ORSAY
N^o d'ordre : 9747

UNIVERSITÉ DE PARIS-SUD 11
CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

le grade de docteur en sciences DE L'UNIVERSITÉ PARIS XI

PAR

Nicolas AYACHE

—×—

SUJET :

Vérification Formelle, Compositionnelle et Automatique de Systèmes de Composants

soutenue le 5 janvier 2010 devant la commission d'examen

MM.	**Brigitte Rozoy**	Présidente
	Emmanuelle Encrenaz	Rapporteur
	Laurence Pierre	Rapporteur
	Daniel Krob	Examinateur
	Christine Paulin-Mohring	Directrice de Thèse
	Loïc Correnson	Encadrant
	Franck Védrine	Encadrant

Nicolas Ayache
Laboratoire de Sûreté du Logiciel
CEA - Centre de Saclay, F-91191 Gif-sur-Yvette Cedex

Résumé Aujourd'hui, de nombreux systèmes à base de composants sont critiques ; leur fonctionnement requiert un niveau de confiance maximal. Les méthodes formelles comme le Model-Checking sont utilisées pour garantir les propriétés cruciales, avec un haut degré d'automatisation. Cependant, le Model-Checking souffre d'explosion combinatoire lorsque les systèmes deviennent trop grands. Cette thèse propose un cadre de description haut niveau, où la vérification accompagne la modélisation du système, et limite le problème de l'explosion par une vérification modulaire des composants du système.
L'approche se base sur les observateurs, qui s'exécutent en parallèle d'un système sans en modifier le comportement global et permettent de décrire des propriétés. La modularité de la vérification vient de la possibilité de remplacer avec sûreté tout ou partie d'un système par un autre. Ceci est rendu possible par l'utilisation conjointe du Model-Checking et de l'Interprétation Abstraite, ce qui assure le calcul d'un sur-ensemble des comportements du système. Alors que l'Interprétation Abstraite introduit des approximations, les observateurs permettent de distinguer des états qui auraient été fusionnés pour gagner la précision nécessaire. Ils pilotent les analyses en introduisant des instants d'observation. Décrits par l'utilisateur, ils permettent une mise au point interactive des états à fusionner.
Notre approche s'intègre dans le cycle de développement d'un système où raffinements et abstractions sont légions. Nous définissons le langage SystemD, proche de SystemC, intégrant la description, la spécification et la vérification de systèmes, tout en restant accessible au monde de l'ingénierie.

Abstract Nowadays, critical systems based on components are widely used and require a maximal level of trust. Formal methods such as Model-Checking are used to guarantee crucial properties, with a high level of automation. However, Model-Checking suffers from combinatorial explosion when the size of the system becomes too big. This thesis defines a framework for high level description, where verification accompanies the modelisation of the system, and limits the combinatorial explosion problem with a compositional verification of the system.
Our approach is based on observer systems. They are executed in parallel of the studied system without interfering on its global behavior, and allow the expression of system properties. The compositional aspect of the verification comes from the possibility to safely replace a part of or a whole system by another one. This is done through the conjoined use of Model-Checking and Abstract Interpretation, which assures the computation of a subset of the behaviors of the system. When Abstract Interpretation introduces approximations, observers can distinguish states that would have been joined otherwise to bring the needed precision. They guide the analysis and are described by the user who explicitly specifies instants of observation.
This framework is adapted to the conception flow of a system, where abstractions and refinements are intensively used. For experimental purposes, we developed the SystemD language, resembling SystemC, that integrates the description, the specification and the verification of a system, and stays close to the engineering world.

Remerciements

Soyons direct : rédiger ce manuscrit a été dur. Aujourd'hui, la rédaction est finie et il ne me reste qu'une "formalité" : remercier tout ceux qui ont rendu ce travail possible. Mais en fait c'est carrément difficile ça ! Comment rendre l'hommage qui leur ai dû, en deux pages, à toutes les personnes qui m'ont permi d'en arriver là ? Ça ne sera sûrement pas suffisant, mais je souhaiterais tout de même leur adresser quelques mots.

Je tiens tout d'abord à remercier mes rapporteurs, Emmanuelle Encrenaz et Laurence Pierre, pour avoir accepté de se pencher sur mes modestes travaux, pour les avoir plus que nettement améliorés grâce à leurs suggestions et enfin pour m'avoir appris tant de chose sur les systèmes matériels en si peu de temps.

Je remercie également Daniel Krob et Brigitte Rozoy d'avoir accepté d'être respecitvement examinateur et présidente du jury de ma défense de thèse, une journée mémorable pour moi malgré le vol de mes bouteilles de champagne >_<

Un énorme merci à Loïc Correnson et Franck Védrine pour m'avoir encadré et supporté pendant ces quatre ans. Arriver au bout a été difficile pour moi, mais ça n'a pas dû être facile pour vous non plus parce que j'en avais des choses à apprendre. Vous avez su être d'une patience sans faille et vous êtes venus à mon secours toutes les nombreuses fois où j'en ai eu besoin.

Mes prochains remerciements s'adressent à Christine Paulin, ma directrice de thèse, qui a su me remettre dans le droit chemin lorsque mes idées devenaient trop obscures, et qui ne m'a pas abandonné même dans les moments les plus critiques. Elle m'a trouvé une place dans son laboratoire lorsque je me suis retrouvé sans endroit où travailler ; sans cela, je ne sais pas combien de temps encore il m'aurait fallu pour finir.

Je remercie tous les membres de mes laboratoires d'accueil, le Laboratoire de Sûreté du Logiciel au CEA-LIST et l'équipe ProVal du LRI et de l'INRIA. J'ai énormément appris et j'ai passé de très bons moments en votre compagnie. J'ai une pensée toute particulière pour Patricia Mouy, Muriel Roger, Sarah Zennou, Géraud Canet, Pascal Cuoq, Nicolas Bertaux, François Bobot, Romain Bardou et Guillaume Melquiond.

En continuant sur les membres de mes laboratoires d'accueil, je tiens à remercier mes amis Alexandre Chapoutot, Olivier Bouissou et Michel Hirschowitz qui ont été les premiers à reconnaître mes pouvoirs surnaturels. Et maintenant je peux vous l'avouer les gars : je faisais exprès de perdre au poker. Vous pensez vraiment qu'on peut être aussi nul ? ! Allez, deux spéciales dédicaces : "je chèèèque" et "XR8U6...". Je donnerais cher pour partager un bureau à nouveau tous les quatre.

Un gros big up à mes amis qui ont toujours été là pour moi. Entre autres, Nicolas Bouquet, Vincent Dupont et Pierre Vacheret. Je ne peux pas vous expliquer à quel point votre amitié est précieuse et à quel point je vous kiffe.

Comment ne pas remercier également toute ma famille, surtout ma mère et mon père que j'aime tant. Vous avez été mes modèles pendant toutes ces années et je fais tout pour devenir quelqu'un d'aussi bien que vous.

Enfin, je tiens à clore cette section de remerciements en déclarant tout l'amour que j'ai pour elle à Lauranne Bérengué, ma chère et tendre sans qui je n'aurais pu parvenir au bout de ce manuscrit tant l'amour et le soutien qui nous lie m'a été primordial pendant mon doctorat. J'ai hâte que l'on puisse réaliser tous nos projets ensemble (oui oui, tous !).

Table des matières

Première partie

Système de Composants

CHAPITRE 1

Introduction

Au cours du XX^{ème} siècle, les humains ont appris à contrôler un puissant outil d'automatisation des tâches : l'ordinateur. Cet outil s'accompagne d'une science, l'informatique, qui a pour but d'expliquer ce qui est faisable ou non avec un tel outil, comment et avec quelle efficacité. Alors que les aspects théoriques ont pu être étudiés très tôt, il a fallu attendre la fin du siècle et l'avènement du transistor avant de voir les ordinateurs présents dans la vie quotidienne de toute société moderne. Aujourd'hui, des systèmes de composants mélangeant électronique et programmes informatiques se trouvent dans de nombreux domaines : téléphonie, ordinateurs personnels, automobiles, avionique, etc. On appelle *systèmes embarqués* les systèmes qui sont destinés à être intégrés dans de tels appareils et qui offrent des fonctionnalités particulières, comme un décodeur de flux vidéo dans une télévision par exemple. Parmi ces systèmes embarqués, certains sont qualifiés de *critiques* lorsque de leur bon fonctionnement dépendent la vie d'êtres humains ou d'énormes sommes d'argent. On distingue deux types de dysfonctionnement des systèmes. Il y a d'une part les problèmes matériels, qui se produisent lorsqu'un composant connaît un endommagement physique. D'autre part, il y a les problèmes conceptuels, appelés *bugs* par abus de langage, qui surviennent lorsqu'un composant ne réalise pas précisément les fonctionnalités pour lesquelles il a été conçu. Parmi les bugs les plus célèbres, on pourra citer celui de la fusée Ariane 5 [Boa96] (700 millions de dollars de perdus) ou encore celui du missile Patriot [GAO92] (28 morts) causés tous deux par des erreurs de conversion de représentations des nombres par les systèmes.

