA+ $spinlock_acquire()$ modélise la fonction C d'acquisition du verrou. Il consiste à définir deux opérateurs temporaires mem_preCAS et mem_CAS. L'opérateur mem_preCAS renvoie la valeur de la mémoire mem juste avant l'appel à l'instruction CAS. Il empile $stack[id]$ avec les paramètres de la fonction `CAS()` et met à jour le sommet de la pile des registres $mem.register[id]$. L'opérateur $mem_CAS()$ appelle l'instruction CAS en lui passant en paramètre l'état de la mémoire retourné par mem_preCAS. L'opérateur $spinlock_acquire()$ consiste ensuite à tester la valeur retournée par l'expression $mem_CAS.ret[id]$, qui correspond à la valeur retournée par l'instruction CAS. Si cette valeur vaut $[val \mapsto 1]$, alors la mémoire mem est mise à jour et les mémoires $stack[id]$ et $register[id]$ sont dépilées. Sinon, l'état de la mémoire mem_preCAS est retourné. Ce qui permet de reboucler sur l'appel à l'instruction CAS.

L'opérateur $spinlock_release()$ met à jour la mémoire $data$ en mettant le verrou à la

```
Addr_param_spinlock_acquire_lock ≜ [loc ↦ "stack", offs ↦ 0]
Addr_param_spinlock_release_lock ≜ [loc ↦ "stack", offs ↦ 0]

spinlock_acquire(id, mem) ≜
  LET mem_preCAS ≜ [mem EXCEPT
          !.stack = [mem.stack EXCEPT ![id] = mem.stack[id] ∘
                      ⟨load(id, mem, Addr_param_spinlock_acquire_l), [val ↦ 1], [val ↦ 0]⟩],
          !.register = [mem.register EXCEPT ![id] = ⟨[pc ↦ "CAS", fp ↦ Len(mem.stack[id]) + 1]⟩ ∘
                      mem.register[id]]]
       mem_CAS ≜ CAS(id, mem_preCAS)
  IN  IF (mem_preCAS.ret[id] = [val ↦ 1])THEN mem_preCAS
       ELSE [mem_preCAS EXCEPT
          !.stack = [mem_preCAS.stack EXCEPT ![id] =
                         SubSeq(mem_preCAS.stack[id], 1, Head(mem_preCAS.register[id]).fp − 1)],
          !.register = [mem_preCAS.register EXCEPT ![id] = Tail(mem_preCAS.register[id])]]

spinlock_release(id, mem) ≜
    [mem EXCEPT
       !.data = store(id, mem, load(id, mem, Addr_param_spinlock_release_lock), [val ↦ 0]).data,
       !.stack = [mem.stack EXCEPT ![id] = SubSeq(mem.stack[id], 1, Head(mem.register[id]).fp − 1)],
       !.register = [mem.register EXCEPT ![id] = Tail(mem.register[id])]]
```

Listing 4.10 – Définition TLA+ des opérateurs $spinlock_acquire()$ et $spinlock_release()$

valeur $[val \mapsto 1]$. Il dépile ensuite les mémoires $stack[id]$ et $register[id]$.

L'utilisateur peut utiliser la définition TLA+ du *spinlock* donnée par le listing 4.10 en déclarant dans le code C les primitives `spinlock_acquire()` et `spinlock_release()` à travers le mécanisme d'attribut et en spécifiant comme paramètre d'attribut le module où les primitives sont spécifiées.

4.2.4.2 Sémaphore

Le sémaphore est un mécanisme de synchronisation proposé par Dijkstra en 1965 [Dij65]. Il s'agit d'une solution par attente passive qui permet d'attribuer le processeur à un autre processus, quand un processus est en attente d'une section critique.

Un sémaphore est composé d'un compteur sem et une file d'attente de processus $queue_{procs}$. Il est géré par trois opérations, notées $Init()$, $P()$ et $V()$. Ces opérations sont illustrées par l'algorithme 4.2 et permettent de gérer l'accès à la SC et de synchroniser les processus.

- $Init()$ permet d'initialiser le sémaphore avec une valeur positive non nulle. Cette initialisation ne doit être faite qu'une seule fois ;

- $P()$ est une opération de demande d'accès, qui décrémente sem. Lorsqu'un processus p tente d'acquérir un sémaphore indisponible (la valeur de sem est inférieure à 0), il est alors placé à la fin de la file $queue_{procs}$ et son état est mis à *bloqué* ;

ALGORITHME 4.2 : Algorithme des opérations P() et V() sur un sémaphore

Variables :
  sem is an integer
  p is the current process
  $queue_{procs}$ is an empty queue
Function : INIT(sem)
  $sem \leftarrow postivevalue$

Function : P(sem)
  $sem \leftarrow sem - 1$
  if ($sem < 0$) **then**
    $set_process(p, \text{“}blocked\text{”})$
    $push(p, queue_{procs})$

Function : V(sem)
  if ($sem < 0$) **then**
    $p \leftarrow pop(queue_{procs})$
    $set_process(p, \text{“}ready\text{”})$
  $sem \leftarrow sem + 1$

- $V()$ est une opération de libération du sémaphore, qui incrémente sem. Le processus en tête de la file $queue_{procs}$ est débloqué et peut ainsi refaire une demande d'acquisition du sémaphore.

Quand elle devient négative, sa valeur absolue indique le nombre de processus bloqués se trouvant dans la file $queue_{proc}$. Quand la valeur de sem est positive, elle indique le nombre de processus autorisés à accéder à la section critique. Dans le cas où sem est initialisé à 1, le sémaphore permet d'assurer l'exclusion mutuelle pour l'accès à la section critique. Dans ce cas, il s'agit d'un sémaphore binaire, appelé aussi *mutex*.

(a) Spécification TLA+ d'un sémaphore. Comme le sémaphore est un mécanisme basé sur l'attente passive dans lequel l'état d'un processus peut changer, nous étendons le modèle d'exécution afin de gérer les états possibles que peut prendre un processus.

Nous définissons un nouveau champ appelé $procs$ dans la variable TLA+ $memory$. Ce dernier contient un tableau d'enregistrements dont chaque élément correspond à un processus. Chaque enregistrement contient un champ st qui spécifie l'état du processus. Ce champ peut prendre deux valeurs : “blocked” (état *bloqué*) ou “ready” (état *prêt*) :

- l'état bloqué désigne que le processus est en attente de ressource autre que le processeur ;

- l'état prêt indique que le processus peut s'exécuter ou qu'il s'exécute sur le processeur.

Ces deux états, d'une part, suffisent pour modéliser les applications concurrentes que nous considérons dans ce travail et, d'autre part, cela permet d'éviter des entrelacements inutiles que l'on obtiendrait si on considérait plusieurs états. Cependant, l'utilisateur peut considérer d'autres états d'un processus si l'application à étudier le nécessite.

Il est à noter que le fait d'avoir dans le champ $procs$ un enregistrement par processus permet d'étendre facilement le modèle d'exécution pour rajouter d'autres champs comme par exemple la priorité d'un processus.

Pour modéliser la file des processus bloqués sur un sémaphore, nous rajoutons un champ appelé $locks$ qui permet de stocker les sémaphores utilisés dans le code C. Ce champ est modélisé à travers une séquence d'enregistrements, dont chaque enregistrement contient :

- un champ key qui contient l'identificateur du sémaphore. Il s'agit de l'adresse TLA+ du compteur associé au sémaphore ;
- un champ $queue$ modélisant la file (de type FIFO) des processus bloqués sur le sémaphore. Ce champ contient une séquence d'identificateurs de processus.

Nous considérons que tous les sémaphores sont déclarés et initialisés comme des variables globales. Dans TLA+, ces sémaphores sont créés à l'état initial.

Exemple 4.2.1. *Prenons un exemple de code C déclarant un sémaphore* $sem1$ *(dont l'adresse TLA+ est* $Addr_sem1$*). L'initialisation du champ* $locks$ *par le prédicat* $Init$ *se fait comme suit :*

$$\begin{aligned} Init \triangleq \; memory = [&data \mapsto \ldots \\ &locks \mapsto \langle [key \mapsto Addr_sem1, queue \mapsto \langle\rangle] \rangle \\ &\ldots \end{aligned}$$

Le champ $locks$ *est composé d'une séquence contenant un enregistrement dont le champ* key *comprend l'adresse du sémaphore* $sem1$*, à savoir* $Addr_sem1$ *et dont la séquence* $queue$ *est initialement vide.*

L'appel aux opérations d'acquisition et de libération de sémaphore dans le code C se fait en appelant respectivement des fonctions `P()` et `V()`. Ces fonctions prennent comme paramètre l'adresse du compteur associé au sémaphore. La spécification TLA+ de ces opérations est décrite dans ce qui suit.

Spécification de l'opération $P()$. Comme le montre le listing 4.11, la spécification de l'opération $P()$ comporte la déclaration d'une constante $Addr_P_param_sem$ ainsi que la définition de l'opérateur $P()$. La constante $Addr_P_param_sem$ modélise l'adresse mémoire du paramètre passé en argument à la fonction `P()`. Il s'agit de l'adresse du compteur associé au sémaphore. Par défaut, ce paramètre est stocké au sommet du cadre de pile de la fonction `P()`.

L'opérateur $P()$ prend en argument l'identificateur de processus id et la mémoire mem et retourne une nouvelle valeur de la mémoire mem. Sa définition consiste à tester la valeur du compteur, dont la valeur est donnée par l'expression TLA+

```
Addr_P_param_sem ≜ [loc ↦ "stack", offs ↦ 0]

P(id, mem) ≜
   IF (load(id, mem, load(id, mem, Addr_P_param_sem)).val ≤ 0)
  THEN [mem EXCEPT
      !.procs = [mem.procs EXCEPT ![id].st = "blocked"],
      !.locks = LET idSem ≜ getIndexSem(load(id, mem, Addr_P_param_sem), mem)
                IN [mem.locks EXCEPT ![idSem].queue = mem.locks[idSem].queue ∘ ⟨id⟩]
   ELSE [mem EXCEPT
      !.data = store(id, mem, load(id, mem, Addr_P_param_sem),
               minus(load(id, mem, load(id, mem, Addr_P_param_sem)), [val ↦ 1])).data,
      !.stack = [mem.stack EXCEPT ![id] = SubSeq(mem.stack[id], 1, Head(mem.register[id]).fp − 1)],
      !.register = [mem.register EXCEPT ![id] = Tail(mem.register[id])]]
```

Listing 4.11 – Spécification TLA+ de l'opération $P()$ d'un sémaphore

$(load(id, mem, load(id, mem, Addr_P_param_sem)).val$:

- si cette valeur est supérieure à zéro, alors le compteur associé est décrémenté, les piles $register[id]$ et $stack[id]$ sont ensuite dépilées ;

- sinon, le processus courant id est mis à l'état bloqué. Pour mettre à jour le champ $locks$, nous définissons une fonction $getIndexSem()$ (voir listing 4.12) qui prend en argument l'identifiant du sémaphore (l'adresse du compteur associé au sémaphore) et retourne l'index associé au sémaphore dans le champ $locks$.

```
getIndexSem(key, mem) ≜
   IF ∃ j ∈ 1..Len(mem.locks) : mem.locks[j].key = key
   THEN CHOOSE j ∈ 1..Len(mem.locks) : mem.locks[j].key = key
   ELSE 0
```

Listing 4.12 – Définition TLA+ de la fonction $getIndexSem()$

Par la suite, la valeur retournée par cette fonction est testée. Deux cas se présentent :
- Si l'identificateur du sémaphore n'existe pas dans $locks$ (la valeur retournée par $idSem$ est 0 car il s'agit d'un premier appel à une opération $P()$ du sémaphore), alors un nouvel enregistrement est empilé. Cet enregistrement est composé d'un champ key contenant l'adresse du compteur associé au sémaphore et d'un champ $queue$ comprenant l'identificateur du processus id.
- Sinon, l'identificateur du processus courant id est empilé dans la file $queue$ du sémaphore.

Spécification de l'opération $V()$. Comme pour l'opération $P()$, la spécification TLA+ de $V()$ (listing 4.13) utilise une constante $Addr_V_param_sem$ et définit l'opérateur $V()$. La constante $Addr_P_param_sem$ modélise l'adresse mémoire du paramètre passé en argument à la fonction `V()`.

L'opérateur $V()$ prend en arguments l'identificateur de processus id et la mémoire mem et retourne une nouvelle valeur de mem. La définition de $V()$ consiste à incrémenter le compteur associé au sémaphore. Les piles $register[id]$ et $stack[id]$ sont ensuite dépilées.

$Addr_V_param_sem \triangleq [loc \mapsto \text{"stack"}, offs \mapsto 1]$

$$
\begin{array}{l}
V_stmt(id, mem) \triangleq \\
\quad \text{LET } idSem \triangleq getIndexSem(load(id, mem, Addr_V_param_sem), mem) \\
\quad \text{IN } [mem \text{ EXCEPT} \\
\qquad !.data = store(id, mem, load(id, mem, Addr_V_param_sem), \\
\qquad\qquad plus(load(id, mem, load(id, mem, Addr_V_param_sem)), [val \mapsto 1])).data, \\
\qquad !.stack = [mem.stack \text{ EXCEPT } ![id] = SubSeq(mem.stack[id], 1, Head(mem.register[id]).fp - 1)], \\
\qquad !.register = [mem.register \text{ EXCEPT } ![id] = Tail(mem.register[id])], \\
\qquad !.procs = \text{IF } mem.locks[idSem].queue = \langle\rangle \text{THEN } mem.procs \\
\qquad\qquad \text{ELSE } [mem.procs \text{ EXCEPT } ![Head(mem.locks[idSem].queue)].st = \text{"ready"}], \\
\qquad !.locks = \text{IF } mem.locks[idSem].queue = \langle\rangle \text{THEN } mem.locks \\
\qquad\qquad \text{ELSE } [mem.locks \text{ EXCEPT } ![idSem].queue = Tail(mem.locks[idSem].queue)]]
\end{array}
$$

Listing 4.13 – Spécification TLA+ de l'opération $V()$ d'un sémaphore

Le processus à la tête de la file $mem.locks[idSem].queue$ est dépilé et son état est mis à l'état prêt.

(b) Spécification de la fonction $procsExec()$. Comme le mécanisme de sémaphore est basé sur une attente passive, il convient de définir la fonction $procsExec()$ afin de ne choisir que les processus qui ne sont pas bloqués sur un sémaphore. Cette fonction s'écrit alors comme suit :

$$procsExec(mem) \triangleq \{id \in ProcSet : memory.procs[i].st \neq \text{"blocked"}\}$$

Dans ce cas, l'appel à l'action $Next$ permet de choisir un processus non bloqué pour exécuter une de ces actions. Avec les définitions TLA+ des opérations $P()$ et $V()$ présentées ci-dessus, il est possible de vérifier un programme C utilisant le sémaphore comme primitive de synchronisation. Il suffit de déclarer ces opérations dans le code C à traduire avec le mécanisme d'attribut tout en précisant le module où les spécifications TLA+ des opérations $P()$ et $V()$ sont définies.

4.3 TEMPS-RÉEL ET ORDONNANCEMENT

Dans cette section, nous présentons l'extension du modèle d'exécution pour spécifier le temps-réel et construire une spécification temps-réel vérifiable par TLC. Nous commençons dans la section 4.3.1 par présenter la modélisation du temps dans TLA+. Nous décrivons par la suite dans la section 4.3.2 un exemple de modélisation TLA+ d'un système avec des contraintes temporelles. En se basant sur cette modélisation, nous illustrons dans la section 4.3.3 des exemples d'ordonnancement temps-réel que nous spécifions formellement en TLA+.

4.3.1 Évolution du temps

```
MODULE RealTime
This module declares the variable now, which represents real time, and defines operator NowNext which
increases now until the maximum MaxTime is reached
EXTENDS Naturals
CONSTANT MaxTime
VARIABLE now
NowNext(v) ≜ ∧ now ≤ Maxtime
             ∧ now' = now + 1
             ∧ UNCHANGED v
```

Listing 4.14 – Définition du module $RealTime$

Lamport [BL02] décrit dans son livre « *Specifying Systems* » un module appelé $RealTime$ permettant d'écrire des spécifications temps-réel. Son principe consiste à définir une variable, nommée *now*, qui représente le temps. Dans ce travail, nous n'utilisons pas le module $RealTime$ proposé par Lamport, vu que les contraintes de temps définies dans ce module sont exprimées sur le temps d'exécution d'une action alors que nous nous intéressons à définir des contraintes de temps sur l'exécution d'un processus, c'est-à-dire sur une suite d'actions. Pour cela, nous redéfinissons le module $RealTime$ dont le contenu est donné par le listing 4.14. Ce module déclare la variable now représentant le temps et une constante $MaxTime$ désignant la valeur maximum que peut prendre la variable now. Il définit ensuite l'opérateur $NowNext(v)$ qui prend en argument un tuple de variables v et qui incrémente la variable now de 1 jusqu'à la valeur $MaxTime$ en gardant les valeurs des variables contenues dans v inchangées. Le fait de borner l'évolution de now permet d'éviter que l'exploration d'espace d'état s'exécute indéfiniment.

4.3.2 Modèle avec contraintes temporelles

Pour modéliser un système temps-réel, nous considérons qu'un processus temps-réel est caractérisé par :

- une date d'activation, qui représente le moment où le processus est activable ;
- le délai critique acceptable pour exécuter le processus, appelé *échéance*.

Pour modéliser ces deux paramètres dans notre modèle d'exécution TLA+, nous rajoutons pour chaque processus deux champs, un champ appelé act qui spécifie la date d'activation du processus et un champ ddl qui modélise l'échéance de ce dernier. Ces deux champs sont spécifiés pour chaque enregistrement contenu dans $procs$.

D'autres paramètres peuvent être ajoutés à ce modèle. Nous pourrons par exemple définir la période d'un processus quand il s'agit d'un processus périodique. Mais, nous nous limitons dans cette section à la modélisation de processus sans contrainte de periodicité. En se basant sur ces paramètres, nous supposons qu'un système temps-réel

est composé de plusieurs processus. La contrainte temporelle que nous considérons consiste à ce que chaque processus achève son exécution avant le dépassement de son échéance. L'action $Next$ de la spécification TLA+ de ce système est donnée par le listing 4.15.

$$procsExec(mem) \triangleq \{id \in ProcSet : mem.procs[id].act \leq now \land mem.procs[i].ddl - 1 \geq now\}$$

$$procToBeCompletedNow(mem) \triangleq$$
$$\{id \in ProcSet : mem.procs[id].act \leq now \land mem.procs[i].ddl - 1 = now \land mem.register[i] \neq \langle\rangle\}$$

$$
\begin{array}{l}
Next \triangleq \\
\quad \land (NowNext(memory) \lor \text{UNCHANGED } now) \\
\quad \land \\
\qquad \lor \exists id \in procsExec(memory) : \\
\qquad\quad \land memory.register[id] \neq \langle\rangle \\
\qquad\quad \land memory' = dispatch(id, memory) \\
\qquad \lor procToBeCompletedNow = \{\} \land \text{UNCHANGED } memory
\end{array}
$$

Listing 4.15 – Spécification TLA+ de l'action $Next$ pour un système temps-réel

La fonction $procsExec()$ est redéfinie afin de retourner l'ensemble des processus dont l'instant d'activation est atteint et dont l'échéance n'est pas imminente, c'est-à-dire la valeur $ddl - 1$ est supérieure à la valeur de now.

La fonction $procToBeCompletedNow()$ retourne les processus dont l'échéance est atteinte et qui n'ont pas encore terminé leur exécution.

La définition de l'action $Next$ consiste en la conjonction de deux formules :

- la première formule est une disjonction de deux actions : l'action $NowNext()$ qui incrémente le temps en gardant la variable $memory$ inchangée et l'action qui garde le temps inchangé ;
- la deuxième formule est aussi une disjonction de deux actions : une action qui choisit de manière indéterministe un processus parmi les processus retournés par la fonction $procsExec()$ et exécute une de ses actions. La deuxième action exprime qu'il n'y a pas de processus dont l'exécution est imminente et que la variable $memory$ reste inchangée.

L'écriture de l'action $Next$ permet d'exprimer l'entrelacement entre l'action qui incrémente le temps et une action d'un processus activable dont l'exécution n'est pas imminente. Quand un processus actif atteint son échéance et n'a pas encore achevé son exécution, le temps reste inchangé. La spécification TLA+ de ce système correspond alors à tous les entrelacements possibles entre le temps et l'exécution des différentes actions des processus.

4.3.3 Modèle d'ordonnancement temps-réel

Vu que les processus temps-réel n'ont pas souvent la même importance vis-à-vis du système. Une priorité est attribuée à chaque processus afin de les distinguer. Parmi les

types d'affection de priorité, on trouve :

- la priorité fixe : la priorité d'un processus est indépendante du temps et reste toujours identique durant l'exécution de l'application. Elle est initialement fixée par le concepteur du système ;
- la priorité dynamique : la priorité d'un processus varie au cours du temps.

Le choix du processus à exécuter dépend de l'algorithme d'ordonnancement implémenté. Nous décrivons dans la suite de ce manuscrit comment le modèle d'exécution temps-réel présenté ci-dessus peut s'étendre pour modéliser différents types d'ordonnancement.

4.3.3.1 Ordonnancement avec priorité fixe

Dans ce type d'ordonnancement, c'est le processus le plus prioritaire qui s'exécute. Pour modéliser ce type de système, nous étendons notre modèle d'exécution en rajoutant dans chaque enregistrement contenu dans le champ $procs$ un nouveau champ que nous appelons pr et qui spécifie la priorité d'un processus (nous considérons que 0 est la priorité la plus basse).

Il suffit ensuite de définir l'ensemble des processus activables, à savoir spécifier la fonction $procsExec()$, que nous redéfinissons comme suit :

$$procsExec(mem) \triangleq \{id1 \in ProcSet : \forall id2 \in ProcSet : mem.procs[id1].pr \geq mem.procs[id2].pr\}$$

Si nous considérons que les processus utilisent des primitives de synchronisation passive. La fonction $procsExec(mem)$ devient :

$$\begin{aligned} procsExec(mem) \triangleq \{id1 \in ProcSet : \forall id2 \in ProcSet : & \; mem.procs[id1].st \neq \text{“blocked”} \\ & \wedge mem.procs[id1].pr \geq mem.procs[id2].pr\} \end{aligned}$$

4.3.3.2 Ordonnancement avec priorité dynamique

Dans le cas d'un ordonnancement à priorité dynamique, les priorités entre processus sont calculées en fonction de différents critères comme l'utilisation des ressources et l'échéance du processus. Dans cette section, nous présentons un exemple d'ordonnancement dynamique avec héritage de priorité.

(a) Héritage de priorité. L'héritage de priorité consiste à gérer l'accès à une ressource partagée de façon à éviter l'inversion de priorité. Le phénomène d'inversion de priorité se produit quand un processus de haute priorité voulant accéder à la section critique se trouve bloqué par un processus moins prioritaire détenant la SC. Pour éviter ce problème, le principe de l'héritage de priorité consiste à attribuer à un processus qui bloque sur une SC une priorité temporaire, et ce jusqu'à la libération de la SC. Cette priorité temporaire correspond à la priorité la plus haute des processus bloqués sur cette SC.

Dans ce qui suit, nous proposons une modélisation dans laquelle nous considérons que les processus sont synchronisés à travers un sémaphore dont la définition TLA+ est

donnée dans la section 4.2.4.2. Cette modélisation utilise la même définition de l'action $Next$ définie dans le listing 4.15. Par contre, nous redéfinissons la fonction $procsExec()$ qui retourne les processus activables. La nouvelle définition de cette fonction doit prendre en compte le fait que la priorité de chaque processus peut s'inverser.

Nous calculons alors pour chaque processus une nouvelle priorité. Cette priorité est déterminée par la fonction TLA+ $newPrioProc()$ définie dans le listing 4.16. Elle prend en argument l'identificateur de processus et la mémoire mem et retourne la priorité maximale entre la priorité de base du processus (désignée par $mem.procs[id].pr$) et la prioritaire temporaire calculée par $tempPrio()$.

$$newPrioProc(id, mem) \triangleq getMax(mem.procs[id].pr, tempPrio(id, mem))$$

Listing 4.16 – Définition de la fonction $newPrioProc()$

Le calcul de la priorité temporaire revient à déterminer les processus qui sont bloqués par le sémaphore détenu par le processus id et à choisir la priorité la plus haute parmi ces processus. Mais, avec la spécification TLA+ du sémaphore donnée dans la section 4.2.4.2, il n'est pas possible de déterminer pour chaque processus le sémaphore qu'il détient. Par conséquent, nous étendons la définition TLA+ des opérations $P()$ et $V()$ du sémaphore afin de sauvegarder quel sémaphore le processus détient. Dans le listing 4.17, nous présentons seulement les définitions que nous avons ajoutées aux opérations $P()$ et $V()$ spécifiées dans la section 4.2.4.2.

```
P(id, mem) ≜
      IF (load(id, mem, load(id, mem, Addr_P_param_sem)).val ≤ 0)
      THEN [mem EXCEPT ....
       ELSE [mem EXCEPT ....
                !.procs = [mem.procs EXCEPT ![id].acqLock = load(id, mem, Addr_P_param_sem)],
                ...
V(id, mem) ≜
      LET ...
      IN   [mem EXCEPT
                ...
                !.procs = IF mem.locks[idSem].queue = ⟨⟩THEN ...
                            ELSE [mem.procs EXCEPT ![Head(mem.locks[idSem].queue)].st = "ready",
                                                   ![id].acqLock = Null]]
```

Listing 4.17 – Extension des définitions des fonctions $P()$ et $V()$

Nous ajoutons un nouveau champ appelé $acqLock$ dans le champ $procs$. Quand le processus détient le sémaphore, il assigne à ce champ l'identifiant de ce sémaphore. Quand ce dernier libère la section critique en appelant la fonction $V()$, il met le champ $acqLock$ à la valeur $Null$.

$$
\begin{array}{l}
tempPrio(id, mem) \triangleq \\
\quad \text{IF } \land\ mem.procs[id].acqLock \neq Null \land getSemByKey(mem.procs[id].acqLock, mem) \neq 0 \\
\quad\quad \land\ mem.locks[getSemByKey(mem.procs[id].acqLock, mem)].queue \neq \langle\rangle \\
\quad \text{THEN } maxPrio(mem.locks[getSemByKey(mem.procs[id].acqLock, mem)].queue, mem) \\
\quad \text{ELSE } mem.procs[id].pr
\end{array}
$$

Listing 4.18 – Définition de la fonction $tempPrio()$

Une fois que nous avons sauvegardé le sémaphore acquis par un processus, il est simple de déterminer la priorité temporaire d'un processus. Cette priorité est calculée par la fonction $tempPrio()$ dont la spécification TLA+ est illustrée par le listing 4.18. Cette définition consiste tout d'abord à calculer l'identifiant du sémaphore détenu par le processus id (s'il en détient un), désigné par $getSemByKey(mem.procs[id].acqLock, mem)$. Elle teste par la suite si la liste des processus bloqués sur le sémaphore détenu par le processus id (désignée par $mem.locks[getSemByKey(mem.procs[id].acqLock, mem)].queue$) est non vide. Si oui, elle retourne la priorité maximale de ces processus bloqués, en appelant la fonction $maxPrio()$. Sinon, elle retourne 0 (0 est la priorité minimale que prendre un processus).

$$
\begin{array}{l}
maxPrio(seq, mem) \triangleq \\
\quad \text{IF } Len(seq) = 1 \text{THEN } seq[1] \\
\quad \text{ELSE CHOOSE } id1 \in 1..Len(seq) : \forall id2 \in 1..Len(seq) : id1 \neq id2 \\
\quad\quad\quad\quad \land\ mem.procs[seq[id1]].pr \geq mem.procs[seq[id2]].pr
\end{array}
$$

Listing 4.19 – Définition de la fonction $maxPrio()$

La fonction $maxPrio()$ (voir listing 4.19) prend en argument une séquence de processus et retourne le processus ayant la priorité maximale. Quand cette séquence est de taille 1, elle retourne l'unique élément contenu dans celle-ci.

$$
\begin{array}{l}
procsExec(mem) \triangleq \\
\quad \text{LET } activeProcs \triangleq \{id1 \in ProcSet : \land\ (mem.procs[id1].st = \text{“ready”}) \land (mem.procs[id].act \leq now \\
\quad\quad\quad\quad \land\ (mem.procs[id1].ddl - 1 \geq now) \land (mem.register[id1] \neq \langle\rangle)\} \\
\quad \text{IN } \{id1 \in activeProcs : \forall id2 \in activeProcs : newPrioProc(id1, mem) \geq newPrioProc(id2, mem)\}
\end{array}
$$

Listing 4.20 – Définition de $procsExec()$ pour un ordonnancement avec héritage de priorité

Après avoir calculé la priorité temporaire d'un processus id, nous définissons dans le listing 4.20 la fonction $procsExec()$ qui retourne l'ensemble des processus prioritaires parmi les processus prêts et dont l'échéance n'est pas encore atteinte.

4.4 Autres alternatives de traduction proposées

Avant de mettre en place le modèle d'exécution décrit précédemment, plusieurs propositions ont été étudiées et discutées. Dans cette section, nous décrivons les idées de traduction proposées et leurs limitations afin de justifier les choix de traduction que nous avons adoptés.

4.4.1 Première proposition

Une première proposition consistait à assigner à chaque variable C globale et locale une variable TLA+ et à chaque fonction deux variables TLA+ : une qui contrôle le flot d'exécution des instructions contenue dans celle-ci et une deuxième variable qui stocke la valeur de retour de cette fonction. Chaque instruction possède un identifiant. Plus particulièrement, l'instruction `return` est identifiée avec une étiquette spéciale, égale à "done".

```
1 int x = 0;
2 void f() {
3   int a;
4   a = x;
5   x++;
6   return;
7 }
```

(a) Code C initial

$$
\begin{array}{l}
f(id) \triangleq \\
\quad \lor \land f_label[id] = \text{"f_4"} \\
\qquad \land f_a' = x \\
\qquad \land f_label' = [f_label \text{ EXCEPT } ![id] = \text{"f_5"}] \\
\qquad \land \text{UNCHANGED } \langle x, f_ret \rangle \\
\quad \lor \land f_label[id] = \text{"f_5"} \\
\qquad \land x' = x + 1 \\
\qquad \land f_label' = [f_label \text{ EXCEPT } ![id] = \text{"done"}] \\
\qquad \land \text{UNCHANGED } \langle f_a, f_ret \rangle \\
\quad \lor \land f_label[id] = \text{"done"} \\
\qquad \land \text{UNCHANGED } \langle x, f_label, f_a, f_ret \rangle \\
Spec \triangleq \\
\qquad \land x = 0 \land f_a = Undef \land f_ret = Undef \land f_label = (1 :> \text{"f_4"}) \\
\qquad \land \Box[f \lor (f_label = \text{"done"} \\
\qquad\qquad \land \text{UNCHANGED } \langle x, f_a, f_label, f_ret \rangle)]_{\langle x, f_a, f_label, f_ret \rangle}
\end{array}
$$

(b) Code TLA+

Listing 4.21 – Exemple de traduction d'une définition de fonction

Le listing 4.21a présente un code C contenant une fonction `f()` manipulant une variable globale `x` et une variable locale `a`. Nous supposons que cette fonction est exécutée par un processus ayant un identifiant égal à 1. Le listing 4.21b illustre une traduction TLA+ du code C. La traduction de la fonction `f()` est modélisée par l'opérateur $f()$. Cet opérateur consiste en la disjonction d'actions dont chacune modélise une instruction C. Chaque action met à jour f_label avec l'identifiant de l'instruction suivante.

Avec cette proposition, il est difficile de modéliser les pointeurs et les tableaux, car les variables C sont représentées directement par des variables TLA+. Pour pallier à ce problème, nous avons étudié une deuxième solution que nous décrivons dans la suite.

4.4.2 Deuxième proposition

Cette deuxième proposition consistait à associer aux variables référencées, pointeurs et tableaux, des adresses (constantes) dans TLA+, spécifiées par l'utilisateur. Il fallait ensuite stocker dans un tableau TLA+ (indexé par ces adresses) le contenu des tableaux C et les valeurs pointées par les pointeurs. Ce tableau est appelé $heap_int$.

```
1 p = &i;
2 *p = 4;
```

(a) Code C initial

$$\begin{array}{l} \lor \land main_label[id] = \text{"f_1"} \\ \quad \land p' = [p \text{ EXCEPT } ![id] = Addr_i] \\ \quad \land f_label' = [f_label \text{ EXCEPT } ![id]\text{"f_2"}] \\ \quad \ldots \\ \lor \land f_label[id] = \text{"f_2"} \\ \quad \land heap_int' = [heap_int \text{ EXCEPT } ![Addr_p] = 4] \\ \quad \ldots \end{array}$$

(b) Code TLA+

Listing 4.22 – Exemple de traduction d'instructions manipulant un pointeur

L'exemple donné par le listing 4.22 illustre deux instructions C (contenues dans une fonction `f()`) et leur traduction TLA+ avec la proposition énoncée. L'affectation du pointeur `p` est traduite en TLA+ par une affectation de la variable TLA+ p par la constante $Addr_i$. Le dé-référencement consiste à mettre à jour l'élément $heap_int[Addr_p]$ qui contient la valeur pointée par `p`. Mais le dé-référencement du pointeur devra affecter la valeur de la variable `i`, ce qui n'est pas le cas.

4.4.3 Troisième proposition

Pour résoudre le problème de dé-référencement posé précédemment, une solution consistait à associer à chaque pointeur un enregistrement contenant un identificateur v (sous forme de chaîne de caractères) de la variable vers laquelle le pointeur pointe ainsi qu'un déplacement $offs$ (entier) si le pointeur pointe vers un élément d'un tableau ou vers un champ d'une structure. Le dé-référencement consiste alors à parcourir toutes les variables référencées et déterminer quelle variable le champ $addr$ fait référence. La génération de la fonction de dé-référencement dans ce cas doit parcourir tout le programme C pour déterminer les variables qui sont référencées. De plus, cette fonction est compliquée à écrire.

En ce qui concerne l'appel de fonction, le listing 4.23 illustre une traduction d'un appel à la fonction `f()` depuis la fonction `main()`. La traduction consiste à tester la variable $f_label[id]$. Quand celle-ci est égale à "done", le contrôle d'exécution de la fonction `main()` est passé à l'action suivante (l'instruction suivant l'appel de fonction) et la variable f_label est initialisée avec la valeur "main_3". Sinon, les variables f_label et $main_label$ restent inchangées.

La difficulté qui se présente avec cette traduction concerne la spécification des variables inchangées. Pour illustrer ce problème, prenons l'exemple donné par le listing

$$
\begin{array}{l}
main(id) \triangleq \lor \land main_label[id] = \text{"main_1"} \\
\quad \land \text{IF } f_label[id] = \text{"done"} \\
\quad \text{THEN} \\
\qquad \land main_label' = [main_label \text{ EXCEPT } ![id] = \text{"main_2"}] \\
\qquad \land f_label' = [main_label \text{ EXCEPT } ![id] = \text{"f_1"}] \\
\qquad \land \text{UNCHANGED } \langle x, f_a, f_ret \rangle \\
\quad \text{ELSE} \\
\qquad \land \text{UNCHANGED } \langle main_label, x, f_a, f_ret \rangle \\
\qquad \land f(id) \\
\lor \land main_label[id] = \text{"main_2"} \\
\quad \ldots
\end{array}
$$

Listing 4.23 – Traduction d'un appel de fonction

```c
1  void h(){
2   ...
3  }
4
5  void g(){
6   h();
7   ...}
8
9  int f(){
10  g();
11  ...}
```

(a) Code C

$$
\begin{array}{l}
f(id) \triangleq \lor \land f_label[id] = \text{"f_10"} \\
\quad \land \text{IF } g_label[id] \neq \text{"done"} \\
\quad \text{THEN} \land f_label' = [f_label \text{ EXCEPT } ![id] = \text{"f_11"}] \\
\qquad \land g_label' = [g_label \text{ EXCEPT } ![id] = \text{"g_6"}] \\
\qquad \land \text{UNCHANGED } \langle x, h_label \rangle \\
\quad \text{ELSE} \land \text{UNCHANGED } \langle g_label, h_label \rangle \\
\qquad \land f(id) \\
\qquad \ldots
\end{array}
$$

(b) Code TLA+

Listing 4.24 – Exemple de traduction d'un appel de fonction

4.24a et sa traduction TLA+. Dans le code C, la fonction `f()` appelle une fonction `g()` qui elle fait appel à une fonction `h()`. La traduction de l'appel à la fonction `g()` spécifie que les variables inchangées sont x et h_label. Or, la fonction `g()` appelle `h()` et donc la variable h_label change de valeur. Donc, à un état donné, h_label change de valeur et reste inchangée en même temps. Ce qui mène à une contradiction.

Modéliser le flot de contrôle avec une variable pour chaque fonction nécessite une phase d'analyse de tous les appels de fonctions. Aussi, cette proposition ne permet pas de modéliser la récursivité et la liste des variables inchangées est trop longue, ce qui rend le code TLA+ illisible et rend la tâche de la traduction plus complexe.

Par conséquent, pour pouvoir modéliser les pointeurs, les tableaux et la récursivité, nous avons proposé une traduction qui est proche de la sémantique de C dans laquelle le modèle mémoire est proche du modèle concret du programme C. Le choix de la traduction adopté dans ce travail nous a permis d'appliquer une technique de réduction de l'espace d'états. Cette partie est détaillée dans le chapitre 5.

4.5 CONCLUSION

Dans ce chapitre, nous avons présenté une méthode pour modéliser des programmes C concurrents. Dans un premier temps, nous avons présenté un modèle mémoire basé sur une mémoire partagée par l'ensemble des processus composant le programme ainsi qu'un modèle de concurrence basé sur l'entrelacement. En se basant sur ces modèles, nous avons défini des extensions permettant de spécifier des exemples de primitives de synchronisation avec différents types d'ordonnancement ainsi que des contraintes temps-réel. Enfin, nous avons présenté dans la dernière section les propositions de traduction que nous avons étudiées et ensuite discutées. Cette étape nous a permis de déterminer les limitations de ces propositions et de justifier les choix de modélisation que nous avons proposés dans C2TLA+.

Les spécifications TLA+ générées par C2TLA+ sont vérifiées ensuite par TLC. Et comme la concurrence joue un rôle principal dans l'explosion combinatoire, nous proposons dans le chapitre suivant une technique de réduction de l'espace d'états que nous avons implémentée dans C2TLA+.

5

Une technique de réduction de l'espace d'états

Le *model-checking* se heurte au problème connu sous le nom de l'explosion combinatoire du nombre d'états (« *state explosion* » en anglais). Ce problème est dû à la taille des systèmes à vérifier, en particulier en présence de la concurrence et d'entrelacement. Par exemple, pour un système composé de n processus indépendants dont chaque processus est représenté par p états, la taille de l'espace d'états est de l'ordre de p^n états. Cette taille augmente alors exponentiellement en fonction du nombre de processus, limitant les perspectives d'utilisation du *model-checking*.

Face à ce constat, plusieurs techniques ont été proposées pour pallier ce problème. Ces techniques ne permettent pas de résoudre complètement le problème, mais elles contribuent significativement à réduire la complexité de la vérification.

Dans ce chapitre, nous commençons dans la section 5.1 par dresser un état de l'art des techniques de réduction de l'espace d'états. Puisque nous nous intéressons à la vérification de l'espace d'états généré par une spécification TLA+, nous décrivons dans la section 5.2 le principe de réduction que nous proposons pour générer une spécification TLA+ réduite et nous expliquons ensuite comment cette technique est intégrée lors de la traduction des programmes C. Afin d'évaluer cette technique, nous présentons dans la section 5.3 les résultats obtenus sur un ensemble de benchmarks. Finalement, nous exposons dans la section 5.4, les alternatives que nous avons étudiées pour implémenter

la réduction dans C2TLA+.

5.1 État de l'art des techniques de réduction de l'espace d'états

Nous regroupons dans cette section un état de l'art (non exhaustif) des techniques de réduction utilisées pour pallier le problème de l'explosion combinatoire. Nous commençons par celles qui s'appliquent dans la phase amont de l'exploration de l'espace d'états, à savoir l'abstraction et les techniques structurelles. Nous poursuivons ensuite avec les techniques intermédiaires, qui incluent les ordres partiels et la réduction par symétrie. Enfin, nous exposons les techniques qui s'appliquent en phase aval de l'exploration de l'espace d'états, à savoir les techniques de représentation efficace.

5.1.1 Technique par abstraction

L'abstraction est une technique qui s'applique sur le modèle du système et consiste à générer une version abstraite du modèle, à savoir un modèle plus simple. Plusieurs types d'abstractions existent, souvent basées sur l'interprétation abstraite [CC92] (introduite dans la section 1.1.3.3).

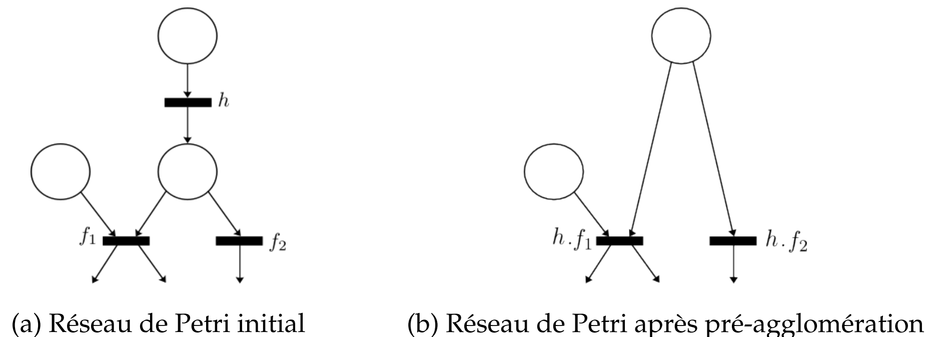
La construction d'un modèle abstrait peut être manuelle ou automatique. L'idée principale réside sur le fait que la propriété à vérifier doit être préservée par l'abstraction [LGS+95]. Les outils comme BLAST et SLAM implémentent l'approche CEGAR et reposent sur une phase de construction automatique d'un modèle abstrait (voir section 1.2.2) basées sur l'utilisation de l'abstraction des prédicats. D'autres méthodes implémentent des techniques d'interprétation abstraite pour construire une abstraction.

5.1.2 Techniques structurelles

Les techniques de réduction structurelles permettent de raisonner sur la structure d'un modèle. Elles s'appliquent avant l'exploration de l'espace d'états du modèle et consistent à transformer un modèle en un modèle équivalent tout en préservant un ensemble de propriétés. Parmi les techniques structurelles les plus connues, nous présentons dans cette section celle basée sur l'*agglomération des transitions* et la *réduction de places implicites*, dans le cadre des réseaux de Petri.

(a) Agglomération des transitions. Cette technique a été initialement proposée dans le cadre des réseaux de Petri par Berthelot [Ber85]. Étant donné une place, l'agglomération de transitions consiste à agglomérer les transitions qui produisent dans la place avec les transitions qui consomment dans la place. La définition d'une telle réduction repose sur trois éléments clés : les conditions d'application, la transformation qui génère le réseau de Petri réduit et les propriétés préservées par cette réduction.

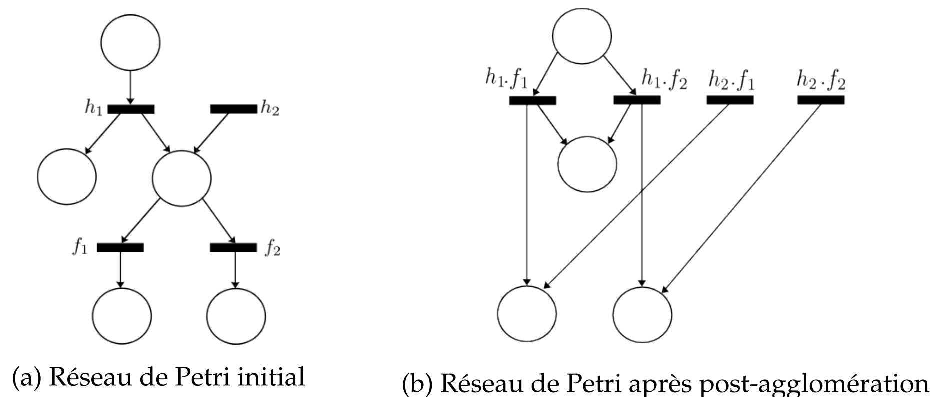
Berthelot a défini [Ber85] un ensemble de réductions dont les deux les plus connues sont : la *pré-* et la *post-agglomération de transitions*.



(a) Réseau de Petri initial (b) Réseau de Petri après pré-agglomération

FIGURE 5.1 – Exemple d'application d'une pré-agglomération d'un réseau de Petri

Dans un réseau de Petri, pour deux ensembles de transitions H et F. La pré-agglomération, sous réserve d'un ensemble de conditions, consiste à retarder le franchissement de $h \in H$ jusqu'au franchissement de $f \in F$. La figure 5.1 illustre un exemple d'application d'une pré-agglomération sur un réseau de Petri. Dans le réseau de Petri pré-aggloméré (figure 5.1b), le franchissement de $h.f_1$ et $h.f_2$ se fait de façon atomique.



(a) Réseau de Petri initial (b) Réseau de Petri après post-agglomération

FIGURE 5.2 – Exemple d'application d'une post-agglomération d'un réseau de Petri

Pour la post-agglomération, le principe repose sur le fait qu'on peut avancer le franchissement de toute transition $f \in F$ dès que l'on a franchi une transition $h \in H$, si certaines conditions sont vérifiées. Nous illustrons dans la figure 5.2 un exemple d'application d'une post-agglomération sur un réseau de Petri. Dans le réseau Petri pré-aggloméré (figure 5.2b), les transitions f_1 et f_2 sont post-agglomérables.

Dans les deux cas, le franchissement de la séquence de transitions $h.f$ entraîne la suppression d'un état intermédiaire. Ce qui permet de réduire la taille du graphe des marquages accessibles dans le réseau de Petri. L'application des réductions de Berthelot repose sur la définition d'un ensemble de conditions locales structurelles sur

le modèle. Ces conditions garantissent que les agglomérations sont correctes, c'est-à-dire les propriétés à vérifier sont préservées.

Il a été démontré dans [PPP00] que les pré- et les post-agglomérations définies par Berthelot préservent des propriétés $LTL_{\setminus X}$ (LTL sans l'opérateur « suivant ») sous certaines conditions. Dans [HP06], des conditions suffisantes plus fines pour les pré- et post-agglomérations ont été définies afin d'avoir plus de réductions dans les réseaux de Petri. À titre d'exemple, citons l'outil Helena [Eva05] qui a implémenté des réductions à base d'agglomérations de transitions dans les réseaux de Petri.

(b) Réduction des places implicites. D'autres types de réductions ont été proposées dans la littérature, par exemple la réduction des places implicites qui consiste à éliminer une place dont le marquage n'a pas d'incidence sur le franchissement des transitions d'un réseau de Petri [Ber85]. Une telle place peut apparaître à la suite d'application d'autres réductions et rendre ensuite elle-même de nouvelles réductions applicables.

5.1.3 Techniques intermédiaires

Cette classe regroupe les techniques de réduction dynamiques qui sont réalisées lors de l'exploration de l'espace d'états. Nous présentons dans cette classe deux techniques : la réduction d'ordre partiel et celle par symétrie.

(a) La réduction d'ordre partiel. Dans un système concurrent, les entrelacements possibles des actions des processus peuvent comporter des scénarios équivalents. La technique de réduction d'ordre partiel exploite la commutativité des actions concurrentes en ignorant lors de l'exploration de l'espace d'états les transitions qui génèrent des scénarios équivalents. Ce sous-ensemble de scénarios constitue une représentation partielle du système, qui doit être suffisante pour vérifier la propriété d'intérêt sur le système.

Cette technique est aussi appelée *model-checking d'ensembles représentatifs* (« *model-checking using representatives* » en anglais). Elle a été initialement proposée pour les réseaux de Petri et elle consiste à parcourir l'espace d'états en calculant pour chaque état visité un sous-ensemble T' des transitions exécutables, et à explorer seulement les transitions contenues dans T'. Les transitions qui n'appartiennent pas à T' ne sont pas explorées. Le calcul de l'ensemble T' repose sur l'*indépendance* des transitions. Deux transitions t_1 et t_2 sont dites indépendantes dans un état s si l'exécution de l'une des transitions ne désactive par l'autre et si les transitions sont commutatives et mènent vers un même état.

Le calcul de l'ensemble T' des transitions à exécuter peut se faire selon plusieurs techniques. Nous présentons dans ce qui suit, deux techniques principales : la technique des ensembles têtus (« *stubborn set* » en anglais) et celles des ensembles dormants (« *sleep set* » en anglais).

- La technique de « *stubborn set* » a été développée par Valmari [Val89]. Elle consiste à

calculer l'ensemble des transitions exécutables de l'état s, appelé l'ensemble têtu T'_s. L'ensemble T'_s est dit têtu à l'état s si et seulement si toutes les séquences d'exécution partant de s et ne passant pas par une transition de T'_s sont des séquences composées que de transitions indépendantes de T'_s.
Plusieurs algorithmes ont été proposés pour calculer à chaque état l'ensemble de transitions têtues. Certains de ces derniers se basent sur des heuristiques.

- La technique de « *sleep set* » a été développée par Godefroid [God91]. Elle consiste à construire pour chaque état s un ensemble contenant les transitions exécutables depuis cet état mais qui ne seront pas exécutées. Cet ensemble est appelé l'*ensemble dormant* de l'état s. Cette technique ne réduit pas le nombre d'états à visiter, elle permet plutôt d'éviter des transitions qui mènent à des états déjà visités.

La technique des ensembles têtus tente de vérifier les conflits qui pourraient survenir entre les transitions du système alors que celle des ensembles dormants se base sur les informations collectées, à savoir les transitions déjà exécutées lors de l'exploration de l'espace d'états. La technique des ensembles têtus est utile pour vérifier des propriétés de terminaison. Néanmoins, cette technique, comme l'a décrit Valmari [Val91], fait face à un problème d'*ignorance* (« *ignoring problem* » en anglais) quand la propriété à vérifier est autre que la terminaison. Ce problème survient quand une transition t n'est jamais choisie pour s'exécuter alors que la propriété porte sur les états atteignables depuis t. Ce problème a été résolu par Godefroid [God96] qui a défini un ensemble de conditions supplémentaires, appelé « *proviso* » qui assure l'équité d'exécution d'une transition.

Dans [CGMP99], les auteurs ont montré que les propriétés $\text{LTL}_{\setminus X}$ (la logique LTL sans l'opérateur « suivant ») sont préservées en appliquant les techniques des ensembles têtus et dormants. Ces dernières ont été implémentées dans le *model-checker* SPIN. D'autre techniques de réduction d'ordre partiel ont été proposées, par exemple la technique *des pas couvrants* [VAM96].

(b) La réduction par symétrie. Cette technique a été initialement exploitée pour la vérification des réseaux de Petri de haut niveau [HJJJ86, CDFH97]. Elle a été proposée suite à la constatation suivante : certains systèmes concurrents sont la composition parallèle de n processus ayant un même comportement. Il convient alors de détecter les comportements équivalents qui ne différent que par une permutation des états/transitions et de n'explorer qu'un seul état parmi l'ensemble des états équivalents. Cet ensemble constitue une *classe d'équivalence*. Le graphe construit est appelé *graphe quotient* et l'exploration des états consiste à visiter un état par classe. Ainsi, l'outil TOPSpin [DM06] permet de détecter automatiquement des symétries dans des modèles Promela.

5.1.4 Techniques basées sur la représentation des espaces états

Ces techniques permettent la gestion de l'espace d'états lors de la construction de celui-ci. Elles peuvent s'appliquer sur n'importe quel type de modèle : automate, réseau

de Petri ou algèbre de processus. Nous décrivons ici deux approches : symbolique et par compression.

(a) Approche symbolique. L'approche symbolique a été introduite par McMillan [McM92] en 1992 et consiste à utiliser une représentation symbolique des états, dont la plus connue est le *diagramme de décision binaire* ou BDD (« *Binary Decision Diagram* » en anglais) introduit par Bryant en 1986 [Bry86].

Un BDD permet de représenter des fonctions booléennes. Il dispose d'un seul nœud initial et des nœuds terminaux marqués par une valeur booléenne 0 ou 1 et des nœuds non terminaux marqués par des variables booléennes. En appliquant un ordre dans l'occurrence des variables et des règles de réduction sur un BDD, Bryant a défini un BDD réduit et ordonné, appelé ROBDD. Nous utilisons dans la suite, l'appellation BDD pour désigner un ROBDD. La compacité de la représentation de la fonction booléenne dépend fortement de l'ordre choisi des variables. Cette compacité permet à des algorithmes de manipuler efficacement les BDD.

Dans le *model-checking* symbolique, l'ensemble d'états et les transitions sont représentés par des BDD. La satisfaisabilité d'une formule sur un modèle se ramène à des opérations sur ces BDD. L'exploration de l'espace d'états se fait en visitant un ensemble d'états simultanément. Contrairement à l'approche explicite où l'exploration de l'espace d'états se fait état par état.

L'utilisation des approches symboliques a contribué dans la vérification des systèmes complexes et pour la première fois, des espaces d'états allant jusqu'à 10^{20} ont été vérifiés [BCM+92]. L'approche symbolique a été fortement utilisée dans le domaine de la vérification des circuits matériels. Le plus ancien des *model-checker* symboliques est SMV [McM93]. On trouve aussi NuSMV [CCGR99] et son successeur nuXmv [CCD+14].

(b) Compression mémoire. Quand le *model-checking* génère un grand espace d'états, les limites physiques de la mémoire sont vite atteintes. Face à ce problème, des techniques probabilistes basées sur des fonctions de hachage ont été explorées.

La technique « *hash compaction* » a été proposée par Wolper et Leroy [WL93]. Au lieu de stocker un descripteur de l'état (contenant les valeurs des variables en cet état), cette technique utilise une fonction de *hachage* pour calculer un *hash* qui représente une compression du descripteur de l'état. Ce hash est stocké dans une table de hachage T_h.

La technique de « *supertrace* » (appelée aussi « *bitstate hashing* ») a été introduite par Holzmann [Hol87] pour son *model-checker* SPIN. Cette technique consiste à utiliser une table de hachage T_b dans laquelle chaque état est codé par un bit. Initialement, les bits de T_b sont positionnés à 0. Lors de l'exploration d'un nouvel état, un hash est calculé pour cet état qui représente sa position dans la table T_b. Cette position est ensuite marquée par 1, indiquant que l'état a été visité.
Cette technique permet d'accélérer la recherche d'états lors de la phase de comparaison pour la compression des états. Mais comme la fonction de hachage

n'est pas injective, deux états différents peuvent avoir la même valeur du hash. Dans ce cas, on risque de se retrouver dans une situation de collision dans laquelle deux états sont confondus et par conséquent une partie de l'espace d'états ne sera pas explorée. Pour réduire le risque de collision, une amélioration a été apportée par Holzmann et est appelé *k-fold bitstate hashing*. Cette amélioration consiste à utiliser k fonctions de hachage différentes [Hol96]. Un état est considéré comme *visité* si les positions retournées par les k fonctions dans la table T_b sont marquées.
Dans cette technique, la taille de la table T_b est constante alors que dans la technique « *hash compaction* » T_h est proportionnelle au nombre d'états.

Une autre forme de compression appelée « *collapse* » (ou « *state collapsing* ») est utilisée par SPIN [Hol97]. Cette dernière part du fait que deux états peuvent contenir des informations identiques, c'est-à-dire un sous-ensemble de variables dont les valeurs sont identiques. Pour éviter la duplication de ces parties identiques dans la mémoire, il convient de ne les stocker qu'une seule fois et d'utiliser des références vers ces parties.

D'autres techniques ont été mises en œuvre pour mieux améliorer l'efficacité de stockage des états lors du *model-checking*. Citons par exemple la technique « *state space caching* » [GHP95]. Une étude des techniques probabilistes pour la compression des états est donnée dans [KL04].

5.1.5 Autres techniques

Plusieurs autres techniques ont été proposées dans la littérature. Citons par exemple la technique de minimisation par bisimulation (« *bisimulation minimization* » en anglais) qui consiste à générer l'automate bissimilaire minimal à partir de l'automate initial du modèle [LY92]. Cette technique est utile pour les modèles compositionnels où elle opére avant composition.

Une autre technique de réduction est appelée « *program slicing* » [Wei84]. Elle consiste à retirer du programme (ou du modèle) les parties qui ne sont pas pertinentes par rapport à la propriété à vérifier.

5.1.6 Discussion

Chaque technique de réduction présente des avantages et des inconvénients. L'efficacité d'une technique dépend de la nature du système à vérifier. Par exemple, les techniques de réduction par ordre partiel exploitent l'entrelacement des actions des processus pour réduire l'espace d'états. Elles ne se sont pas efficaces sur les systèmes synchrones à la différence de l'approche symbolique qui s'applique aussi bien sur les systèmes synchrones qu'asynchrones.

Il est à noter que toutes les techniques décrites précédemment sont souvent complémentaires et peuvent se combiner entre elles afin de mieux réduire l'espace

d'états. Citons à titre d'exemple, le *model-checker* Murϕ [Dil96] qui combine la réduction par symétries et la compression *hash compaction*.

5.2 APPROCHE DE RÉDUCTION PROPOSÉE

Nous présentons dans cette section le schéma général de la technique de réduction que nous appliquons sur un programme C. Nous décrivons ensuite le principe de cette réduction et nous détaillons par la suite comment appliquer cette réduction sur chaque type d'instruction C. Enfin, nous décrivons les propriétés préservées par cette réduction.

5.2.1 Schéma général

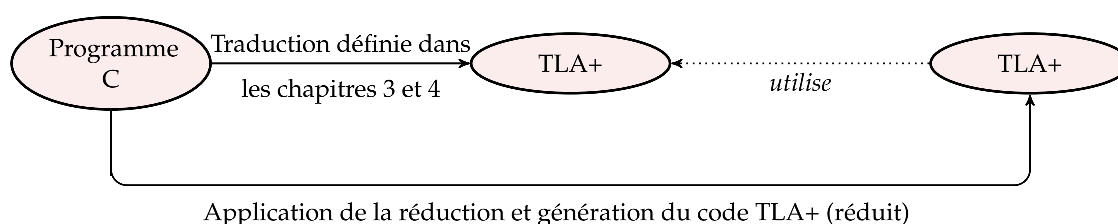


FIGURE 5.3 – Processus de génération d'une spécification TLA+ réduite

La processus de réduction que nous appliquons sur un programme C est illustré par la figure 5.3. La réduction proposée est proche de la technique d'agglomération des transitions des réseaux de Petri décrite dans la section 5.1.2. La différence qui réside entre les deux est que la technique que nous présentons est opérée en amont lors de la génération du code TLA+ à partir du C. L'application de la réduction n'affecte pas les règles de traduction définies dans le chapitre 4. Elle correspond alors à une extension qui utilise le code TLA+ généré par la traduction définie dans les chapitres 3 et 4. Puisque notre réduction s'applique sur le code C, nous appelons cette réduction *agglomération des instructions*.

5.2.2 Principe de réduction

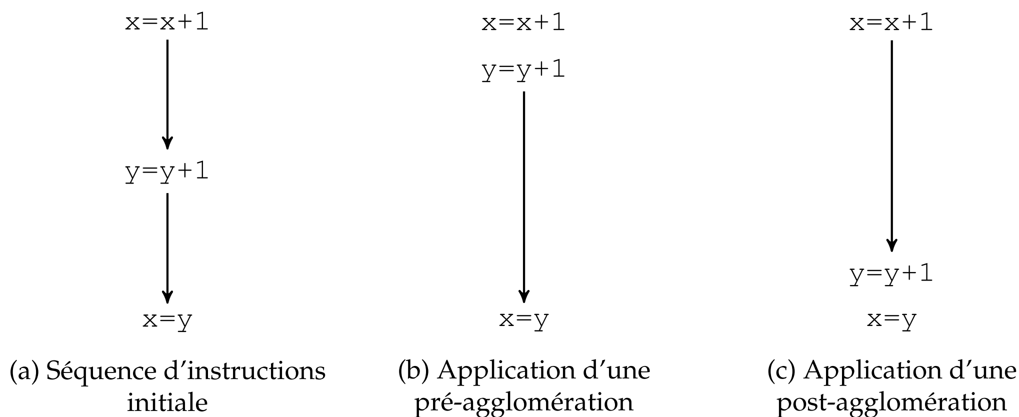
La réduction consiste à rendre atomique l'exécution de certaines instructions C lors de la génération de la spécification TLA+. La nouvelle spécification TLA+ générée après application de la réduction est une spécification dont l'espace d'états est réduit. Cette réduction est caractérisée par les éléments suivants :

- les conditions d'agglomération, qui définissent les cas où la réduction par agglomération peut être appliquée ;
- la transformation f du STE $\mathcal{T}$ de la spécification TLA+ (générée à partir du C) en un STE réduit $\mathcal{T}_r$ tel que $\mathcal{T}_r = f(\mathcal{T})$;

- les propriétés préservées par la réduction.

L'ensemble de conditions caractérisant une transformation par agglomération définit ce qu'on appelle un *prédicat d'agglomération*.

Définition 5.2.1. *Un prédicat d'agglomération $p()$ prend en argument un programme C Prg et une instruction $inst$ et renvoie* vrai *si $inst$ ne fait intervenir ni une lecture ni une écriture dans une variable partagée entre les processus et faux sinon.*



(a) Séquence d'instructions initiale (b) Application d'une pré-agglomération (c) Application d'une post-agglomération

FIGURE 5.4 – Application d'une pré et post-agglomération

Pour deux instructions consécutives, une pré-agglomération consiste à agglomérer la deuxième instruction avec la première. Une post-agglomération consiste à agglomérer la première instruction avec la deuxième. Soit l'exemple d'instructions C donné par la figure 5.4a, dans lequel `x` est une variable partagée et `y` est une variable locale. Comme l'instruction `y=y+1` ne fait intervenir que la variable locale `y`, elle est alors agglomérable. La pré-agglomération de l'instruction `y=y+1`, illustrée par la figure 5.4b, consiste à avancer l'exécution de cette instruction avec l'instruction `x=x+1`. La figure 5.4c illustre une post-agglomération de l'instruction `y=y+1`. Dans ce cas, l'exécution de cette instruction est retardée avec l'instruction suivante, à savoir l'instruction `x=y`.

Dans notre cas, comme la réduction est appliquée sur chaque processus à part, le fait d'appliquer une pré- ou une post-agglomération ne change pas le résultat de la réduction. Nous avons fait le choix d'appliquer des post-agglomérations. Ce choix est arbitraire. Pour un flot d'exécution séquentiel, agglomérer une instruction C consiste à retarder son exécution à l'instruction suivante.

Proposition 5.2.1. *Le test du prédicat d'agglomération $p()$ sur une instruction C est un problème indécidable.*

Démonstration. Pour démontrer l'indécidabilité du test du prédicat $p()$, nous considérons le fragment de code suivant :

```
1    for (i = 0; i<= 1; i++)
```

```
2   {
3     if (i)
4       { f(); p = &g;}
5     else p = &l;
6     *p = 3;
7   }
```

Dans ce code, on dispose d’une variable partagée g, une variable locale l, un pointeur local p et une fonction f(). La question consiste à déterminer si l’instruction de la ligne 6 est agglomérable ou pas. La décidabilité que cette instruction est agglomérable dépend de la terminaison de f(). Étant donné que la terminaison d’un programme est un problème indécidable [Tur36], le test d’agglomération sur une instruction C est aussi indécidable. □

Comme le test d’agglomérabilité d’une instruction est un problème indécidable, nous proposons un prédicat $q()$ basé sur des conditions suffisantes, tel que pour un programme C Prg et une instruction $inst$ on a $q(Prg, instr) \Rightarrow p(Prg, instr)$. Dans ce cas, l’ensemble des instructions C agglomérables par le prédicat $q()$ est un sous ensemble des instructions C agglomérables par $p()$. Nous proposons un prédicat basé sur une analyse syntaxique dans laquelle nous considérons comme agglomérables les instructions suivantes :

1. Toute instruction manipulant des variables locales non référencées.

2. Toute instruction ne modifiant que le contrôle de flot du processus, à savoir les instructions goto, break et continue.

Le choix de ces conditions revient au fait que ces instructions s’exécutent indépendamment des données partagées par les processus. Nous avons introduit la condition d’agglomérer les variables locales non référencées car une variable locale peut pointer vers une variable partagée et modifier ensuite sa valeur. Nous illustrons ce problème dans l’exemple de code C qui suit.

Exemple 5.2.1. *Soit le code C suivant :*

```
1   void f(){
2     int *l;
3     l = &g;
4     *l= 12;
5   }
```

La variable g *est partagée par plusieurs processus et* l *est une variable locale. L’instruction de la ligne 3 fait pointer* l *vers* g *et l’instruction de la ligne 4 modifie la valeur pointée par* l*, donc la valeur de* g *passe à* 12.

5.2.3 Application de la réduction

Nous rappelons que la traduction réduite est une surcouche qui se base sur la traduction TLA+ générée selon les règles définies dans le chapitre 3. Pour un programme

C concurrent composé de n processus P_i, la réduction consiste à agglomérer le code C appelé par chaque processus avant la composition asynchrone de ces derniers. Le principe de réduction consiste à varier la granularité d'exécution des instructions. Au lieu de modéliser l'exécution d'une instruction par une action (comme le cas dans la traduction du chapitre 3), nous modélisons l'exécution d'un bloc d'instructions. Pour appliquer ce principe dans TLA+, nous attribuons pour chaque instruction C <s : stmt> un opérateur récursif TLA+ *stmt_*⟦<s : stmt>⟧$_{sid}$*_long()*. Quand une instruction C <s : stmt> est agglomérable, l'opérateur TLA+ correspondant *stmt_*⟦<s : stmt>⟧$_{sid}$*_long()* appelle l'opérateur de l'instruction suivante (selon le GFC) en lui passant en argument l'état de la mémoire après exécution de l'instruction <s : stmt>. Dans le cas où il n'est pas possible de déterminer quelle est l'instruction suivante, *stmt_*⟦<s : stmt>⟧$_{sid}$*_long()* appelle l'opérateur *dispatch_red()* qui gère le flot de contrôle du programme C aggloméré. Il est à noter que les appels aux opérateurs récursifs sont terminaux. Mais si l'agglomération est appliquée sur un programme séquentiel contenant une boucle infinie, l'appel infini aux opérateurs récursifs est détecté par TLC. Un exemple de code C illustrant ce cas est présenté dans la section 5.3.

Dans la suite de cette section, nous détaillons les étapes de génération d'une spécification réduite à partir d'un programme C.

5.2.3.1 Principe général

Dans la suite de cette section, nous notons **p_aggl()** la fonction OCaml qui implémente le prédicat d'agglomération $q()$ pour un programme C donné en entrée à C2TLA+. Ce dernier prend en argument une instruction <s : stmt> à partir du GFC de la fonction appelée par le processus P_i et retourne vrai si <s : stmt> est agglomérable et faux sinon. Pour simplifier l'écriture des règles, nous employons la notation ⟦<s : stmt>⟧$_{stmt'}$ pour désigner la traduction réduite d'une instruction <s : stmt>.

La traduction réduite pour un programme C s'écrivant sous la forme d'une suite de déclarations <d : decls> et de définitions de fonctions <f$_1$: fun_def>...<f$_k$: fun_def> est donnée par la règle 5.1. Cette règle génère le module *Program_red*, qui étend le module *Program* résultant de la traduction des chapitres 3 et 4. Le module définit ensuite les opérateurs TLA+ récursifs que nous expliquons dans ce qui suit. La règle 5.1 consiste ensuite en les étapes suivantes :

1. Parcourir le GFC de chaque fonction <f$_i$: fun_def> appelée par le processus P_i et appliquer la règle ⟦ <f$_i$: fun_def>⟧$_{fun_def'}$. Cette règle permet de parcourir chaque instruction <s : stmt> de la fonction <f$_i$: fun_def> et génère un opérateur de la forme *stmt_*⟦ <s : stmt>⟧$_{sid}$ *_long()* dont le principe est détaillé dans la section 5.2.3.5.

2. Générer un opérateur appelé *dispatch_red()* qui modélise le contrôle de flot du programme C après réduction (détaillé dans la section 5.2.3.4).

3. Générer la formule *Spec_red* qui définit le comportement du programme après réduction (détaillé dans la section 5.2.3.3).

$[\![$<d : decls><f$_1$: fun_def>. . .<f$_k$: fun_def>$]\!]_{prg'}$ $\rightarrow$

```
──────────────── MODULE Program_red ────────────────
EXTENDS Program
RECURSIVE ⟦ <s1: stmt>⟧rec . . . ⟦ <sn: stmt>⟧rec . . . ⟦ <sh: stmt>⟧rec . . . ⟦ <sm: stmt>⟧rec

⟦ <f1: fun_def> : := <type> FUN_ID (<params>) { <decls><s1: stmt>. . .<sn: stmt> } ⟧fun_def'
...
⟦ <fk: fun_def> : :=<type> FUN_ID (<params>) { <decls><sh: stmt>. . .<sm: stmt> } ⟧fun_def'

dispatch_red(id, mem) ≜
  CASE mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<f1: fun_def>⟧fid lbl (<s1: stmt>)"
            → stmt_⟦<s1: stmt>⟧sid_long(id, mem)
       . . .
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<f1: fun_def>⟧fid lbl (<sn: stmt>)"
            → stmt_⟦<sn: stmt>⟧sid_long(id, mem)
       . . .
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<fk: fun_def>⟧fid lbl (<sh: stmt>)"
            → stmt_⟦<sh: stmt>⟧sid_long(id, mem)
       . . .
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "⟦<fk: fun_def>⟧fid lbl (<sm: stmt>)"
            → stmt_⟦<sm: stmt>⟧sid_long(id, mem)
    □ mem.register[id] = ⟨⟩ → mem
    □ OTHER → "error"


ProcsExec(mem) ≜ ProcSet

Next_red ≜ ∨ ∃ id ∈ ProcSet :
               ∧ memory.register[id] ≠ ⟨⟩
               ∧ memory' = dispatch_red(id, memory)
           ∨ ∀ id ∈ ProcSet :
               ∧ memory.register[id] = ⟨⟩
               ∧ UNCHANGED memory

Spec_red ≜ Init ∧ □[Next_red]memroy ∧ WF⟨memory⟩(Next_red)
────────────────────────────────────────────────────
```

Règle 5.1 – Traduction réduite d'un programme C

5.2.3.2 Illustration

Nous reprenons dans cette section l'exemple de code C donné par la figure 4.2a. Après application de toutes les règles de traduction réduite, le code TLA+ résultant est donné par le module *Program_red* (voir listing 5.1).

Le module généré porte le nom de *Program_red*. Il étend le module *Program* dont la définition est donnée dans l'annexe A. Il déclare ensuite les opérateurs TLA+ récursifs. Les instructions de la fonction `inc()` sont tous agglomérables. Pour la fonction `p1()`, l'appel à la fonction `inc(a)` et l'instruction `return` sont agglomérables. Pour la fonction `p2()`, c'est seulement l'instruction `return` qui est agglomérable.

La spécification réduite est donnée par la formule *Spec_red*. Elle modélise le

```
MODULE Program_red
EXTENDS Program
RECURSIVE dispatch_red(_, _), stmt_inc_5_long(_, _), stmt_inc_6_long(_, _), stmt_p1_10_long(_, _),
        stmt_p1_11_pre_long(_, _), stmt_p1_11_post_long(_, _), stmt_p1_12_long(_, _),
        stmt_p1_13_long(_, _), stmt_p2_16_long(_, _), stmt_p2_17_long(_, _), p2_stmt_18_long(_, _)
Generating operators for inc() function
tmp = i + 1 ;
stmt_inc_5_long(id, mem) ≜ stmt_inc_6_long(id, stmt_inc_5(id, mem))
return (tmp) ;
stmt_inc_6_long(id, mem) ≜ dispatch_red(id, stmt_inc_6(id, mem))

Generating operators for p1() function
a = 1 ;
stmt_p1_10_long(id, mem) ≜ stmt_p1_11_pre_long(id, stmt_p1_10(id, mem))
x = inc(a) ;
stmt_p1_11_pre_long(id, mem) ≜ dispatch_red(id, stmt_p1_11_pre(id, mem)
stmt_p1_11_post_long(id, mem) ≜ stmt_p1_11_post(id, mem))
x ++ ;
stmt_p1_12_long(id, mem) ≜ stmt_p1_12(id, mem)
return ;
stmt_p1_13_long(id, mem) ≜ dispatch_red(id, stmt_p1_13(id, mem))

Generating operators for p2() function
b = 0 ;
stmt_p2_16_long(id, mem) ≜ stmt_p2_17_long(id, stmt_p2_16(id, mem))
x += b ;
stmt_p2_17_long(id, mem) ≜ stmt_p2_17(id, mem)
return ;
p2_stmt_18_long(id, mem) ≜ dispatch_red(id, stmt_p2_18(id, mem))

dispatch_red(id, mem) ≜
  CASE mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "inc_5" → stmt_inc_5_long(id, mem)
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "inc_6" → stmt_inc_6_long(id, mem)
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p1_10" → stmt_p1_10_long(id, mem)
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p1_11_pre" →
          stmt_p1_11_pre_long(id, mem)
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p1_11_post" →
          stmt_p1_11_post_long(id, mem)
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p1_12" → stmt_p1_12_long(id, mem)
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p1_13" → stmt_p1_13_long(id, mem)
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p2_16" → stmt_p2_16_long(id, mem)
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p2_17" → stmt_p2_17_long(id, mem)
    □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p2_18" → stmt_p2_18_long(id, mem)
    □ mem.register[id] = ⟨⟩ → mem
    □ OTHER → "error"

ProcsExec(mem) ≜ ProcSet

Next_red ≜ ∨ ∃ id ∈ ProcsExec(mem) :
              ∧ mem.register[id] ≠ ⟨⟩
              ∧ mem' = dispatch_red(id, mem)
           ∨ ∀ id ∈ ProcsExec(mem) :
              ∧ mem.register[id] = ⟨⟩
              ∧ UNCHANGED memory
Spec_red ≜ Init ∧ □[Next_red]_memory ∧ WF_memory(Next_red)
```

Listing 5.1 – Spécification réduite d'un exemple de code C

comportement du programme C après réduction. L'action $Next_red$ modélise les transitions possibles du programme. Elle est définie par l'entrelacement des actions des deux processus. Quand l'une des piles de registre est non vide, l'appel à $dispatch_red()$ met à jour la variable $memory$. Cette mise à jour simule l'exécution d'une instruction ou plusieurs instructions d'un processus. Quand la pile de tous les processus est vide, la variable $memory$ reste inchangée. La spécification $Spec_red$ génère un espace d'états de 19 états et 12 états différents. Alors que la spécification $Spec$ du module $Program$ génère 54 états et 32 états différents.