Il a donc fallu trouver des façons de développer des systèmes sûrs. Par sûreté, on entend le fait que le système est accompagné d'une spécification qu'il respecte. Trois notions se dégagent alors : celle de développement du système, celle de sa spécification et enfin la vérification que le système développé satisfait sa spécification. Des méthodologies sont apparues afin de donner un cadre qui prend en compte les contraintes bien distinctes de développement, de spécification et de validation. Parmi ces méthodologies, le cycle en V [FM92] (figure 1.1) est devenu un standard depuis les années 80.

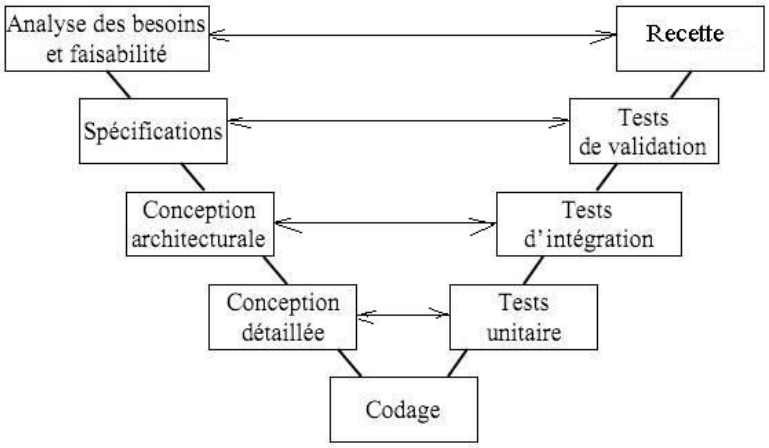

FIG. 1.1 – Cycle de développement en V

Plus on s'approche de la base du V, plus le travail se rapproche de celui de l'ingénieur. Les étapes de conception et de validation se correspondent afin d'identifier d'éventuels problèmes de l'un ou de l'autre et d'y remédier rapidement. La gestion d'un projet selon le cycle en V commence par identifier les fonctionnalités générales du système : c'est la branche gauche du V. Les étapes suivantes consistent à préciser le projet jusqu'à obtenir le code qui réalise le système. La seconde branche du cycle consiste à valider le système par des techniques de vérification, depuis le plus simple composant jusqu'au système tout entier.

Afin d'obtenir rapidement et à moindre coût une représentation du comportement d'un système, des langages de programmation ont vu le jour. Ils sont utilisés dans la partie codage du cycle ; on les appelle alors des langages de description de systèmes. En plus des avantages en coût de développement, les langages de description permettent d'automatiser une grande partie du travail de gestion de projet ; par exemple on peut construire des outils de tests pour la validation. Les étapes de conception et de validation doivent s'adapter aux contraintes liées aux langages.

Cette thèse s'intéresse tout particulièrement à la partie validation. Elle propose une méthode de vérification qui s'intègre dans le cycle en V et repose sur les langages de description. Nous commençons donc par présenter le type de langage de programmation particulier que sont les langages de description, avant d'introduire les techniques de validation principalement utilisées aujourd'hui sur tout type de programme et de présenter notre contribution à ces techniques.

1.1 Langages Synchrones

1.1.1 Généralités

Il existe différents types de systèmes selon les interactions qu'ils requièrent avec leur environnement. Les systèmes *réactifs* sont ceux qui contraignent le moins l'environnement et

autorisent le plus les interactions. Un système réactif doit gérer l'évolution dynamique de son environnement et doit en contrôler le comportement. Quelques exemples de systèmes réactifs sont : les jeux vidéo, le régulateur de papillon des gaz dans une automobile, le système de stabilisation d'un avion.

Comme les systèmes réactifs doivent être en permanence au service de leur environnement, il est souvent difficile de traiter tous les cas possibles d'imbrication des actions de l'environnement et du système. Pour pallier cette difficulté, la plupart des systèmes et des langages de description s'appuient sur le *paradigme synchrone* : on suppose que l'émission d'une requête sur un système provoque une réponse instantanée de sa part. En pratique, le paradigme synchrone s'exprime par le fait que la réponse du système à une requête est plus rapide que l'émission de la requête suivante. Le paradigme synchrone permet, comme son nom l'indique, de synchroniser les composants d'un système pour qu'ils communiquent avec une horloge commune ; on contrôle ainsi les imbrications de leurs actions.

Mais même si les langages de description s'appuient tous sur le paradigme synchrone, ils adoptent différentes positions dans la structure même des langages d'une part et dans leur niveau de description d'autre part.

Pour ce qui est des différences structurelles, certains langages sont fonctionnels et orientés flot de données, comme Lucid Synchrone [Luc] où les systèmes sont représentés par des signaux qui sont des séquences infinies de valeurs. D'autres sont impératifs, comme Esterel [Tec], voire orientés objet comme SystemC [Sysb]. Tous modélisent des systèmes et leurs comportements. Cependant, le type de langage n'est qu'un obstacle technique et ne change pas la méthode générale de vérification. Ce qui nous intéresse ici, c'est le niveau de description que permettent les langages.

Les langages de description ou HDL (pour *Hardware Description Langages*) modélisent des systèmes en abstrayant des détails très bas niveau comme leur implantation physique (matériaux, longueurs des fils, etc). Ils donnent une vue plutôt structurelle ou plutôt comportementale des systèmes. Dans une vue structurelle, un système est considéré comme un assemblage de composants et l'accent est mis sur les interconnexions entre ces composants. On peut utiliser des portes logiques pour décrire des systèmes comme des bus, des processeurs, des unités mémoires. Dans une vue comportementale, un système est une fonction de transfert qui explicite comment les sorties réagissent selon les entrées. C'est la vue la plus abstraite qui se décrit en général par des algorithmes très expressifs. Dans une vue comportementale, on parle de Register Transfert Modelling [TLN+89], ou RTL, lorsqu'un système est défini en termes d'envois et de réceptions de signaux, d'opérations logiques sur ces signaux, et de transferts de données entre registres.

Parmi les langages de description les plus utilisés en industrie, citons Verilog [Ver], VHDL [VHD] et SystemC. SystemC est très haut niveau et permet un degré d'abstraction supplémentaire par rapport à RTL que ne permet pas les deux autres : le Transaction Level Modelling [Ghe06, CG03] ou TLM. En TLM, les détails de communication entre composants peuvent être abstraits. La modularité de SystemC permet de définir des interfaces que les canaux de communication doivent respecter et de les instancier différemment a posteriori. La définition d'un canal de communication peut donc contenir le protocole de communication sur ce canal (signal simple, FIFO, mutex, ethernet, etc).

Dans nos travaux, nous nous sommes particulièrement intéressé au langage SystemC car c'est le langage le plus complexe d'un point de vue algorithmique, et qu'il nécessite d'adapter les techniques de validation à son haut niveau de description.

Quel que soit le langage, plusieurs notions sont au cœur des descriptions de systèmes. La notion la plus fondamentale est celle de *composant*. Tous les langages de description permettent de définir des composants qui seront amenés à être interconnectés. On peut se représenter un composant comme une boîte contenant au moins les éléments suivants :

- une interface : elle définit les points de communication, ou *ports*, avec l'environnement extérieur (les autres composants, un être humain, la force du vent, etc) ;
- une fonction de transfert : elle définit le comportement du composant. Elle est souvent représentée sous forme d'un processus.

La seconde notion commune à ces langages est celle d'*évènement*. Les évènements sont produits par les fonctions de transfert des composants et sont la brique de base de la communication entre composants. En effet, les composants sont reliés entre-eux par des canaux de communication. Ces canaux véhiculent des informations que les composants peuvent émettre ou recevoir via leurs ports. Les processus des composants définissent la vitesse des écritures et les valeurs écrites en fonction des valeurs lues. Ceci permet donc l'échange d'informations entre composants (par exemple des valeurs de types structurés) et plus particulièrement de simples envois/réceptions de messages ou notifications qu'une certaine action a eu lieu. Ce sont ces messages que nous appelons "évènement". Ajoutons à cela le rôle particulier du temps. Dans les langages synchrones, le temps est un paramètre discret, divisé en périodes plus ou moins grandes (seconde, nanoseconde, etc). Nous considérons l'écoulement du temps comme un évènement produit à des instants particuliers. C'est d'ailleurs souvent le choix de modélisation du temps dans les langages de description.

Les composants, ports et évènements permettent de représenter les *systèmes* ; la figure 1.2 en est un exemple.