Dans ce qui suit, nous détaillons les étapes de génération d'une spécification TLA+ réduite.

5.2.3.3 Génération de la formule $Spec_red$

Nous générons la formule $Spec_red$, qui définit le comportement des processus après application de la réduction sur un programme C donné en entrée à C2TLA+. Sa définition fait appelle au prédicat $Init$ et à l'action $Next_red$:

- le prédicat $Init$ est défini tel que décrit dans la section 4.1.1.3,
- l'action $Next_red$ consiste à appeler l'opérateur $dispatch_red()$ au lieu de $dispatch()$. La définition de $dispatch_red()$ est détaillée dans la section suivante.

5.2.3.4 Génération de l'opérateur $dispatch_red()$

La gestion du flot de contrôle d'une spécification réduite est gérée par l'opérateur $dispatch_red()$. Cet opérateur appelle pour chaque instruction <s_i : stmt> un opérateur noté $stmt_[\![\text{<s}_\text{i}\text{ : stmt>}]\!]_{sid}_long(id, mem)$ dont la définition est détaillée dans la section suivante. Nous décrivons le principe de cet opérateur à travers l'exemple qui suit.

Exemple 5.2.2. *Pour l'exemple du code C donné par la figure 4.2a, la définition de l'opérateur* $dispatch_red()$ *est la suivante :*

$$
\begin{array}{l}
dispatch_red(id, mem) \triangleq \\
\quad \text{CASE } mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"inc_5"} \rightarrow stmt_inc_5_long(id, mem) \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"inc_6"} \rightarrow stmt_inc_6_long(id, mem) \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p1_10"} \rightarrow stmt_p1_10_long(id, mem) \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p1_11_pre"} \rightarrow \\
\qquad\qquad stmt_p1_11_pre_long(id, mem) \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p1_11_post"} \rightarrow \\
\qquad\qquad stmt_p1_11_post_long(id, mem) \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p1_12"} \rightarrow stmt_p1_12_long(id, mem) \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p1_13"} \rightarrow stmt_p1_13_long(id, mem) \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p2_16"} \rightarrow stmt_p2_16_long(id, mem) \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p2_17"} \rightarrow stmt_p2_17_long(id, mem) \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{"p2_18"} \rightarrow stmt_p2_18_long(id, mem) \\
\qquad \Box\ mem.register[id] = \langle\rangle \rightarrow mem \\
\qquad \Box\ \text{OTHER} \rightarrow \text{"error"}
\end{array}
$$

L'opérateur $dispatch_red()$ *gère le flot de contrôle de la spécification TLA+ réduite. Il teste les*

valeurs possibles que peut prendre le champ pc contenu dans le sommet de la pile du processus id et en fonction de cette valeur, il appelle l'opérateur de l'instruction C correspondante (l'opérateur que nous définissons dans la section suivante). Quand le registre $mem.register[id]$ du processus courant est vide, il retourne l'état de la mémoire qui lui est passé en argument.

5.2.3.5 Génération des opérateurs pour les instructions

Dans la suite, nous utilisons une structure **if/then/else** pour désigner un choix conditionnel. Par exemple, la règle suivante :
if p_aggl(<s:stmt>**)**
then $expr_1$
else $expr_2$
permet de générer l'expression TLA+ $expr_1$ si le prédicat **p_aggl()** retourne vrai et $expr_2$ sinon. Dans ce qui suit nous décrivons les règles de traduction réduite pour chaque type d'instruction.

Nous rappelons que dans la traduction du chapitre 3, chaque instruction <s:stmt> du programme C est traduite en un opérateur qui prend en argument l'identificateur id du processus et l'état de la mémoire mem et qui retourne l'état de la mémoire après exécution de l'instruction C.

⟦ <fun_def> : := <type> **FUN_ID** (<params>) { <decls> <s$_1$:stmt> <s$_2$:stmt>. . .<s$_n$:stmt> }⟧$_{fun_def'}$ $\rightarrow$
⟦<s$_1$:stmt>⟧$_{stmt'}$
⟦<s$_2$:stmt>⟧$_{stmt'}$
. . .
⟦<s$_n$:stmt>⟧$_{stmt'}$

Règle 5.2 – Traduction réduite d'une définition de fonction

Comme mentionné précédemment, la réduction s'applique sur chaque fonction appelée par le processus P_i séparément. Pour un processus P_i appelant une fonction <fun_def>, la traduction réduite est donnée par la règle 5.2.

Cette règle consiste à parcourir le GFC de la fonction <fun_def> et appliquer la règle ⟦<s$_i$: stmt>⟧$_{stmt'}$ sur chaque instruction <s$_i$: stmt>. Nous décrivons dans la suite de cette section comment nous générons les opérateurs TLA+ pour chaque type d'instruction.

(a) Instruction d'affectation et de saut inconditionnel. Pour une instruction <s : stmt> d'affectation ou de saut inconditionnel (`goto`, `break` et `continue`), la règle de traduction réduite est donnée par la règle 5.3.

⟦<s:stmt> : : = <lval> = <expr> **;** | `goto` **LABEL_ID;** | `break;` | `continue;` ⟧$_{stmt'}$ $\rightarrow$
$stmt_$⟦<s:stmt>⟧$_{sid}_long(id, mem)$ $\triangleq$ **if p_aggl(**<s:stmt>**)**
then $stmt_$⟦**succ(**<s:stmt>**)**⟧$_{sid}_long(id, stmt_$⟦<s:stmt>⟧$_{sid}(id, mem))$
else $stmt_$⟦<s:stmt>⟧$_{sid}(id, mem)$

Règle 5.3 – Traduction réduite d'une instruction d'affectation ou de saut inconditionnel

La définition de l'opérateur $stmt_[\![\texttt{<s : stmt>}]\!]_{sid_}long()$ dépend de la réductibilité de l'instruction <s : stmt>. Le principe de cette règle est le suivant : si l'instruction <s : stmt> est agglomérable, la définition de $stmt_[\![\texttt{<s:stmt>}]\!]_{sid_}long()$ consiste à appeler l'opérateur TLA+ modélisant l'instruction C suivante, désignée par $stmt_[\![\textbf{succ(}\texttt{<s : stmt>}\textbf{)}]\!]_{sid_}long()$. Ceci, en transmettant en argument une copie de la mémoire après exécution de l'instruction courante. Cet état mémoire est retourné par l'opérateur $stmt_[\![\texttt{<s : stmt>}]\!]_{sid}()$ généré par la règle 3.15 présentée dans le chapitre 3 et dont la définition TLA+ est donnée dans le module *Program*. Quand l'instruction n'est pas agglomérable, la définition de $stmt_[\![\texttt{<s : stmt>}]\!]_{sid_}long()$ consiste à appeler l'opérateur $stmt_[\![\texttt{<s:stmt>}]\!]_{sid}()$.

Il est à noter que chaque opérateur dans TLA+ ne peut appeler qu'un opérateur qui a été déjà déclaré auparavant. Et comme une instruction C agglomérable est traduite en un opérateur TLA+ qui appelle l'opérateur de l'instruction suivante, l'opérateur $stmt_[\![\texttt{<s : stmt>}]\!]_{sid_}long()$ de toute instruction agglomérable est déclaré comme étant un opérateur récursif, avec le mot clé RECURSIVE.

Exemple 5.2.3. *Reprenons l'exemple du code C donné par la figure 4.2a. L'application de la règle 5.3 sur l'instruction de la ligne 16 se fait comme suit :*

$$[\![\texttt{b = 0;}]\!]_{stmt'} \underset{R.5.3}{\longrightarrow} stmt_[\![\texttt{b =0;}]\!]_{sid_}long(id, mem) \triangleq \quad (1)$$
$$\textbf{\textit{if p_aggl(}}\texttt{b = 0;}\textbf{\textit{) then}}\ stmt_[\![\textbf{\textit{succ(}}\texttt{b = 0;}\textbf{\textit{)}}]\!]_{sid_}long(id, stmt_[\![\texttt{b = 0;}]\!]_{sid}(id, mem))$$
$$\textbf{\textit{else}}\ stmt_[\![\texttt{b = 0;}]\!]_{sid}(id, mem)$$

Comme l'instruction `b = 0;` *fait intervenir une écriture dans une variable locale, elle est alors agglomérable. Le résultat de l'évaluation du prédicat d'agglomération donne le code suivant :*

$$[\![\texttt{b = 0;}]\!]_{stmt'} \underset{R.5.3}{\longrightarrow} stmt_[\![\texttt{b = 0;}]\!]_{sid_}long(id, mem) \triangleq \quad (1)$$
$$stmt_[\![\textbf{\textit{succ(}}\texttt{b = 0;}\textbf{\textit{)}}]\!]_{sid_}long(id, stmt_[\![\texttt{b = 0;}]\!]_{sid}(id, mem))$$

L'instruction en question a comme identificateur $p2_16$ et l'instruction suivante a comme identificateur $p2_17$, en appliquant la règle 3.15 (règle de génération d'identificateur d'instruction), la traduction réduite finale est :

$$(1) \underset{R.5.3}{\longrightarrow} stmt_p2_16_long(id, mem) \triangleq stmt_p2_17_long(id, stmt_p2_16(id, mem))$$

L'opérateur $stmt_p2_16_long()$ consiste à appeler l'opérateur de l'instruction suivante $stmt_p2_17_long()$ en lui passant en argument le résultat retourné par l'opérateur $stmt_p2_16()$. Il est à rappeler que $stmt_p2_16()$ retourne une copie de la mémoire telle que le champ $mem.register[id]$ est mis à jour pour pointer vers l'instruction suivante et la case mémoire correspondante à la variable `b` *contient la valeur $[val \mapsto 0]$.*

Exemple 5.2.4. *Reprenons l'instruction* `goto` *de l'exemple du listing 3.4b (ligne 10). Comme cette instruction est agglomérable alors l'application de la règle 5.3 donne la traduction suivante :*

$$[\![\texttt{goto}\ \textbf{\textit{while_0_break ;}}]\!]_{stmt} \underset{R.5.3}{\longrightarrow} stmt_[\![\texttt{goto}\ \textbf{\textit{while_0_break ;}}]\!]_{sid_}long(id, mem) \triangleq \quad (1)$$
$$stmt_[\![\textbf{\textit{succ(}}\texttt{goto}\ \textbf{\textit{while_0_break ;)}}]\!]_{sid_}long(id, stmt_[\![\texttt{goto}\ \textbf{\textit{while_0_break ;}}]\!]_{sid}(id, mem))$$

Comme l'instruction `goto` *a comme identificateur "f_10" et l'instruction suivante a un*

identificateur égal à "f_13". La traduction réduite donne le code TLA+ suivant :

$$(1) \underset{R.5.3}{\longrightarrow} stmt_f_10_long(id, mem) \triangleq stmt_f_13_long(id, stmt_f_10(id, mem))$$

Ici, l'opérateur $stmt_f_10_long()$ appelle l'opérateur de l'instruction suivante et lui passe en argument $stmt_f_10()$ qui change le registre $mem.register[id]$ pour pointer vers l'instruction qui suit l'instruction `goto` *dans le GFC.*

(b) Instruction conditionnelle `if`. La règle de traduction réduite d'une instruction <s: stmt> de la forme `if` <expr> `then` <s_1: stmt> `else` <s_2: stmt> est donnée par la règle 5.4 La réductibilité de cette instruction dépend de l'expression <expr>. Si cette expression ne fait pas intervenir une lecture dans une variable globale, l'instruction conditionnelle est alors agglomérable.

```
⟦ if <e:exp> then <s1:stmt> else <s2:stmt> ⟧stmt' →
    stmt_⟦<s:stmt>⟧sid_long(id, mem) ≜
      IF ⟦<s:stmt>⟧cond
      THEN if p_aggl(<s:stmt>))
             then stmt_⟦<s1:stmt>⟧sid_long(id, stmt_⟦<s:stmt>⟧sid(id, mem))
             else stmt_⟦<s:stmt>⟧sid(id, mem)
      ELSE if p_aggl(<s:stmt>)
             then stmt_⟦<s2:stmt>⟧sid_long(id, stmt_⟦<s:stmt>⟧sid(id, mem))
             else stmt_⟦<s:stmt>⟧sid(id, mem)
    ⟦<s1:stmt>⟧stmt'
    ⟦<s2:stmt>⟧stmt'
```

Règle 5.4 – Traduction réduite d'une instruction conditionnelle

Nous rappelons que dans le GFC, une instruction `if` dispose de deux successeurs : le bloc d'instructions `THEN` qui est désigné par <s_1: stmt> et le bloc d'instructions `ELSE` designé par <s_2: stmt>.

Exemple 5.2.5. *Soit l'instruction* `if (i>j) tmp = i; else tmp = j;` *de l'exemple 3.3.10. Supposons que cette instruction est agglomérable. L'application de la règle 5.4 sur cette structure conditionnelle donne le résultat suivant :*

```
⟦<s:stmt> ::= <s:stmt> ::= if (i > j) tmp = i; else tmp = j; ⟧stmt' ──→R.5.4      (1)
  stmt_⟦ if i>j tmp = i; else tmp = j;⟧sid_long(id, mem) ≜
   IF ⟦(i > j)⟧cond
   THEN stmt_⟦<s1:stmt>⟧sid_long(id, stmt_⟦if (i > j) tmp = i; else tmp = j;⟧sid(id, mem))
   ELSE stmt_⟦<s2:stmt>⟧sid_long(id, stmt_⟦if (i > j) tmp = i; else tmp = j;⟧sid(id, mem))
  ⟦ tmp = i ;⟧stmt'
  ⟦ tmp = j ;⟧stmt'
```

La traduction de ⟦`(i > j)`⟧$_{cond}$ *consiste à appliquer la règle 3.17.1. Nous utilisons dans la suite de cet exemple le résultat de traduction de cette expression conditionnelle dont les étapes de traduction sont détaillées dans l'exemple 3.3.10. Supposons que l'instruction conditionnelle en question a un identificateur égal "max_13" et l'identificateur de l'instruction* `tmp = i` *vaut*

"max_14" et celui de `tmp = j` *vaut "max_15", en appliquant la règle 3.15 la traduction réduite donne le code suivant :*

$$(1) \underset{R.3.15,R.3.17.1}{\longrightarrow} [\![\texttt{if (i > j) tmp = i; else tmp = j;}]\!]_{stmt'} \rightarrow \quad (2)$$

$$stmt_max_13_long(id, mem) \triangleq$$
$$\text{IF } (not_zero(gt(load(id, mem, Addr_max_param_i), load(id, mem, Addr_max_param_j))))$$
$$\text{THEN } stmt_max_14_long(id, stmt_max_13(id, mem))$$
$$\text{ELSE } stmt_max_15_long(id, stmt_max_13(id, mem))$$
$$[\![tmp = i\,;]\!]_{stmt'}$$
$$[\![tmp = j\,;]\!]_{stmt'}$$

Dans cette traduction, nous passons l'état de la mémoire mis à jour par l'opérateur $stmt_max_13()$ *à l'opérateur* $stmt_max_14_long()$ *ou* $stmt_max_15_long()$ *en fonction de la validité de la condition du construction* IF *.*

Exemple 5.2.6. *Reprenons l'instruction de l'exemple précédent en supposant que l'instruction* `if (i > j) tmp = i; else tmp = j;` *n'est pas agglomérable. L'application de la règle 5.4 donne le résultat suivant :*

$$[\![\texttt{<s:stmt>} ::= \texttt{if (i > j) tmp = i; else tmp = j;}]\!]_{stmt'} \underset{R.5.4}{\longrightarrow} (1)$$

$$stmt_[\![\texttt{if i>j tmp = i; else tmp = j;}]\!]_{sid}_long(id, mem) \triangleq$$
$$\text{IF } [\![\texttt{(i > j)}]\!]_{cond}$$
$$\text{THEN } stmt_[\![\texttt{<s:stmt>}]\!]_{sid}(id, mem)$$
$$\text{ELSE } stmt_[\![\texttt{<s:stmt>}]\!]_{sid}(id, mem)$$
$$[\![tmp = i\,;]\!]_{stmt'}$$
$$[\![tmp = j\,;]\!]_{stmt'}$$

En appliquant les règles 3.17.1 et 3.15, le traduction réduite est la suivante :

$$(1) \underset{R.3.17.1,R.3.15.}{\longrightarrow}$$

$$stmt_max_13_long(id, mem) \triangleq$$
$$\text{IF } (not_zero(gt(load(id, mem, Addr_max_param_i), load(id, mem, Addr_max_param_j))))$$
$$\text{THEN } stmt_max_13(id, mem)$$
$$\text{ELSE } stmt_max_13(id, mem)$$
$$[\![tmp = i\,;]\!]_{stmt'}$$
$$[\![tmp = j\,;]\!]_{stmt'}$$

Dans cette traduction, comme l'expression `(i > j)` *fait intervenir des variables globales. La définition de* $stmt_max_13_long()$ *appelle* $stmt_max_13()$ *qui retourne l'état mémoire après mise à jour de la pile des registres* $mem.register[id]$*.*

(c) Instruction itérative. La traduction réduite d'une instruction itérative (normalisée par CIL), de la forme `while(1){`<d : decls> <s_1 : stmt> . . . <s_n : stmt>`}`, est donnée par la règle 5.5.

L'instruction `while` dans le GFC fait passer le contrôle à la première instruction contenue dans le bloc de `while`. Cette instruction est agglomérable vu qu'elle modifie seulement le registre $mem.register[id]$ du processus id. Donc, la traduction réduite de `while` appelle l'opérateur de l'instruction <s_1 : stmt>. La traduction réduite consiste ensuite à appliquer pour chacune des instructions <s_1 : stmt> . . . <s_n : stmt> la règle de traduction réduite correspondante.

$$
\begin{array}{l}
[\![\text{<s:stmt> ::= } \texttt{while(1)\{} \text{<d:decls> <s}_1\text{:stmt> ... <s}_n\text{:stmt>} \texttt{\}}]\!]_{stmt'} \rightarrow \\
\quad stmt_[\![\text{<s:stmt>}]\!]_{sid}_long(id, mem) \triangleq \; stmt_[\![\text{<s}_1\text{:stmt>}]\!]_{sid}_long(id, stmt_[\![\text{<s:stmt>}]\!]_{sid}(id, mem)) \\
\quad [\![\text{<s}_1\text{:stmt>}]\!]_{stmt'} \\
\quad [\![\text{<s}_2\text{:stmt>}]\!]_{stmt'} \\
\quad \ldots \\
\quad [\![\text{<s}_n\text{:stmt>}]\!]_{stmt'}
\end{array}
$$

Règle 5.5 – Traduction réduite d'une instruction itérative

Exemple 5.2.7. *Reprenons l'instruction itérative de la ligne 1 de l'exemple donné par le listing 3.4b. L'application de la règle 5.5 donne la traduction suivante :*

$$
\begin{array}{l}
[\![\texttt{while(1)\{int tmp; \{tmp = i; i++;\}...\}}]\!]_{stmt'} \underset{R.5.5}{\longrightarrow} \qquad \textit{(1)} \\
\quad stmt_[\![\texttt{while(1)\{int tmp; \{tmp = i; i++;\}...\}}]\!]_{sid}_long(id, mem) \triangleq \\
\qquad stmt_[\![\texttt{tmp = i}]\!]_{sid}_long(id, stmt_[\![\texttt{while(1)\{int tmp;\{tmp = i; i++;\}...\}}]\!]_{sid}(id, mem)) \\
\quad [\![\textit{\{ int tmp ; \{tmp = i ; i++ ;\}...\}}]\!]_{stmt'}
\end{array}
$$

Sachant que l'identificateur de l'instruction itérative (ligne 1) vaut "f_6", celui de `tmp = i` *est "f_8", en appliquant la règle 3.15, le résultat de la traduction réduite est le suivant :*

$$
\begin{array}{l}
\textit{(1)} \underset{R.3.15}{\longrightarrow} stmt_f_6_long(id, mem) \triangleq stmt_f_8_long(id, stmt_f_6(id, mem)) \\
\qquad [\![\textit{\{ int tmp ; \{tmp = i ; i++ ;\}...\}}]\!]_{stmt'}
\end{array}
$$

L'opérateur $stmt_f_6_long()$ fait directement appel à $stmt_f_8_long()$, l'opérateur réduit de la première instruction contenue dans le bloc `while`*. Ceci en passant en argument l'état mémoire retourné par l'opérateur $stmt_f_6()$. Nous rappelons que $stmt_f_6()$ met à jour la pile des registres $mem.register[id]$ pour pointer vers la première instruction du bloc* `while`*.*

Il est possible qu'un programme C contienne une séquence d'instructions agglomérables causant une boucle infinie. Par exemple, une séquence d'instructions contenant l'instruction `lbl: goto lbl`. L'exploration d'états de la spécification TLA+ réduite du code C, résulte en une suite infinie d'appels récursifs de l'opérateur TLA+ modélisant l'instruction en question. Ces cas sont détectables par TLC. Nous décrivons dans la section 5.3 un exemple qui met en œuvre ce cas de figure.

(d) Appel de fonction. La traduction réduite d'un appel de fonction suit la règle 5.6.

$$
\begin{array}{l}
[\![\text{<s:stmt> ::= <lval> = } \mathbf{FUN_ID}\,(\text{<e}_1\text{:expr>...<e}_n\text{:expr>})]\!]_{stmt'} \rightarrow \\
\quad stmt_[\![\text{<s:stmt>}]\!]_{sid}_long_pre(id, mem) \triangleq \\
\qquad \textbf{if } \mathbf{p_aggl}\,(\mathbf{FUN_ID}\,(\text{<e}_1\text{:expr>...<e}_n\text{:expr>})) \\
\qquad \textbf{then } stmt_[\![\mathbf{fst}\,(\mathbf{FUN_ID})]\!]_{sid}(id, stmt_[\![\text{<s:stmt>}]\!]_{sid}_pre(id, mem)) \\
\qquad \textbf{else } stmt_[\![\text{<s:stmt>}]\!]_{sid}_pre(id, mem) \\
\\
\quad stmt_[\![\text{<s:stmt>}]\!]_{sid}_long_post(id, mem) \triangleq \\
\qquad \textbf{if } \mathbf{p_aggl}\,(\text{<s:stmt>}) \\
\qquad \textbf{then } stmt_[\![\mathbf{succ}\,(\text{<s:stmt>})]\!]_{sid}_long(id, stmt_[\![\text{<s:stmt>}]\!]_{sid}_post(id, mem)) \\
\qquad \textbf{else } stmt_[\![\text{<s:stmt>}]\!]_{sid}_post(id, mem)
\end{array}
$$

Règle 5.6 – Traduction réduite d'un appel de fonction

Nous rappelons qu'un appel à une fonction de type non `void`, s'écrivant sous la

forme <lval> = **FUN_ID** (<e$_1$: expr>. . .<e$_n$: expr>) est traduit en deux opérateurs TLA+ (voir règle 3.3.7.1). Le premier opérateur (avec un suffixe $_pre$) empile le cadre de pile de la fonction appelée dans $mem.stack[id]$. Il met ensuite à jour la pile de registres $mem.register[id]$ avec l'identifiant de la première instruction de la fonction appelée en sauvegardant l'identifiant de l'instruction suivant l'appel de fonction. Le deuxième opérateur (avec un suffixe $_post$) permet de charger la valeur retournée par la fonction appelée dans le registre $mem.ret[id]$.

Pour la traduction réduite, l'appel de fonction **FUN_ID** (<e$_1$: expr>. . .<e$_n$: expr>) est agglomérable si chaque argument de **FUN_ID** ne fait pas intervenir une lecture dans une variable globale.

Ici, la règle 5.6 génère deux opérateurs.

- $stmt_$⟦<s : stmt>⟧$_{sid}_long_pre(id, mem)$ est défini tel que : si l'appel de fonction est agglomérable, l'opérateur de la première instruction de la fonction **FUN_ID** appelée est appelé, en lui passant en argument l'état mémoire retourné par l'opérateur $stmt_$⟦<s : stmt>⟧$_{sid}_pre()$. Si l'appel de fonction n'est pas agglomérable, on appelle directement $stmt_$⟦<s : stmt>⟧$_{sid}_pre()$.

- $stmt_$⟦<s : stmt>⟧$_{sid}_long_post(id, mem)$ est défini tel que : si le retour de fonction ne se fait pas dans une variable globale alors l'affectation de <lval> est agglomérable et dans ce cas on appelle l'instruction suivante en passant en argument l'opérateur $stmt_$⟦<s : stmt>⟧$_{sid}_post()$. Sinon, on appelle directement $stmt_$⟦<s : stmt>⟧$_{sid}_post()$.

Exemple 5.2.8. *La traduction réduite de l'instruction d'appel à la fonction* `inc(a)` *(ligne 11 de la figure 4.2a) est comme suit :*

$$[\![\texttt{x = inc(a);}]\!]_{stmt'} \underset{R.5.6}{\longrightarrow} \quad (1)$$
$$stmt_[\![\texttt{x = inc(a);}]\!]_{sid}_long_pre(id, mem) \triangleq stmt_[\![\textbf{fst}\texttt{(inc)}]\!]_{sid}(id, mem))$$
$$stmt_[\![\texttt{x = inc(a);}]\!]_{sid}_long_post(id, mem) \triangleq stmt_[\![\texttt{x = inc(a);}]\!]_{sid}_long(id, stmt_[\![\texttt{x = inc(a);}]\!]_{sid}_post(id, mem))$$

L'identificateur de l'instruction d'appel de fonction est égal à "p1_11" et la première instruction de la fonction `inc()` *est "inc_15". Nous appliquons la règle 3.15 nous obtenons :*

$$(1) \underset{R.3.15}{\longrightarrow} stmt_p1_11_long_pre(id, mem) \triangleq stmt_inc_5(id, stmt_p1_11_pre(id, mem))$$
$$stmt_p1_11_long_post(id, mem) \triangleq stmt_p1_11_post(id, mem)$$

Ici, l'appel à `inc(a)` *est agglomérable car l'argument de la fonction fait intervenir une lecture dans une variable locale. Dans ce cas, $stmt_p1_11_long_pre()$ appelle l'opérateur de l'instruction suivante, à savoir la première instruction de la fonction* `inc()`. *L'affectation du retour de fonction dans la variable* `x` *n'est pas agglomérable puisque* `x` *est une variable globale. Dans ce cas, l'opérateur $stmt_p12_long()$ appelle l'opérateur $stmt_p1_11_post()$. Celui-ci retourne l'état mémoire après avoir mis dans le registre $mem.ret[id]$ la valeur retournée par la fonction* `inc()`.

(e) Retour de fonction. La traduction réduite d'un retour de fonction suit la règle 5.7.

$$
\begin{array}{l}
[\![\text{<s:stmt> ::= } \texttt{return;} \mid \texttt{return} \text{ <e:expr> }]\!]_{stmt'} \rightarrow \\
\quad stmt_[\![\text{<s:stmt>}]\!]_{sid}_long(id, mem) \triangleq \\
\qquad\qquad \textbf{if p_aggl(}\text{<s:stmt>}\textbf{)} \\
\qquad\qquad \textbf{then } dispatch_red(id, stmt_[\![\text{<s:stmt>}]\!]_{sid}(id, mem)) \\
\qquad\qquad \textbf{else } stmt_[\![\text{<s:stmt>}]\!]_{sid}(id, mem)
\end{array}
$$

Règle 5.7 – Traduction réduite d'un retour de fonction

L'instruction `return;` est toujours agglomérable vu qu'elle met à jour le registre local $mem.register[id]$ du processus id. L'instruction `return` <e : expr> n'est agglomérable que si <e : expr> ne fait pas intervenir une lecture dans une variable globale ou dans une variable locale référencée. Quand cette instruction est agglomérable, on fait appel à l'opérateur $dispatch_red()$. L'appel à $dispatch_red()$ permet de mettre à jour le flot d'exécution, vu que l'instruction `return` n'a de successeur dans le GFC. Sinon, on appelle l'opérateur $stmt_[\![\text{<s : stmt>}]\!]_{sid}()$ qui va dépiler le registre $mem.register[id]$ du processus id et la pile $mem.stack[id]$.

Exemple 5.2.9. *La traduction réduite du retour de fonction (ligne 6 du code C de la figure 4.2a) consiste à appliquer la règle 5.7 comme suit :*

$$
[\![\texttt{return tmp;}]\!]_{stmt'} \underset{R.5.7}{\longrightarrow} \qquad (1)
$$
$$
stmt_[\![\texttt{return tmp;}]\!]_{sid}_long(id, mem) \triangleq dispatch_red(id, stmt_[\![\texttt{return tmp;}]\!]_{sid}(id, mem))
$$

Supposons que cette instruction a un identificateur égal à "inc_6". Appliquer la règle 3.15, la traduction réduite finale est :

$$
(1) \underset{R.3.15}{\longrightarrow} stmt_inc_6_long(id, mem) \triangleq dispatch_red(id, stmt_inc_6(id, mem))
$$

L'appel à l'opérateur $stmt_inc_6_long(id, mem)$ permet d'appeler $dispatch_red()$ qui met à jour le flot de contrôle, car l'instruction `return` n'a pas de successeur dans le GFC.

Si `tmp` n'était pas une variable locale mais plutôt une variable partagée, l'instruction `return tmp;` ne serait pas agglomérable et dans ce cas la traduction réduite serait :

$$
[\![\texttt{return tmp;}]\!]_{stmt'} \underset{R.5.7}{\longrightarrow} stmt_inc_6_long(id, mem) \triangleq stmt_inc_6(id, mem)
$$

L'opérateur $stmt_inc_6_long()$ fait appel à l'opérateur $stmt_inc_6()$ de la traduction standard. Ce dernier retourne l'état de la mémoire après avoir stocké dans $mem.ret[id]$ la valeur de `tmp` et dépilé le registre $mem.register[id]$ et la pile $mem.stack[id]$ du processus id.

5.2.4 Préservation des propriétés

Comme décrit dans la section 5.2.2, une réduction est caractérisée par une transformation f. Dans notre cas la transformation appliquée consiste à générer une spécification TLA+ telle que l'espace d'états résultant $\mathcal{T}_r$ est réduit par rapport l'espace d'état $\mathcal{T}$ généré par la spécification TLA+ sans réduction. Nous notons $\mathcal{T}_r = f(\mathcal{T})$.

Définition 5.2.2. *Soit φ une propriété TLA+. La transformation f est dite complète pour un prédicat d'agglomération $p()$ si $\mathcal{T}_r \models \varphi$ est équivalente à $\mathcal{T} \models \varphi$.*

Proposition 5.2.2. *Si la propriété à vérifier φ s'exprime que sur les variables partagées par les processus, alors la transformation f est complète pour le prédicat d'agglomération $p()$.*

Sketch de preuve. Soit L un ensemble de locations (adresses) mémoire. Soit la fonction $proj_L(s)$ la projection de l'état par rapport à L. Cette fonction retourne une copie de l'état ne contenant que la valuation de l'ensemble L de locations mémoire. Soit $projb_L()$ une fonction définie telle que $projb_L(\langle s_0, s_1, \ldots, s_n\rangle) = \langle proj_L(s_0), proj_L(s_1), \ldots, proj_L(s_n)\rangle$.

Supposons que $\mathcal{T}_r \models \varphi$ et prouvons que $\mathcal{T} \models \varphi$. Supposons que φ est exprimée que sur l'ensemble des locations mémoire L. Soit $\sigma = \langle s_0, \ldots, s_i, \ldots, s_j, \ldots, s_n\rangle$ un comportement tel que $\sigma \in \mathcal{T}$ et considérons que la séquence d'états $s_i, \ldots, s_j$ correspond à une séquence d'instructions C agglomérables. On a $projb_L(\sigma) = \langle proj_L(s_0), \ldots, proj_L(s_i) \ldots, proj_L(s_j), \ldots, proj(s_n)\rangle$. Comme $\mathcal{T}_r = f(\mathcal{T})$ alors il existe un comportement $\sigma' \in \mathcal{T}_r$ tel que $\sigma' = \langle s_0, \ldots, s'_{i\ldots j}, \ldots, s_n\rangle$ et $s'_{i\ldots j}$ est l'état après agglomération des instructions C qui correspondent à la séquence d'états $s_i, \ldots, s_j$. Comme l'agglomération ne porte pas sur les locations mémoire L et étant donné que les propriétés TLA+ sont invariantes au bégaiement alors $proj_L(s'_{i\ldots j}) = proj_L(s_i) = proj_L(s_j)$. Et comme $projb_L(\sigma') \models \varphi$ alors $projb_L(\sigma) \models \varphi$.

Prouver que si $\mathcal{T} \models \varphi$ alors $\mathcal{T}_r \models \varphi$ revient à prouver que tout comportement σ dans $projb_L(\mathcal{T})$ correspond à un comportement $\sigma' \in \mathcal{T}_r$ tel que $projb_L(\sigma) = projb_L(\sigma')$ (à cause l'insensibilité au bégaiement). Étant donné que $projb_L(\sigma) \models \varphi$ et comme nous considérons que l'exécution des actions est équitable alors $projb_L(\sigma') \models \varphi$.

□

Corollaire 5.2.1. *Soit $q()$ un prédicat d'agglomération basé sur des conditions suffisantes du prédicat d'agglomération $p()$, tel que $q() \Rightarrow p()$. Si la propriété à vérifier φ ne s'exprime que sur les variables partagées alors la transformation f induite par $q()$ est complète.*

5.3 EXPÉRIMENTATIONS ET RÉSULTATS

Nous illustrons dans cette section les résultats obtenus en appliquant la technique d'agglomération. Nous utilisons dans ces expérimentations le prédicat d'agglomération basé sur une analyse syntaxique du code C. Les programmes C que nous considérons sont les suivants :

- code *Zunebug* : il s'agit d'un programme erroné du pilote de l'horloge interne d'un lecteur mp3 appelé *Zune*. Le code source de la fonction qui a causé le *bug* est donné dans [WFLGN10] ;

- les protocoles d'entrée et de sortie de la section critique, à savoir la solution de Dekker [Dij68], celle de Peterson [Pet81] et de la boulangerie [Lam74] ;

- le dîner des philosophes : nous étudions la solution proposée par Tanenbaum [Tan07].

TABLE 5.1 – Résultats expérimentaux du *model-checking* avant et après application de la réduction (temps en secondes)

#Prog	#Proc	avant réduction		après réduction		#P1(%)	#P2(%)
		nombre d'états	temps (s)	nombre d'états	temps (s)		
Zunebug	1	389	0.147	2	0.136	99.48	7.48
Dekker	2	173	0.128	70	0.109	59.53	14.84
Peterson	2	107	1.37	22	0.131	79.43	90.43
	4	$1.08 * 10^6$	59.2	$3.12 * 10^4$	4.82	97.11	91.52
Bakery	2	$2.3 * 10^3$	1.91	223	1.67	88.86	12.56
	4	$5.05 * 10^7$	1560	$8.35 * 10^5$	76.6	99.83	95.08
Philos	4	$9.8 * 10^7$	366	$1.46 * 10^5$	12	99.85	96.7
	5	$>6.19 * 10^8$	>25340	$4.17 * 10^6$	352	99.32	98.61

En utilisant le *model-checker* TLC, nous calculons le nombre total d'états des spécifications TLA+ générées par C2TLA+ sans et après application de la technique de réduction. Les résultats des expériences sont présentés dans la table 5.1, où #Proc désigne le nombre de processus. Les colonnes 3 à 6 présentent la taille de l'espace d'états ainsi que le temps de vérification avant et après application de la technique de réduction. #P1 indique le pourcentage de réduction en terme d'états et #P2 indique le pourcentage de réduction en terme de temps de vérification.

Pour le code *zunebug*, l'espace d'états après réduction comprend 2 états. Ceci s'explique par le fait que le code C n'utilise que des variables locales et donc toutes les instructions contenues dans le programme sont agglomérables. Comme cet exemple contient une boucle C infinie, l'exécution de la spécification réduite par TLC génère l'erreur suivante :

`This was a Java StackOverflowError. It was probably the result of an incorrect recursive function definition that caused TLC to enter an infinite loop when trying to compute the function or its application`. Cette erreur indique qu'un opérateur récursif est appelé indéfiniment. L'utilité d'appliquer la réduction sur un code séquentiel est de déterminer si le programme termine ou pas en un peu de temps.