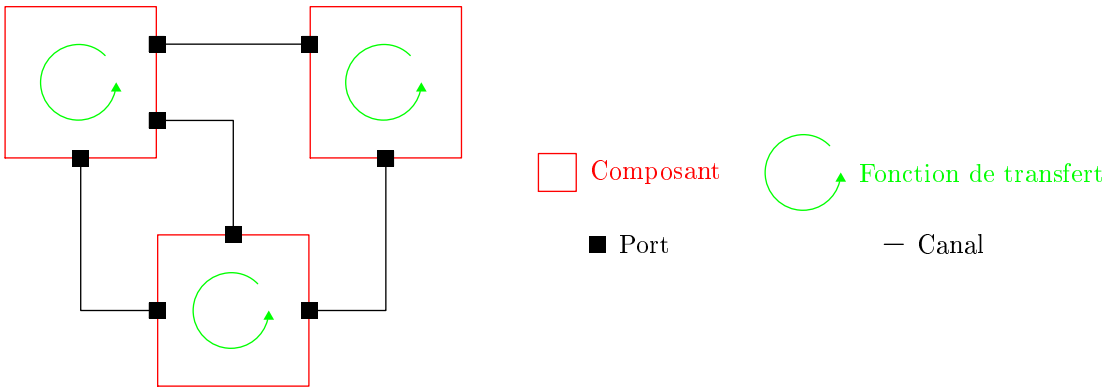

Fig. 1.2 – Exemple de système

Nous détaillons un peu plus la notion de système en donnant un exemple concret qui servira à illustrer les principes développés dans nos travaux.

1.1.2 L'exemple des Pompes

Tout d'abord, on considère un système où un **réservoir** est rempli d'un certain liquide par un composant externe. Le réservoir a une contenance maximale ainsi que deux *capteurs*. L'un réagit lorsque le niveau de liquide dans le réservoir monte au-dessus d'un certain niveau, et l'autre réagit lorsque le liquide descend en-dessous d'un certain autre niveau. Lorsque les capteurs sont activés, ils envoient un signal à un **contrôleur**. Lorsque celui-ci reçoit le signal de dépassement de niveau, il met en marche une **pompe** connectée au réservoir en lui commandant de le vider à un certain *débit*, le déversant dans un autre réservoir externe au système. Lorsque le contrôleur reçoit le signal de passage en-dessous de niveau, il éteint la pompe, ce qui interrompt son action sur le réservoir. Ce que l'on souhaite observer, c'est l'évolution du niveau de liquide dans le réservoir, en fonction du flux d'entrée. La figure 1.3 représente schématiquement le système.

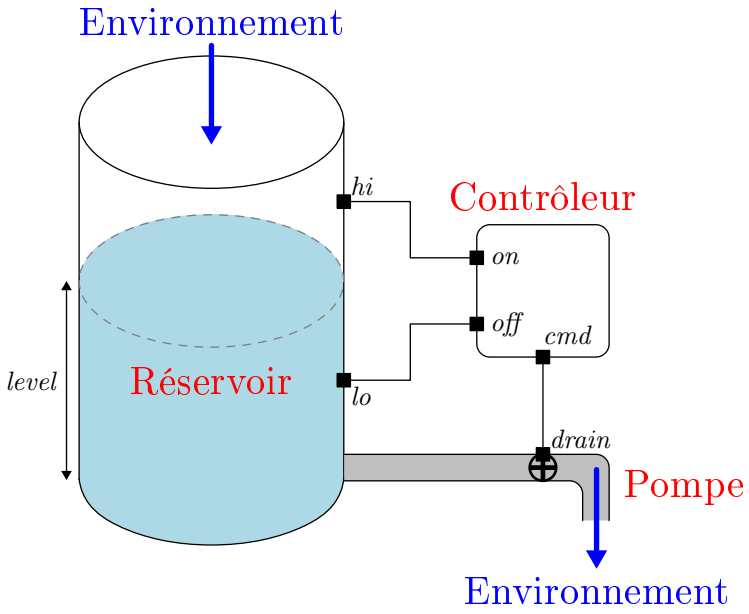

Fig. 1.3 – Schéma d'une pompe

Pour illustrer les travaux de la thèse, nous nous intéresserons également à un système plus complexe, où deux pompes sont mises en séquence. Dans ce système, une première pompe draine du liquide d'un réservoir pour le déverser dans un autre, et une seconde pompe draine le liquide de l'autre réservoir pour le déverser ailleurs. L'intérêt est alors d'étudier l'évolution des niveaux de liquide dans les deux réservoirs au cours du temps, pour s'assurer en particulier que les pompes ne drainent pas à vide et que le niveau dans les réservoirs ne dépassent pas leur contenance maximale respective. La figure 1.4 schématise ce que nous appelons *le système des pompes*.

Plusieurs paramètres influent directement sur le comportement global du système. Tout d'abord, le flux d'entrée du système, qui n'est pas spécifié pour l'instant. L'extérieur peut donc déverser du liquide à n'importe quel débit. Analyser le système va donc consister à déterminer un débit autorisé. Ensuite, le système est également très sensible au débit de cha-

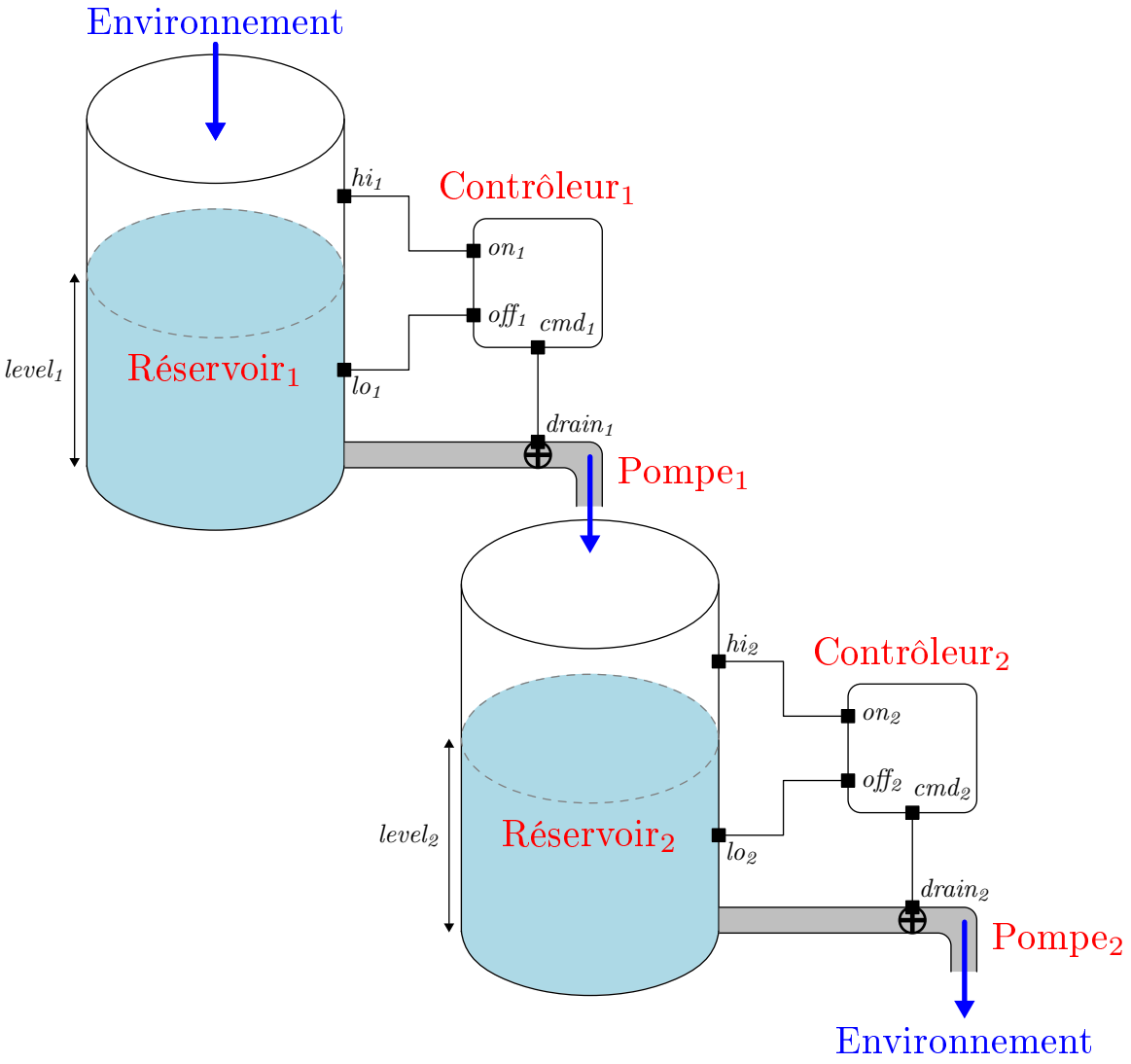

FIG. 1.4 – Le système des pompes

cune des pompes. Si le débit de la première pompe est trop puissant par rapport à celui de la seconde, un réservoir peut déborder. La figure 1.5 montre l'évolution des niveaux de chaque réservoir dans deux cas particuliers. Dans le premier cas, le flux externe est constant et de 5 volumes de liquide par unité de temps, soit 5 u (unité de débit). La vitesse de vidange de la première pompe est de 10 u, et celui de la deuxième de 1 u. Dans le second cas, le flux externe est de 5 u, et les vidanges des pompes est de 10 u et 15 u respectivement pour la première et la seconde pompe.

Comme nous l'avons dit, nous souhaitons analyser des descriptions du niveau du langage SYSTEMC. Nous présentons donc le langage et nous y exprimons le système des pompes afin de comprendre les difficultés que posent la validation des programmes SYSTEMC.

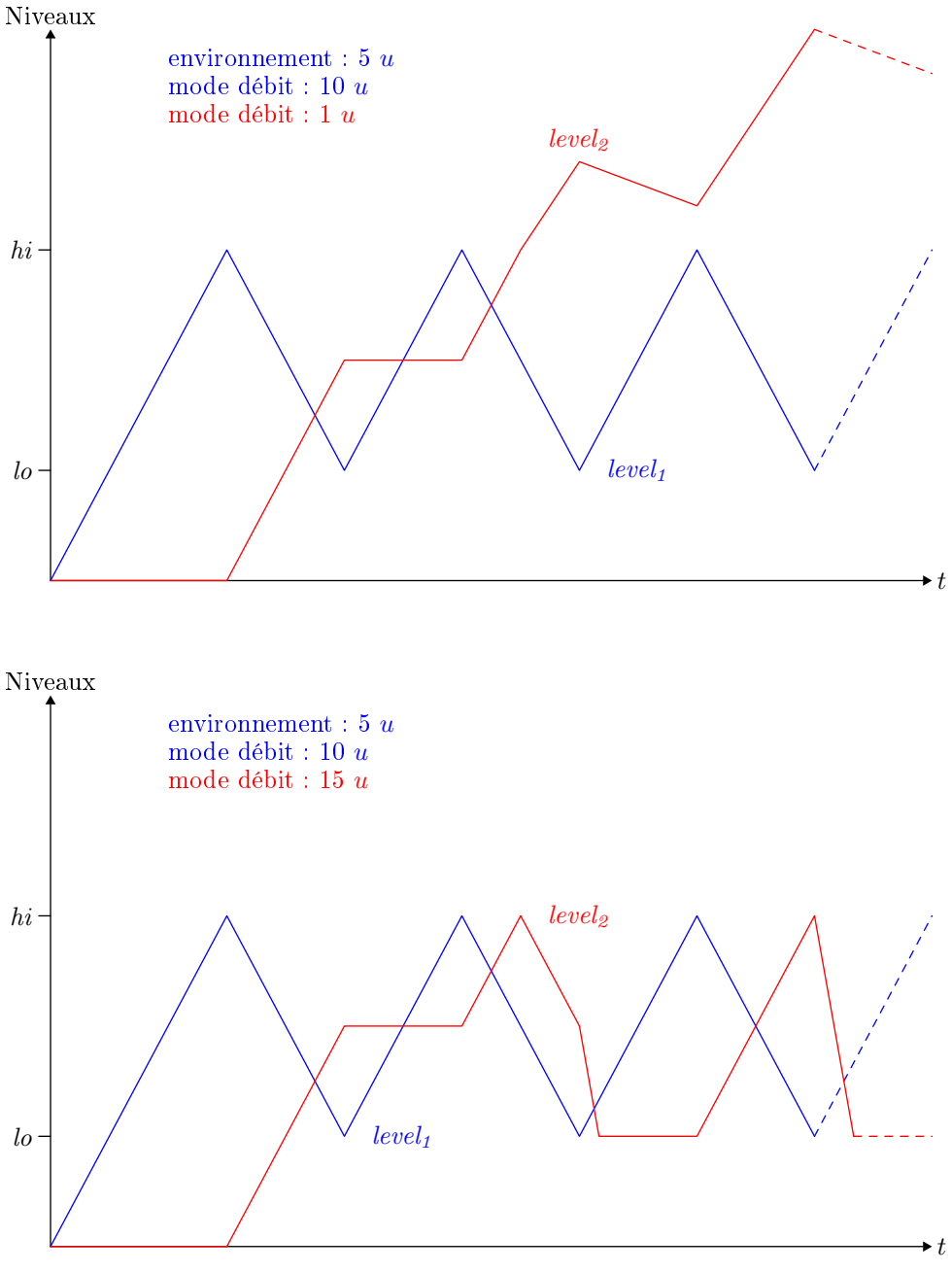

FIG. 1.5 – Évolutions possibles des niveaux de liquide

1.1.3 SystemC

Nous ne présentons que quelques aspects du langage. Pour une présentation complète, le lecteur pourra s'intéresser au manuel de référence du langage [16605].

SystemC est une bibliothèque C++ qui définit des primitives haut niveau pour modéliser et simuler des systèmes de composants. Un développement SystemC se déroule en trois étapes :

- la description de chaque composant ;
- l'élaboration, c'est-à-dire connecter les composants les uns aux autres et initialiser l'environnement de simulation ;
- la simulation, c'est-à-dire exécuter le programme créé aux étapes précédentes afin d'observer le comportement du système.

La partie description se fait en C++, dans un style de programmation impérative et objet, à ceci près que la bibliothèque SystemC définit des mots-clefs propres aux systèmes de composants. La figure 1.6 donne un code SystemC pour représenter le composant contrôleur.

```
SC_MODULE( controller ) {

  sc_in<bool> on;
  sc_in<bool> off;
  sc_out<int> cmd;
  int value;

  SC_CTOR( controller ) {
    SC_METHOD(adjust);
    sensitive << on << off;
  }

  void adjust() {
    if (off) cmd = 0;
    else if (on) cmd = value;
  }

}
```

Fig. 1.6 – Code SystemC du composant contrôleur

Les mots-clefs font partie du vocabulaire des langages de descriptions. Ainsi, un `SC_MODULE` désigne un composant ou un système, `sc_in` est un port d'entrée, etc. On peut alors connecter les composants de base pour former un bout du système, et instancier puis connecter deux de ces bouts pour former le système des pompes. Un exemple de code de connexion est décrit en figure 1.7. On y remarque l'élaboration du système qui se fait dans le constructeur des modules, qui connectent les composants, qui partage certains de leurs segments mémoire, et qui initialise l'environnement d'exécution.

Les étapes de description des composants et d'élaboration sont communes à tous les HDLs, seuls les langages et les niveaux de description changent. Cependant, le langage et le niveau de description jouent un rôle prépondérant dans l'analyse des descriptions. En plus des aspects système du langage, analyser du code SystemC signifie également analyser du code C++, ce qui est déjà un travail extrêmement complexe en soi (modularité, objets, hiérarchie, pointeurs etc). En parlant des aspects système, ceux-ci sont décrits en C++ et ont une signification particulière que comprend le moteur de simulation, lui aussi écrit en C++. Pour comprendre les parties système du langage, il faut donc comprendre comment se conduit une simulation.

```
SC_MODULE( system ) {

  tank tk(``Tank'');
  pump filler(``Filler'');
  pump drainer(``Flusher'');
  controller ctrl(``Controller'');
  sc_signal<int> sign1, sign2, sign3;

  SC_CTOR( system ) {
    filler.tk = &tk;
    drainer.tk = &tk;
    ctrl.cmd(sign1); drainer.drain(sign1);
    ctrl.on(sign2); tank.high(sign2);
    ctrl.off(sign3); tank.low(sign3);
    filler.faucet = 1;
    ctrl.value = -5;
  }

}
```

FIG. 1.7 – Exemple de code de connexion en SYSTEMC

Moteur de Simulation

Le moteur de simulation SYSTEMC permet de visualiser de façon rapide et peu coûteuse un comportement possible qu'aurait un système une fois implanté physiquement. Les composants d'un système sont exécutés en parallèle et en concurrence sur les segments mémoires qu'ils partagent. Le moteur de simulation définit l'ordonnancement et gère la concurrence en adoptant une politique précise. Cependant, avant de rentrer dans les détails du moteur, insistons sur sa nature logicielle afin de mieux comprendre son action.

SYSTEMC est une bibliothèque C++. Dans cette bibliothèque, des primitives de manipulation de systèmes sont définies de façon haut niveau. De ce fait, l'exécution d'un programme SYSTEMC (et donc la simulation d'un système de composants) repose sur deux éléments :

La bibliothèque SYSTEMC. Elle définit les primitives des aspects systèmes du langage (composant, port, canal de communication, etc) en terme d'expressions et d'instructions C++. Elle implante le moteur de simulation qui est donc lui aussi défini par un programme C++.

Le compilateur C++. C'est lui qui, à partir d'un projet SYSTEMC, va créer l'exécutable qui représentera la simulation d'un système, en s'appuyant sur la bibliothèque SYSTEMC et sa connaissance (traduction dans un langage plus bas niveau) du langage C++.

Ainsi, on entend par moteur de simulation l'action conjointe de ses deux éléments.