En ce qui concerne l'exemple des philosophes (avec 5 processus), le temps de vérification de la spécification avant réduction prend plus de 7 heures alors que celle après réduction nécessite seulement 6 minutes. Pour ce même exemple, le facteur de réduction #P2 atteint son maximum, à savoir 98.61%.

Comme le montrent les résultats, appliquer une transformation en utilisant un prédicat basé sur une analyse syntaxique du code C réduit, de manière exponentielle la taille de

l'espace d'états ainsi que le temps du *model-checking*.

5.4 Alternatives d'implémentation de la réduction

Avant d'implémenter l'idée de réduction dans C2TLA+, plusieurs propositions ont été étudiées et discutées. Dans cette section, nous décrivons les idées proposées et leurs limitations.

5.4.1 Alternative erronée

Dans C2TLA+, chaque instruction C <s_i : stmt> est traduite en un opérateur TLA+, noté $stmt_[\![$<s_i: stmt>$]\!]_{stmt}()$ prenant en argument l'identificateur de processus id et l'état de la mémoire mem et retourne un enregistrement contenant l'état de la mémoire après exécution de l'instruction C. La première alternative s'inspirait du *style de programmation par continuation* appelé « *CPS* » (en anglais « *continuation passing style* »). Ce style est utilisé dans le λ-calcul et dans les langages fonctionnels où une fonction ne renvoie pas le résultat de son calcul, mais le passe en argument à une autre fonction, appelée *continuation*. Cette continuation représente la suite d'instructions à exécuter.

Dans notre cas, l'idée consistait à passer en argument pour chaque opérateur $stmt_[\![$<s_i: stmt>$]\!]_{stmt}()$ modélisant une instruction <s_i : stmt> un argument supplémentaire. Cet argument se présente sous la forme d'un opérateur générique, noté $K(_)$ et qui correspond à la continuation, c'est-à-dire la suite d'instructions à exécuter.

$$stmt_f_3(id, mem, K(_)) \triangleq K([mem \text{ EXCEPT } mem.register = update_reg(mem.register[id], \text{“f_6”})])$$

Listing 5.2 – Exemple de traduction d'une instruction `goto` en utilisant les opérateurs génériques

Par exemple, pour une instruction `goto lbl2;` contenue dans une fonction `f()` et ayant un identifiant égal à “f_1”, la traduction de cette instruction est donnée par le listing 5.2. L'opérateur $stmt_f_1()$ prend en argument l'identifiant du processus id, l'état de la mémoire mem et l'opérateur générique $K(_)$ qui représente la continuation. Nous rappelons que l'opérateur $update_reg()$ (introduit dans la section 3.3.6.1) permet de mettre à jour la pile des registres $mem.register[id]$ du processus id. La définition de $stmt_f_1()$ consiste à appliquer l'opérateur $K()$ à l'état mémoire retourné après exécution de l'instruction `goto`, à savoir la mise à jour de la pile des registres du processus id par l'appel à l'opérateur $update_reg()$.

La modélisation d'un flot d'exécution atomique consistait à appeler l'opérateur TLA+ d'une instruction C en lui passant en paramètre la continuation. Cette continuation est exprimée en TLA+ par une lambda-expression. Une lambda-expression permet de définir des fonctions anonymes et locales. Ces expressions sont introduites dans le

chapitre 2 (voir section 2.2.9). Nous illustrons comment les opérateurs génériques et les lambda-expressions sont utilisés pour modéliser un flot d'exécution atomique d'un programme en TLA+. Pour ce faire, nous considérons le code C donné par le listing 5.3a. La traduction de la définition de la fonction `f()` se fait comme le montre le listing 5.3b. Cette traduction consiste en une disjonction d'actions. Chaque action correspond à une exécution atomique d'une ou plusieurs instructions C. Comme l'instruction de la ligne 2 (listing 5.3a) est agglomérable, la première action modélise l'exécution des instructions de la ligne 2 et de la ligne 3. L'instruction de la ligne 3 est modélisée par l'opérateur TLA+ $stmt_f_3()$. Cet opérateur est passé en argument lors de l'appel à l'opérateur $stmt_f_2()$. Il désigne la continuation, c'est-à-dire la suite d'opérations à exécuter après $stmt_f_2()$ et il est exprimé par une lambda-expression. L'appel à l'opérateur $stmt_f_3()$ prend en argument l'opérateur $result$, qui permet de retourner le résultat de l'exécution de l'instruction courante ou du bloc d'instructions courant. Quand une instruction n'est pas agglomérable, l'opérateur $result()$ est passé en argument. C'est le cas pour l'instruction de la ligne 4 du listing 5.3a. La traduction de cette instruction consiste à appeler $stmt_f_4()$ en passant en argument l'opérateur $result()$.

La traduction de la fonction `main()` (ligne 9 du listing 5.3a) est modélisée par l'opérateur $main()$. Les opérateurs $stmt_main_9_pre()$ et $stmt_main_10()$ désignent respectivement la traduction des instructions de la ligne 9 et 10, selon le principe décrit par l'exemple du listing 5.2. L'exécution du programme est modélisée par la formule $Spec$. Quand le registre d'un processus $memory.register[id]$ est non vide, une action parmi $f()$ ou $main()$ est exécutée.

L'inconvénient de cette traduction est l'impossibilité de modéliser l'agglomération d'une instruction d'appel de fonction. En effet, quand le registre $memory.register[id]$ est non vide, une des actions $main()$ ou `f()` est exécutable. Quand l'action $main()$ l'action `f()`, il est nécessaire d'exécuter l'action $stmt_main_9()$ afin de mettre à jour la pile des registres du processus, pour que l'action $f()$ devienne exécutable. Le deuxième inconvénient est qu'il est impossible de modéliser l'agglomération d'une instruction de retour de fonction, car on ne sait pas la continuation de cette instruction.

D'après l'alternative décrite dans cette section, on aperçoit que les choix de traduction sont importants pour appliquer la technique de réduction. Les règles de traduction établies dans les chapitres 3 et 4 nous ont permis de gérer la traduction réduite dans C2TLA+.

5.4.2 Optimisation de la réduction basée sur une analyse sémantique

En utilisant une analyse syntaxique du code C, il est difficile de déterminer si une instruction utilisant des références ou des pointeurs est agglomérable ou pas, car une variable locale peut faire référence à une zone mémoire partagée (voir exemple 5.2.1). L'analyse syntaxique est alors insuffisante pour déterminer tel cas. Pour cela, une analyse sémantique basée sur l'interprétation abstraite du code peut s'avérer intéressante. Nous

```
void f(int a){
  x = 1;
  goto l3;
  x=x+a;
  ...
  }

void main(){
  f(1);
  b = x+1;
  ...
  }
```

(a) Code C

$stmt_f_2(id, mem, K(_)) \triangleq \ldots$
$\ldots$
$stmt_main_9(id, mem, K(_)) \triangleq \ldots$
$\ldots$
$result(id, mem) \triangleq mem$

$f(id, mem, K(_)) \triangleq$
$\quad \lor \land Head(mem.register[id]).pc = \text{"f_2"}$
$\quad\quad \land memory' \triangleq stmt_f_2(id, mem,$
$\quad\quad\quad \text{LAMBDA } mem1 : stmt_f_3(id, mem1, result))$
$\quad \lor \land Head(mem.register[id]).pc = \text{"f_4"}$
$\quad\quad \land memory' = stmt_f_4(id, mem, result)$
$\quad \ldots$
$main(id, mem, K(_)) \triangleq$
$\quad \lor \land Head(mem.register[id]).pc = \text{"main_9"}$
$\quad\quad \land memory' = stmt_main_9_pre(id, mem, result))$
$\quad \lor \land Head(mem.register[id]).pc = \text{"main_10"}$
$\quad\quad \land memory' = stmt_main_10(id, mem, result)$
$\quad \ldots$
$Next \triangleq \lor \exists\, id \in ProcSet :$
$\quad\quad \land memory.register[id] \neq \langle\rangle$
$\quad\quad \land \lor main(id, memory, result)$
$\quad\quad\quad \lor f(id, memory, result)$
$\quad \lor \forall\, id \in ProcSet :$
$\quad\quad \land memory.register[id] = \langle\rangle$
$\quad\quad \land \text{UNCHANGED } vars$

$Spec \triangleq Init \land \Box[Next]_{memory} \land \mathrm{WF}_{memory}(Next)$

(b) Code TLA+

Listing 5.3 – Alternative de traduction d'un code C en TLA+ en utilisant les opérateurs génériques et les expressions λ

proposons une analyse sémantique par le plugin Mthread [CL15b] de la plateforme Frama-C. Ce prédicat n'est pas implémenté dans C2TLA+, nous décrivons seulement dans ce qui suit comment ce plugin peut s'intégrer dans C2TLA+. Pour plus de détails sur le fonctionnement de ce plugin, nous invitons le lecteur à consulter le manuel [YB12].

Mthread est basé sur le plugin *value analysis* [CCM09]. Il prend en entrée un programme C concurrent (utilisant la bibliothèque *pthread*) et fournit de multiples informations sur le code C, à savoir une sur-estimation des zones mémoires accessibles simultanément par plusieurs processus. Le calcul de cette sur-estimation est basé sur les étapes suivantes :

- calculer pour chaque processus i, exécuté symboliquement, l'ensemble des variables globales lues $R(i)$ et écrites $W(i)$;

- calculer l'ensemble RW des zones potentiellement partagées. Cet ensemble

comprend les zones mémoires lues par un autre processus i et écrites par un processus j, tel que $RW = \bigcup_{i,j,i \neq j} (R(j) \cap W(j))$;

- calculer pour chaque processus j l'ensemble $Rp^z(j)$ (respectivement $Wp^z(j)$) événements de lecture (respectivement écriture) de toute zone z appartenant à RW ;

- calculer le prédicat $shared(z)$ de chaque zone z dans RW, défini par : $shared(z) = \exists i, j, i \neq j : \exists st_1 \in RSt^z(i) : \exists st_2 \in WSt^z(j) : live(i@st_1) \wedge live(j@st_2)$
  avec $RInst^z(i)$ et $WInst^z(i)$ désignent respectivement l'ensemble des instructions de lecture et l'ensemble des instructions d'écriture opérées par le processus i sur la zone mémoire z. Le prédicat $live(i@st)$ désigne que le processus i est actif à l'instruction st.
  Le prédicat $shared(z)$ détermine pour chaque zone mémoire z dans RW, s'il existe une opération de lecture st_1 de la zone z opérée par le processus i telle qu'il existe une opération d'écriture st_2 opérée par un processus j sur la même zone z.

En se basant sur ce résultat, Mthread retourne pour chaque instruction les variables qui sont accédées simultanément en lecture/écriture par plusieurs processus. Nous illustrons l'utilité cette analyse dans l'exemple donné par le listing 5.4.

```
1  int x,y,z;
2  int lock = 1;
3
4  void P1() {
5    int l;
6    l=&x;
7    (*l)++;
8    y = *l;
9    P(&lock);
10   z = z-1;
11   V(&lock);
12   ... }
```

```
13 void P2() {
14   x--;
15   P(&lock);
16   z = x+1;
17   V(&lock);
18   ...
19 }
```

Listing 5.4 – Exemple de code C concurrent

Dans le code C du listing 5.4, chacune des fonctions `P1()` et `P2()` est exécutée par un processus. Les deux processus partagent les variables `x`, `y`, `z` et `lock`. La variable `lock` implémente un mutex, accessible par le biais des primitives atomiques `P()` et `V()`.

Une analyse sémantique avec Mthread détecte que l'instruction de la ligne 7 fait intervenir la variable `x`, qui est accédée simultanément en lecture et en écriture par les deux processus. Mthread détecte aussi que l'instruction de la ligne 8 fait intervenir une variable accédée seulement par un seul processus.

Il est à noter aussi que Mthread détecte les portions du code C protégées par des mutex. Dans cet exemple, l'accès à la variable `z` est protégé par le mutex `lock`. Nous invitons le lecteur à consulter [YB12] pour plus de détails sur le principe d'analyse opéré par Mthread.

Si on utilisait une analyse sémantique pour implémenter notre prédicat

d'agglomération, les instructions agglomérables de l'exemple sont les instructions de la ligne 8, 10 et 16. De telles détections ne sont pas possibles avec une analyse syntaxique.

5.5 CONCLUSION

Nous avons présenté dans ce chapitre une technique de réduction de l'espace d'états d'une spécification TLA+ basée sur l'agglomération des instructions C. Nous avons décrit comment cette technique s'intègre dans C2TLA+ afin de générer une spécification réduite à partir des programmes C. Cette intégration se fait en définissant un prédicat d'agglomération qui détermine si une instruction est agglomérable ou pas. Nous avons proposé un exemple de prédicat basé sur une analyse syntaxique du code C. Nous avons ensuite montré sur des exemples de code que cette technique permet de réduire le nombre d'états générés lors de la phase du *model-checking*. Il est à noter que cette réduction est opérée lors de la génération du code TLA+ et se fait sur les instructions C de chaque processus, c'est-à-dire avant la composition asynchrone des processus. Néanmoins, il est possible de définir des conditions plus fines afin d'avoir plus d'agglomérations.

Nous développons dans le prochain chapitre l'application de notre méthodologie de spécification et de vérification avec TLA+ sur une étude de cas réel.

Troisième partie

Expérimentation

6

Spécification et vérification de l'ordonnanceur du micronoyau PharOS

Les logiciels systèmes critiques sont des systèmes complexes qui sont soumis à des exigences de fiabilité et de sûreté de fonctionnement. Durant les dernières années, de nombreuses recherches ont été menées sur l'application des méthodes formelles pour la vérification des logiciels systèmes critiques. Comme évoqué dans le chapitre 1, citons par exemple, parmi les plus spectaculaires, la vérification du micronoyau seL4 [KEH+09].

Les micronoyaux sont souvent multitâches. Ils interagissent avec des équipements matériels et leur développement met en œuvre des primitives système complexes. Les traitements à réaliser par un micronoyau doivent respecter des contraintes temporelles et les données doivent être cohérentes durant l'évolution du système. Face à ces exigences, la vérification formelle est un élément clé pour garantir une meilleure fiabilité de tels systèmes.

Nous consacrons ce chapitre à la spécification et la vérification d'un cas d'étude réel issu de l'implémentation d'un micronoyau, appelé PharOS [LOC+11], qui assure l'ordonnancement de tâches temps-réel. Nous nous intéressons en particulier à la vérification des propriétés fonctionnelles de ce dernier, et non pas aux propriétés temps-réel de l'application qu'il gère. Ce chapitre est structuré comme suit : nous commençons dans la section 6.2 par donner un bref aperçu des travaux existants portant sur la vérification des micronoyaux. Nous décrivons ensuite dans la section 6.2 la

structure générale du micronoyau et ses principes de fonctionnement. Ce dernier dispose de deux implémentations, une version monoprocesseur et une version s'exécutant sur un processeur multi-cœur. Dans la section 6.3, nous détaillons la modélisation TLA+ de la version monoprocesseur. La section 6.4 est consacrée à la modélisation TLA+ de la version multi-cœur. Dans ces deux sections, nous expliquons comment nous avons étendu le modèle d'exécution proposé dans le chapitre 4 afin de spécifier formellement l'interaction du micronoyau avec l'environnement matériel. Nous exprimons ensuite, pour chacune des versions, les propriétés à vérifier. Enfin, nous détaillons et discutons les résultats de vérification obtenus ainsi que l'importance de l'utilisation de la technique de réduction proposée dans le chapitre 5.

6.1 Travaux sur la vérification formelle des micronoyaux

La vérification formelle a été appliquée par une équipe de chercheurs du laboratoire NICTA de Sydney à un micronoyau de système d'exploitation orienté système embarqué, appelé seL4 [KEH+09]. Ce micronoyau cible les architectures matérielles x86 et ARM. seL4 est considéré comme le premier micronoyau dont la conformité entre la spécification du système et le binaire produit à partir du code C a été vérifiée. L'effort de vérification de ce micronoyau a été considérable. Il a fallu écrire des preuves de 200,000 lignes de code ISABELLE et un effort qui correspond à 20 personnes-année pour 8700 lignes de code C et 600 lignes de code assembleur.

Verisoft XT [ER11] est un projet allemand qui vise la vérification formelle de l'implémentation des systèmes d'exploitation comme PikeOS [BBBB09]. Le micronoyau consiste en 6000 lignes de code C. Il supporte les architectures x86, PowerPC et ARM. La vérification utilise l'environnement VCC [CDH+09] (voir section 1.2.5.1), qui utilise Z3 comme prouveur de théorèmes.

En 2015, une entreprise française Prove&Run a proposé une preuve d'un micronoyau appelé ProvenCore [Les15], qui garantit l'intégrité et la confidentialité des différents processus exécutés par le micronoyau. Ce micronoyau cible les architectures ARM. Il a été spécifié et développé dans un langage de spécification appelé Smart, un langage fonctionnel créé par Prove&Run. Ce langage permet de spécifier des modèles et des spécifications logiques, à savoir des pré-/post-conditions, des lemmes ainsi que des prédicats inductifs. Le code Smart est ensuite traduit en Smil (« *SMart Intermediate Language* »), langage d'entrée du prouveur Smart. Un code C est enfin généré à partir de Smil.

6.2 DESCRIPTION DU MICRONOYAU PHAROS

PharOS est un micronoyau temps-réel, préemptif, développé en 2009, par le CEA-List. Le micronoyau effectue l'ordonnancement et l'exécution des tâches temps-réel. Il implémente l'algorithme d'ordonnancement EDF (« *Earliest Deadline First* »), dans lequel la tâche la plus prioritaire est celle dont l'échéance est la plus proche. La conception du micronoyau est minimaliste afin qu'il puisse être intégré sur des architectures avec des ressources réduites. Par exemple, il requiert 5Koctets avec le microcontrôleur S12XE. Il a été porté sur plusieurs architectures. Citons à titre d'exemple PowerPC 551x, PowerPC 560xB et ARMv7M. Il a été utilisé au départ dans le domaine du nucléaire. Puis, il a été adapté aux contraintes du monde de l'automobile [CL12]. PharOS a permis la création, en 2011, par le CEA-List d'une start-up appelée *Krono-safe*, un éditeur logiciel pour l'embarqué temps-réel critique. *Krono-safe* a industrialisé son produit *Kron-OS*, un noyau temps-réel et un outil de configuration fondé sur la technologie PharOS [CL11].

6.2.1 Structure de PharOS

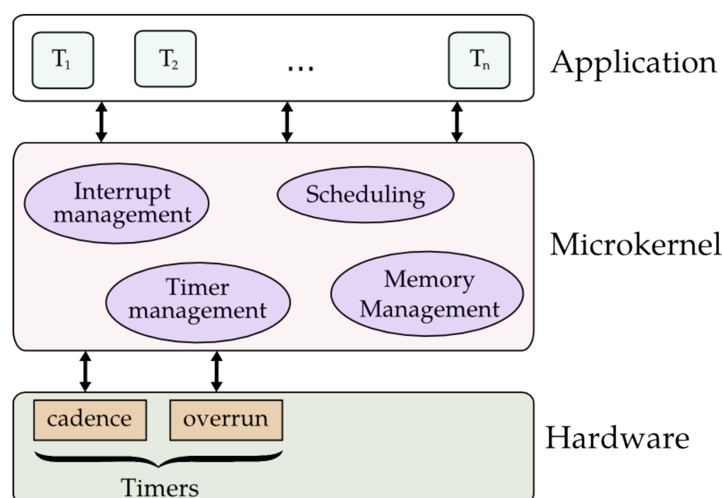


FIGURE 6.1 – Structure générale de PharOS

Pour rester fidèle à l'implémentation, nous préservons dans la suite de ce document les termes en anglais. Le schéma donné par la figure 6.1 montre la structure simplifiée de PharOS. Dans ce travail, nous nous intéressons au niveau micronoyau. Ce dernier assure l'ordonnancement et l'exécution des tâches temps-réel du niveau applicatif. Il est implémenté en C avec des sections qui sont codées en assembleur afin d'assurer la portabilité avec les différentes architectures matérielles. Le micronoyau dispose de deux implémentations, une version destinée pour une architecture mono-cœur et une deuxième implémentation dédiée à une architecture multi-cœur. Dans la version mono-cœur, l'exécution du micronoyau est séquentielle. C'est-à-dire, à tout moment, le micronoyau exécute soit une tâche soit l'ordonnanceur. La version multi-cœur de PharOS

est dédiée à une architecture multi-cœur asymétrique, c'est-à-dire une architecture composée de cœurs hétérogènes. Dans notre cas, un cœur est dédié à la gestion des listes de tâches, appelé cœur de contrôle (« *control core* », abrégé *CC*) et un (ou plusieurs) cœur(s) d'exécution, appelé(s) cœur(s) d'exécution (« *executing core(s)* », abrégé *XC*) dont le rôle principal est l'exécution des tâches du niveau applicatif. Le nombre de cœurs d'exécution est adaptable en fonction de l'architecture matérielle. Dans la suite du travail, afin de simplifier la présentation nous considérons l'implémentation multi-cœur asymétrique composée d'un cœur d'exécution et d'un cœur de contrôle.

Le micronoyau utilise deux *timers* : « *cadence* » et « *overrun* ». Ces timers sont utilisés afin de respecter les contraintes temps-réel des tâches PharOS. Le micronoyau peut recevoir soit une interruption venant d'un timer soit un appel système venant d'une tâche. Dans ce dernier cas, il sauvegarde le contexte de la tâche en cours d'exécution (s'il en existe) et bascule en mode d'exécution d'une routine d'interruption ou d'un appel système.

Le niveau applicatif de PharOS est composé d'un ensemble de tâches créées statiquement. Dans ce qui suit, nous présentons le modèle de tâche dans PharOS et la structure et le fonctionnement de ce dernier.

6.2.2 Description et modèle des tâches PharOs

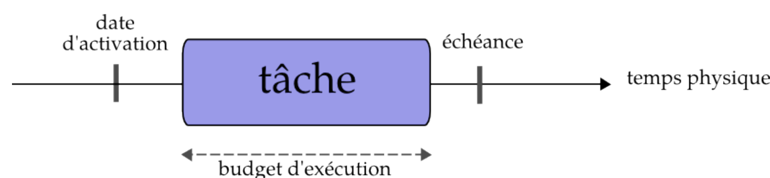


FIGURE 6.2 – Exécution d'une tâche PharOS

Étant donné que le micronoyau PharOS implémente une approche cadencée par le temps [KB03] (« *time-triggered* » en anglais), les tâches initient et finissent leur exécution à des instants prédéterminés (voir figure 6.2).

Une tâche PharOS est caractérisée par un contexte d'exécution qui définit des données caractérisant la tâche en question. Le contexte d'une tâche est implémenté par une structure de données contenant trois paramètres :

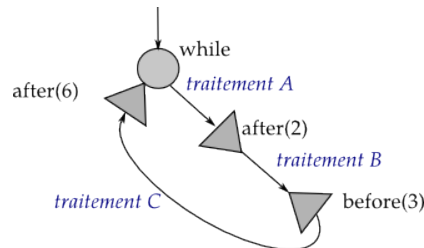
- une date d'activation : la date à laquelle une tâche peut commencer son exécution ;
- un budget d'exécution : la durée d'exécution maximale d'une tâche ;
- une échéance : l'instant auquel l'exécution d'une tâche doit être terminée.

Une application PharOS est composée de plusieurs tâches. Les traitements effectués par une tâche ainsi que les contraintes temporelles sont spécifiés à travers un modèle sous forme d'automate orienté [LDaGVN10]. Les arcs de l'automate définissent les différents

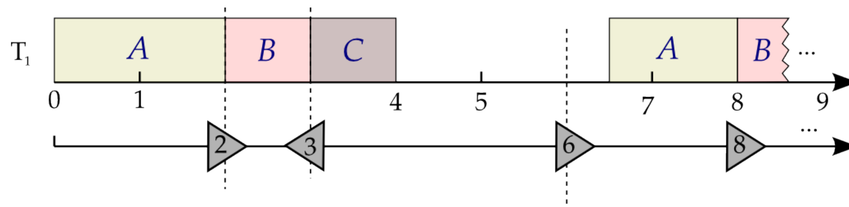
traitements à exécuter et les nœuds expriment des contraintes temporelles sur l'exécution de ces traitements. L'automate est généré à partir d'un langage spécifique appelé ΨC (« *Parallel SYnchronous C* »), inspiré du C, permettant de spécifier des contraintes temporelles. Parmi les instructions ΨC, citons l'instruction *after(d)* qui spécifie que le traitement qui suit l'instruction *after* ne peut s'exécuter par la tâche courante qu'après la date d. L'instruction *before(d)* spécifie que le traitement qui précède l'instruction *before* doit s'exécuter par la tâche courante avant la date d. À travers les contraintes temporelles et les constructeurs du langage C, ΨC permet d'exprimer une exécution périodique d'une tâche en modifiant dynamiquement ses contraintes temporelles.

```
1     while(1) {
2       A: // Executing job A
3       after(2);
4       B : // Executing job B
5       before(3);
6       C: // Executing job C
7       after(6);
8     }
```

(a) Code ΨC de T



(b) Automate correspond au code ΨC



(c) Exemple d'une exécution correcte de l'automate 6.3b

FIGURE 6.3 – Spécification d'une tâche dans PharOS

Exemple 6.2.1. *Nous illustrons dans la figure 6.3a un exemple de code ΨC d'une tâche T composée de trois traitements A, B et C exécutés de manière itérative. L'automate correspondant est donné par la figure 6.3b. Les instructions ΨC sont transformées en des nœuds (spécifiques sous formes de triangles) qui expriment des contraintes temporelles sur les traitements composant la tâche. Nous présentons dans la figure 6.3c une exécution possible de T respectant les contraintes temporelles spécifiées par l'automate. Initialement, la tâche peut exécuter le traitement A. Le traitement B doit être initié après la date 2 et s'achever avant la date 3. Le traitement C s'exécute après B. Dans la première itération de la boucle* `while`*, la tâche T a une date d'activation égale à 0 et une échéance égale à 6. À la rencontre du nœud* `while`*, l'exécution du traitement A est reprise, mais seulemnt après la date 6 et le processus continue. La date d'activation de la tâche est alors repoussée à la date 6 et son échéance à 12.*

Initialement, les tâches sont créées statiquement, c'est-à-dire que les contextes d'exécution des tâches sont statiquement alloués dans la mémoire et les dates d'activation et les échéances sont initialement spécifiées par le développeur du système.

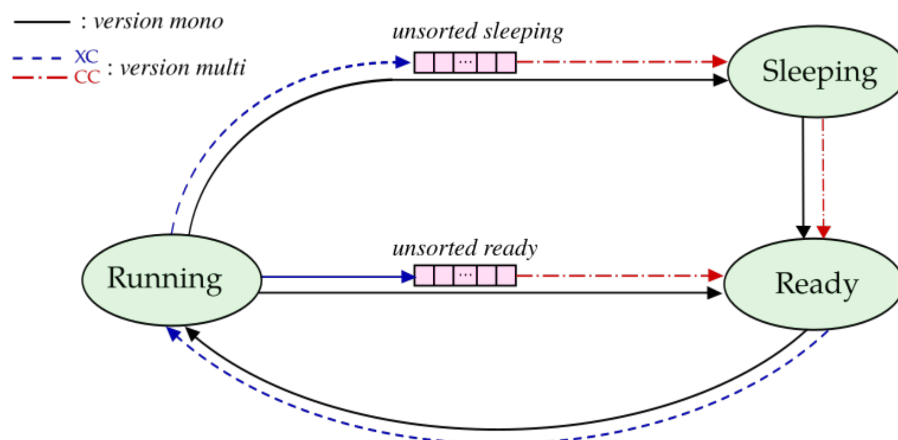


FIGURE 6.4 – Listes des tâches PharOS

Pour ordonnancer l'exécution des tâches, le micronoyau organise ces dernières à travers des listes. La figure 6.4 illustre les différentes listes des tâches gérées par PharOS.

- La liste *ready* stocke les contextes des tâches prêtes, c'est-à-dire les tâches dont la date d'activation est atteinte. Cette liste est triée selon les échéances des tâches. L'insertion d'un contexte dans cette liste se fait en gardant la liste triée selon un ordre croissant des échéances des tâches et le retrait se fait en tête de liste.

- La liste *sleeping* contient les contextes des tâches endormies, c'est-à-dire les tâches dont la date d'activation n'est pas encore atteinte. Cette liste est triée selon les dates d'activation des tâches. L'insertion d'un contexte d'une tâche dans cette liste se fait en respectant un ordre croissant de la date d'activation des tâches et le retrait se fait en fin de liste.

- La tâche en exécution est définie par une variable globale, appelée `ctx_current`, dans laquelle le micronoyau mémorise l'adresse du contexte de la tâche en exécution. L'échéance de la tâche en exécution est la plus proche parmi les tâches prêtes.

Si aucune tâche n'est prête, une tâche spéciale « idle » est exécutée. Cette tâche est définie initialement par la constante `CTX_IDLE`. Elle spécifie que le micronoyau est en attente de la prochaine date d'activation d'une nouvelle tâche.

Pour la version multi-cœur, l'insertion dans *ready* et *sleeping* se fait en passant par des tampons intermédiaires bornés appelés *unsorted ready* et *unsorted sleeping*. Ces derniers sont implémentés à travers des listes FIFO, le retrait d'un contexte se fait en tête de liste et l'insertion en fin de liste. L'utilisation de ces listes intermédiaires permet de réduire le temps de réquisition des listes *ready* et *sleeping* par l'un des cœurs. Le retrait et l'insertion d'un contexte dans l'une des listes se font en un temps constant ($O(1)$).

Toutes les listes sont stockées dans la mémoire partagée entre les cœurs. Ces derniers

insèrent et retirent des tâches dans les listes de manière concurrente. Pour synchroniser l'accès concurrent entre les deux cœurs, PharOS utilise le mécanisme de *spinlock* (introduit dans la section 4.1.2.2) avec l'instruction matérielle « *Compare-and-Swap* ». Le processus qui rentre dans la section critique prend le verrou en modifiant la valeur du *spinlock*, et le libère à la fin de la section ; les autres bouclent en lisant la variable et attendent que le verrou se libère.

L'exécution d'une routine d'interruption permet la mise à jour des listes des tâches. Par exemple, l'exécution de la routine `syscall()` permet d'insérer le contexte courant dans l'une des listes *sleeping* ou *ready*. Pour la version multi-cœur, l'insertion se fait dans l'une des listes non triées (*unsorted*). Le choix d'insertion dans l'une des listes est indéterministe.

Nous détaillons dans la section suivante les différentes routines d'interruption de PharOS.

6.2.3 Gestion des interruptions dans PharOS

Afin d'assurer un ordonnancement préemptif, le micronoyau PharOS gère des interruptions de différents types : matérielles, logicielles et des exceptions. Ces interruptions permettent de préempter la tâche en exécution afin d'appeler l'ordonnanceur. Chaque interruption provoque l'exécution d'une routine d'interruption (une fonction C). L'exécution de ces routines se fait de manière atomique, c'est-à-dire que les deux cœurs ne peuvent pas être interrompus au cours du traitement d'un événement pour gérer un autre événement. Ces routines sont décrites dans les paragraphes suivants.

6.2.3.1 Interruption du timer « *cadence* »

À l'expiration du timer « *cadence* », une interruption est envoyée. La routine d'interruption associée est appelée `rcvCadence()`. Dans la version multi-cœur, cette routine est exécutée par le cœur de contrôle.
L'exécution de `rcvCadence()` consiste en les étapes suivantes :

(i) insérer les tâches contenues dans la liste *sleeping* et dont la date d'activation est atteinte dans la liste *ready*. Pour la version multi-cœur, cette opération est précédée par l'action qui consiste à insérer toutes les tâches de *unsorted sleeping* dans *sleeping* et toutes les tâches de *unsorted ready* dans *ready* ;

(ii) dans la version multi-cœur, le cœur de contrôle envoie une notification (interruption inter-processeur) au cœur d'exécution, pour lui signaler de la mise à jour de la liste *ready* (suite à l'exécution de l'opération précédente).

(iii) si la liste *sleeping* est non vide, mettre à jour le timer « *cadence* » avec la date à laquelle il doit se réveiller ultérieurement, à savoir la date d'activation de la tâche en tête de la liste *sleeping* ;

(iv) retirer et exécuter la tâche en tête de la liste *ready*, si son échéance est plus proche que celle de la tâche courante. Dans ce cas, la tâche courante est insérée dans *ready*. Pour la version multi-cœur cette dernière insertion se fait dans *unsorted ready*. Sinon, rendre la main à la tâche courante.

6.2.3.2 Exception du timer « *overrun* »

Pour assurer la sécurité dans PharOS, le micronoyau utilise un timer appelé « *overrun* » pour vérifier en permanence que la tâche en exécution ne dépasse pas son échéance. Quand le timer expire, une exception est envoyée au micronoyau qui cesse immédiatement son exécution.

6.2.3.3 Interruptions logicielles

À tout moment pendant son exécution, une tâche peut envoyer une interruption logicielle (ou appel système), que nous appelons « *syscall* ». À l'issue de cette interruption, le micronoyau appelle la routine `syscall()`, qui prend en paramètre soit le type `datTard`, soit le type `pretRepos`.

Il est à noter que le modèle d'une tâche PharOS contient des instructions ΨC permettant à la tâche de reprendre son exécution après émission d'un « *syscall* », c'est-à-dire que sa date de début et/ou son échéance sont repoussées.

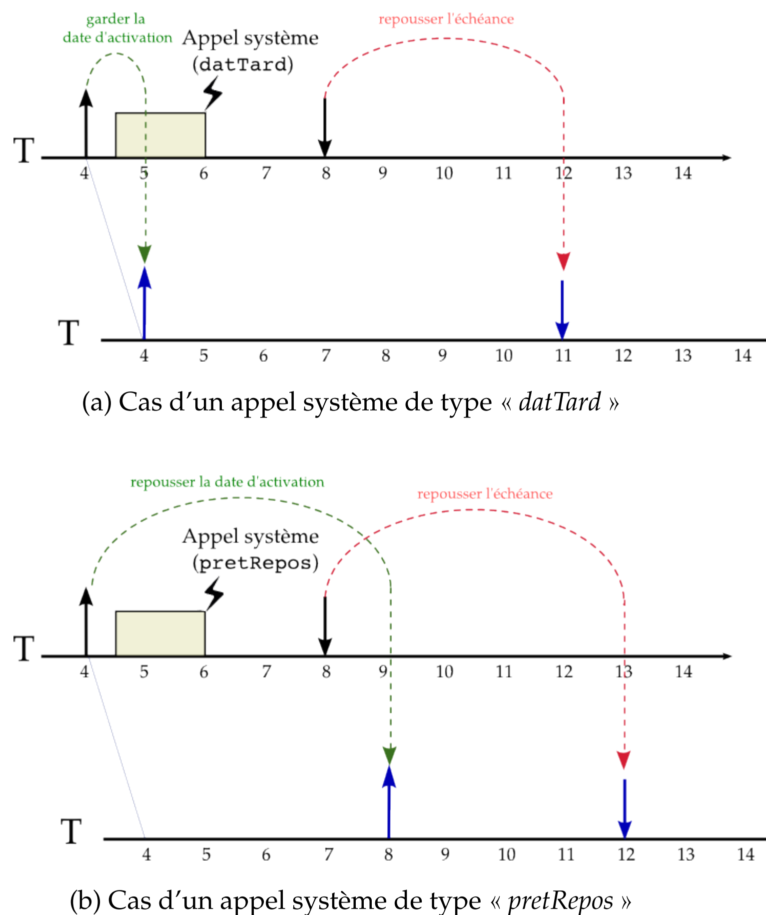
Nous illustrons dans la figure 6.5, la mise à jour du contexte d'une tâche, par le niveau applicatif de PharOS, suite à un appel système. Initialement, la tâche T a le contexte d'exécution suivant : $[t_{act} = 4, t_{dead} = 8]$. À $t = 6$, T émet un appel système de type « *datTard* ». Suite à cet événement (voir figure 6.5a), la tâche exécute un code ΨC qui repousse l'échéance de la tâche. La date d'activation reste inchangée. Le nouveau contexte de T est tel que $[t'_{act} = 4, t'_{dead} = 11]$. La valeur avec laquelle l'échéance de la tâche T est repoussée est définie par le développeur de l'application.

Dans la figure 6.5b, la tâche T fait un appel système de type « *pretRepos* ». Sa date d'activation est repoussée à la date de son échéance initiale, à savoir la date 8. Son échéance est repoussée à une date d'_{dead} telle que $d'_{dead} = d'_{act} + (d_{dead} - d_{act})$. Le nouveau contexte de T est tel que $[t'_{act} = 8, t'_{dead} = 12]$.

Nous décrivons dans la suite le comportement du micronoyau en fonction du type de l'appel « *syscall* » émis par la tâche.

(a) Routine `syscall(datTard)`**.** Cette routine insère le contexte de la tâche courante dans la liste *ready*. Pour la version multi-cœur, cette insertion se fait dans *unsorted ready*. Par la suite, elle retire une tâche à exécuter parmi les tâches prêtes en activant le timer « *overrun* ». Ce timer sera alors activé jusqu'à l'échéance de la tâche choisie pour s'exécuter.

(b) Routine `syscall(pretRepos)`**.** Cette routine consiste à vérifier si la liste *sleeping* est vide (*unsorted sleeping* pour la version multi-cœur) ou si l'échéance de la tâche



(a) Cas d'un appel système de type « *datTard* »

(b) Cas d'un appel système de type « *pretRepos* »

FIGURE 6.5 – Mise à jour d'un contexte de tâche suite à un appel système

courante est postérieure à celle située en tête de la liste *sleeping*. Si c'est le cas, le micronoyau active le timer « *cadence* ». Il insère ensuite la tâche courante dans la liste *sleeping* (*unsorted sleeping* pour la version multi-cœur). Enfin, il retire une nouvelle tâche de la liste *ready* et pour l'exécuter que le cœur d'exécution.

6.2.3.4 Interruptions inter-processeurs

Ces interruptions sont utilisées dans la version multi-cœur afin de gérer la communication entre le cœur de contrôle et le cœur d'exécution. La figure 6.6 illustre le schéma de la communication (niveau code) par des interruptions inter-processeurs entre les deux cœurs. Il existe deux types d'interruption inter-processeurs : « *syncPreemption* » et « *syncSortList* », qui appellent respectivement les routines `handlePreemptionNotif()` et `handleSortListNotif()`.

(a) La routine `handlePreemptionNotif()`. Cette routine est gérée par le cœur d'exécution, quand celui-ci reçoit une interruption « *syncPreemption* » de la part du cœur CC. L'interruption « *syncPreemption* » est déclenchée quand le cœur de contrôle exécute

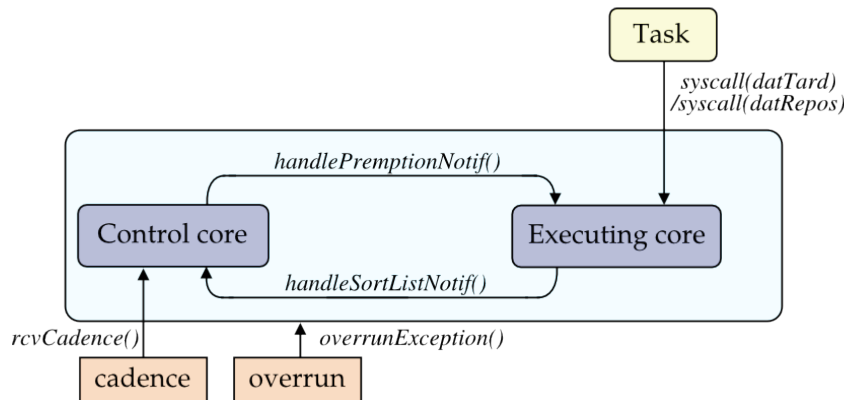


FIGURE 6.6 – Interruptions inter-processeurs dans PharOS (version multi-cœur)

la routine d'interruption `rcvCadence()`. Elle permet de réinsérer le contexte de la tâche courante dans la liste *unsorted ready* et retire une nouvelle tâche de la liste *ready* pour l'exécuter.

(b) La routine `handleSortListNotif()`**.** Cette routine est gérée par le cœur de contrôle quand le cœur d'exécution lui envoie une interruption « *syncSortList* ». Cette interruption est déclenchée par le *XC* lors de l'exécution de la routine `syscall()`, plus particulièrement quand le contexte courant est inséré dans une des listes non triées. La routine `handleSortListNotif()` permet de mettre à jour toutes les listes des tâches. Elle place toutes les tâches de *unsorted sleeping* dans *sleeping* et toutes les tâches de *unsorted ready* dans *ready*. Elle place ensuite les tâches de la liste *sleeping* et dont la date d'activation est atteinte dans la liste *ready*. Enfin, elle met à jour le timer « *cadence* » avec la date d'activation de la tâche en tête de la liste *sleeping*.

Exemple 6.2.2 (Scénario d'exécution de PharOS dans la version mono-cœur)**.** *Considérons initialement trois tâches T_1, T_2 et T_3 ayant les contextes ci-dessous, avec t_{act} désignent la date d'activation de la tâche et t_{dead} son échéance.*

- $T_1 = [t_{act} = 4, t_{dead} = 9]$
- $T_2 = [t_{act} = 3, t_{dead} = 10]$
- $T_3 = [t_{act} = 8, t_{dead} = 12]$

Le budget d'exécution est de 2 unités de temps pour toutes les tâches. Nous supposons qu'au démarrage, la tâche T_1 est en exécution. La liste ready contient T_2 et la liste sleeping contient T_3. Nous illustrons dans la figure 6.7 un scénario d'exécution de ces trois tâches dans PharOS.

- *À $t = 6$, la tâche T_1 envoie une interruption logicielle « syscall(pretRepos) » au micronoyau. Le contexte de T_1 est mis à jour (voir principe décrit dans la section 6.2.3.3). La date de début et l'échéance de T_1 sont alors repoussées, tel que $[t'_{act} = 9, t'_{dead} = 14]$.*

- *Le micronoyau reçoit l'interruption et exécute la routine associée, à savoir* `syscall(pretRepos)`*. À l'issue de cette exécution, le timer* cadence *est activé*

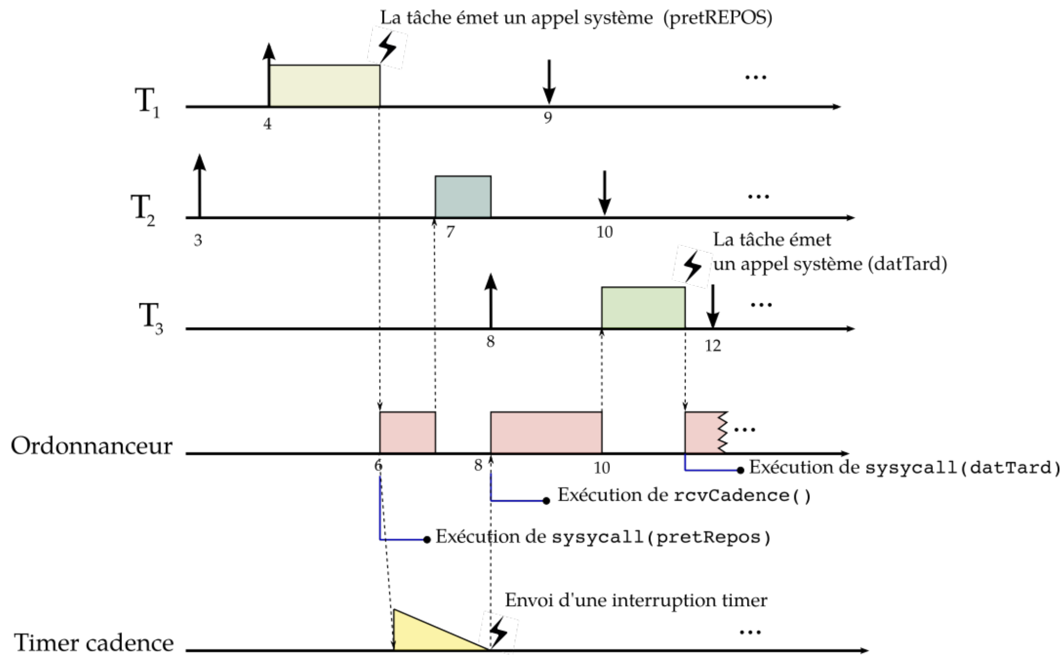


FIGURE 6.7 – Exemple de scénario d'exécution dans PharOS version monoprocesseur

en lui affectant la valeur à laquelle la première tâche dans sleeping (T_3) sera activée, à savoir la valeur 2. Le micronoyau place ensuite le contexte de la tâche courante dans la liste sleeping, et retire un nouveau contexte de tâche de la liste ready pour l'exécuter.

- *À $t = 7$, la tâche T_2 s'exécute. Quand le timer* cadence *expire, il envoie une interruption. Le micronoyau suspend l'exécution de la tâche T_2 pour exécuter la routine d'interruption provoquée par le timer, à savoir* `rcvCadence()`*. Cette routine place toutes les tâches de sleeping et dont la date d'activation est atteinte dans ready. T_3 est alors insérée dans ready.*

- *À $t = 10$, la routine* `rcvCadence()` *retire le contexte à la tête de la liste ready, à savoir la tâche T_3 et remet la tâche T_2 dans la ready.*

- *Et le processus d'exécution se poursuit.*

Exemple 6.2.3 (Scénario d'exécution de PharOS dans la version multi-cœur)**.** *Soient trois tâches T_1, T_2 et T_3 avec les contextes ci-dessous :*

- $T_1 = [t_{act} = 4, t_{dead} = 9]$
- $T_2 = [t_{act} = 3, t_{dead} = 12]$
- $T_3 = [t_{act} = 8, t_{dead} = 16]$

Nous supposons qu'au démarrage, la tâche T_1 est en exécution. La liste ready contient T_2 et la liste sleeping contient T_3. Les deux listes unsorted sont vides. Le scénario que nous considérons est donné par la figure 6.8.

- *À $t = 5$, la tâche T_1 émet un appel système de type «* pretRepos *», qui sera traitée par le*

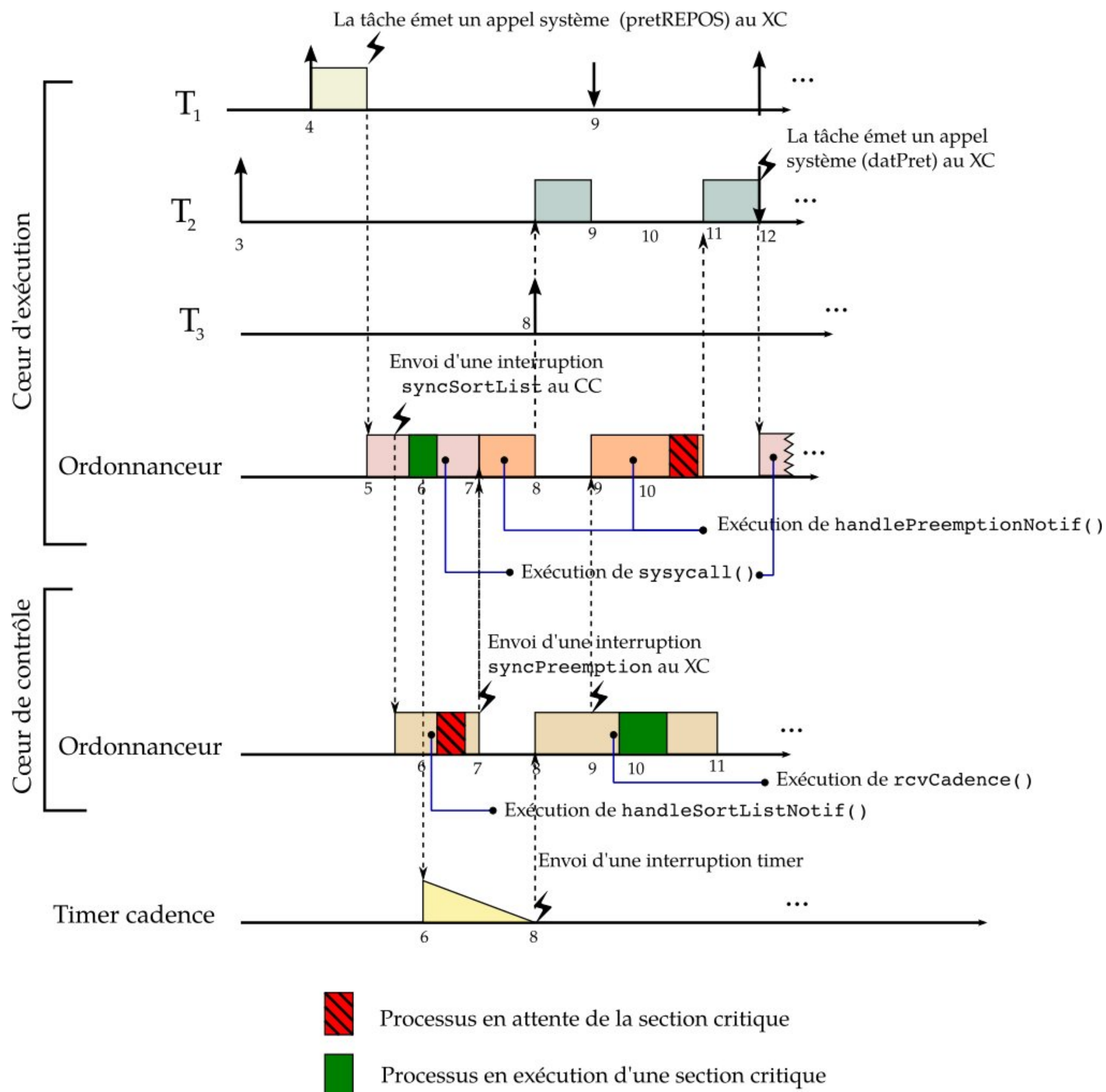


FIGURE 6.8 – Exemple de scénario d'exécution dans PharOS version multi-cœur

XC. Le niveau applicatif met à jour le contexte de T_1 (selon le principe décrit dans la section 6.2.3.3). Le nouveau contexte de T_1 est tel que $[t'_{act} = 9, t_{dead} = 14]$.

- *Le XC exécute la routine* `syscall(pretRepos)` *qui envoie une interruption «* syncSortList *» pour réveiller le CC. À $t = 6$, la routine* `syscall()` *met à jour le timer* cadence *en lui affectant la valeur à laquelle la première tâche dans sleeping (T_3) sera activable, à savoir la valeur 2. La tâche T_1 est ensuite insérée dans la liste unsorted sleeping.*

- *Entre $t = 5$ et $t = 6$ le XC reçoit l'interruption «* syncSortList *» et appelle la routine*

`handleSortListNotif()`.

- *Entre $t = 6$ et $t = 7$, les deux cœurs accèdent mutuellement à la liste ready. Comme l'accès aux listes est géré par des spinlock, le CC est mis en attente (section colorée en rouge) jusqu'à la libération de la section critique de la part du cœur d'exécution.*

- *À $t = 7$, le CC envoie une interruption «* syncPreemption *» à XC. À la réception de cette interruption, le cœur d'exécution exécute la routine* `handlePreemptionNotif()`. *Cette routine retire le contexte à la tête de la liste ready, à savoir la tâche T_2, qui va prendre la main à $t = 8$. À cet instant, le cœur de contrôle reçoit l'interruption «* cadence *» et exécute la routine associée, à savoir* `rcvCadence()`.

- *À $t = 9$, CC met à jour les listes unsorted sleeping, sleeping et ready. Les tâches T_3 et T_1 sont placées (selon leur échéance) dans la liste ready, vu que leur date d'activation est atteinte. CC envoie ensuite une interruption «* syncPreemption *» à XC.*

- *À la réception de l'interruption «* syncPreemption *» ($t = 9$), XC exécute la routine* `handlePreemptionNotif()`. *Il met à jour le budget d'exécution de T_2, qui a été interrompue (à $t = 9$). À $t = 10$, XC se met en attente pour accéder à la liste ready. Quand CC libère la section critique, XC accède à la liste ready pour vérifier s'il y a des tâches dont l'échéance est plus proche que la tâche T_2. Comme la tâche T_2 a l'échéance la plus proche et elle lui reste une unité de temps dans son budget d'exécution, elle reprend alors la main à $t = 11$. À $t = 12$, T_2 émet un appel système de type «* datTard *» et le processus d'exécution se poursuit.*

6.3 Modélisation TLA+ du micronoyau mono-cœur

Dans la version mono-coeur de PharOS, à tout moment le processeur exécute soit une tâche soit le code de l'ordonnanceur. Il est à noter que le comportement du micronoyau a un caractère indéterministe. En effet, la tâche en exécution peut, à tout moment faire un appel système. Le choix du type d'appel système (« datTard » ou « preRepos ») est indéterministe. Elle peut même dépasser son échéance. Aussi, la durée d'exécution de la tâche influence l'état des autres tâches contenues dans le système. Le comportement indéterministe d'une tâche a alors des conséquences sur le comportement du micronoyau.

Dans cette section, nous décrivons le modèle d'exécution utilisé pour la modélisation du micronoyau mono-cœur. Nous détaillons comment nous avons spécifié le comportement du micronoyau et son interaction avec l'environnement. À partir de la spécification TLA+, nous spécifions l'ensemble de propriétés à vérifier sur celle-ci. Nous discutons les résultats de vérification obtenus avant et après application de la technique de réduction proposée dans le chapitre précédent.

6.3.1 Modèle d'exécution

Pour modéliser le micronoyau PharOS en utilisant C2TLA+, nous étendons le modèle d'exécution décrit dans le chapitre 4. Les fonctions C du micronoyau faisant intervenir le matériel sont déclarées à travers le mécanisme des attributs et elles sont modélisées manuellement en TLA+, comme par exemple la fonction C qui active ou désactive un timer. Le code C du micronoyau est passé en entrée à C2TLA+ qui génère les fichiers TLA+ correspondant. Des nouveaux modules TLA+ sont ensuite intégrés avec les modules générés. Ces derniers modélisent les aspects non définis par le code C, comme les primitives de gestion des interruptions. Le micronoyau et son environnement sont modélisés par la formule $Spec$. Il est à noter que nous ne vérifions pas le niveau applicatif de PharOS. Dans la suite de cette section, nous décrivons comment nous avons étendu le modèle d'exécution pour modéliser l'ordonnanceur PharOS et son environnement.

Pour modéliser les timers, nous avons défini une nouvelle variable TLA+ appelée $timers$. Cette variable est représentée sous la forme d'un tableau à deux éléments dont chaque élément présente un timer. L'index de chaque élément est l'identifiant du timer, à savoir "cadence" ou "overrun".

Pour la gestion des interruptions, nous avons ajouté dans la variable $memory$ un nouveau champ que nous avons appelé $interrupts$. Ce dernier contient un tableau de booléens, dont chaque élément représente une interruption. Dans notre cas, le tableau $interrupts$ contient trois éléments et il est indexé par les identifiants d'interruptions, à savoir "cadence" (pour l'interruption générée par le timer « *cadence* »), "overrun" (pour l'interruption générée par le timer « *overrun* »), et l'identifiant "syscall" pour l'interruption générée par une tâche en exécution.

Le temps-réel est modélisé par la variable now dont le principe est énoncé dans le chapitre 4. La variable now est considérée comme étant l'horloge du système. Nous rappelons que nous ne vérifions pas les propriétés temps-réel des tâches, à savoir le respect des échéances des tâches. L'introduction de la variable now nous permet de vérifier un ensemble de propriétés fonctionnelles que nous décrivons dans la section 6.3.3.

L'évolution du temps et des timers se fait en entrelacement avec l'exécution du micronoyau. À tout moment, le micronoyau peut exécuter soit une routine d'interruption soit une tâche du niveau applicatif. Nous considérons le processus ordonnanceur avec l'identifiant "sched", un processus "idle" et pour chaque tâche du niveau applicatif l'identifiant "t1", "t2", ..., "tn" . Tous ces processus partagent la mémoire globale $data$ et chacun dispose de sa propre mémoire locale. Les identifiants de ces processus sont donnés par la constante $ProcSet$. La mémoire $data$ contient les variables utilisées par le code C du micronoyau, comme les adresses des listes des tâches et les contextes des tâches. Chaque processus contient une mémoire locale contenant une pile de registres $register$, une pile d'exécution $stack$ et un registre pour stocker le dernier retour de fonction ret.

L'opérateur $procsExec()$ définit l'ensemble des processus activables. Dans la version

mono-cœur, un seul processus à un instant donné peut s'exécuter sur le processeur.

Les codes C des routines d'interruptions `syscall()` et `rcvCadence()` contiennent des instructions faisant intervenir le matériel comme les primitives d'activation et de désactivation d'un timer. Comme mentionné précédemment, ces instructions sont directement spécifiées en TLA+.

6.3.2 Spécification TLA+ du comportement du micronoyau

Le micronoyau PharOS assure l'ordonnancement des tâches temps-réel. Sa spécification TLA+ complète est donnée par la formule :

$$Spec \triangleq Init \wedge \Box[Next]_{\langle memory, timers, now \rangle} \wedge \mathrm{WF}_{\langle memory, timers, now \rangle} Next$$

Nous détaillons dans la suite de cette section la définition du prédicat $Init$ ainsi que l'action $Next$ qui spécifie les transitions du micronoyau.

6.3.2.1 Définition du prédicat $Init$

Le prédicat $Init$ initialise les variables du système, à savoir now, $memory$ et $timers$. Le listing 6.1 illustre la définition de ce prédicat.

$$\begin{array}{l} Init \triangleq \wedge\ memory = [data \mapsto [(0 :> [val \mapsto 0]\ @@\ 1 :> [val \mapsto 0] \ldots)\\ \qquad register \mapsto (\text{“sched”} :> \langle\rangle\ @@\\ \qquad\qquad \text{“t1”} :> \langle [pc \mapsto \langle \text{“task_init”} \rangle, fp \mapsto 1] \rangle @@\\ \qquad\qquad \text{“t2”} :> \langle [pc \mapsto \langle \text{“task_init”} \rangle, fp \mapsto 1] \rangle @@ \ldots @@\\ \qquad\qquad \text{“idle”} :> \langle [pc \mapsto \langle \text{“idle_init”} \rangle, fp \mapsto 1] \rangle),\\ \qquad stack \mapsto [i \in ProcSet \mapsto \langle\rangle],\\ \qquad ret \mapsto [i \in ProcSet \mapsto [val \mapsto Undef]],\\ \qquad interrupts \mapsto (\text{“overrun”} :> \text{FALSE}\ @@\ \text{“cadence”} :> \text{TRUE}\ @@\ \text{“syscall”} :> \text{FALSE}),\\ \qquad syscalltype \mapsto Null,\\ \qquad nsyscall \mapsto [i \in tasks \mapsto 0]]\\ \quad \wedge\ timers = (\text{“overrun”} :> 2\ @@\ \text{“cadence”} :> 4)\\ \quad \wedge\ now = 0 \end{array}$$

Listing 6.1 – Définition TLA+ du prédicat $Init$ de la version mono-cœur

Initialement, la mémoire globale $data$ contient les initialisations des variables statiques du système, à savoir le nombre de tâches, le contexte de chacune des tâches, l'adresse des listes $ready$ et $sleeping$ et l'adresse du contexte de la tâche courante. Nous supposons que le micronoyau à l'état initial a une tâche en exécution, dont l'identifiant est “t1”. Les piles $stack$ et $register$ de tous les processus sont vides. Le registre ret contient la valeur $Null$ pour tous les processus. Le champ $interrupts$ contient la valeur FALSE pour toutes les interruptions sauf pour « *cadence* ». Le champ $syscalltype$ contient la valeur $Null$. Cette valeur est modifiée par une tâche quand elle émet une interruption de type « *syscall* ». Le champ $nsyscall$ stocke le nombre d'appels système de chaque tâche du système. Nous détaillons l'utilité de ce champ dans la section 6.3.2.4. Le timer « *overrun* » contient la date d'activation minimale des tâches contenues dans *sleeping* (ici elle vaut 2) et le timer

« *cadence* » contient la valeur de l'échéance de la tâche ayant l'identifiant "t1" (la tâche en exécution).

Le prédicat $Next$ spécifie toutes les transitions possibles du système. Sa définition est illustrée par le listing 6.2.

$$\begin{array}{l} Next \triangleq \lor \\ \quad\quad \land\ now < MaxTime \\ \quad\quad \land\ \lor HardwareNext(memory, timers) \\ \quad\quad\quad \lor\ kernel \\ \quad \lor \\ \quad\quad \land\ now \geq maxTime \\ \quad\quad \land\ \text{UNCHANGED } \langle now, timers, memory \rangle \end{array}$$

Listing 6.2 – Spécification TLA+ de l'action $Next$

L'action $Next$ définit le comportement du micronoyau ainsi que son environnement. Cette action est exprimée en une disjonction d'actions :

- La première action est exécutable si le temps now n'a pas encore atteint la valeur maximale $MaxTime$ définie par l'utilisateur. Cette action est exprimée par la disjonction de deux actions : l'action $HardwareNext()$ qui simule les aspects matériels du système et l'action $kernel()$ qui spécifie le comportement du micronoyau PharOS. Les deux actions $HardwareNext()$ et $kernel()$ sont décrites dans les paragraphes suivants.

- La deuxième action permet de borner le temps d'exécution de TLC. Quand la variable now atteint la borne maximale $MaxTime$, les variables du système restent inchangées.

Nous rappelons que la définition d'une borne maximale $MaxTime$ pour la variable now permet d'éviter que le *model-checker* s'exécute indéfiniment.

6.3.2.2 Définition de l'action $HardwareNext()$

$$\begin{array}{l} HardwareNext(mem, tims) \triangleq \\ \quad \land\ now' = now + 1 \\ \quad \land\ timers' = updateTimers(tims) \\ \quad \land\ memory' = interruptRequest(mem, tims) \end{array}$$

Listing 6.3 – Spécification TLA+ de l'action $HardwareNext()$

La spécification TLA+ de l'action $HardwareNext()$ est définie par le listing 6.3. Elle est modélisée par une conjonction d'actions :

- une action qui fait évoluer le temps en incrémentant la variable now ;

- une action qui met à jour la variable $timers$ en appelant l'opérateur $updateTimers()$ (défini dans le paragraphe (a) de cette section) ;

- une action qui met à jour le champ $interrupts$ contenu dans la variable $memory$. Cette mise à jour est faite par l'opérateur $interruptRequest()$ qui gère le déclenchement des interruptions. Cette action est détaillée dans le paragraphe (b) de cette section.

(a) Définition de l'opérateur $updateTimers()$. Sa spécification TLA+ est donnée par le listing 6.4. Dans cette spécification, un timer est décrémenté automatiquement en fonction du temps. Quand sa valeur devient inférieure ou égale à zéro, il est désactivé en lui attribuant la valeur $Null$.

$$
\begin{array}{l}
updateTimers(tims) \triangleq \\
\quad \text{LET } updateOneTimer(t) \triangleq \text{ IF } tims[t] = Null \text{THEN } tims[t] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE IF } tims[t] \leq 0 \text{THEN } Null \text{ ELSE } tims[t] - (now' - now) \\
\quad \text{IN } [tims \text{ EXCEPT } ![\text{“cadence”}] = updateOneTimer(\text{“cadence”}), \\
\qquad\qquad\qquad\quad ![\text{“overrun”}] = updateOneTimer(\text{“overrun”})]
\end{array}
$$

Listing 6.4 – Spécification TLA+ de la mise à jour des timers

(b) Spécification de l'opérateur $interruptRequest()$. L'envoi d'une interruption par un timer est modélisé par l'opérateur TLA+ $interruptRequest()$ défini dans le listing 6.5.

$$
\begin{array}{l}
interruptRequest(mem, tims) \triangleq \\
\quad \text{LET } sendCadenceOrOverrun(int) \triangleq \text{ IF } tims[int] \neq Null \land tims[int] \leq 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN TRUE ELSE } mem.interrupts[int] \\
\quad \text{IN } [mem \text{ EXCEPT } !.interrupts = [mem.interrupts \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad\qquad\qquad ![\text{“cadence”}] = sendCadenceOrOverrun(\text{“cadence”}), \\
\qquad\qquad\qquad\qquad\qquad\qquad ![\text{“overrun”}] = sendCadenceOrOverrun(\text{“overrun”})]]
\end{array}
$$

Listing 6.5 – Spécification TLA+ de l'opérateur $interruptRequest()$

Cet opérateur spécifie qu'à l'expiration d'un timer $tims[int]$ la valeur de l'interruption associée $mem.interrupts[int]$ est mise à TRUE, signifiant que l'interruption est transmise au micronoyau.

6.3.2.3 Définition de l'action $kernel()$

À tout moment, le micronoyau exécute soit une tâche (dont la tâche « idle »), soit l'ordonnanceur. La spécification du comportement du micronoyau est donnée par l'action $kernel()$ que nous définissons dans le listing 6.6.

La définition de l'action $kernel$ est exprimée comme une disjonction de deux actions :

- La première action spécifie le cas où le micronoyau reçoit une exception « *overrun* ». Dans ce cas, le système arrête son exécution et les variables du système restent inchangées. Il s'agit d'un état où le système reporte une erreur.

- La deuxième action modélise le cas où le micronoyau est en exécution, c'est-à-dire la pile $register[\text{“sched”}]$ est non vide, ou le cas où aucune interruption n'est reçue. Dans ce cas, l'action $executeReadyProcess()$ (voir section 6.3.2.8) est appelée

$$
\begin{array}{l}
kernel \triangleq \\
\quad \lor \\
\qquad \land\ memory.interrupts[\text{"overrun"}] = \text{TRUE} \\
\qquad \land\ \text{UNCHANGED}\ \langle memory, now, timers \rangle \\
\quad \lor \\
\qquad \land\ memory.interrupts[\text{"overrun"}] \neq \text{TRUE} \\
\qquad \land\ \text{IF}\ memory.register[\text{"sched"}] \neq \langle\rangle \lor interruptionsSet(memory) = \{\} \\
\qquad\quad \text{THEN}\ executeReadyProcess(memory) \\
\qquad\quad \text{ELSE}\ \exists\, idInt \in interruptionsSet(memory) : \\
\qquad\qquad\quad \text{LET}\ nmem \triangleq initTaskInterruption(idInt, memory) \\
\qquad\qquad\qquad\quad ntims \triangleq initTimers(idInt, timers) \\
\qquad\qquad\quad \text{IN}\quad executeReadyProcess(nmem, ntims) \\
\qquad \land\ \text{UNCHANGED}\ now
\end{array}
$$

Listing 6.6 – Spécification TLA+ de la gestion des tâches et des interruptions

afin d'exécuter une action d'un processus activable. Sinon, le micronoyau vérifie en permanence le déclenchement d'une interruption « *syscall* » ou « *cadence* », c'est-à-dire l'ensemble des interruptions déclenchées défini par l'opérateur $interruptionsSet()$ (défini dans la section 6.3.2.7) est non vide. Si le micronoyau reçoit une ou plusieurs interruptions, une interruption à traiter $idInt$ est choisie de manière indéterministe. Dans ce cas, le registre et la pile du processus "sched" ainsi que la variable $timers$ sont mises à jour en appelant respectivement l'opérateur $initTaskInterruption()$ et $initTimers()$. L'opérateur $initTaskInterruption()$ permet d'initialiser le registre et la pile du micronoyau afin d'exécuter la routine d'interruption. L'opérateur $initTimers()$ permet d'initialiser les timers. Ces deux derniers opérateurs sont décrits dans les sections suivantes.

6.3.2.4 Initialisation d'une routine d'interruption

La fonction $initTaskInterruption()$ présentée dans le listing 6.7 prend en argument la mémoire mem et l'identifiant d'une interruption inT et permet de mettre à jour les champs de la variable $memory$.

Si l'interruption est de type « *syscall* » ou « *cadence* », le cadre de pile de la routine associée est d'abord empilé dans la pile $stack[\text{"sched"}]$ de l'ordonnanceur. L'identifiant de l'instruction à exécuter et le pointeur de cadre de pile sont ensuite insérés au sommet du registre $mem.register[\text{"sched"}]$. Par la suite, l'élément du champ $interrupts$ associé à l'interruption inT est remis à la valeur FALSE. Le champ $syscalltype$ définit le type de l'interruption « *syscall* » envoyée, est réinitialisé à $Null$.

Les fonctions $updateDatTard()$ et $updatePretRepos()$ permettent de mettre à jour le contexte de la tâche qui a émis l'appel système. Comme décrit dans la section 6.2.3.3, suite à l'envoi d'une interruption « *syscall* » de type « *datTard* », l'échéance de la tâche est repoussée (par le niveau applicatif). Cette opération est simulée par la fonction $updateDatTard()$ qui retourne l'état de la mémoire $data$ après avoir modifié l'échéance de la tâche courante. La fonction $updatePretRepos()$ permet de repousser la date d'activation et l'échéance de la tâche courante.

$$
\begin{array}{l}
initTaskInterruption(inT, mem) \triangleq \\
\quad \text{CASE } inT = \text{“cadence”} \rightarrow [mem \text{ EXCEPT} \\
\qquad\qquad !.register = [mem.register \text{ EXCEPT } ![\text{“sched”}] = \rangle[pc \mapsto \text{“rcvCadence_494”}, fp \mapsto 1]\rangle], \\
\qquad\qquad !.stack = [mem.stack \text{ EXCEPT } ![\text{“sched”}] = mem.stack[\text{“sched”}] \circ \\
\qquad\qquad\; \langle Undef, [loc \mapsto Undef, offs \mapsto Undef], Undef\rangle], \\
\qquad\qquad !.interrupts = [mem.interrupts \text{ EXCEPT } ![inT] = \text{FALSE}]] \\
\qquad \square\; inT = \text{“syscall”} \rightarrow \\
\qquad\qquad \text{IF } mem.syscall = DATARD \text{THEN } [mem \text{ EXCEPT} \\
\qquad\qquad\quad !.data = updateDatTard(load(\text{“unused”}, mem, Addr_ctxcurrent), mem), \\
\qquad\qquad\quad !.register = [mem.register \text{ EXCEPT } ![\text{“sched”}] = \langle[pc \mapsto \text{“syscall_436”}, fp \mapsto 1]\rangle], \\
\qquad\qquad\quad !.stack = [mem.stack \text{ EXCEPT } ![\text{“sched”}] = mem.stack[\text{“sched”}] \circ \langle[val \mapsto 0], \\
\qquad\qquad\qquad [loc \mapsto Undef, offs \mapsto Undef], Undef, [loc \mapsto Undef, offs \mapsto Undef], Undef, Undef, \\
\qquad\qquad\qquad [loc \mapsto Undef, offs \mapsto Undef]\rangle], \\
\qquad\qquad\quad !.interrupts = [mem.interrupts \text{ EXCEPT } ![inT] = \text{FALSE}], \\
\qquad\qquad\quad !.nsyscall = [mem.nsyscall \text{ EXCEPT } ![load(\text{“unused”}, mem, Addr_ctxcurrent)] = @ + 1], \\
\qquad\qquad\quad !.syscalltype = Null] \\
\qquad\qquad \text{ELSE } [mem \text{ EXCEPT} \\
\qquad\qquad\quad !.data = updatePretRepos(load(\text{“unused”}, mem, Addr_ctxcurrent), mem), \\
\qquad\qquad\quad !.register = [mem.register \text{ EXCEPT } ![\text{“sched”}] = \langle[pc \mapsto \text{“syscall_436”}, fp \mapsto 1]\rangle], \\
\qquad\qquad\quad !.stack = [mem.stack \text{ EXCEPT } ![\text{“sched”}] = mem.stack[\text{“sched”}] \circ \\
\qquad\qquad\qquad \langle[val \mapsto 1], [loc \mapsto Undef, offs \mapsto Undef], Undef, [loc \mapsto Undef, offs \mapsto Undef], Undef, \\
\qquad\qquad\qquad Undef, [loc \mapsto Undef, offs \mapsto Undef]\rangle], \\
\qquad\qquad\quad !.interrupts = [mem.interrupts \text{ EXCEPT } ![inT] = \text{FALSE}], \\
\qquad\qquad\quad !.nsyscall = [mem.nsyscall \text{ EXCEPT } ![load(\text{“unused”}, mem, Addr_ctxcurrent)] = @ + 1], \\
\qquad\qquad\quad !.syscalltype = Null] \\
\qquad \square\; \text{OTHER} \rightarrow \text{“ERROR”}
\end{array}
$$

Listing 6.7 – Spécification TLA+ de l'initialisation des routines d'interruptions

Nous rappelons que le champ $nsyscall$ indique le nombre d'appels système émis par une tâche. À chaque émission d'un appel système ce nombre est incrémenté de 1. Nous expliquons dans la section 6.3.2.9, l'utilité de ce champ.

6.3.2.5 Initialisation des timers

La spécification de l'opérateur $initTimers()$ est donnée par le listing 6.8. Cet opérateur prend en argument un identifiant d'une interruption $idInt$ déclenchée par un timer et un argument $tims$ désignant la variable $timers$. Il permet d'initialiser le timer associé à l'interruption $idInt$ en mettant la valeur de celui-ci à la valeur $Null$.

$$
\begin{array}{l}
initTimers(idInt, tims) \triangleq \\
\quad \text{CASE } idInt = \text{“cadence”} \rightarrow [tims \text{ EXCEPT } ![idInt] = Null] \\
\qquad \square\; idInt = \text{“overrun”} \rightarrow [tims \text{ EXCEPT } ![idInt] = Null] \\
\qquad \square\; \text{OTHER} \rightarrow tims
\end{array}
$$

Listing 6.8 – Spécification TLA+ de l'initialisation des timers

6.3.2.6 Définition de l'opérateur $procsExec()$

L'opérateur $procsExec()$ retourne l'ensemble des processus activables. Sa définition est illustrée par le listing 6.9.

$$
\begin{array}{l}
procsExec(mem) \triangleq \text{IF } mem.register[\text{"sched"}] \neq \langle\rangle \text{THEN } \{\text{"sched"}\} \\
\quad \text{ELSE LET } currentTask \triangleq load(\text{"unused"}, mem, Addr_ctxcurrent) \\
\quad\quad \text{IN CASE } current = load(\text{"unused"}, memory, Addr_CTX_IDLE) \rightarrow \{\text{"idle"}\} \\
\quad\quad\quad \square\ current = Addr_c1 \rightarrow \{\text{"t1"}\} \\
\quad\quad\quad \square\ current = Addr_c2 \rightarrow \{\text{"t2"}\} \\
\quad\quad\quad \square\ \ldots \\
\quad\quad\quad \square\ current = Addr_cn \rightarrow \{\text{"tn"}\} \\
\quad\quad\quad \square\ \text{OTHER} \rightarrow \text{"ERROR"}
\end{array}
$$

Listing 6.9 – Spécification TLA+ de l'opérateur $procsExec(mem)$

Si la pile $register[\text{"sched"}]$ est non vide puisque le processus "sched" est plus prioritaire que les autres processus, alors l'opérateur $procsExec()$ retourne l'identifiant "sched". Sinon, il retourne l'identifiant de la tâche en exécution dont l'adresse de son contexte est contenue dans la case mémoire donnée par $load(\text{"unused"}, mem, Addr_ctxcurrent)$.