Pendant une simulation, les composants exécutent des processus. Des points de synchronisation sont définis dans les processus par des appels explicites à la procédure `wait` qui endort le processus qui l'invoque jusqu'à ce que certains évènements (passés en argument de la procédure) soient notifiés. Lorsque tous les évènements qu'attend un processus ont été notifiés et

qu'il est prêt à être exécuté, on dit qu'il est *éligible*. La simulation se déroule comme suit : l'ordonnanceur choisit un processus éligible à qui il donne la main. Ce processus s'exécute de façon non-préemptive en émettant potentiellement des évènements. Son exécution s'interrompt lorsqu'il atteint un point de synchronisation, ce qui l'endort. Nous rappelons que cette exécution est l'exécution d'un code C++. L'ordonnanceur reprend la main et la donne à un autre processus éligible s'il y en a. Cette phase, appelée phase d'évaluation, se répète tant qu'il y a des processus éligibles. Lorsque tous les processus sont endormis, la simulation passe en phase de mise à jour. Les valeurs sur les ports sont stabilisées et les évènements notifiés pendant la phase d'évaluation réveillent les processus qui en sont en attente, et la simulation reprend en phase d'évaluation. C'est l'hypothèse synchrone qui autorise à simuler l'exécution de plusieurs processus en parallèle comme une succession d'exécution. Ceci étant dit, l'aspect synchrone du système s'écroule si plusieurs processus ont des actions différentes sur une même variable au cours d'une phase d'évaluation (par exemple, lorsque deux processus écrivent des valeurs différentes sur un même port). Notons qu'il n'existe pas d'outil statique dans le langage SystemC pour garantir qu'un tel comportement n'arrive pas. D'ailleurs, de manière plus générale, puisque SystemC est avant tout une bibliothèque C++, il est possible d'utiliser des instructions fondamentalement asynchrones volontairement. C'est le cas par exemple du niveau d'un réservoir dans le système des pompes qui nous avons choisi de représenter comme un segment de mémoire partagée. Les processus y lisent et écrivent sans contrainte, modélisant le fait qu'un réservoir peut se remplir, se vider, ou les deux en même temps. Si un tel comportement peut être décrit de façon synchrone, la souplesse et le haut niveau de description de SystemC permet également de le représenter de façon asynchrone, ce qui complexifie nettement le langage d'un point de vue vérification.

1.2 Techniques de Validation

On entend par "techniques de validation" des outils qui permettent de vérifier, dans une certaine mesure, que le code satisfait une spécification. Le but est d'augmenter la confiance que l'on a sur le système développé, ce qui se traduit en pratique par différentes approches.

1.2.1 Test

Le test dynamique consiste à soumettre à un programme un ensemble d'entrées et à exécuter le programme pour vérifier qu'il se comporte sur ces entrées comme le décrit sa spécification [EI90]. Le test s'applique donc à la fin du développement d'une partie du système, pour s'assurer que cette partie est correcte. Il s'adapte bien à la validation de descriptions de systèmes car chaque composant peut être testé séparément (test unitaire), et on peut aussi tester un assemblage de composants validés (test d'intégration) [Mye79, Bei90]. Il s'intègre donc bien dans une logique de validation modulaire, ce qui en fait la technique la plus utilisée dans l'industrie pour la validation de programmes. On distingue deux types de test : le test fonctionnel (ou boîte noire) et le test structurel (ou boîte de verre). Le test fonctionnel valide un programme en le soumettant à des entrées sur lesquelles on vérifie que le programme satisfait sa spécification. Plus on teste de valeurs différentes, plus grande est la confiance accordée au programme (exemple : l'outil GATeL [MA00]). Le test structurel analyse le

code du programme, afin d'établir des jeux de tests adaptés et de couvrir un maximum de cas possibles en un minimum de tests (exemple : l'outil PATHCRAWLER [NWR05]). Citons : « Program Testing can be used to prove the presence of bugs, but never their absence » (Dijkstra, 1974), c'est-à-dire que le test peut trouver des erreurs, mais ne peut pas vérifier qu'il n'y en a aucune. Aujourd'hui, des normes de sécurité (dans le sens "sûreté") de systèmes à base de composants électroniques demandent des critères de confiance supérieur au test [RF].

1.2.2 UML et Scénario

Afin de faciliter le développement d'un système et d'organiser toutes les phases de son cycle de vie, des langages de modélisation sont apparus à la fin du siècle dernier. Certains d'entre-eux ont fusionné pour donner naissance à l'*Unified Modelling Langage* (UML) [BJR01]. UML s'abstrait du langage de programmation utilisé pour représenter un système et s'attache à définir les liens entre les différentes partie du système, mais aussi entre les différentes parties de son développement. Pour cela, le langage — très graphique — définit plusieurs diagrammes standards. D'un point de vue programmation, on trouve par exemple les *diagrammes de classe* pour modéliser une classe d'un langage orienté objet. Si les aspects les plus implantatoires comme le code d'une méthode de classe ne se décrivent pas en UML, il est tout de même possible de spécifier les cas d'utilisation (ou *use cases*) d'un système. Des diagrammes UML leur sont également dédiés, et permettent de représenter l'envoi et la réception de messages dans le temps entre différents acteurs. Les diagrammes des use cases ont évolué en expressivité jusqu'à devenir de véritables *scénarios d'exécution* (envois gardés, comportements obligatoires ou potentiels, etc) décrits par des *Live Sequence Charts* (LSC) [BDK+04]. Ces scénarios peuvent être traduits automatiquement en du code exécutable implantant rigoureusement le scénario [HM03]. Ici, on obtient donc un code qui est correct par construction.

1.2.3 Model Checking

Le model checking [EMC81, QS82] d'un système est une technique qui repose sur trois notions : un modèle du système étudié, une spécification sous la forme d'une propriété (éventuellement temporelle) du système, et des algorithmes qui permettent de vérifier que le modèle satisfait sa spécification. Le modèle est donné par une *structure de Kripke* [BCG87] qui décrit quels sont les états possibles du système, ses évolutions (transitions entre états) et les propriétés atomiques que vérifient chaque état. Les propriétés plus complexes sont données dans des logiques temporelles [Pri67, Pnu77a] qui permettent d'exprimer des relations de causalité entre les états. Le model checking couvre l'ensemble de tous les états du système, ce qui en fait une technique de vérification formelle : toutes les exécutions du système sont considérées. De plus, mise à part la modélisation du système et l'expression de sa spécification, le model checking est une technique entièrement automatique ce qui la rend très utilisée en industrie aujourd'hui [HJM+02, BCLR04]. Cependant, assembler des systèmes multiplie leurs états. Un phénomène d'*explosion combinatoire* apparaît alors sur les grands systèmes formés de nombreux sous-systèmes. Le model checking de tels systèmes peut alors souffrir d'une (trop) grande complexité en temps ou en espace. Ces dernières années, les principaux travaux en model checking tentent

de réduire ce problème en proposant par exemple des méthodes symboliques [McM93] avec des représentations plus compactes pour les états et les propositions atomiques (Binary Decision Diagram [Bry92]). De nombreux outils utilisés dans l'industrie implantent cette technologie et ses dérivés (exemples : SMV [SMV], SPIN [SPI]).

1.2.4 Interprétation Abstraite

L'interprétation abstraite [CC77, CC92] est un cadre théorique qui fournit des définitions et des critères pour simplifier ou *abstraire* un objet tout en assurant que l'abstraction est correcte vis-à-vis d'une classe de propriété : une propriété de cette classe non satisfaite par l'abstraction n'est pas satisfaite par l'objet initial. Elle s'appuie sur les théories du point fixe et des domaines pour introduire des approximations qui permettent de réduire à un temps fini des calculs qui pouvaient être potentiellement infiniment longs. En informatique, cela se traduit par des outils qui calculent en temps fini un sur-ensemble des comportements d'un programme, alors que le problème d'un calcul exact des comportements est indécidable [Ric53]. La vérification d'une propriété à partir de l'abstraction des comportements d'un programme peut donner trois verdicts : *oui* la propriété est vérifiée, *non* elle ne l'est pas, ou *peut-être* mais l'abstraction ne permet pas de décider. Dans ce dernier cas, on parle de *fausse alarme* lorsque la propriété est effectivement vérifiée par le programme mais que l'abstraction ne parvient pas à le montrer du fait des approximations. L'enjeu d'un interpréteur abstrait consiste à trouver le juste équilibre entre précision et complexité des calculs d'un côté, et approximation et rapidité des calculs de l'autre. De nombreux domaines plus ou moins complexes ont été élaborés à la recherche de bons compromis [CC92, Min06, CMC08]. L'interprétation abstraite est aujourd'hui utilisée avec succès dans l'industrie avec des outils comme Astrée [BCC+03] capables de vérifier des milliers de lignes de code embarqué critique, mais en se focalisant uniquement sur certains types de propriétés.

1.2.5 Preuve Interactive

La preuve interactive a donné naissance à des outils comme Coq [Coq, BC04], HOL [HOL, AJLW92] ou Isabelle [Isa, NPW02] qui implantent des logiques très expressives. Dans ces outils, l'utilisateur communique avec l'environnement pour construire la preuve de propositions qu'il souhaite vérifier. Plusieurs travaux ont été menés afin d'utiliser la puissance expressive des logiques sous-jacentes pour la preuve de programmes [FM04, MPMU04]. Pour cela, on annote le code source des programmes en y insérant des hypothèses et des conclusions dans une logique à la Hoare [Hoa83] ; un calcul de plus faible pré-condition [Dij75] établit alors quelles propositions doivent être vérifiées afin de prouver la correction des programmes. On appelle ces propositions des *obligations de preuve*. La preuve interactive permet de prouver une large classe de propriétés sur les programmes mais requiert une main experte pour annoter et certifier des programmes pour l'instant encore relativement petits comparés à la taille de certains codes embarqués. La faute également à un manque d'automatisation même si des travaux mettent en place la coopérations de la preuve interactive avec d'autres techniques de vérification (exemple : Frama-C [Fra]).

1.3 Travaux Connexes

Le langage SystemC est devenu un standard de fait pour la description de systèmes grâce à TLM, son haut niveau de description, qui est adapté à la complexité des systèmes d'aujourd'hui et permet d'avoir une description exécutable très tôt dans leur processus de développement. Cependant un tel niveau d'abstraction nécessite l'adaptation des techniques de validation qui étaient appliquées à plus bas niveau (RTL), comme le constate Moshe Vardi [Var07]. Dans son article, Vardi présente les nombreuses méthodes existantes pour traiter RTL, déjà complexe, ainsi que quelques pistes pour la vérification formelle de descriptions TLM en précisant que cette communauté est encore jeune mais croissante. À ce propos, Vardi insiste sur le besoin de méthodes formelles sur TLM du fait de la criticité des systèmes décrits. Nous nous intéressons tout particulièrement au problème de la vérification formelle de description haut niveau, tout en gardant comme objectifs une bonne intégration au flot de développement et l'accessibilité des techniques pour l'ingénieur.

Propriété par Assertion. Pour ce qui est de l'accessibilité des techniques de vérification pour l'ingénieur, de nombreux efforts ont été fournis depuis plusieurs années dans ce sens, en particulier pour l'expression des propriétés des systèmes. Très rapidement, c'est l'approche par *assertions* [FLK03] qui s'est imposée. Les assertions sont des expressions dans le code des systèmes — d'où leur popularité — qui provoquent une erreur à l'exécution lorsqu'elles sont violées. Cette approche est par exemple directement utilisée dans le langage SystemVerilog [Sysa] (c'est une extension de Verilog), dont le langage de propriétés prend alors le nom de SystemVerilog Assertions. En SystemC/TLM, l'idée consiste souvent à user de *moniteurs* qui vont observer la correction d'exécutions par des assertions, ce qui permet un déploiement rapide pour vérifier plusieurs types de propriétés, avec des références au temps par exemple, pendant la simulation [PF08]. Une partie des travaux sur l'approche par assertions a également donné naissance au *Property Specification Language* (PSL) [Acc04], un langage de spécification dont les instructions sont proches des langages de description. Une spécification PSL peut alors être utilisée pour construire automatiquement des moniteurs dans le langage de description souhaité [OMAB08, GTMH07], bien que cette technique soit également utilisée à partir de langages de spécification différents [ABG+00]. D'autres travaux permettent même à partir d'une spécification PSL de construire automatiquement un système qui valide la spécification par construction [BENS07]. Cependant, la plupart des travaux que nous avons étudiés concerne des langages plus bas niveau que SystemC/TLM, comme Verilog ou VHDL, et les moniteurs sur des programmes SystemC/TLM sont principalement utilisés pour le test. Nous pouvons tout de même nous appuyer sur tous ces travaux pour établir un langage de propriétés intégré au langage de description, et dans lequel les propriétés s'expriment assez naturellement pour l'ingénieur.

Sémantique SystemC. Pour passer à la vérification formelle, on doit disposer d'une sémantique précise du langage. Ainsi, SystemC est souvent critiqué pour son absence de sémantique formelle. Ce qui explique sans doute un certain retard dans l'application des méthodes formelles à ce langage. Plusieurs projets s'y sont attachés et la communauté dispose aujourd'hui de différentes sémantiques.

Remarquons avant de voir plus en détails ces sémantiques qu'utiliser une sémantique de C++ pour traiter formellement un développement en SystemC ne paraît pas être une solution efficace. En effet, l'encodage des composants, de leurs communications et du moteur de simulation aura tôt fait de perturber les outils de vérification du fait de leur complexité. La vérification d'un développement SystemC gagnera à considérer certains aspects propres aux systèmes — nous expliciterons lesquels — de façon primitive, plutôt que de se focaliser sur leur implantation en C++.

Ashraf Salem donne une sémantique dénotationnelle d'un sous-ensemble synchrone de SystemC [Sal03]. Cette sémantique aide à comprendre la façon dont SystemC s'appuie sur le paradigme synchrone, mais SystemC comprend également une grande part de constructions asynchrones (à cause en particulier de la non-préemption et de la possibilité d'écrire sur des segments de mémoire partagée) qu'il nous faut traiter pour une vérification formelle. De plus, une sémantique dénotationnelle aide à déterminer les propriétés d'un langage, alors qu'une sémantique opérationnelle est plus adaptée à la compréhension et à la vérification des programmes du langage. À propos d'une approche synchrone contre une approche asynchrone, nous choisissons de nous intéresser à la seconde, bien que la première associée à des mécanismes de verrous logiciels apporte plus de confort lors de la description de systèmes [HP08]. Cependant, notre but est de traiter le plus grand ensemble possible de descriptions et de coller autant que possible aux exécutions produites par le moteur de simulation. Pour cela, nous avons besoin de distinguer les aspects synchrones et asynchrones d'une part, et matériels et logiciels d'autre part [MMC+08], et de les intégrer en une seule et même sémantique. À notre connaissance, il y a quatre travaux aboutis concernant la sémantique de SystemC/TLM, qui consiste tous en sa traduction vers un formalisme intermédiaire qu'il est possible de manipuler afin de créer des instances compréhensibles par de nombreux outils de vérification formelle externes. Le premier [GHT04] consiste à traduire le langage vers des *Abstract State Machines* [GKOT00], un formalisme très souple pour la description de systèmes. Le second concerne l'outil LusSy [MMMC05] dont le format interne est une forme de machines communicantes (de façon synchrone ou asynchrone) qui offre d'ores et déjà des connexions vers Lesar [NHR92] (un model checker symbolique) ou encore NBac [Jea] (un interpréteur abstrait adapté à l'analyse de systèmes) entre autres. Dans ce même style, notre troisième référence est l'outil PRES+ [KEP06] qui traite SystemC/TLM en le traduisant dans une forme augmentée de réseau de Petri qui est utilisée pour du model checking avec UPPAAL [UPP]. Enfin, *SystemC*$^{\mathbb{FL}}$ [MFMS06] est un formalisme mathématique qui donne une sémantique opérationnelle à SystemC/TLM. Ses auteurs montrent comment l'utiliser comme formalisme intermédiaire pour appliquer une vérification du système par model checking, typiquement SPIN.

Ces sémantiques nous donnent une bonne compréhension du fonctionnement de TLM, et donc des systèmes décrits à ce niveau. Les concepteurs de LusSy ont même montré que l'on peut grâce à leur outil utiliser l'approche par assertions et l'interprétation abstraite (avec NBac dans leur cas) pour prouver formellement des propriétés de sûreté (un mauvais état n'est pas atteignable) sur les systèmes. En effet, l'interprétation abstraite introduit des approximations correctes, et permet donc de calculer un sur-ensemble des états atteignables par un système. Ce travail est donc plus restrictif qu'une approche réalisée sur RTL par model checking [DG02] où l'on calcule exactement l'ensemble des états atteignables par un système fini (comme ceux considérés par RTL), ce qui rend possible la vérification de propriétés de type sûreté aussi bien que la vérification de propriétés de type vivacité (un bon état est atteignable) [AS86, Sch97].

Compositionnalité. Malgré l'efficacité de tous ces travaux, un problème subsiste : l'explosion combinatoire en nombre d'états d'un système. Par exemple, l'outil PRES+ [KEP06] met moins d'une seconde à vérifier une propriété sur un modèle Maître/Esclave de l'implantation de référence de SystemC formé d'un maître et d'un esclave, alors que la vérification de la même propriété prend plus de quatre heures lorsqu'on considère le même type de système mais formé de deux maîtres et de deux esclaves. Vu le nombre d'états dans ce système cela est plutôt performant, mais cela souligne également une explosion selon le nombre de composants qui forment le système. Pour remédier à ce problème, une approche consiste à diviser le système en sous-parties et à analyser chacune de ces sous-parties, plus simples, indépendamment. On peut ajouter à cela l'abstraction de certaines parties afin de se débarrasser de détails inutiles pour la vérification de propriétés globales au système (cette abstraction peut même se décrire par des formules de logiques temporelles [BE07]). Cette technique a été utilisée par Zohar Manna et al. [FMS98] pour développer leur outil STeP [MCF+98]. Comme les sous-parties ne sont généralement pas indépendantes, il faut pouvoir supposer l'action des autres sous-parties lorsqu'on en analyse une particulière. Il est courant d'utiliser pour cela des *assume-guarantee* [HQR98] qui donnent un cadre général correct pour poser des hypothèses sur l'environnement d'exécution d'un composant qui seront déchargées lors de l'assemblage avec le reste du système. Des travaux sur la préservation de propriétés par abstractions de systèmes [LGS+95] ciblent les conditions de correction de l'application de ces abstractions. Alors que les abstractions introduisent des approximations, il est tout de même possible de regagner en précision en considérant des exécutions de l'abstraction qui ne sont pas des exécutions du système de départ [CGJ+03].