6.3.2.7 Définition de la fonction $interruptionsSet()$.

$$interruptionsSet(mem) \triangleq \{id \in \text{"syscall"}, \text{"cadence"} : mem.interrupts[id] = \text{TRUE}\}$$

Listing 6.10 – Spécification TLA+ de l'ensemble des interruptions déclenchées

L'ensemble des interruptions déclenchées est donné par l'opérateur $interruptionsSet()$ dont la définition est illustrée par le listing 6.10. Cet opérateur prend en paramètre la mémoire mem et retourne l'ensemble des identifiants des interruptions envoyées parmi les interruptions « *overrun* » et « *cadence* ». Il est à noter que lorsque le micronoyau reçoit une interruption (exception) « *overrun* », le système cesse son exécution.

6.3.2.8 Définition de l'action $executeReadyProcess()$

$$
\begin{array}{l}
executeReadyProcess(mem) \triangleq \exists\, id \in procsExec(mem) : \\
\quad \lor \land id \in \{\text{"t1"}, \text{"t2"}, \ldots, \text{"tn"}\} \\
\quad\quad \land taskExecution \\
\quad \lor \land id \in \{\text{"sched"}, \text{"idle"}\} \\
\quad\quad \land memory' = dispatch(id, mem) \\
\quad\quad \land timers' = dispatch_timers(id, mem, timers)
\end{array}
$$

Listing 6.11 – Spécification TLA+ de l'exécution d'un processus activable

Cette action choisit un identifiant de processus activable parmi l'ensemble des identifiants retournés par $procsExec()$ et exécute le processus associé. Si le processus activable est une tâche, alors on appelle l'action $taskExecution$ qui simule l'exécution d'une tâche (voir section 6.3.2.9). Sinon, le processus activable est soit l'ordonnanceur soit la tâche "idle". Dans ce cas, les variables $memory$ et $timers$ sont mises à jour en appelant respectivement les opérateurs $dispatch()$ et $dispatch_timers()$. Nous rappelons que $dispatch()$ est l'opérateur généré par C2TLA+ qui modélise le flot de contrôle des instructions C du code du micronoyau. L'opérateur $dispatch_timers()$ quant à lui spécifie comment les timers sont modifiés par les fonctions C appelées par les routines

d'interruptions. Nous expliquons la spécification de cet opérateur dans le paragraphe suivant.

$$
\begin{array}{l}
dispatch(id, mem) \triangleq \\
\quad \text{CASE } mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“syscall_436”} \\
\qquad\qquad \rightarrow stmt_syscall_436(id, mem) \\
\qquad \ldots \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“rcvCadence_494”} \\
\qquad\qquad \rightarrow stmt_rcvCadence_494(id, mem) \\
\qquad \ldots \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“ctx_updateAll_415”} \\
\qquad\qquad \rightarrow stmt_rcvCadence_494(id, mem) \\
\qquad \ldots \\
\qquad \Box\ \text{OTHER} \rightarrow mem
\end{array}
$$

Listing 6.12 – Définition de l'opérateur $dispatch()$ pour la gestion du flot d'exécution du code C généré par C2TLA+

(a) Gestion du flot de contrôle par $dispatch()$. L'opérateur $dispatch()$ permet de gérer le flot d'exécution de l'implémentation C du micronoyau. La définition de cet opérateur s'étend sur plusieurs lignes, nous illustrons alors dans le listing 6.12 un aperçu de la définition de $dispatch()$ générée par C2TLA+. Cet opérateur est appelé par l'action $executeReadyProcess()$ et retourne l'état de la mémoire mem après exécution d'une action TLA+ qui correspond à une opération atomique du code C.

(b) Gestion des timers par l'opérateur $dispatch_timers()$. Le code C des routines d'interruption contient des appels à des primitives matérielles, comme les fonctions `timer_cadenceWakeIn()` et `timer_overrunWakeIn()` qui permettent d'initialiser respectivement les timers « *cadence* » et « *overrun* ». L'opérateur $dispatch_timers()$ (défini dans le listing 6.13) permet de mettre à jour la variable $timers$ quand le code C fait appel à l'une des fonctions `timer_cadenceWakeIn()`, `timer_overrunWakeIn()` et `timer_overrunDisable()`.

$$
\begin{array}{l}
dispatch_timers(id, mem, tims) \triangleq \\
\quad \text{CASE } mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“timer_cadenceWakeIn_init”} \rightarrow \\
\qquad\qquad [tims \text{ EXCEPT } ![\text{“cadence”}] = load(id, mem, Addr_timer_cadenceWakeIn_param_d).val] \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“timer_overrunWakeIn_init”} \rightarrow \\
\qquad\qquad [tims \text{ EXCEPT } ![\text{“overrun”}] = load(id, mem, Addr_timer_overrunWakeIn_param_d).val] \\
\qquad \Box\ mem.register[id] \neq \langle\rangle \land Head(mem.register[id]).pc = \text{“timer_overrunDisable_init”} \rightarrow \\
\qquad\qquad [tims \text{ EXCEPT } ![\text{“overrun”}] = Null] \\
\qquad \Box\ \text{OTHER} \rightarrow tims
\end{array}
$$

Listing 6.13 – Spécification TLA+ de l'opérateur $dispatch_timers()$

À l'appel d'une des fonctions `timer_cadenceWakeIn()` et `timer_overrunWakeIn()` l'opérateur $dispatch_timers()$ modifie respectivement les timers « *cadence* » et « *overrun* » par la somme des valeurs passées en paramètres à la fonction appelée. Quand il s'agit d'un appel à la fonction `timer_overrunDisable()` qui permet de désactiver le timer « *overrun* », l'opérateur $dispatch_timers()$ met le timer « *overrun* » à la valeur $Null$.

6.3.2.9 Spécification d'une tâche en cours d'exécution

Une tâche en cours d'exécution est modélisée par l'action TLA+ $taskExecution()$ définie dans le listing 6.14.

$$
\begin{array}{l}
taskExecution(memory) \triangleq \\
\quad \land \text{UNCHANGED } timers \\
\quad \land \text{IF } memory.nsyscall[load(\text{"unused"}, memory, Addr_ctxcurrent)] \geq maxSyscall \\
\quad\quad \text{THEN UNCHANGED } memory \\
\quad\quad \text{ELSE } memory' \in \{memory, \\
\quad\quad\quad [memory \text{ EXCEPT } !.interrupts[\text{"syscall"}] = \text{TRUE}, !.syscalltype = DATARD], \\
\quad\quad\quad [memory \text{ EXCEPT } !.interrupts[\text{"syscall"}] = \text{TRUE}, !.syscalltype = PRETREPOS]\}
\end{array}
$$

Listing 6.14 – Spécification TLA+ du comportement d'une tâche en exécution

L'action $taskExecution()$ est une conjonction de deux formules :

- La première formule garde la variable $timers$ inchangée.
- La deuxième formule est exprimée ainsi : si la tâche courante dépasse le nombre maximal d'appels système, désigné par la constante $maxSyscall$, alors la variable $memory$ reste inchangée. Le fait de borner le nombre d'appels système autorisés par une tâche permet d'éviter l'exécution infinie de TLC. Ce scénario se présente quand une même tâche est choisie pour s'exécuter sur le processeur et qu'elle émet par la suite un appel système de type « *datTard* » (sa date d'activation reste inchangée et son échéance est repoussée) sans que la variable now change. Cette tâche est toujours prête à s'exécuter puisque sa date d'activation est toujours inférieure à now.
  Sinon, la tâche courante peut soit s'exécuter (dans ce cas $memory$ reste inchangée), soit faire un appel système. Dans ce dernier cas, la variable $memory$ est modifiée de manière indéterministe, en affectant au champ $syscalltype$ la valeur $datTard$ ou $PretRepos$, le champ $interrupts[\text{"syscall"}]$ est ensuite mis à TRUE. Il est à noter que la tâche en cours d'exécution peut à tout moment faire un appel système.

6.3.3 Spécification des propriétés du micronoyau mono-cœur

Les modules TLA+ de la version mono-cœur font en total 217Koctets, à savoir 2900 lignes de code, dont 2340 lignes sont générées automatiquement par C2TLA+ et 560 lignes sont écrites manuellement. Dans cette section, nous décrivons les propriétés fonctionnelles de la version mono-cœur de PharOS à vérifier sur la spécification TLA+.

6.3.3.1 Propriétés des listes des tâches

La liste des tâches *ready* est triée à tout instant par ordre croissant selon l'échéance des tâches et la liste des tâches *sleeping* est triée à tout instant par ordre croissant selon la date d'activation des tâches. Comme ces listes sont implémentées à travers des listes chaînées, nous définissons dans le listing 6.15 la fonction TLA+ itérative $getSeq()$

qui prend en argument l'adresse d'une liste ptr et un déplacement $offs$ qui désigne le déplacement requis pour accéder à un champ donné de cette structure. Dans la définition d'une fonction récursive dans TLA+, il est nécessaire de spécifier l'ensemble des valeurs possibles de chaque paramètre de fonction. Ici, $setTasks$ est l'ensemble qui contient toutes les adresses des contextes de tâches. La fonction $getSeq()$ retourne la séquence des valeurs des champs $offs$ des tâches qui composent la liste des tâches et dont l'adresse est ptr.

```
getSeq[off ∈ {Offset_context_act, Offset_context_dead}, ptr ∈ setTasks] ≜
  IF (ptr ≠ [loc ↦ Null, offs ↦ Null])
  THEN IF load("unused", memory, [loc ↦ ptr.loc, offs ↦ ptr.offs + Offset_context_nxt])
              ≠ [loc ↦ Null, offs ↦ Null]
       THEN ⟨load("unused", memory, [loc ↦ ptr.loc, offs ↦ ptr.offs + off])⟩
            ∘ getSeq[offset, load("unused", memory, [loc ↦ ptr.loc, offs ↦ (ptr.offs + Offset_context_nxt)])]
       ELSE ⟨load("unused", memory, [loc ↦ ptr.loc, offs ↦ ptr.offs + off])⟩
  ELSE ⟨⟩

SortedSeq(Seq) ≜ Seq ≠ ⟨⟩ ⇒ ∀i ∈ 1..Len(Seq), j ∈ 1..Len(Seq) :
                                 (i ≠ j) ∧ (i <= j) ⇒ (Seq[i].val ≤ Seq[j].val)
```

Listing 6.15 – Définition de l'opérateur $getSeq()$ et $SortedSeq(Seq)$

Nous définissons le prédicat $SortedSeq()$ (voir listing 6.15). Ce dernier retourne vrai si une séquence Seq est triée, et faux sinon. À partir de $getSeq()$ et $SortedSeq()$, nous exprimons l'invariant $SortedReady$ qui atteste que les tâches de *ready* sont triées par leur échéance. De la même manière, nous exprimons l'invariant $SortedSleeping$ qui exprime que les tâches de *sleeping* sont triées par leur date d'activation.

```
SortedReady ≜ □ SortedSeq(getSeq[Offset_context_dead, load("unused", memory, Addr_ready_contexts)])
SortedSleeping ≜ □ SortedSeq(getSeq[Offset_context_act, load("unused", memory, Addr_sleeping_contexts)])
```

Listing 6.16 – Spécification TLA+ des invariants $SortedReady$ et $SortedSleeping$

Nous spécifions dans le listing 6.17 l'invariant $actReadyList$ qui atteste que toutes les tâches prêtes ont une date d'activation antérieure à la date courante définie par now.

```
actReadyList ≜
  □ LET list ≜ getSeq[Offset_context_datPret, load("unused", memory, Addr_ready_contexts)]
    IN  list ≠ ⟨⟩ ⇒ ∀i ∈ 1..Len(list) : list[i].val ≤ now
```

Listing 6.17 – Spécification TLA+ de l'invariant $actReadyList$

Nous définissons dans le listing 6.18 la spécification de l'invariant $disjointListsTasks$ qui exprime qu'une tâche appartient soit à la liste des tâches prêtes ou endormies, soit elle est en exécution.

Les ensembles des adresses des contextes des tâches prêtes et celles des tâches endormies sont spécifiés respectivement par les variables temporaires $addrReadyTasks$ et $addrSleeppingTasks$. La tâche en exécution est désignée par la variable temporaire $currentTask$. L'invariant $disjointListsTasks$ exprime que l'intersection entre les ensembles $addrReadyTasks$, $addrSleeppingTasks$ et $\{currentTask\}$ est l'ensemble vide.

$$
\begin{aligned}
&\mathit{disjointListsTasks} \triangleq \\
&\quad \Box\ \text{LET } \mathit{addrReadyTasks} \triangleq \mathit{getSetTasks}[\mathit{load}(\text{“unused”}, \mathit{memory}, \mathit{Addr_ready_contexts})] \\
&\qquad\quad \mathit{addrSleepingTasks} \triangleq \mathit{getSetTasks}[\mathit{load}(\text{“unused”}, \mathit{memory}, \mathit{Addr_sleeping_contexts})] \\
&\qquad\quad \mathit{currentTask} \triangleq \mathit{load}(\text{“unused”}, \mathit{memory}, \mathit{Addr_ctxcurrent}) \\
&\qquad \text{IN } \mathit{addrReadyTasks} \cap \mathit{addrSleepingTasks} \cap \mathit{currentTask} = \{\}
\end{aligned}
$$

Listing 6.18 – Spécification TLA+ de l'invariant $\mathit{disjointListsTasks}$

$$
\begin{aligned}
&\mathit{getSetTasks}[\mathit{ptr} \in \mathit{setTasks}] \triangleq \\
&\quad \text{IF } (\mathit{ptr} \neq [\mathit{loc} \mapsto \mathit{Null}, \mathit{offs} \mapsto \mathit{Null}]) \\
&\quad \text{THEN IF } \mathit{load}(\text{“unused”}, \mathit{memory}, [\mathit{loc} \mapsto \mathit{ptr.loc}, \mathit{offs} \mapsto \mathit{ptr.offs} + \mathit{Offset_context_suiv}]) \\
&\qquad\qquad \neq [\mathit{loc} \mapsto \mathit{Null}, \mathit{offs} \mapsto \mathit{Null}] \\
&\qquad \text{THEN } \{[\mathit{loc} \mapsto \mathit{ptr.loc}, \mathit{offs} \mapsto \mathit{ptr.offs}]\} \cup \mathit{getSetTasks}[\mathit{load}(\text{“unused”}, \mathit{memory}, [\mathit{ptr} \text{ EXCEPT} \\
&\qquad\qquad !.\mathit{offs} = @ + \mathit{Offset_context_suiv}])] \\
&\qquad \text{ELSE } \{[\mathit{loc} \mapsto \mathit{ptr.loc}, \mathit{offs} \mapsto \mathit{ptr.offs}]\} \\
&\quad \text{ELSE } \{\}
\end{aligned}
$$

Listing 6.19 – Spécification TLA+ de la fonction $\mathit{getSetTasks}[]$

Les ensembles $\mathit{addrReadyTasks}$ et $\mathit{addrSleeppingTasks}$ sont calculés par la fonction $\mathit{getSetTasks}[]$ (dont la définition est donnée par le listing 6.19). Cette fonction prend en argument l'adresse d'une liste ptr et retourne l'ensemble d'adresses des contextes des tâches que la liste contient.

6.3.3.2 Propriété de la tâche en cours d'exécution

Nous spécifions dans le listing 6.20 l'invariant $\mathit{taskActivation}$ qui énonce que le système exécute à tout instant une tâche dont la date d'activation est antérieure à la date courante now.

$$
\begin{aligned}
&\mathit{taskActivation} \triangleq \\
&\quad \Box\ \text{LET } \mathit{current} \triangleq \mathit{load}(\text{“unused”}, \mathit{memory}, \mathit{Addr_ctxcurrent}) \\
&\qquad \text{IN } \mathit{current} \neq [\mathit{loc} \mapsto \mathit{Null}, \mathit{offs} \mapsto \mathit{Null}] \land \mathit{memory.register}[\text{“sched”}] = \langle\rangle \\
&\qquad\qquad \Rightarrow \mathit{load}(\text{“unused”}, \mathit{memory}, \mathit{current}).\mathit{val} \leq \mathit{now}
\end{aligned}
$$

Listing 6.20 – Spécification TLA+ de l'invariant $\mathit{taskActivation}$

6.3.3.3 Propriété de l'ordonnancement EDF

Une propriété très importante à vérifier par le micronoyau est que ce dernier implémente un ordonnancement EDF. Cette propriété est spécifiée par l'invariant edf défini dans le listing 6.21.

La formule edf atteste que si le micronoyau est au niveau applicatif alors la tâche en exécution est celle ayant l'échéance la plus proche par rapport à toutes les tâches prêtes (contenues dans *ready*). Dans cette définition, la variable temporaire $\mathit{seqReady}$ définit la séquence des valeurs des échéances des tâches prêtes et la variable $\mathit{deadCurr}$ spécifie l'échéance de la tâche en exécution.

$$
\begin{array}{l}
edf \triangleq \\
\quad \Box \text{ LET } curr \triangleq load(\text{"unused"}, memory, Addr_ctxcurrent) \\
\quad\quad \text{IN} \quad \text{IF } memory.register[\text{"sched"}] \neq \langle\rangle \vee curr = [loc \mapsto Null, offs \mapsto Null] \\
\quad\quad\quad\quad \text{THEN TRUE} \\
\quad\quad\quad\quad \text{ELSE LET} \\
\quad\quad\quad\quad\quad seqReady \triangleq getSeq[Offset_context_dead, load(\text{"unused"}, memory, Addr_ready_contexts)] \\
\quad\quad\quad\quad\quad deadCurr \triangleq load(\text{"unused"}, memory, [curr \text{ EXCEPT } !.offs = offs + Offset_context_dead]) \\
\quad\quad\quad\quad \text{IN} \quad \forall\, i \in 1..Len(seqReadys) : deadCurr.val \leq seqReady[i].val
\end{array}
$$

Listing 6.21 – Spécification TLA+ de l'invariant edf

6.3.3.4 Propriété du traitement d'une interruption « *syscall* »

La propriété $syscallCalled$ donnée par le listing 6.22 spécifie qu'à tout instant si une interruption « *syscall* » est déclenchée, alors la routine d'interruption `syscall()` sera inévitablement appelée par le micronoyau, c'est-à-dire que l'identifiant de la première instruction de la routine `syscall()` sera empilé dans la pile $memory.register[\text{"sched"}]$ du micronoyau.

$$
\begin{array}{l}
syscallCalled \triangleq \Box(memory.interrupts[\text{"syscall"}] = \text{TRUE} \wedge memory.register[\text{"sched"}] \neq \langle\rangle \\
\quad\quad \Rightarrow (\Diamond\; Head(memory.register[\text{"sched"}]).pc = \text{"syscall_436"}))
\end{array}
$$

Listing 6.22 – Spécification TLA+ de l'invariant $syscallCalled$

6.3.3.5 Propriété du traitement d'une interruption « *cadence* »

Comme pour l'interruption « *syscall* », nous spécifions dans le listing 6.23 la propriété $rcvCadenceCalled$ qui énonce qu'à tout instant si une interruption « *cadence* » est déclenchée, alors la routine `rcvCadence()` sera inévitablement exécutée par le micronoyau, c'est-à-dire l'identifiant de la première instruction de la routine `rcvCadence()` sera empilé dans la pile $memory.register[\text{"sched"}]$ du micronoyau.

$$
\begin{array}{l}
rcvCadenceCalled \triangleq \Box(memory.interrupts[\text{"cadence"}] = \text{TRUE} \wedge memory.register[\text{"sched"}] \neq \langle\rangle \\
\quad\quad \Rightarrow (\Diamond Head(memory.register[\text{"sched"}]).pc = \text{"rcvCadence_494"}))
\end{array}
$$

Listing 6.23 – Spécification TLA+ de la propriété $rcvCadenceCalled$

6.3.3.6 Propriété de la génération de l'interruption « *overrun* »

Quand une tâche en exécution dépasse son échéance, une interruption est envoyée au micronoyau pour le notifier de ce dépassement. Nous spécifions dans le listing 6.24 la formule temporelle $overrunDetection$ qui exprime qu'à tout instant si la tâche en exécution dépasse son échéance, alors inévitablement une exception est envoyée (la valeur de $memory.interrupts[\text{"overrun"}]$ est égale à TRUE).

$$
\begin{array}{l}
overrunDetection \triangleq \\
\quad \Box\, ((\text{LET } curr \triangleq load(\text{"unused"}, memory, Addr_ctxcurrent) \\
\qquad \text{IN} \\
\qquad\quad \land\, curr \neq [loc \mapsto Null, offs \mapsto Null] \land memory.register[\text{"sched"}] = \langle\rangle \\
\qquad\quad \land\, now > load(\text{"unused"}, memory, [loc \mapsto curr.loc, offs \mapsto curr.offs + Offset_context_dead]).val)) \\
\quad \Rightarrow \Diamond(memory.interrupts[\text{"overrun"}] = \text{TRUE}))
\end{array}
$$

Listing 6.24 – Spécification TLA+ de la propriété $overrunDetection$

6.3.4 Vérification des propriétés

Dans cette section, nous exposons les résultats de vérification des propriétés présentées dans la section précédente.

Pour vérifier la spécification TLA+ décrite dans la section 6.3 nous avons initié le contenu des listes *sleeping* et *ready*. Nous avons fixé la valeur de $MaxTime$ et le nombre maximum d'appels systèmes $nsyscall$ autorisés par une tâche.

6.3.4.1 Application de la réduction

Pour réduire la complexité du *model-checking*, nous avons appliqué sur notre cas d'étude la technique de réduction que nous avons présentée dans le chapitre 5. Pour cela, nous considérons dans cette partie le prédicat énoncé dans la section 5.2.2, dont la transformation consiste à agglomérer toute instruction C ne faisant intervenir ni une lecture ni une écriture dans une variable partagée entre plusieurs processus. Dans cette version du micronoyau, la variable $timers$ est modifiée par le code C du micronoyau à travers l'appel à des instructions matérielles et aussi par l'environnement matériel modélisé par l'action $HardwareNext()$. Pour cela, nous considérons le prédicat d'agglomération qui consiste à agglomérer toute instruction C ne faisant intervenir ni une lecture ni une écriture dans la variable $timers$. Dans ce cas, toutes les instructions du code C sont agglomérables à l'exception des appels aux fonctions suivantes :

- `timer_cadenceWakeIn()` et `timer_overrunWakeIn` qui permettent la mise à jour respectivement du timer « *cadence* » et « *overrun* » ;

- `timer_overrunDisable` qui permet de désactiver le timer « *overrun* », c'est-à-dire le remettre à la valeur $Null$.

6.3.4.2 Résultats obtenus

Après implémentation de ce prédicat dans C2TLA+, nous générons la spécification réduite de l'implémentation C du micronoyau. Pour vérifier la modélisation du micronoyau et son environnement dans TLA+, il suffit de faire appel à l'opérateur $dispatch_red()$ à la place de $dispatch()$ dans la définition de l'action $executeReadyProcess()$ donnée dans la section 6.3.2.8. Nous rappelons que $dispatch_red()$ est l'opérateur qui permet la gestion du flot de contrôle des instructions C agglomérables (voir section

5.2.3.4).

La vérification par TLC a été menée sur une machine Intel Core Pentium i7-2760QM avec 8 cœurs (2.40 GHz chacun) dotée d'une mémoire de 8 Goctets. Après exécution de TLC, la vérification des propriétés $syscallCalled$ et $rcvCadenceCalled$ a échoué. Le scénario d'erreur est le suivant : une interruption de type « *syscall* » est envoyée, mais now atteint sa borne maximale avant que l'interruption ne soit traitée. Ce cas d'erreur ne provient pas de l'implémentation du micronoyau mais il est dû au fait qu'on borne TLC et qu'on interrompt son exécution quand now atteint la valeur maximale. Afin de remédier à ce problème, nous avons changé l'action dans laquelle now est égale à $maxTime$ dans la définition de l'action $Next$. La nouvelle action est définie dans le listing 6.25.

```
∧ now = maxTime
∧ memory.interrupts["overrun"] ≠ TRUE
∧ UNCHANGED now
∧
   ∨ IF memory.register["sched"] ≠ ⟨⟩
     THEN ∧ memory' = dispatch_red("sched", memory.register["sched"])
          ∧ UNCHANGED timers
     ELSE ∃ idInt ∈ {i ∈ {"syscall", "cadence"} : memory.interrupts[i] = TRUE}
           LET nmem = initTaskInterruption(idInt, memory)
           IN  ∧ memory' = dispatch_red("sched", nmem)
               ∧ UNCHANGED timers
   ∨
     ∧ memory.register["sched"] = ⟨⟩
     ∧ {i ∈ {"syscall", "cadence"} : memory.interrupts[i] = TRUE} = {}
     ∧ UNCHANGED ⟨memory, timers⟩
```

Listing 6.25 – Nouvelle action d'arrêt

Cette action d'arrêt permet quand la pile des registres $memory.register[“sched”]$ du micronoyau est non vide de continuer à exécuter la routine d'interruption inachevée. Si la pile $memory.register[“sched”]$ est vide et l'ensemble des interruptions envoyées est non vide, on initialise les routines d'interruptions en appelant $initTaskInterruption()$ afin de les exécuter. Si l'ensemble d'interruptions est vide et le micronoyau n'a rien à exécuter, on garde les variables du système inchangées.

Après ajout de la nouvelle condition d'arrêt, nous montrons dans la table 6.2 les résultats de vérification des propriétés $SortedReady$, $SortedSleeping$, $disjointListsTasks$, $readyDatAct$, $taskActivation$, edf, $syscallCalled$ et $rcvCadenceCalled$. Ces résultats sont obtenus pour une valeur de $MaxTime$ égale à 10 et une valeur de $nsyscall$ égale à 3. Ces valeurs sont fixées pour que le model checker ne s'exécute pas indéfiniment

La première colonne de la table indique le nombre de tâches considérées. Les colonnes 2 à 5 présentent la taille de l'espace d'états ainsi que le temps de vérification avant et après l'application de la technique de réduction. #P1 indique le pourcentage de réduction en terme d'états et #P2 indique le pourcentage de réduction en terme de temps de vérification.

Sans application de la réduction, la vérification du micronoyau avec 4 tâches prend

TABLE 6.1 – Résultats du *model-checking* de la spécification TLA+ du micronoyau (version mono-coeur)

tâches	avant réduction		après réduction		#P1(%)	#P2(%)
	nombre d'états	temps (s)	nombre d'états	temps (s)		
1	262.925	12	26.816	4	89.9	66.6
2	490.520	26	41.559	7	91.52	73
3	14.409.020	723	1.010.891	93	92.98	83.7
4	278,535,021	19.080	151,301,517	6120	45.6	67.9

plus de 5 heures d'exécution. L'application de la réduction a permis de vérifier en un temps réduit les propriétés du micronoyau. Comme le montrent les résultats obtenus, l'espace d'états est réduit de manière considérable. Le facteur #P1 atteint 92% pour 3 tâches.

6.3.4.3 Propriété invalide

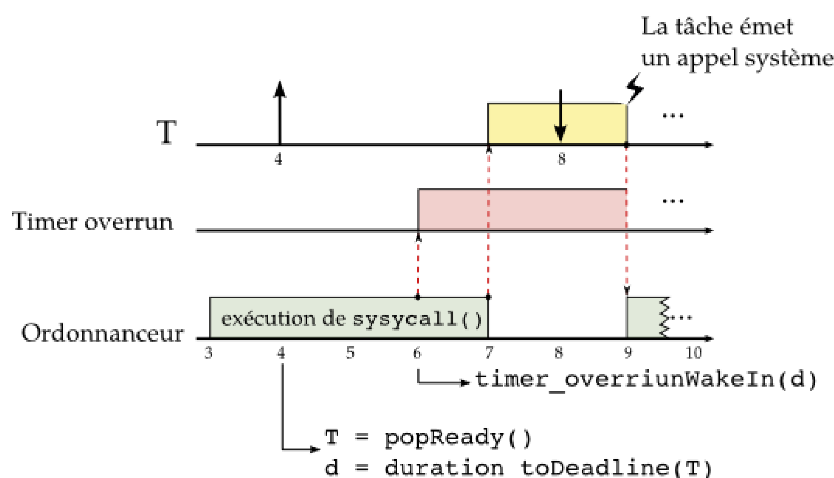


FIGURE 6.9 – Scénario ne satisfaisant pas la propriété *overrunDetection*

La vérification de la propriété *overrunDetection* par TLC a échoué. La trace d'erreur produite par TLC nous a permis de construire le scénario causant cette erreur. Ce scénario est décrit par la figure 6.9 présente.

Dans ce scénario, l'ordonnanceur exécute à $t = 3$ la routine d'interruption `syscall()`. À $t = 4$, cette routine retire un nouveau contexte T de la liste *ready* en appelant `popReady()`, tel que $T = [t_{act} = 4, t_{dead} = 8]$. La routine appelle la fonction `duration_toDeadline()` pour calculer la durée restante à l'échéance de la tâche T,

à savoir la valeur 4. L'exécution de cette routine est lente. À $t = 6$ la routine appelle `timer_cadenceWakeIn()` qui active et lance le timer avec une durée égale à 4 (valeur d lue à $t = 4$). La routine `syscall()` se termine et la tâche T commence son exécution. À $t = 9$ la tâche T fait un appel système avant l'expiration du timer « *overrun* ». Par conséquent, le timer s'interrompt et l'ordonnanceur exécute l'interruption $syscall()$ et l'exécution continue.

Dans ce scénario, le décalage entre la lecture de la tâche en tête de *ready*, le lancement du timer « *overrun* » et le début d'exécution d'une tâche fait que cette tâche dépasse son échéance. Ce scénario est possible dans le cas réel si le `timer_cadenceWakeIn()` s'exécute lentement.

6.4 Spécification et vérification de la version multi-cœur de PharOS

Dans cette section, nous commençons par décrire le modèle d'exécution et la spécification TLA+ complète de la version multi-cœur du micronoyau. Par la suite, nous spécifions les propriétés à vérifier et nous discutons les résultats de vérification obtenus.

6.4.1 Modèle d'exécution

Pour modéliser la version multi-cœur du micronoyau PharOS, nous nous basons sur le modèle d'exécution de la version mono-cœur, introduit dans la section 6.3.1. Comme pour la version mono-cœur, les fonctions C du micronoyau faisant intervenir le matériel sont déclarées à travers le mécanisme des attributs et elles sont modélisées manuellement en TLA+, comme les primitives de gestion des interruptions inter-processeur et les primitives de synchronisation. Les modules TLA+ générés par C2TLA+ sont ensuite intégrés avec d'autres modules, contenant des définitions TLA+ spécifiées manuellement et modélisant les aspects non définis par le code C. Le micronoyau et son environnement sont modélisés par la formule $Spec$, dont la définition est celle donnée dans la version mono-cœur (voir section 6.3.2).

Comme décrit dans la section 6.2, la version multi-cœur du micronoyau gère quatre type d'interruptions, à savoir « *cadence* », « *syscall* », « *syncPreemption* » et « *syncSortList* ». Le champ $interrupts$ devient alors un tableau à quatre éléments dont chaque élément modélise une interruption. Les deux cœurs ainsi que l'environnement s'exécutent en concurrence. Cette concurrence est modélisée par l'entrelacement des actions.

Dans cette version du micronoyau on dispose de deux cœurs. Le cœur d'exécution peut exécuter soit une tâche parmi les tâches désignées par "t1", "t2", ..., "tn", soit la tâche « idle » identifiée par "XC_idle", soit le micronoyau dont le processus est identifié par "sched_XC". Le cœur de contrôle peut exécuter soit le micronoyau qui correspond à un processus identifié par "sched_CC", soit la tâche « idle ». Tous les processus composant

le système partagent la mémoire globale $data$ et chacun dispose de sa propre mémoire locale.

Comme l'accès concurrent aux listes des tâches est géré par des *spinlocks*, nous utilisons dans la spécification du micronoyau multi-cœur les définitions TLA+ des primitives `spinlock_acquire()` et `spinlock_release()` proposées dans l'annexe B.

6.4.2 Spécification du comportement du micronoyau multi-cœur

La spécification TLA+ complète du micronoyau multi-cœur est donnée par la formule $Spec$ dont la définition est celle donnée dans la version mono-cœur (voir listing 6.3.2). La définition du prédicat $Next$ est la même que celle de la version mono-cœur. De même pour la définition de l'action $HardwareNext()$. Ce qui change dans cette version multi-cœur est l'initialisation du prédicat $Init$, la définition du prédicat $kernel$ et les actions de gestion des interruptions. Cette partie sera détaillée dans les paragraphes suivants.

6.4.2.1 Définition du prédicat $Init$

```
Init ≜ ∧ memory = [data ↦ [(0 :> [val ↦ 0] @@ 1 :> [val ↦ 0] ...]
                   register ↦ ("sched_XC" :> ⟨⟩ @@
                               "sched_CC" :> ⟨⟩ @@
                               "t1" :> ⟨[pc ↦ ⟨"task_init"⟩, fp ↦ 1]⟩@@
                               "t2" :> ⟨[pc ↦ ⟨"task_init"⟩, fp ↦ 1]⟩@@ ... @@
                               "idle" :> ⟨[pc ↦ ⟨"idle_init"⟩, fp ↦ 1]⟩),
                               "XC_idle" :> ⟨[pc ↦ ⟨"idle_init"⟩, fp ↦ 1]⟩),
                   stack ↦ [i ∈ ProcSet ↦ ⟨⟩],
                   ret ↦ [i ∈ ProcSet ↦ [val ↦ Undef]],
                   interrupts ↦ ("overrun" :> FALSE @@ "cadence" :> TRUE @@ "syscall" :> FALSE,
                                 "syncPreemption" :> FALSE @@"syncSortList" :> FALSE),
                   syscalltype ↦ Null
       ∧ timers = ("overrun" :> 2 @@ "cadence"  :>  5)
       ∧ now = 0
```

Listing 6.26 – Définition du prédicat $Init$ pour la spécification du micronoyau multi-cœur

Dans le listing 6.26 nous illustrons la définition du prédicat $Init$. Elle est proche à celle définie dans la version mono-cœur à part quelques ajouts. Ces ajouts incluent la spécification des valeurs initiales des interruptions « *syncPreemption* » et « *syncSortList* ». Ces dernières sont initialisées à la valeur FALSE. Aussi, les piles des registres des processus "sched_XC" et "sched_CC" sont initialement vides. La mémoire $data$ dont nous n'explicitons pas la définition, contient les valeurs initiales des adresses des listes des tâches, les contextes d'exécution des tâches "t1", "t2", ..., "tn" et les verrous utilisés.

$$
\begin{aligned}
kernel \triangleq\ & \land (CC \lor XC) \\
& \land \text{UNCHANGED } now \\[1em]
XC \triangleq\ & \lor \\
& \quad \land memory.interrupts[\text{“overrun”}] = \text{TRUE} \\
& \quad \land \text{UNCHANGED } \langle memory, now, timers \rangle \\
& \lor \\
& \quad \land memory.interrupts[\text{“overrun”}] \neq \text{TRUE} \\
& \quad \land \text{IF } memory.register[\text{“sched_XC”}] \neq \langle\rangle \lor interruptionsSetXC(memory) = \{\} \\
& \quad\quad \text{THEN } executeReadyProcessXC(memory) \\
& \quad\quad \text{ELSE } \exists\, idInt \in interruptionsSetXC(memory) : \\
& \quad\quad\quad \text{LET } nmem \triangleq initTaskInterruption(idInt, memory) \\
& \quad\quad\quad\quad\ \ ntims \triangleq initTimers(idInt, timers) \\
& \quad\quad\quad \text{IN } \ executeReadyProcessXC(nmem) \\[1em]
CC \triangleq\ & \land memory.interrupts[\text{“overrun”}] \neq \text{TRUE} \\
& \land \text{IF } memory.register[\text{“sched_CC”}] \neq \langle\rangle \lor interruptionsSetCC(memory) = \{\} \\
& \quad \text{THEN } executeReadyProcessCC(memory) \\
& \quad \text{ELSE } \exists\, idInt \in interruptionsSetCC(memory) : \\
& \quad\quad \text{LET } nmem \triangleq initTaskInterruption(idInt, memory) \\
& \quad\quad \text{IN } \ executeReadyProcessCC(nmem)
\end{aligned}
$$

Listing 6.27 – Spécification des actions $kernel$, XC et CC

6.4.2.2 Définition de l'action $kernel$

La spécification de l'action $kernel$ est donnée par le listing 6.27. Cette dernière consiste à exécuter en entrelacement le cœur d'exécution et le cœur de contrôle dont le comportement est spécifié respectivement par les actions CC et XC.