Interprétation Abstraite de Systèmes. En plus de la question du langage de propriétés des systèmes, nous devons également étudier la façon de vérifier qu'un système satisfait une propriété. Clairement, pour obtenir un environnement confortable pour l'ingénieur, cette vérification doit être automatique. L'interprétation abstraite a particulièrement retenu notre attention pour cela. Tout d'abord, cette technique a déjà été utilisée avec succès pour la vérification de systèmes. La preuve en est l'interpréteur abstrait de systèmes NBac ou encore la vérification par interprétation abstraite de descriptions VHDL [Hym03]. Ensuite, l'interprétation abstraite est particulièrement adaptée à l'étude de systèmes comprenant un grand nombre d'états, potentiellement infini même, comme cela peut être le cas des descriptions TLM. Des études ont d'ailleurs été menées afin de combiner le model checking et l'interprétation abstraite [CC99], et nous pourrons nous appuyer sur leurs résultats. Enfin, la compositionnalité comme nous l'avons présentée s'exprime très bien dans le cadre de l'interprétation abstraite, si bien que nous pourrons utiliser un cadre commun pour la vérification et la compositionnalité.

Nous souhaitons permettre à l'ingénieur de décrire, de spécifier et de vérifier une description TLM dans un seul et même langage dont les constructions pour la spécification restent proches de celles pour la description. Pour contourner le problème de l'explosion combinatoire, nous souhaitons également que le langage permette de remplacer un système (ou une partie du système) par un objet potentiellement plus simple d'un point de vue combinatoire. Pour cela, les travaux du type de LusSy ne conviennent pas tout à fait puisqu'ils traitent SystemC en le traduisant vers un formalisme intermédiaire. L'intégration de la modularité au niveau du langage de description nécessiterait donc d'augmenter les constructions du langage SystemC

et d'adapter la chaîne d'outils LusSy en conséquence. D'autre part, les techniques du type de STeP nécessitent des spécifications qui s'expriment en logique temporelle, un formalisme pas toujours maîtrisé par les ingénieurs. Finalement, les travaux de Nicolas Halbwachs et al. [HLR93] sont les plus proches de notre objectif puisqu'ils étudient la spécification de systèmes par observateurs (même formalisme que pour la description des systèmes), caractérisent le type de propriétés ainsi exprimables (propriétés de sûreté), montrent qu'il est possible d'incorporer des hypothèses sur l'environnement d'exécution, et donnent un cadre pour la vérification modulaire. Le seul inconvénient par rapport à notre ambition est l'utilisation de ces techniques dans un contexte purement synchrone et déterministe, ce qui n'est pas le cas de SystemC/TLM. La différence est telle que nous n'allons pas définir la modularité par le remplacement d'un système par sa spécification, mais plutôt par le remplacement d'un système par un autre.

1.4 Contributions

Nous voulons donc établir un langage qui permette de décrire des systèmes TLM. Pour sa sémantique, nous nous inspirerons des sémantiques de SystemC/TLM que nous avons citées. Les propriétés des systèmes devront pouvoir se faire directement dans ce langage, en ajoutant un minimum de constructions en plus de celles propres aux descriptions. Ces constructions doivent rester confortables pour l'ingénieur si bien que l'approche par assertions semble tout à fait adaptée à cette fin. Enfin, ce langage doit également permettre l'utilisation d'abstractions pour la modularité des analyses.

Ce langage que nous nommons SystemD est au centre de nos contributions. En plus de la possibilité de décrire des systèmes en TLM (avec ou sans référence au temps), il permet l'expression de systèmes qui contiennent des assertions pour représenter des propriétés. Le langage permet également de construire des abstractions afin de remplacer des composants par d'autres pour contourner le problème de l'explosion combinatoire. Ces abstractions jouent le rôle d'assume-guarantee par l'utilisation naturelle de la conditionnelle pour représenter l'implication (hypothèses). Elles permettent d'abstraire mais aussi de raffiner des systèmes, et s'inscrivent donc confortablement dans le cycle de développement d'un système. Nous accompagnons SystemD d'un moteur de simulation et d'un moteur d'analyse. Ce dernier repose sur l'utilisation conjointe de techniques empruntées au model checking pour conserver une précision exacte sur les configurations des processus du système, et de techniques empruntées à l'interprétation abstraite pour réduire le nombre de valeurs pour les variables, comme conseillé par Zohar Manna et al. [MCF+98] et déjà utilisé par NBac. Les analyses permettent de couvrir automatiquement et en temps fini (et relativement court) un sur-ensemble assez précis des exécutions d'un système avec des domaines abstraits simples (intervalles). Elles permettent également de s'assurer que le remplacement d'un composant par un autre est correct. Enfin, nous montrons un intérêt tout nouveau des observateurs dans ce contexte. En effet, alors que l'interprétation abstraite fusionne certains états, les observateurs apportent de la précision en les distinguant. Les analyses sont donc pilotées par des observateurs qui sont définis en SystemD, par l'utilisateur. Nous distinguerons donc deux types d'observateurs : ceux qui servent à représenter des propriétés et que nous appellerons *systèmes propriété*, et ceux qui servent à apporter de la précision aux analyses et que nous appellerons *systèmes discriminants*.

Pour résumer, SystemD est un langage de description TLM et de spécification dans lequel

les propriétés sont des systèmes (approche par assertions). Nous implantons une analyse des programmes du langage en empruntant des techniques de model checking et d'interprétation abstraite. Lorsque l'interprétation abstraite introduit des approximations trop grossières, SystemD offre la possibilité de définir des systèmes discriminants qui apportent de la précision en ajoutant leurs configurations au reste du système étudié. Enfin, le problème de l'explosion combinatoire est contourné en SystemD par la possibilité de remplacer un système par un autre.

1.5 Plan

La partie I du document donne un cadre théorique à nos travaux. Le chapitre 2 de cette partie présente les définitions formelles des systèmes et leur sémantique *ouverte*. Nous y définissons également un opérateur pour assembler les systèmes et nous montrons sa relation sémantique avec les sous-systèmes. Le chapitre 3 qui suit présente les notions de propriétés des systèmes, de remplacement (modularité) et de systèmes discriminants. Nous montrons comment utiliser les systèmes pour représenter des propriétés des systèmes, comment remplacer un système par un système potentiellement plus simple pour les mécanismes de vérification et comment utiliser des systèmes discriminants pour gagner en précision lors des analyses.

La partie II concerne le langage SystemD et commence par montrer en chapitre 4 l'exemple des pompes dans ce langage afin de donner l'intuition du langage. En particulier, nous y voyons comment s'exprime la notion de propriétés, de remplacement et de discrimination. Ensuite, nous donnons la grammaire du langage, puis une sémantique informelle. Le chapitre 5 définit un langage bas niveau dont les programmes sont des graphes de flot de contrôle : KernelD. Ce langage est un noyau pour SystemD. C'est pourquoi nous en donnons une sémantique formelle, avant de présenter la traduction de SystemD à KernelD qui explicite donc la sémantique du langage SystemD. Enfin, ce chapitre se clôt, et avec lui cette partie, sur une étude des langages SystemD/KernelD pour ce qui est de l'expressivité des propriétés sur les systèmes.

La partie III montre comment réaliser la vérification de programmes KernelD, ce dernier rentrant dans le cadre théorique de la partie I. Tout d'abord, le chapitre 6 explique ce qui nous a amené à utiliser conjointement le model checking et l'interprétation abstraite, et montre de manière générale comment appliquer ces techniques sur un développement KernelD. Nous instancions concrètement une méthode d'analyse avant (avec analyse arrière sur les gardes). Le chapitre 7 qui suit donne des algorithmes de vérification pour la sûreté des systèmes, la garantie de remplacements corrects, et le fait qu'un système discrimine bien un autre. Le chapitre 8 présente les résultats de ces algorithmes sur le système des pompes en SystemD.

Enfin, nous discutons brièvement dans le chapitre IV de travaux perspectifs qui peuvent améliorer la précision des analyses, ou réduire leur complexité en espace ou aider un peu plus l'ingénieur à qui se destine nos outils. Finalement, nous concluons.

CHAPITRE 2

Systèmes

Dans ce chapitre, nous décrivons une sémantique pour des systèmes de composants. Cette sémantique servira de référence pour des analyses statiques que nous mettons en œuvre dans les chapitres suivants. C'est un cadre mathématique à partir duquel nous donnerons une sémantique au langage SystemD.

Nous nous sommes attachés à donner une sémantique *ouverte* aux systèmes de composants, avec comme objectif principal la vérification *compositionnelle* d'un assemblage de différents sous-systèmes. Outre une sémantique opérationnelle tout à fait standard pour les composants élémentaires, nous introduisons une opération d'assemblage de deux systèmes permettant d'exprimer des techniques de vérification modulaires.

2.1 Définitions

Les systèmes de composants sont des transformateurs d'*états*. On distingue dans ces états une partie *mémoire* et une partie *contrôle*, elle-même composée de *processus* qui communiquent via la mémoire et par *évènements*.