6.4.2.3 Spécification du comportement du cœur d'exécution *XC*

Dans la suite du document, nous employons le terme *processus XC* (respectivement *processus CC*) pour désigner le processus identifié par “sched_XC” (respectivement “sched_CC”) et exécuté par XC (respectivement CC) .

$$interruptionsSetXC(mem) \triangleq \{id \in \text{“syscall”}, \text{“syncPreemption”} : mem.interrupts[id] = \text{TRUE}\}$$

Listing 6.28 – Spécification de l'ensemble des interruptions gérées par *XC*

L'action XC qui modélise le comportement du cœur d'exécution est donnée par le listing 6.27. Cette action consiste en la disjonction de deux actions :

- une première action qui garde les variables du système inchangées. Elle correspond à l'état où *XC* reçoit une interruption « *overrun* » qui cause l'arrêt du processus *XC* ;

- une action qui exprime que le processus *XC* peut exécuter soit une tâche prête, soit une routine d'interruption. Si sa pile de registres n'est pas vide ou s'il n'a pas encore reçu des interruptions, le processus *XC* continue son exécution en appelant l'action $executeReadyProcessXC()$ (définie dans le paragraphe suivant).

Quand le processus *XC* reçoit une ou plusieurs interruptions, il choisit de manière indéterministe dans l'ensemble retourné par l'opérateur $interruptionsSetXC()$ une interruption $idInt$ et il la traite.
La définition de l'opérateur $interruptionsSetXC()$ est donnée par le listing 6.28. Elle retourne l'ensemble des interruptions traitées par le *XC*.
Le processus *XC* met ensuite à jour la mémoire $memory$ et les timers en appelant respectivement les opérateurs $initTaskInterruption()$ et $initTimers()$. L'opérateur $initTimers()$ est défini dans la spécification de la version mono-cœur du micronoyau (voir paragraphe 6.3.2.5). Par la suite, il démarre l'exécution de la routine en appelant l'action $executeReadyProcessXC()$.

```
initTaskInterruption(inT, mem) ≜
  CASE inT = "syscall" →
      IF mem.syscall = DATARDTHEN [mem EXCEPT
        !.data = updateDatTard(load("unused", mem, Addr_ctxcurrent), mem),
        !.register = [mem.register EXCEPT ![“sched_XC”] = ⟨[pc ↦ “syscall_574”, fp ↦ 1]⟩],
        !.stack = [mem.stack EXCEPT ![“sched_XC”] = mem.stack[“sched_XC”] ∘ ⟨[val ↦ 0],
           [loc ↦ Undef, offs ↦ Undef], Undef, [loc ↦ Undef, offs ↦ Undef]⟩]
        !.interrupts = [mem.interruptsEXCEPT ![inT] = FALSE],
        !.syscalltype = Null,
        !.nsyscall = [mem.nsyscall EXCEPT ![load("unused", mem, Addr_ctxcurrent)] = @ + 1]]
      ELSE [mem EXCEPT
        !.data = updatePretRepos(load("unused", mem, Addr_ctxcurrent), mem),
        !.register = [mem.register EXCEPT ![“sched_XC”] = ⟨[pc ↦ “syscall_574”, fp ↦ 1]⟩],
        !.stack = [mem.stack EXCEPT ![“sched_XC”] = mem.stack[“sched_XC”] ∘
           ⟨[val ↦ 1], [loc ↦ Undef, offs ↦ Undef], Undef, [loc ↦ Undef, offs ↦ Undef]⟩]
        !.interrupts = [mem.interrupts EXCEPT ![inT] = FALSE],
        !.syscalltype = Null,
        !.nsyscall = [mem.nsyscall EXCEPT ![load("unused", mem, Addr_ctxcurrent)] = @ + 1]]
  □ inT = "syncPreemption" → [mem EXCEPT
      !.register = [mem.register EXCEPT ![“sched_XC”] = ⟨[pc ↦ “syncPreemption_591”, fp ↦ 1]⟩],
      !.stack = [mem.stack EXCEPT ![“sched_XC”] = mem.stack[“sched_XC”] ∘ ⟨Undef⟩],
      !.interrupts = [mem.interrupts EXCEPT ![inT] = FALSE]]
  □ inT = "cadence" → [mem EXCEPT
      !.register = [mem.register EXCEPT ![“sched_CC”] =⟩[pc ↦ “rcvCadence_610”, fp ↦ 1]⟩],
      !.interrupts = [mem.interrupts EXCEPT ![inT] = FALSE]]
  □ inT = "syncSortList" → [mem EXCEPT
      !.register = [mem.register EXCEPT ![“sched_CC”] = ⟨[pc ↦ “syncSortList_614”, fp ↦ 1]⟩],
      !.interrupts = [mem.interrupts EXCEPT ![inT] = FALSE]]
  □ OTHER → "ERROR"
```

Listing 6.29 – Spécification de l'initialisation des routines d'interruptions (version multi-cœur)

(a) Initialisation d'une routine d'interruption. Le listing 6.29 illustre la spécification TLA+ de l'initialisation d'une routine d'interruption.

Quand il s'agit d'une interruption de type « *syscall* » ou « *syncPreemption* », le cadre de pile de la routine associée est d'abord empilé dans la pile $mem.stack[\text{“sched_CC”}]$ du cœur d'exécution. L'identifiant de l'instruction à exécuter et le pointeur de cadre de pile sont insérés au sommet de la pile $mem.register[\text{“sched_XC”}]$.

Quand il s'agit d'une interruption de type « *cadence* » ou « *syncSortList* », le cadre de pile de la routine associée est empilé dans la pile $mem.stack[\text{"sched_CC"}]$ du cœur du contrôle et l'identifiant de l'instruction à exécuter ainsi que le pointeur de cadre de pile sont insérés au sommet du registre $mem.register[\text{"sched_CC"}]$.

(b) Définition de l'action $executeReadyProcessXC()$**.** La spécification de l'action $executeReadyProcessXC()$ est donnée par le listing 6.30. Cette action permet de choisir un processus prêt pour s'exécuter sur le *XC*.

$$
\begin{array}{l}
procsExecXC(mem) \triangleq \\
\quad \text{IF } mem.register[\text{"sched_XC"}] \neq \langle\rangle \text{THEN } \{\text{"sched_XC"}\} \\
\quad \text{ELSE} \\
\quad\quad \text{LET } current \triangleq load(\text{"unused"}, mem, Addr_ctxcurrent) \\
\quad\quad \text{IN} \quad \text{CASE } current = load(\text{"unused"}, memory, Addr_CTX_IDLE) \rightarrow \{\text{"idle_XC"}\} \\
\quad\quad\quad\quad \Box\ current = Addr_c1 \rightarrow \{\text{"t1"}\} \\
\quad\quad\quad\quad \Box\ current = Addr_c2 \rightarrow \{\text{"t2"}\} \\
\quad\quad\quad\quad \Box \ldots \\
\quad\quad\quad\quad \Box\ current = Addr_cn \rightarrow \{\text{"tn"}\} \\
\quad\quad\quad\quad \Box\ \text{OTHER} \rightarrow \text{"ERROR"}
\end{array}
$$

$$
\begin{array}{l}
executeReadyProcessXC(nmem) \triangleq \\
\quad \exists\, id \in procsExecXC(nmem) : \\
\quad\quad \lor \\
\quad\quad\quad \land\ id \in \{\text{"t1"}, \text{"t2"}, \ldots\} \\
\quad\quad\quad \land\ taskExecution \\
\quad\quad \lor \\
\quad\quad\quad \land\ id \in \{\text{"sched_XC"}\} \\
\quad\quad\quad \land\ memory' = dispatch(id, nmem.register[id], nmem) \\
\quad\quad\quad \land\ timers' = dispatch_timers(id, nmem.register[id], nmem, timers)
\end{array}
$$

Listing 6.30 – Spécification de l'exécution d'une action sur *XC* d'un processus prêt

Le choix d'un processus à exécuter sur le *XC* se fait en choisissant un identifiant dans l'ensemble retourné par l'opérateur $procsExecXC()$. Cet opérateur retourne l'identifiant "sched_XC" lorsque la pile des registres du processus XC est non vide, c'est-à-dire le processus XC est en cours d'exécution d'une routine d'interruption. Sinon, l'opérateur $procsExecXC()$ retourne l'adresse du contexte de la tâche en exécution.

L'action $executeReadyProcessXC()$ appelle l'opérateur $taskExecution$ quand une tâche est en cours d'exécution. Sinon, elle appelle les opérateurs $dispatch()$ et $dispatch_timers$ qui permettent de mettre à jour respectivement les variables $memory$ et $timers$. Nous rappelons que l'opérateur $dispatch()$ permet de simuler le flot de contrôle du code C exécuté par le micronoyau.

Les définitions des actions $taskExecution$ et $dispatch_timers$ utilisées dans cette version du micronoyau sont celles définies dans la version version mono-cœur (voir section 6.3.2.8).

(c) L'envoi d'une interruption « *syncSortList* ». Comme décrit dans la section 6.2.3.4, l'interruption « *syncSortList* » est déclenchée par le cœur *XC* quand celui-ci exécute

la routine `syscall()`. L'envoi de cette interruption dans l'implémentation C du micronoyau se fait par un appel à la fonction `sync_notify_sortList()`. Cette instruction est implémentée par un mécanisme matériel. Sa modélisation TLA+ est donnée par l'opérateur $sync_notify_sortList()$ (voir listing 6.31).

$$
\begin{aligned}
& sync_notify_sortList(id, st, mem) \triangleq [mem \text{ EXCEPT} \\
& \quad !.register = [mem.register \text{ EXCEPT } ![id] = Tail(mem.register[id])], \\
& \quad !.interrupts = [mem.interrupts \text{ EXCEPT } ![\text{"syncSortList"}] = \text{TRUE}]]
\end{aligned}
$$

Listing 6.31 – Spécification TLA+ de l'instruction d'envoi de l'interruption « *syncSortList* »

L'opérateur $sync_notify_sortList()$ consiste à mettre la valeur de l'élément $mem.interrupts[\text{"syncSortList"}]$ à TRUE modélisant l'envoi d'une interruption « *syncSortList* » et à dépiler la pile des registres $memory.register[\text{"sched_CC"}]$, pour revenir à l'instruction qui suit l'appel à la fonction `sync_notify_sortList()`.

6.4.2.4 Spécification du comportement du cœur de contrôle *CC*

Le comportement du processus "sched_CC" (voir listing 6.27) consiste à exécuter les routines d'interruptions « *cadence* » et « *syncSortList* ». Si sa pile est non vide ou s'il n'a encore pas reçu une interruption, le processus *CC* appelle l'action $executeReadyProcessCC()$. La routine à traiter est choisie parmi l'ensemble retourné par l'opérateur $interruptionsSetCC()$ (défini dans le listing 6.32). Cet opérateur retourne l'ensemble des interruptions déclenchées et traitées par le *CC*.

$$
interruptionsSetCC(mem) \triangleq \{id \in \text{"cadence"}, \text{"syncSortList"} : mem.interrupts[id] = \text{TRUE}\}
$$

Listing 6.32 – Spécification de l'ensemble des interruptions déclenchées et gérées par *CC*

(a) Définition de l'action $executeReadyProcessCC()$. La spécification de l'action $executeReadyProcessCC()$ est illustrée par le listing 6.33.

$$
\begin{aligned}
& executeReadyProcessCC(nmem, tims) \triangleq \\
& \quad \text{IF } nmem.register[\text{"sched_CC"}] \neq \langle\rangle \\
& \quad \text{THEN } \wedge memory' = dispatch(id, nmem.register[id], nmem) \\
& \qquad\quad \wedge timers' = dispatch_timers(id, nmem.register[id], nmem, tims) \\
& \quad \text{ELSE UNCHANGED } \langle memory, timers \rangle
\end{aligned}
$$

Listing 6.33 – Spécification de l'exécution d'une action sur *CC* d'un processus prêt

L'action $executeReadyProcessCC()$ simule l'exécution d'une opération atomique par le cœur *CC*. Lorsque la pile du processus $nmem.register[\text{"sched_CC"}]$ est non vide, le cœur *CC* exécute une routine d'interruption. Ceci en appelant $dispatch()$ qui simule le flot l'exécution du code C et l'opérateur $dispatch_timers$ qui met à jour les valeurs des timers. Sinon, les variables $memory$ et $timers$ restent inchangées. Cette action correspond au cas où le cœur *CC* exécute la tâche *idle*.

(b) L'envoi d'une interruption « *syncPreemption* ». Comme décrit dans la section 6.2.3.4, l'interruption « *syncPreemption* » est déclenchée par le cœur *CC* lors de l'exécution de la routine `rcvCadence()`.

$$\begin{aligned}&sync_notify_preemption(id, st, mem) \triangleq [mem \text{ EXCEPT} \\ &\quad !.register = [mem.register \text{ EXCEPT } ![id] = Tail(mem.register[id])], \\ &\quad !.interrupts = [mem.interrupts \text{ EXCEPT } ![\text{“syncPreemption”}] = \text{TRUE}]]\end{aligned}$$

Listing 6.34 – Spécification TLA+ de l'instruction d'envoi de l'interruption « *syncPreemption* »

L'envoi de cette interruption dans l'implémentation C du micronoyau se fait par un appel à la fonction `sync_notify_preemption()`. Comme pour l'instruction `sync_notify_sortList()` (introduite dans le paragraphe (c)), l'instruction `sync_notify_preemption()` est implémentée par un mécanisme matériel et sa modélisation TLA+ est donnée par l'opérateur $sync_notify_preemption()$ défini dans le listing 6.34.

6.4.3 Spécification des propriétés du micronoyau multi-cœur

Les modules TLA+ de la version multi-cœur font en total 263Koctets de taille, à savoir 4500 lignes de code, dont 3790 lignes sont générées automatiquement par C2TLA+ et 710 lignes sont écrites manuellement. Toutes les propriétés TLA+ définies pour la version mono-cœur du micronoyau sont aussi à vérifier sur la version multi-cœur, avec quelques modifications mineures pour certaines propriétés.

6.4.3.1 Propriétés de la version multi-cœur

Les propriétés $SortedReady$ et $SortedSleeping$ (voir listing 6.16) définies dans la version mono-cœur, restent inchangées pour la version multi-cœur. La propriété $disjointListsTasks$ (voir listing 6.18), qui atteste qu'une tâche n'appartient qu'à une seule liste a été étendue pour prendre en compte les listes *unsorted ready* et *unsorted sleeping*.

Dans la version multi-cœur, le cœur d'exécution *XC* est responsable du traitement des interruptions « *cadence* » et « *syscall* » ainsi que l'exécution d'une tâche prête. Les propriétés suivantes : $actReadyList$ (voir listing 6.17), edf (voir listing 6.21), $overrunDetection$ (voir listing 6.24), $syscallCalled$ (voir listing 6.22) et $rcvCadenceCalled$ (voir listing 6.23), définies pour la version mono-cœur, ont été alors modifiées en remplaçant l'expression $memory.register[\text{“sched”}]$ par $memory.register[\text{“sched_XC”}]$.

6.4.3.2 Propriété sur l'interruption « *syncSortList* »

$$\begin{aligned}&interruptionSyncSortListSent \triangleq \\ &\quad \Box((memory.register[\text{“sched_XC”}] \neq \langle\rangle \land Head(memory.register[\text{“sched_XC”}]).pc = \text{“sched_syscall_547”}) \\ &\qquad \Rightarrow \Diamond(memory.interrupts[\text{“syncSortList”}] = \text{TRUE}))\end{aligned}$$

Listing 6.35 – Spécification TLA+ de la propriété $interruptionSyncSortListSent$

Le listing 6.35 définit la propriété $interruptionSyncSortListSent$. Cette propriété exprime que lorsque le cœur *XC* exécute la routine d'interruption `syscall()`, une interruption inter-processeur de type « *syncSortList* » sera inévitablement envoyée au cœur de contrôle.

$$\begin{array}{l} handleSortListCalled \triangleq \\ \quad \Box(memory.register[\text{"syncSortList"}] = \text{TRUE} \land memory.register[\text{"sched_CC"}] = \langle\rangle \\ \quad\quad \Rightarrow (\Diamond(memory.register[\text{"sched_CC"}] \neq \langle\rangle \\ \quad\quad\quad \Rightarrow (Head(memory.register[\text{"sched_CC"}]).pc = \text{"handleSortListNotif_614"})))) \end{array}$$

Listing 6.36 – Spécification TLA+ de la propriété $handleSortListCalled$

Dans le listing 6.36, nous spécifions la propriété $handleSortListCalled$. Cette dernière exprime qu'à tout instant lorsqu'une interruption « *syncSortList* » est envoyée, la routine d'interruption `handleSortListNotif()` sera inévitablement exécutée par le cœur *CC*.

6.4.3.3 Propriété sur l'interruption « *syncPreemption* »

$$\begin{array}{l} interruptionSyncPreemtionSent \triangleq \\ \quad \Box((memory.register[\text{"sched_CC"}] \neq \langle\rangle \\ \quad\quad \land Head(memory.register[\text{"sched_CC"}]).pc \in \{\text{"rcvCadence_610"}, \text{"handleSortListNotif_614"}\}) \\ \quad \Rightarrow \Diamond(memory.interrupts[\text{"syncPreemption"}] = \text{TRUE}) \end{array}$$

Listing 6.37 – Spécification TLA+ de la propriété $interruptionSyncPreemtionSent$

Nous définissons dans le listing 6.37 la propriété $interruptionSyncPreemtionSent$. Cette propriété exprime que lorsque le cœur *CC* exécute la routine d'interruption `rcvCadence()` ou `handleSortListNotif()` alors une interruption inter-processeur de type « *syncPreemption* » est inévitablement envoyée au cœur *XC*.

$$\begin{array}{l} handlePreemptionNotifCalled \triangleq \\ \quad \Box(memory.register[\text{"syncPreemption"}] = \text{TRUE} \land memory.register[\text{"sched_XC"}] = \langle\rangle \\ \quad\quad \Rightarrow (\Diamond(memory.register[\text{"sched_XC"}] \neq \langle\rangle \\ \quad\quad\quad \Rightarrow (Head(memory.register[\text{"sched_XC"}]).pc = \text{"handlePreemptionNotif_591"})))) \end{array}$$

Listing 6.38 – Spécification TLA+ de la propriété $handlePreemptionNotifCalled$

La propriété $handlePreemptionNotifCalled$, définie par le listing 6.38, exprime qu'à tout instant lorsqu'une interruption « *syncPreemption* » est envoyée, la routine d'interruption `handlePreemptionNotif()` sera inévitablement exécutée par le cœur *XC*.

6.4.3.4 Propriété de l'exclusion mutuelle

Nous rappelons que les deux cœurs accèdent aux listes des tâches (*ready*, *unsorted ready* et *unsorted sleeping*) de manière concurrente (voir figure 6.4). L'accès à ces listes est assuré par le mécanisme de *spinlock* utilisant l'instruction $CAS()$. Une propriété importante à vérifier est que les primitives de synchronisation *spinlock* sont bien implémentées et qu'elles assurent l'exclusion mutuelle.

$$
\begin{array}{l}
mutual_exclusion(sc1, sc2) \triangleq \forall\, stmt1 \in sc1 : \forall\, stmt2 \in sc2 : \\
\quad (\land\ memory.register[\text{“sched_CC”}] \neq \langle\rangle \\
\quad \land\ memory.register[\text{“sched_XC”}] \neq \langle\rangle \\
\quad \land\ Head(memory.register[\text{“sched_CC”}]).pc = stmt1) \\
\qquad \Rightarrow Head(memory.register[\text{“sched_XC”}]).pc \neq stmt2
\end{array}
$$

$$
\begin{array}{l}
mutualexclusionReadyList \triangleq \\
\ \text{LET } sc1 \triangleq \{\text{“insertReady_523”}, \ldots, \text{“insertReady_525”}\} \\
\qquad sc2 \triangleq \{\text{“popReady_447”}, \ldots, \text{“popReady_450”}\} \\
\ \text{IN } \ \Box\, mutual_exclusion(sc1, sc2)
\end{array}
$$

$$
\begin{array}{l}
mutualexclusionUnsortedReadyList \triangleq \\
\ \text{LET } sc1 \triangleq \{\text{“allUnsortedReadyToReady_536”}, \ldots, \text{“allUnsortedReadyToReady_5538”}\} \\
\qquad sc2 \triangleq \{\text{“toReady_455”}, \ldots, \text{“toReady_457”}\} \\
\ \text{IN } \ \Box\, mutual_exclusion(sc1, sc2)
\end{array}
$$

$$
\begin{array}{l}
mutualexclusionSleepingList \triangleq \\
\ \text{LET } sc1 \triangleq \{\text{“allUnsortedSleepingToSleeping_494”}, \ldots, \text{“toSleeping_464”}\} \\
\qquad sc2 \triangleq \{\text{“toSleeping_462”}, \text{“toSleeping_464”}\} \\
\ \text{IN } \ \Box\, mutual_exclusion(sc1, sc2)
\end{array}
$$

Listing 6.39 – Spécification TLA+ de la propriété d'exclusion mutuelle

Pour vérifier cette propriété, nous exprimons dans le listing 6.39 l'opérateur $mutual_exclusion()$. Ce dernier prend en arguments deux ensembles $sc1$ et $sc2$ contenant respectivement les identifiants des instructions C appartenant à la section critique du code exécuté par le cœur de contrôle et le cœur d'exécution. Et il retourne vrai si l'accès à ces sections se fait de manière exclusive, et faux sinon.

Nous exprimons dans le listing 6.39 les opérateurs $mutualexclusionReadyList$, $mutualexclusionUnsortedReadyList$ et $mutualexclusionSleepingList$ qui désignent respectivement la propriété d'exclusion mutuelle pour les listes *ready*, *unsorted ready* et *sleeping*. Dans ces définitions, $sc1$ et $sc2$ contiennent les identifiants des instructions générées par C2TLA+ et appelées respectivement par le cœur de contrôle et le cœur d'exécution. La vérification de cette propriété implique un examen de l'implémentation par l'utilisateur afin de repérer les portions du code qui sont en accès concurrents.

6.4.3.5 Propriétés d'absence de famine

$$
\begin{array}{l}
deadlockFreeProc(p, sc) \triangleq \\
\quad \Box((memory.register[p] \neq \langle\rangle \land Head(memory.register[p]).pc = \text{“spinlock_acquire_init”}) \\
\qquad \Rightarrow \Diamond(Head(memory.register[p]).pc \in sc))
\end{array}
$$

$$
\begin{array}{l}
deadlockFree \triangleq \ \land\ deadlockFreeProc(\text{“sched_XC”}, \text{“ctxlist_popReady_447”}) \\
\qquad\qquad\qquad \land \ldots \\
\qquad\qquad\qquad \land\ deadlockFreeProc(\text{“sched_CC”}, \text{“ctxlist_insertReady_523”})
\end{array}
$$

Listing 6.40 – Spécification TLA+ de la formule $deadlockFree$

Pour exprimer la propriété d'absence de famine dans le micronoyau, nous définissons

dans le listing 6.40 la formule $deadlockFreeProc()$. Cette formule prend en argument un identifiant de processus p et un identifiant d'instruction appartenant à une section critique sc. Elle exprime que si le processus p essaie d'entrer dans une section critique sc (le sommet de sa pile $memory.register[i]$ contient l'identifiant "spinlock_acquire_init"), alors il y parviendra (sa pile contiendra l'identifiant sc). La propriété $deadlockFree$ consiste alors en la conjonction des formules attestant l'absence de famine pour les processus "sched_XC" et"sched_CC" pour toute demande d'accès à une section critique.

La condition $WF_{\langle memory, timers, now \rangle}$ (définie dans la formule $Spec$) est insuffisante pour vérifier l'absence de famine dans le code C. Il est nécessaire de définir une condition d'équité plus forte. Pour cela, nous définissons dans le listing 6.41 une condition d'équité qui exprime que les processus *XC* et *CC* appellent inévitablement la primitive de libération du verrou `spinlock_release()`.

$$\begin{array}{l} \forall\, id \in \{\text{"sched_XC"}, \text{"sched_CC"}\} : \Box\Diamond(memory.register[id] \neq \langle\rangle \\ \quad \Rightarrow Head(memory.register[id]).pc = \text{"spinlock_release_init"}) \end{array}$$

Listing 6.41 – Spécification TLA+ d'une condition d'équité supplémentaire

6.4.4 Vérification des propriétés

Dans cette section, nous exposons les résultats de vérification des propriétés présentées dans la section précédente.

6.4.4.1 Prédicat d'agglomération

Nous considérons dans cette partie le prédicat énoncé dans le chapitre 5, dont la transformation consiste à agglomérer toute instruction C ne faisant intervenir ni une lecture ni une écriture dans une variable partagée entre plusieurs processus. Dans cette version du micronoyau, les variables partagées sont les listes des tâches, les contextes de tâches ainsi que la variable $timers$. Les fonctions qui modifient la variable $timers$ sont celles spécifiées dans la section 6.3.4.1. Les fonctions qui modifient les listes des tâches sont les fonctions C d'insertion et de retrait dans les listes $ready$, $unsortedready$ et $unsortedsleeping$.

6.4.4.2 Application de la réduction et résultats obtenus

La table 6.2 montre les résultats de vérification de la spécification TLA+ de la version multi-cœur de PharOS. Nous avons fixé la valeur maximale du nombre d'appels système ($nsyscall$) par tâche à 3.

Les espaces d'états générés sont relativement grands (des millions d'états). Ceci s'explique par le fait que dans cette version on dispose de plus d'entrelacements. La vérification de la spécification TLA+ avant application de la réduction prend beaucoup de temps. Ici, pour une seule tâche le temps de vérification a dépassé 2 heures.

TABLE 6.2 – Résultats du *model-checking* de la spécification TLA+ du micronoyau (version multi-cœur)

tâches	avant réduction		après réduction		#P1(%)	#P2(%)
	nombre d'états	temps (s)	nombre d'états	temps (s)		
1	>122.008.220	8.200	8.448.431	723	93	91.1
2	>520.526.362	>30.800	95.508.419	2880	81.3	90.6

```
∧ now = maxTime
∧ memory.interrupts["overrun"] ≠ TRUE
∧ UNCHANGED now
∧
  ∨ IF memory.register["sched_XC"] ≠ ⟨⟩
    THEN ∧ memory' = dispatch_withIntMask("sched_XC", memory.register["sched_XC"])
         ∧ UNCHANGED timers
    ELSE ∃ idInt ∈ interruptionsSetXC(memory) :
         LET nmem = initTaskInterruption(idInt, memory)
         IN  ∧ memory' = dispatch_withIntMask("sched_XC", nmem)
             ∧ UNCHANGED timers
  ∨ IF memory.register["sched_CC"] ≠ ⟨⟩
    THEN ∧ memory' = dispatch_withIntMask("sched_CC", memory.register["sched_CC"])
         ∧ UNCHANGED timers
    ELSE ∃ idInt ∈ interruptionsSetCC(memory) :
         LET nmem = initTaskInterruption(idInt, memory)
         IN  ∧ memory' = dispatch_withIntMask("sched_CC", nmem)
             ∧ UNCHANGED timers
  ∨
    ∧ memory.register["sched_XC"] = ⟨⟩
    ∧ memory.register["sched_CC"] = ⟨⟩
    ∧ interruptionsSetXC(memory) = {}
    ∧ interruptionsSetCC(memory) = {}
    ∧ UNCHANGED ⟨memory, now, timers⟩
```

Listing 6.42 – Nouvelle action d'arrêt (version multi-cœur

Les propriétés d'exclusion mutuelle ont été vérifiées correctement sur la spécification TLA+ du micronoyau. Comme pour la version mono-cœur, les propriétés de vivacité $interruptionSyncSortListSent$, $handleSortListCalled$, $interruptionSyncPreemtionSent$ et $handlePreemptionNotifCalled$ ne sont vérifiées qu'en redéfinissant l'action pour laquelle now atteint $maxTime$. Cette action est spécifiée dans le listing 6.42. Elle consiste à continuer à exécuter les routines d'interruption inachevées dans chaque cœur. La différence par rapport à la version mono-cœur est que les deux cœurs s'envoient mutuellement des interruptions inter-processeurs en exécutant les routines « *syscall* », « *syncPreemption* » et « *syncSortList* ». Il convient alors, lorsque now atteint $MaxTime$, de traiter les interruptions envoyées et de désactiver l'envoi de nouvelles interruptions inter-processeurs. Pour désactiver l'envoi des nouvelles interruptions inter-processeurs,

nous avons redéfini un opérateur $dispatch_withIntMask()$ ayant la même définition que l'opérateur $dispatch()$, à l'exception que ce dernier dépile directement le $mem.stack$ du processus qui fait appel à l'une des fonctions `sync_notify_sortList()` et `sync_notify_preemption` au lieu d'appeler les opérateurs TLA+ définis dans les listings 6.31 et 6.34.

6.4.4.3 Propriétés invalides

La vérification de la propriété $overrunDetection$ (définie dans la section 6.24) a échoué. Le scénario d'exécution causant l'erreur repose sur le même principe de scénario défini dans la section 6.3.4.2.

En ce qui concerne l'absence de famine, la vérification de la propriété $deadlockFree$ a échoué. Cet échec se produit dans le cas où un dépassement d'échéance est détecté et une exception « *overrun* » est produite. Le scénario est le suivant : le cœur d'exécution accède à la section critique. Le cœur essaye d'y accéder, mais il se met en attente. Le cœur d'exécution vérifie l'échéance de la tâche courante, qui est dépassée. Il envoie alors une exception « *overrun* » et le système s'interrompt. L'échec de cette propriété est un résultat qui est dû à l'exception « *overrun* » produite par le micronoyau. Ce cas correspond à une vraie erreur.

6.5 CONCLUSION

Dans ce chapitre, nous avons appliqué la méthodologie que nous proposons dans cette thèse sur un cas d'étude réel issu de l'implémentation du micronoyau PharOS. Ce dernier dispose de deux implémentations, une version destinée pour une architecture mono-cœur et une version à une architecture multi-cœur. Nous avons considéré dans cette étude les deux versions de PharOs. Pour cela, nous avons utilisé dans un premier temps C2TLA+ afin de traduire l'implémentation C du micronoyau en TLA+, et ceci pour les deux versions d'implémentation. Dans un second temps, nous avons étendu le modèle d'exécution de C2TLA+ pour modéliser l'environnement de celui-ci, à savoir les aspects matériels comme la gestion des *timers* et des interruptions. Pour la version multi-cœur, nous avons étendu le modèle d'exécution afin de gérer les primitives de synchronisation entre les cœurs lors des accès concurrents aux sections critiques.

À partir des spécifications TLA+ du micronoyau des versions mono et multi-cœur, on s'est intéressé aux propriétés fonctionnelles que PharOS doit vérifier. Afin de réduire la complexité du *model-checking* lors de la vérification de ces propriétés, nous avons appliqué un prédicat d'agglomération basé sur une analyse syntaxique. La transformation induite par ce prédicat consiste à agglomérer toute instruction C ne faisant pas intervenir une lecture ou une écriture dans une variable partagée par plusieurs processus. Les résultats obtenus montrent que la réduction par agglomération permet de réduire significativement l'espace d'états ainsi que le temps de vérification.

Il est important de souligner que dans cette étude de cas, nous ne nous sommes

pas intéressés à la vérification des contraintes temps-réels que les tâches doivent respecter. Nous nous sommes intéressés à la vérification des propriétés fonctionnelles du micronoyau et son interaction avec son environnement. L'adaptation du modèle proposée dans ce chapitre reste possible et doit faire l'objet d'un travail complémentaire.

Conclusion

Les travaux de recherche menés dans cette thèse avaient pour objectif de proposer une méthodologie d'aide à la conception des logiciels systèmes critiques permettant leur spécification et leur vérification formelle. Nous nous sommes intéressés plus particulièrement aux logiciels systèmes implémentés en C. Les travaux effectués ont abouti aux propositions suivantes :

- Formalisation de la sémantique d'un sous-ensemble C en TLA+ : la méthodologie proposée dans cette thèse est basée sur la traduction de la sémantique de C en une sémantique opérationnelle exprimée dans le langage de spécification formel TLA+. Pour cela, nous avons proposé un modèle de mémoire et d'exécution permettant de modéliser les programmes C séquentiels. En se basant sur ce modèle, nous avons défini un ensemble de règles de traduction d'un code C en TLA+ que nous avons implémenté dans l'outil C2TLA+. Comme TLA+ est un langage formel, cette traduction fournit une sémantique formelle du C. Une fois la spécification TLA+ générée par C2TLA+, nous pouvons vérifier un ensemble de propriétés de sûreté et de vivacité exprimées en TLA+ grâce au *model-checker* TLC. L'avantage du modèle d'exécution proposé est qu'il est facilement extensible.

- Spécification des programmes concurrents : en se basant sur le modèle de mémoire et d'exécution défini précédemment, nous avons étendu ce dernier afin de gérer les programmes concurrents, les aspects liés à la synchronisation et la gestion des processus et leur ordonnancement. Nous avons pour cela proposé une bibliothèque contenant des définitions de primitives matérielles et logicielles de synchronisation facilement extensible. Nous avons montré par la suite comment spécifier des processus avec différentes politiques d'ordonnancement et de contraintes temporelles.

- Technique de réduction de l'espace d'états : pour réduire la complexité de la vérification, nous avons proposé une technique de réduction de l'espace d'états. Cette réduction consiste pour un code C à agglomérer une suite d'instructions lors de la génération du code TLA+, sous réserve d'un ensemble de conditions. Cette réduction se fait en définissant un prédicat d'agglomération qui détermine si une instruction est agglomérable ou pas. Nous avons proposé un exemple de

prédicat basé sur une analyse syntaxique du code C. Nous avons décrit comment cette technique s'applique dans C2TLA+ afin de générer une spécification TLA+ réduite. Nous avons montré sur un ensemble d'expérimentations que la technique de réduction permet de réduire le nombre d'états de manière considérable ainsi que le temps nécessaire à la vérification.

- Vérification du micronoyau PharOS : nous avons utilisé C2TLA+ afin de spécifier formellement l'implémentation C du micronoyau PharOS. Nous avons considéré deux implémentations, une version monoprocesseur et une version dédiée pour un processeur multi-cœur. À partir du code TLA+ généré à partir du code C, nous avons étendu le modèle avec des modules TLA+ modélisant l'environnement matériel du micronoyau comme les timers et la gestion des interruptions. Nous avons ensuite appliqué la technique de réduction basée sur l'agglomération des instructions C. Nous avons spécifié et vérifié un ensemble de propriétés de sûreté et de vivacité qui porte sur le fonctionnement du micronoyau. Les résultats obtenus montrent bien l'efficacité de la technique d'agglomération dans la réduction de la complexité de vérification en terme de temps et d'espace d'états.

PERSPECTIVES

Les résultats obtenus dans cette thèse permettent d'explorer de nombreuses perspectives. Nous présentons dans ce qui suit les principales perspectives sous la forme de trois axes de recherche.

Extension du sous-ensemble de C considéré

Une perspective évidente consiste en l'extension de C2TLA+ afin de supporter les aspects du C non gérés jusqu'à présent. Nous présentons dans ce qui suit chaque aspect en décrivant comment il est possible de le supporter dans C2TLA+.

- Affectation des structures de données : cet aspect peut, facilement, être supporté en définissant des opérations d'affectations entre les champs de structures.

- L'allocation dynamique de la mémoire peut être modélisée en étendant le modèle mémoire avec une nouvelle région mémoire représentant le *tas* (« *heap* ») du programme. Cette région mémoire peut être modélisée par une séquence TLA+ dans laquelle les opérations d'allocation et de désallocation de mémoire consistent en des opérations sur cette séquence.

- La prise en compte des tailles réelles d'allocation : actuellement, nous avons considéré que tous les objets de base C (entier, pointeur, structure) avaient une taille 1, ce qui peut poser problème pour certains codes dépendants de l'architecture. La principale nouveauté pour les gérer est la nécessité d'introduire une valeur représentant une « tranche de pointeur », dont on a besoin lorsqu'on copie un

pointeur octet par octet.

- Représentation du type `float` : pour modéliser cet aspect, il serait inutile de modifier le modèle mémoire, car on peut représenter les opérations flottantes comme prenant 2 entiers de même taille, les convertir en triple (signe, mantisse, exposant), faire l'opération flottante, puis reconvertir le résultat en binaire.

- Pointeurs de fonction : il est possible de gérer cet aspect en définissant un nouveau champ dans l'adresse d'un pointeur. Ce champ spécifie si le pointeur pointe vers une fonction et si c'est le cas, il indique son identificateur et l'identifiant de sa première instruction. Il suffirait alors de prendre en compte cela dans le traitement de l'instruction d'appel de fonction.

Une fois que tous les aspects du C sont supportés, une piste de recherche (à long terme) consiste à prouver que la sémantique proposée est conforme à la sémantique définie dans CompCert [Ler09].

Vérification de l'implémentation d'un code C par rapport à sa spécification

Une autre piste de recherche consiste à permettre à l'utilisateur de vérifier si un code C est conforme à sa spécification. Dans ce cas, d'une part le code C est traduit en TLA+ que nous appelons une *implémentation* et d'autre part une spécification de ce que fait le code C est définie en TLA+, que nous appelons une *spécification abstraite*. L'utilisateur définit la relation de « *mapping* » entre les variables de l'implémentation avec celle de la spécification abstraite. Nous extrayons ensuite les automates (LTS) associés aux spécifications TLA+ et nous les passons à un outil de vérification de relation de simulation entre les automates. Une piste résultante de cette idée consiste à réaliser des vérifications modulaires, dans laquelle une partie du code C pourrait être remplacé par sa spécification TLA+ abstraite, permettant ainsi de diminuer le nombre d'états. Cette idée implique une preuve de congruence des spécifications TLA+. Nous avons commencé durant la thèse à implémenter l'idée de vérifier un code C par rapport à une spécification TLA+ abstraite. Nous l'avons proposé dans [MLH+15]. Mais vu qu'elle nécessitait beaucoup d'effort de formalisation ainsi que d'implémentation, nous avons choisi de ne pas approfondir cette piste et de la considérer comme perspective.

Utilisation de TLAPS comme vérificateur de preuve

Nous pouvons imaginer que l'utilisateur puisse inclure les propriétés à vérifier dans le code C directement, à travers des annotations ACSL [BFM+10]. À partir du code annoté, des obligations de preuve qui garantissent la correction fonctionnelle du code C seront générées. Ces obligations seront ensuite traduites en TLA+. À ce stade, le prouveur TLAPS (ou le model-checker TLC) peut être utilisé pour réaliser les preuves du code C. Pour rendre cela pratique, le générateur devra générer des lemmes utiles à la réalisation de la preuve ; par exemple pour exprimer qu'une variable du programme n'est modifiée que par certaines instructions.

On peut aussi envisager d'utiliser TLAPS comme un vérificateur de preuve ; c'est-à-dire que plutôt que de ne générer que des obligations de preuves, on générerait des théorèmes avec leurs preuves, et on demanderait à TLAPS de les vérifier.

Bibliographie

[Abr05] Jean-Raymond Abrial. *The B-book : Assigning Programs to Meanings*. Cambridge University Press, 2005.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and computation*, 104(1) :2–34, 1993.

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2) :183–235, April 1994.

[AMQ10] Sabina Akhtar, Stephan Merz, and Martin Quinson. A High-Level Language for Modeling Algorithms and Their Properties. In Jim Davies, Leila Silva, and Adenilso da Silva Simão, editors, *SBMF*, volume 6527 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2010.

[AWS16a] Inc. Amazon Web Services. Amazon DynamoDB Documentation. Disponible à `https://aws.amazon.com/fr/documentation/dynamodb`, 2016.

[AWS16b] Inc. Amazon Web Services. Amazon Elastic Block Store. Disponible à `https://aws.amazon.com/fr/ebs`, 2016.

[BBBB09] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. *Computer Safety, Reliability, and Security : 28th International Conference, SAFECOMP 2009, Hamburg, Germany, September 15-18, 2009. Proceedings*, chapter Formal Verification of a Microkernel Used in Dependable Software Systems, pages 187–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor : A Successful Application of B in a Large Project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer Berlin Heidelberg, 1999.

[BC10] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development : Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-critical Software. *SIGPLAN Not.*, 38(5) :196–207, May 2003.

[BCCL08] François Bobot, Sylvain Conchon, Évelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In Clark Barrett and Leonardo de Moura, editors, *SMT 2008 : 6th International Workshop on Satisfiability Modulo*, volume 367 of *ACM International Conference Proceedings Series*, pages 1–5, 2008.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg, 1999.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking : 10^{20} States and beyond. *Information and Computation*, 98(2) :142 – 170, 1992.

[BDL04] Gerd Behrmann, Alexandre David, and KimG. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[Ber85] Gérard Berthelot. *Checking properties of nets using transformations*, volume 222. Springer-Verlag, 1985.

[BFM+10] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL : ANSI/ISO C Specification Language. Technical report, CEA LIST and INRIA, 2010.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3 : Shepherd Your Herd of Provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[BL02] Brannon Batson and Leslie Lamport. High-Level Specifications : Lessons from Industry. In *FMCO'02*, pages 242–261, 2002.

[BR01] Thomas Ball and SriramK. Rajamani. The SLAM Toolkit. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer Berlin Heidelberg, 2001.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, C-35(8) :677–691, Aug 1986.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, USA, 1977. ACM.

[CC92] Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4) :511–547, 1992.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 334–342. Springer-Verlag New York, Inc., 2014.

[CCF+07] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, and Xavier Rival. The AstréeA Static Analyzer. Disponible à `http://www.astreea.ens.fr`, 2007. Consulté en Novembre 2015.

[CCGR99] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, Lecture Notes in Computer Science, pages 495–499. Springer, 1999.

[CCM09] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A Value Analysis for C Programs. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 123–124, Washington, DC, USA, 2009. IEEE Computer Society.

[CDC93] Rosvelter Joao Coelho Da Costa. *Systèmes de transitions étiquetés causaux : une nouvelle approche pour la description du comportement évènementiel de systèmes concurrents*. PhD thesis, Toulouse 3, 1993.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude : specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2) :187 – 243, 2002.

[CDFH97] Giovanni Chiola, Claude Dutheillet, Giuliana Franceschinis, and Serge Haddad. A symbolic reachability graph for coloured Petri nets. *Theoretical Computer Science*, 176(1-2) :39–65, April 1997.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC : A Practical System for Verifying Concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.

[CDMM14] Sylvain Conchon, David Declerck, Luc Maranget, and Alain Mebsout. Vérification de programmes C concurrents avec Cubicle : enfoncer les barrières. In *Les vingt-cinquièmes Journées francophones des langages applicatifs, Fréjus, France*, pages 17–32, 2014.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CEA16] CEA-LIST. Slicing plug-in. Disponible à `http://frama-c.com/slicing.html`, 2016. Consulté en juin 2015.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, April 1986.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer Berlin Heidelberg, 2000.

[CGK+12] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. Cubicle : A Parallel SMT-Based Model Checker for Parameterized Systems - Tool Paper. In *CAV*, pages 718–724, 2012.

[CGMP99] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State Space Reduction using Partial Order Techniques. *International Journal on Software Tools for Technology Transfer*, 2(3) :279–287, 1999.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C : A Software Analysis Perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CL11] CEA-List. Rapport d'activité du CEA-LIST 2011. Technical report, CEA-LIST, 2011. Disponible à `http://www-list.cea.fr/images/stories/decouvrir-le-cea-list/qui-sommes-nous/rapport-dactivite/Rapport_activite_CEA_LIST_2011.pdf`.

[CL12] CEA-List. Systèmes embarqués pour l'automobile : une nouvelle dynamique pour accélérer l'innovation dans la filière électronique automobile. Rapport, CEA-LIST, Septembre 2012.

[CL13] CEA-LIST. Frama-C. Disponble à http://frama-c.com/features.html, 2013. Consulté en janvier 2013.

[CL15a] CEA-LIST. Aoraï plug-in. Disponible à http://frama-c.com/aorai.html, 2015. Consulté en novembre 2015.

[CL15b] CEA-LIST. The Mthread Plugin. Disponible à http://frama-c.com/mthread.html, 2015.

[Cla08] Edmund M. Clarke. The Birth of Model Checking. In *25 Years of Model Checking*, pages 1–26, 2008.

[Cor10] Microsoft Corporation. The TLA Toolbox. Disponible à http://research.microsoft.com/en-us/um/people/lamport/tla/tools.html, 2010. Consulté en octobre 2012.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal Methods : State of the Art and Future Directions. *ACM Comput. Surv.*, 28(4) :626–643, December 1996.

[DD89] Philippe Darondeau and Pierpaolo Degano. Causal Trees. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simonetta Ronchi Della Rocca, editors, *Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 234–248. Springer Berlin Heidelberg, 1989.

[Dij65] Edsger W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Commun. ACM*, 8(9) :569–, September 1965.

[Dij68] Edsger W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages : NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[Dij70] Edsger W. Dijkstra. Notes on Structured Programming. 1970.

[Dil96] David L. Dill. The Mur*phi* Verification System. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 390–393, London, UK, 1996. Springer-Verlag.

[DM06] Alastair F. Donaldson and Alice Miller. Exact and Approximate Strategies for Symmetry Reduction in Model Checking. In *Proceedings of the 14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 541–556. Springer, 2006.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3 : An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify : a Theorem Prover for Program Checking. *J. ACM*, 52(3) :365–473, May 2005.

[Dut14] Bruno Dutertre. *Computer Aided Verification*, chapter Yices 2.2, pages 737–744. Springer International Publishing, 2014.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" Revisited : On Branching Versus Linear Time Temporal Logic. *J. ACM*, 33(1) :151–178, January 1986.

[ER11] German Federal Ministry Of Education and Research. The Verisoft XT Project. `http://www.verisoftxt.de`, 2011.

[ER12] Chucky Ellison and Grigore Rosu. An Executable Formal Semantics of C with Applications. *SIGPLAN Not.*, 47(1) :533–544, January 2012.

[Eva05] Sami Evangelista. High Level Petri Nets Analysis with Helena. In Gianfranco Ciardo and Philippe Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 455–464. Springer Berlin Heidelberg, 2005.

[Fit90] Melvin Fitting. *First-order Logic and Automated Theorem Proving*. Springer-Verlag New York, Inc., New York, USA, 1990.

[Flo67] R. W. Floyd. Assigning Meaning to Programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.

[GHP95] Patrice Godefroid, GerardJ. Holzmann, and Didier Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3) :227–241, 1995.

[GJMS12] M. M. Gallardo, C. Joubert, P. Merino, and D. Sanán. A Model-extraction Approach to Verifying Concurrent C Programs with CADP. *Sci. Comput. Program.*, 77(3) :375–392, March 2012.

[GL03] Eli Gafni and Leslie Lamport. Disk Paxos. *Distrib. Comput.*, 16(1) :1–20, February 2003.

[GLMS13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011 : a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2) :89–107, 2013.

[GNRZ08] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. *Automated Reasoning : 4th International Joint Conference, IJCAR 2008 Sydney, Australia, August 12-15, 2008 Proceedings*, chapter Towards SMT Model Checking of Array-Based Systems, pages 67–82. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[God91] Patrice Godefroid. Using partial orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer Berlin Heidelberg, 1991.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems : An Approach to the State-Explosion Problem*, volume 1032. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[God97] Patrice Godefroid. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, 1997.

[Hai80] Brent Tzion Hailpern. *Verifying Concurrent Processes Using Temporal Logic*. PhD thesis, Stanford, CA, USA, 1980.

[HCE+15] Arvind Haran, Montgomery Carter, Michael Emmi, Akash Lal, Shaz Qadeer, and Zvonimir Rakamaric. SMACK+Corral : A Modular Verifier (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, March 2015.

[HJJJ86] Peter Huber, Arne M. Jensen, Leif O. Jepsen, and Kurt Jensen. Reachability Trees for High-level Petri Nets. *Theor. Comput. Sci.*, 45(3) :261–292, 1986.

[HJMQ03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-Modular Abstraction Refinement. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 262–274, 2003.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 58–70, New York, USA, 2002. ACM.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. *Model Checking Software : 10th International SPIN Workshop Portland, OR, USA, May 9–10, 2003 Proceedings*, chapter Software Verification with BLAST, pages 235–239. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8) :666–677, August 1978.

[Hol87] Gerard J. Holzmann. On limits and possibilities of automated protocol analysis. In *Proceedings of the IFIP WG6.1 Seventh International Conference on Protocol Specification, Testing and Verification VII*, pages 339–344,

Amsterdam, The Netherlands, The Netherlands, 1987. North-Holland Publishing Co.

[Hol96] Gerard J. Holzmann. An Analysis of Bistate Hashing. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 301–314, London, UK, 1996. Chapman & Hall, Ltd.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5) :279–295, 1997.

[Hol00] Gerard J. Holzmann. Logic Verification of ANSI-C Code with SPIN. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2000.

[HP06] Serge Haddad and Jean-François Pradat-Peyre. New efficient Petri nets reductions for parallel programs verification. *Parallel Processing Letters*, 16(1) :101–116, March 2006.

[HS99] Gerard J. Holzmann and Margaret H. Smith. A Practical Method for Verifying Event-driven Software. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 597–607, New York, USA, 1999. ACM.

[INT+16] Omar Inverso, Truc L. Nguyen, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. CSeq. Disponible à `http://users.ecs.soton.ac.uk/gp4/cseq/`, 2016.

[ITF+14] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. *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*, chapter Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization, pages 585–602. Springer International Publishing, 2014.

[Jea96] Jean-Dominique Merchet. L'explosion d'Ariane 5 : c'est la faute au logiciel "Nous sommes tous coupable". Disponible à `http://www.liberation.fr/futurs/1996/07/24/l-explosion-d-ariane-5-c-est-la-faute-au-logicielnous-sommes-tous-coupables-a-affirme-le-directeur-g_176415`, Juillet 1996.

[Jen81] Kurt Jensen. Coloured Petri Nets and the Invariant-method. *Theoretical Computer Science*, 14(3) :317 – 336, 1981.

[JLM+03] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, and Yuan Yu. Checking Cache Coherence Protocols with TLA+. *Journal of Formal Methods in System Design*, 22(2) :125–131, March 2003.

[JM05] Mauro Jaskelioff and Stephan Merz. Proving the Correctness of Disk Paxos Isabelle/HOL. Technical report, LORIA, Nancy, France, 6 2005.

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1) :112–126, Jan 2003.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4 : Formal Verification of an OS Kernel. In *SOSP*, pages 207–220, New York, USA, 2009.

[Kel76] Robert M. Keller. Formal Verification of Parallel Programs. *Commun. ACM*, 19(7) :371–384, July 1976.

[KJ09] Bengt Jonsson Ke Jiang. Using SPIN to Model Check Concurrent Algorithms, Using a Translation From C to Promela. In *Proc. 2nd Swedish Workshop on Multi-Core Computing*, 2009.

[KL04] Matthias Kuntz and Kai Lampka. Probabilistic Methods in State Space Analysis. In Christel Baier, BoudewijnR. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors, *Validation of Stochastic Systems*, volume 2925 of *Lecture Notes in Computer Science*, pages 339–383. Springer Berlin Heidelberg, 2004.

[Lam74] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM*, 17(8) :453–455, August 1974.

[Lam83] Leslie Lamport. What Good is Temporal Logic ? In Elsevier Publishers R. E. A. Mason, ed., editor, *Congress on Information Processing*, pages 657–668, 1983.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3) :872–923, 1994.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Lam09] Leslie Lamport. The PlusCal Algorithm Language. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, ICTAC '09, pages 36–60, Berlin, Heidelberg, 2009. Springer-Verlag.

[LB08] Xavier Leroy and Sandrine Blazy. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. Autom. Reason.*, 41(1) :1–31, July 2008.

[LDaGVN10] Matthieu Lemerre, Vincent David, and Christophe Aussaguès a Guy Vidal-Naquet. An introduction to time-constrained automata. In *EPTCS :*

Proceedings of the 3rd Interaction and Concurrency Experience Workshop, volume 38, page 83–98, 2010.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009.

[Ler15] Xavier Leroy. The CompCert verified compiler. Disponible à `http://compcert.inria.fr`, Décembre 2015.

[Les15] Stéphane Lescuyer. ProvenCore : Towards a Verified Isolation Micro-Kernel. 2015.

[LGS+95] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1) :11–44, 1995.

[LOC+11] Matthieu Lemerre, Emmanuel Ohayon, Damien Chabrol, Mathieu Jan, and Marie-Bénédicte Jacques. Method and Tools for Mixed-Criticality Real-Time Applications within PharOS. In *Proceedings of AMICS 2011 : 1st International Workshop on Architectures and Applications for Mixed-Criticality Systems*, 2011.

[LPD+11] Guodong Li, Robert Palmer, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Formal specification of MPI 2.0 : Case study in specifying a practical concurrent programming API . *Science of Computer Programming*, 76(2) :65 – 81, 2011.

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral : A solver for reachability modulo theories. Technical Report MSR-TR-2012-9, January 2012.

[LY92] David Lee and Mihalis Yannakakis. Online minimization of transition systems (extended abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 264–274, New York, USA, 1992. ACM.

[Mat16] MathWorks. The mathworks Polyspace Verifier. Disponble à `http://fr.mathworks.com/products/polyspace/`, 2016. Consulté en janvier 2016.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking : an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[Mer74] Philip Meir Merlin. *A Study of the Recoverability of Computing Systems*. PhD thesis, 1974.

[Mer03] Stephan Merz. On the Logic of TLA+. *Computers and Artificial Intelligence*, 22(3-4) :351–379, 2003.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1) :73–155, 1992.

[Mey92] Bertrand Meyer. *Eiffel : The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MLH+15] Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Serge Haddad, and Kamel Barkaoui. Verifying and Constructing Abstract TLA Specifications : Application to the Verification of C programs. In *10th International Conference on Software Engineering Advances*, pages 32–39, Barcelone, Spain, 2015.

[MLW11] Stephan Merz, Tianxiang Lu, and Christoph Weidenbach. Towards Verification of the Pastry Protocol using TLA+. In R. Bruni and J. Dingel, editors, *31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, volume 6722 of *FMOODS/FORTE 2011*, Reykjavik, Iceland, June 2011.

[MM15] Claude Marché and Yannick Moy. The Jessie plugin for Deductive Verification in Frama-C. Disponible à `http://krakatoa.lri.fr/jessie.html`, mars 2015. Consulté en juin 2015.

[MPC+02] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC : A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 36(SI) :75–88, December 2002.

[Nec15] George Necula. CIL-Infrastructure for C Program Analysis and Transformation. Disponble à `http://www.cs.berkeley.edu/~necula/cil`, 2015. Consulté en janvier 2015.

[NMRW02] George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. CIL : Intermediate Language and Tools for Analysis and Transformation of C Programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NRZ+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Commun. ACM*, 58(4) :66–73, March 2015.

[Owi75] Susan Speer Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Ithaca, USA, 1975.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Schriften des IIM 2, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, 1962.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3) :115 – 116, 1981.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57, 1977.

[PPP00] Denis Poitrenaud and Jean-Francois Pradat-Peyre. Pre- and Post-agglomerations for LTL Model Checking. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 387–408. Springer Berlin Heidelberg, 2000.

[QW04] Shaz Qadeer and Dinghao Wu. KISS : Keep It Simple and Sequential. *SIGPLAN Not.*, 39(6) :14–24, June 2004.

[Ram74] Chander Ramchandani. Analysis of Asynchronous Concurrent Systems by Timed Petri Nets. Technical report, Cambridge, MA, USA, 1974.

[RE14] Zvonimir Rakamarić and Michael Emmi. *Computer Aided Verification*, chapter SMACK : Decoupling Source Language Details from Verifier Implementations, pages 106–113. Springer International Publishing, 2014.

[RŞ10] Grigore Roşu and Traian Florin Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6) :397–434, 2010.

[SC85] A. Prasad Sistla and Edmund M. Clarke. The Complexity of Propositional Linear Temporal Logics. *J. ACM*, 32(3) :733–749, July 1985.

[SCLP09] Julien Signoles, Loïc Correnson, Matthieu Lemerre, and Virgile Prevosto. *Plug-in Development Guide*. CEA LIST, Software Reliability Laboratory, 2009.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[TIF+14] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore Torre, and Gennaro Parlato. *Tools and Algorithms for the Construction and Analysis*

of Systems : 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, chapter MU-CSeq : Sequentialization of C Programs by Shared Memory Unwindings, pages 402–404. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[Tur36] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42) :230–265, 1936.

[Val89] Antti Valmari. Stubborn Sets for Reduced State Space Generation. In *Proceedings of the Tenth International Conference on Application and Theory of Petri Nets*, pages 1–22, 1989.

[Val91] Antti Valmari. A Stubborn Attack On State Explosion. In *Proceedings of the 2Nd International Workshop on Computer Aided Verification*, CAV ’90, pages 156–165, London, UK, 1991. Springer-Verlag.

[VAM96] François Vernadat, Pierre Azéma, and François Michel. Covering step graph. In Jonathan Billington and Wolfgang Reisig, editors, *Application and Theory of Petri Nets 1996*, volume 1091 of *Lecture Notes in Computer Science*, pages 516–535. Springer Berlin Heidelberg, 1996.

[VdJ07] Eric Verhulst and Gjalt de Jong. OpenComRTOS : An Ultra-Small Network Centric Embedded RTOS Designed Using Formal Modeling. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007 : Design for Dependable Systems*, volume 4745 of *Lecture Notes in Computer Science*, pages 258–271. Springer Berlin Heidelberg, 2007.

[vG90] Rob J. van Glabbeek. The Refinement Theorem for ST-Bisimulation Semantics. In M. Broy and C.B. Jones, editors, *Proceedings IFIP TC2 Working Conference on Programming Concepts and Methods*, pages 27–52, 1990.

[Wei84] Mark Weiser. Program slicing. *Software Engineering, IEEE Transactions on*, SE-10(4) :352–357, July 1984.

[WFLGN10] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic Program Repair with Evolutionary Computation. *Commun. ACM*, 53(5) :109–116, May 2010.

[WL93] Pierre Wolper and Denis Leroy. Reliable hashing without collision detection. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer Berlin Heidelberg, 1993.

[YB12] Boris Yakobowski and Richard Bonichon. Frama-C’s Mthread plug-in : Version 0.9 for Oxygen-20120901. Disponible à `http://frama-c.com/download/frama-c-mthread-manual.pdf`, 2012.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '99, pages 54–66, London, UK, 1999. Springer-Verlag.

Annexes

Traduction TLA+ d'un exemple de code C

Nous illustrons dans le listing A.1 la traduction TLA+ de l'exemple de code C donné par la figure 4.2a. Cette traduction consiste à appliquer la règle 4.1 sur le code C donné en entrée. Nous rappelons que le programme C à traduire contient une variable `x` partagée entre deux processus ayant respectivement les identifiants 1 et 2 et exécutant respectivement les fonctions `p1()` et `p2()`.

Le module $Parameters$ définit les adresses des variables partagées, locales et les paramètres utilisés dans le code C. Le module $Program$ définit la traduction des fonctions C. La fonction `p1()` est traduite en 5 opérateurs TLA+ et la fonction `p2()` est traduite en 3 opérateurs TLA+. Initialement, la mémoire $data$ contient la valeur initiale de `x`, à savoir 0. Les piles $stack[1]$ et $stack[2]$ contiennent respectivement les cadres de piles de `p1()` et `p2()`. Chaque cadre de pile contient une case mémoire pour stocker la valeur de la variable locale. Cette valeur est indéfinie et elle est désignée par l'enregistrement $[val \mapsto Undef]$. Les piles de registres $register[1]$ et $register[2]$ contiennent respectivement un enregistrement composé du champ pc qui spécifie l'identificateur de la première instruction à exécuter par le processus et le champ fp qui contient la valeur 1.

La formule $Spec$ modélise l'exécution concurrente des deux processus. À chaque état, un processus est choisi pour exécuter une action en appelant l'opérateur $dispatch()$ qui gère le flot d'exécution de ces processus.

```
┌──────────────────────────── MODULE Parameters ────────────────────────────┐
EXTENDS Naturals, Integers, Sequences, TLC
CONSTANTS ProcSet, Undef, Null
VARIABLE memory
Generating C variables addresses
Addr_x ≜ [loc ↦ "data", offs ↦ 0]
Addr_inc_param_i ≜ [loc ↦ "stack", offs ↦ 0]
Addr_inc_tmp ≜ [loc ↦ "stack", offs ↦ 1]
Addr_p1_a ≜ [loc ↦ "stack", offs ↦ 0]
Addr_p2_b ≜ [loc ↦ "stack", offs ↦ 0]
Generating macros of offsets and sizes
Size_of_int ≜ 1
└────────────────────────────────────────────────────────────────────────────┘

┌──────────────────────────── MODULE Program ────────────────────────────┐
EXTENDS Parameters
Translation of inc() function
tmp = i + 1 ;
stmt_inc_5(id, mem) ≜
  LET nmem ≜ store(id, mem, Addr_inc_tmp, plus(load(id, mem, Addr_inc_param_i), [val ↦ 1]))
  IN  [mem EXCEPT !.data = nmem.data, !.stack = nmem.stack,
                  !.register = update_reg(mem.register[id], "inc_6")]
return (tmp) ;
stmt_inc_6(id, mem) ≜
  [mem EXCEPT
     !.data = [mem.data EXCEPT ![id] = SubSeq(mem.stack[id], 1, Head(mem.stack[id]).fp − 1)],
     !.register = [mem.register EXCEPT ![id] = Tail(mem.register[id])],
     !.ret = [mem.ret EXCEPT ![id] = load(id, mem, Addr_inc_tmp)]]

Translation of p1() function
a = 1 ;
stmt_p1_10(id, mem) ≜
  LET nmem ≜ store(id, mem, Addr_p1_a], [val ↦ 1])
  IN  [mem EXCEPT !.data = nmem.data, !.stack = nmem.stack,
                  !.register = update_reg(mem.register[id], "p1_11_pre")]
x = inc(a) ;
stmt_p1_11_pre(id, mem) ≜
  [mem EXCEPT
     !.stack = [mem.stack EXCEPT ![id] = mem.stack[id] ∘ ⟨load(id, mem, Addr_p1_a), [val ↦ Undef]⟩],
     !.register = [mem.register EXCEPT ![id] = ⟨[pc ↦ "inc_5", fp ↦ Len(mem.stack[id]) + 1]⟩ ∘
       ⟨[pc ↦ "p1_11_post", fp ↦ Head(mem.register[id]).fp]⟩ ∘ Tail(mem.register[id])]
x = inc(a) ;
stmt_p1_11_post(id, mem) ≜
  LET nmem ≜ store(id, mem, Addr_x, mem.ret[id])
  IN  [mem EXCEPT !.data = nmem.data, !.stack = nmem.stack,
                  !.register = update_reg(mem.register[id], "p1_12")]
x ++ ;
stmt_p1_12(id, mem) ≜
  [mem EXCEPT
  LET nmem ≜ store(id, mem, Addr_x, plus(load(id, mem, Addr_x), [val ↦ 1]))
  IN  [mem EXCEPT !.data = nmem.data, !.stack = nmem.stack,
                  !.register = update_reg(mem.register[id], "p1_13")]
return ;
stmt_p1_13(id, mem) ≜
  [mem EXCEPT
     !.stack = [mem.stack EXCEPT ![id] = SubSeq(mem.stack[id], 1, Head(mem.register[id]).fp − 1)],
     !.register = [mem.register EXCEPT ![id] = Tail(mem.register[id])]]
```

```
Translation of p2() function
 b = 0 ;
stmt_p2_16(id, mem) ≜
  LET nmem ≜ store(id, mem, Addr_p2_b, [val ↦ 0])
  IN  [mem EXCEPT !.data = nmem.data, !.stack = nmem.stack,
                  !.register = update_reg(mem.register[id], "p2_17")]
 x += b ;
stmt_p2_17(id, mem) ≜
  LET nmem ≜ store(id, mem, Addr_x, plus(load(id, mem, Addr_x), load(id, mem, Addr_p2_b))),
  IN  [mem EXCEPT
      !.data = nmem.data, !.stack = nmem.stack,
      !.register = update_reg(mem.register[id], "p2_18")]
 return ;
stmt_p2_18(id, mem) ≜
  [mem EXCEPT
      !.stack = [mem.stack EXCEPT ![id] = SubSeq(mem.stack[id], 1, Head(mem.register[id]).fp − 1)],
      !.register = [mem.register EXCEPT ![id] = Tail(mem.register[id])]]

Init ≜ memory = [data ↦ (0 :> [val ↦ 0]),
                 register ↦ (1 :> ⟨[pc ↦ "p1_10", fp ↦ 1]⟩ @@ 2 :> ⟨[pc ↦ "p2_16", fp ↦ 1]⟩),
                 stack ↦ (1 :> ⟨[val ↦ Undef⟩ @@ 2 :> ⟨[val ↦ Undef]⟩),
                 ret ↦ [i ∈ ProcSet ↦ [val ↦ Undef]]

dispatch(id, mem) ≜
   CASE mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "inc_5" → stmt_inc_5(id, mem)
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "inc_6" → stmt_inc_6(id, mem)
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p1_10" → stmt_p1_10(id, mem)
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p1_11_pre" → stmt_p1_11_pre(id, mem)
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p1_11_post" → stmt_p1_11_post(id, mem)
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p1_12" → stmt_p1_12(id, mem)
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p1_13" → stmt_p1_13(id, mem)
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p2_16" → stmt_p2_16(id, mem)
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p2_17" → stmt_p2_17(id, mem)
     □ mem.register[id] ≠ ⟨⟩ ∧ Head(mem.register[id]).pc = "p2_18" → stmt_p2_18(id, mem)
     □ mem.register[id] = ⟨⟩ → mem
     □ OTHER → "error"

procsExec(mem) ≜ ProcSet

Next ≜ ∨ ∃ id ∈ procsExec(mem) :
             ∧ memory.register[id] ≠ ⟨⟩
             ∧ memory' = dispatch(id, memory)
       ∨ ∀ id ∈ procsExec(mem) :
             ∧ memory.register[id] = ⟨⟩
             ∧ UNCHANGED ⟨memory⟩

Spec ≜ Init ∧ □[Next]_⟨memory⟩ ∧ WF_⟨memory⟩(Next)
```

Listing A.1 – Traduction complète d'un exemple de code C

B

Traduction des primitives `spinlock_acquire()` et `spinlock_release()`

Nous illustrons dans les listing B.2 et B.3 la traduction TLA+ par C2TLA+ des primitives `spinlock_acquire()` et `spinlock_release()` dont les définitions C sont données dans la section 4.2.4.1. Nous rappelons que cette traduction résulte de la normalisation opérée par CIL sur le code C donné en entrée à C2TLA+. Le listing B.1 montre le code C normalisé de ces deux fonctions C.

Chaque instruction C est traduite en un opérateur TLA+ en se basant sur le principe de traduction décrit dans le chapitre 3. La fonction `spinlock_acquire()` est traduite en 7 opérateurs TLA+. L'opérateur $spinlock_acquire_stmt_1()$ est la traduction de l'instruction `while(1)`, qui transmet le contrôle d'exécution à l'instruction suivante dans le GFC. L'opérateur $spinlock_acquire_stmt_5_pre()$ est la traduction de l'appel à l'instruction `CAS()`. Cet opérateur met à jour la pile $mem.register[id]$ afin de pointer vers la première instruction de `CAS()` et insère un nouveau cadre de pile contenant les paramètres et les variables locales de `CAS()` dans la pile $mem.stack[id]$. Nous rappelons que cette instruction est atomique et sa définition TLA+ est donnée dans le listing 4.8. L'opérateur $spinlock_acquire_stmt_5_post()$ stocke la valeur retournée par la fonction `CAS()` dans la variable `tmp` dont l'adresse TLA+ est désignée par $Addr_spinlock_acquire_tmp$. L'opérateur $spinlock_acquire_stmt_6()$ teste la valeur contenue à l'adresse $Addr_spinlock_acquire_tmp$. Si cette valeur est différente de $[val \mapsto 0]$, le flot de contrôle est passé à l'instruction `goto while_0_break` (ayant l'identifiant "spinlock_acquire_7"). Sinon, le flot de contrôle est passé à l'instruction `while(1)` (ayant identifiant "spinlock_acquire_1"). L'opérateur $spinlock_acquire_stmt_7()$ passe le contrôle

```
1  int CAS (int *addr,int old, int new) __attribute__ ((Sync\_primitives,alias ("CAS")));
2  void spinlock_acquire(spinlock *s)
3  {  while (1) {
4       int tmp;
5       tmp = CAS(s,1,0);
6       if (! tmp)
7         { goto while_0_break; }
8     }
9     while_0_break: /* internal */ ;
10    return;
11 }
12
13 void spinlock_release(spinlock *s)
14 {
15    *s = 1;
16    return;
17 }
```

Listing B.1 – Normalisation CIL des fonctions spinlock_acquire() et spinlock_release()

à l'instruction skip qui elle transmet le contrôle à l'instruction return.

Traduction des instructions de *spinlock_release*()

(*s) = 1 ;

$$\begin{array}{l} spinlock_release_stmt_15(id,\ mem)\ \triangleq \\ \quad \text{LET } nmem\ \triangleq\ store(id,\ mem,\ load(id,\ mem,\ Addr_spinlock_release_param_s),\ [val \mapsto 1]) \\ \quad \text{IN}\ \ [mem \text{ EXCEPT } !.data = nmem.data,\ !.stack = nmem.stack, \\ \qquad !.register = update_reg(mem.register[id], \text{“spinlock_release_16”})] \end{array}$$

return ;

$$\begin{array}{l} spinlock_release_stmt_16(id,\ mem)\ \triangleq \\ \quad [mem \text{ EXCEPT} \\ \quad\ !stack = [mem.stack \text{ EXCEPT } ![id] = SubSeq(mem.stack[id],\ 1,\ Head(mem.register[id]).fp - 1)], \\ \quad\ !.register\ = [mem.register \text{ EXCEPT } ![id] = Tail(mem.register[id])]] \end{array}$$

Listing B.3 – Traduction de la fonction spinlock_release()

La fonction spinlock_release() est traduite en deux opérateurs. Le premier opérateur $spinlock_release_stmt15()$ met à jour la valeur pointée par s et passe le contrôle à l'instruction suivante. Le deuxième opérateur $spinlock_release_stmt16()$ dépile les piles $mem.register[id]$ et $mem.stack[id]$.

Traduction des instructions de *spinlock_acquire*()

while(1)

$spinlock_acquire_stmt_3(id,\ mem)\ \triangleq\ [mem\ \text{EXCEPT}$
$\quad !.register = update_reg(mem.register[id],$ "spinlock_acquire_5_pre"$)]$

$tmp = CAS(s, 1, 0)$;

$spinlock_acquire_stmt_5_pre(id,\ mem)\ \triangleq\ [mem\ \text{EXCEPT}$
$\quad !.stack = [mem.stack\ \text{EXCEPT}\ ![id] = mem.stack[id]$
$\qquad \circ \langle load(id,\ mem,\ Addr_spinlock_acquire_param_s),\ [val \mapsto 1],\ [val \mapsto 0], [val \mapsto\ Undef\rangle]],$
$\quad !.register =\ [mem.register\ \text{EXCEPT}\ ![id] = \langle [pc \mapsto$ "CAS_init", $fp \mapsto Len(mem.stack[id]) + 1]\rangle$
$\qquad \circ\ \ \langle [pc \mapsto$ "spinlock_acquire_5_post", $fp \mapsto Head(mem.register[id]).fp]\rangle$
$\qquad \circ\ Tail(mem.register[id])]]$

$spinlock_acquire_stmt_5_post(id,\ mem)\ \triangleq$
$\quad \text{LET}\ nmem\ \triangleq\ store(id,\ mem,\ Addr_spinlock_acquire_tmp,\ mem.ret[id])$
$\quad \text{IN}\ \ [mem\ \text{EXCEPT}\ !.data = nmem.data,\ !.stack = nmem.stack,$
$\qquad !.register = update_reg(mem.register[id],$ "spinlock_acquire_6"$)]$

if (*tmp*) ;

$spinlock_acquire_stmt_6(id,\ mem)\ \triangleq$
$\quad \text{IF}\ (load(id,\ mem,\ Addr_spinlock_acquire_tmp) \neq [val \mapsto 0])$
$\quad \text{THEN}\ [mem\ \text{EXCEPT}\ !.register\ = update_reg(mem.register[id],$ "spinlock_acquire_7"$)]$
$\quad \text{ELSE}\ \ [mem\ \text{EXCEPT}\ !.register\ = update_reg(mem.register[id],$ "spinlock_acquire_1"$)]$

goto while_0_break ;

$spinlock_acquire_stmt_7(id,\ mem)\ \triangleq$
$\quad [mem\ \text{EXCEPT}\ !.register\ = update_reg(mem.register[id],$ "spinlock_acquire_9"$)]$

while_0_break : /* internal */ ;

$spinlock_acquire_stmt_9(id,\ mem)\ \triangleq$
$\quad [mem\ \text{EXCEPT}\ !.register\ = update_reg(mem.register[id],$ "spinlock_acquire_10"$)]$

return ;

$spinlock_acquire_stmt_10(id,\ mem)\ \triangleq$
$\quad [mem\ \text{EXCEPT}$
$\quad !.stack = [mem.stack\ \text{EXCEPT}\ ![id] = SubSeq(mem.stack[id],\ 1,\ Head(mem.register[id]).fp - 1)],$
$\quad !.register\ = [mem.register\ \text{EXCEPT}\ ![id] =\ Tail(mem.register[id])]]$

Listing B.2 – Traduction de la fonction `spinlock_acquire()`