2.1.1 Mémoire

La partie mémoire de l'état d'un système est représentée par un *environnement* qui associe des *valeurs* aux *variables*.

Paramètre 2.1 Variable $x \in \mathcal{X}$ Valeur $v \in \mathcal{V}$

Définition 2.2 État mémoire ou Environnement $\rho \in \Gamma \triangleq \mathcal{X} \to \mathcal{V}$

L'état mémoire du système est donc représenté par une fonction *totale* sur le domaine des variables, conformément à notre objectif d'une sémantique *ouverte*.

Notation : étant donnés un état mémoire ρ, une variable x et une valeur v, on note $\rho[x \leftarrow v]$ la mise à jour de x dans ρ par v, définie comme suit :

$$\rho[x \leftarrow v] \triangleq \{(y, v') \mid v' = v \text{ si } y = x \text{ et } v' = \rho(y) \text{ sinon}\}$$

$\surd$ *Nous utiliserons dans toute la thèse l'égalité* extensionnelle *pour les fonctions, ainsi on aura pour les états mémoire :*

$$\rho = \rho' \quad \triangleq \quad \forall x,\ \rho(x) = \rho'(x)$$

2.1.2 Processus

La partie contrôle d'un système est représentée par des processus. Un processus transforme la mémoire, et éventuellement émet (ou *notifie*) des évènements. Mathématiquement, un processus est donc simplement représenté par une *fonction de transfert* sur les états.

Les processus communiquent en utilisant des segments de mémoire partagée, mais aussi par l'envoi et la réception de messages. On appelle ces messages des *évènements*. Lorsqu'un système s'exécute, certains processus sont actifs — on dit aussi *éligibles* — alors que d'autres sont endormis, en attente d'évènements. Par leur exécution, les processus actifs déclenchent des évènements qui réveillent les processus endormis qui sont en attente de notification.

Nous souhaitons avoir un cadre adapté à des techniques d'analyses qui distinguent la réactivité des processus aux notifications d'évènements. À cette fin, la fonction de transfert d'un processus s'appuie d'une part sur les états mémoire, mais d'autre part sur la *configuration* du processus. Typiquement, une configuration exprime par exemple qu'un processus p attend un évènement e pour exécuter une transformation A, ou bien un évènement e' pour effectuer une transformation B.

Paramètre 2.3 Évènement $e \in \mathcal{E}$ Configuration de processus $c \in \mathcal{C}$

Définition 2.4 Fonction de transfert $\phi \in \Phi \triangleq \mathcal{P}((\mathcal{C} \times \Gamma) \times (\mathcal{C} \times \mathcal{P}(\mathcal{E}) \times \Gamma))$

Une fonction de transfert est donc une relation qui, étant donnés une configuration d'un processus et un état mémoire courant, indique la configuration que va atteindre le processus après exécution, les changements sur l'état mémoire que cela produit, et les évènements qui ont ainsi été notifiés. Les fonctions de transfert sont généralement définies par des séquences d'instructions, ces dernières ayant une incidence directe sur l'expressivité des processus, ainsi que sur la clarté des descriptions. Nous verrons un jeu possible d'instructions dans une partie ultérieure, qui présentera le langage SystemD : c'est un exemple d'instanciation représentatif. Cependant, il est possible d'étudier un jeu d'instructions différent, comme celui de SystemC ; nous en faisons abstraction ici car cela ne change pas la méthodologie de vérification.

Notation : étant donnés deux configurations c et c', un ensemble d'évènements E, deux états mémoire ρ et ρ' et une fonction de transfert ϕ, on notera $(c, \rho) \xrightarrow{\phi} (c', E, \rho')$ le fait que $((c, \rho), (c', E, \rho')) \in \phi$. Cette notation vaudra pour toute relation binaire.

Si dans notre sémantique ouverte, les états mémoire sont des fonctions totales sur les variables, chaque processus est en revanche contraint à n'en modifier qu'une partie. De même pour les évènements qu'un processus est susceptible de recevoir ou notifier.

Un processus est donc un quadruplet formé d'un ensemble de variables et d'évènements qu'il peut lire, d'un ensemble de variables et d'évènements qu'il peut écrire, de l'ensemble de ses configurations, ainsi que d'une fonction de transfert.

Définition 2.5 Processus $p = \langle R, W, C, \phi \rangle \in \Psi \triangleq \mathcal{P}(\mathcal{X} \cup \mathcal{E}) \times \mathcal{P}(\mathcal{X} \cup \mathcal{E}) \times \mathcal{C} \times \Phi$

Clairement, les variables et évènements lus et écrits dans la définition d'un processus doivent contraindre sa fonction de transfert et ses configurations. Dans la suite, on supposera toujours que les processus sont *bien formés*. Pour ce qui est des variables, ceci signifie que le domaine de définition de la fonction de transfert du processus peut se restreindre aux variables lues par le processus dans sa définition, et que l'ensemble image de sa fonction de transfert peut se restreindre à ses variables écrites. Pour ce qui est des évènements, ceci signifie que seuls les évènements écrits par le processus dans sa définition peuvent effectivement être notifiés par sa fonction de transfert, et que les configurations du processus ne peuvent être sensibles qu'aux évènements lus.

La nature relationnelle des fonctions de transfert introduit l'indéterminisme inhérent à l'exécution des processus. En effet, nous supposons que deux exécutions d'une fonction de transfert sur les mêmes arguments peut donner lieu à différents résultats.

Abus de langage : étant donné un processus p, il arrivera que l'on note directement R_p et W_p ses variables et évènements lus et écrits, C_p ses configurations et $\xrightarrow{p}$ sa fonction de transfert.

2.1.3 Système

Un *système* est simplement un ensemble fini de processus. Ces derniers peuvent s'exécuter en parallèle et en concurrence sur la mémoire. Cependant, une notion supplémentaire est ajoutée : les variables *locales* et les évènements *locaux*. Ceux-ci vont nous permettre d'exprimer par la suite la notion d'*interface* ainsi que différentes opérations sur les systèmes. Un processus hors d'un système n'a pas le droit de modifier les valeurs des variables et évènements locaux au système ; cependant, il peut lire leur valeur.

Définition 2.6 Système $S = \langle L, P \rangle \in \mathcal{S} \triangleq \mathcal{P}(\mathcal{X} \cup \mathcal{E}) \times \mathcal{P}(\Psi)$

Notation : on utilise L_S et P_S pour représenter respectivement les variables et évènements locaux et les processus d'un système S.

Nous avons choisi de représenter la notion de variables et évènements locaux pour les systèmes plutôt que la notion plus classique d'*entrées/sorties* [CP04]. Ce choix a été motivé par le fait que nous souhaitons donner un aspect compositionnel à nos analyses. Pour cela, nous serons amenés à étudier l'évolution d'une partie d'un système potentiellement ouverte sur un milieu inconnu. Alors, il est plus confortable de se concentrer sur les objets qui ne peuvent pas être modifiés par le milieu extérieur plutôt que sur ceux qui peuvent être modifiés arbitrairement d'un instant à un autre.

Toujours au sujet des variables et évènements, nous prolongeons la définition de variables et évènements lus et écrits par un processus sur les systèmes. Ce sont simplement des unions de ces notions pour chacun des processus du système.

Variables et évènements lus par un système $R \in \mathcal{S} \to \mathcal{P}(\mathcal{X} \cup \mathcal{E})$

$$R\langle L, P\rangle = \bigcup_{p \in P} R_p$$

Variables et évènements écrits par un système $W \in \mathcal{S} \to \mathcal{P}(\mathcal{X} \cup \mathcal{E})$

$$W\langle L, P\rangle = \bigcup_{p \in P} W_p$$

Abus de langage : nous utiliserons les symboles de fonction R et W pour représenter l'ensemble des variables et évènements lus ou écrits par un ensemble de processus. C'est simplement l'union des variables ou évènements lus ou écrits par chaque processus de l'ensemble, comme pour le cas des systèmes (d'ailleurs, la notion de variables et évènements locaux au système n'intervient pas dans la définition de ses variables et évènements lus ou écrits).

2.2 Encapsulation et Assemblage

Parfois, on souhaite modifier les variables et évènements locaux d'un système (souvent, on en ajoute des supplémentaires). Par cette opération, on "cache" des variables et évènements en interdisant l'extérieur d'un système à y accéder en écriture (mais il peut encore accéder à leur lecture). Lorsque l'on ajoute des variables ou des évènements aux locaux d'un système, on parle d'*encapsulation* car le système sera le seul à avoir un accès total en lecture et en écriture sur ces variables et évènements. Notons que contrairement à l'encapsulation plus classique, comme le masquage en π-calcul [Mil99], notre notion d'encapsulation ne réserve pas l'accès en lecture. Encapsuler les variables et évènements X dans le système S consiste simplement à déclarer locales à S les variables et évènements X.

Définition 2.7 Encapsulation $/ \in \mathcal{S} \times \mathcal{P}(\mathcal{X} \cup \mathcal{E}) \to \mathcal{S}$

$$S/X \quad \triangleq \quad \langle X, P_S\rangle$$

On peut également *assembler* deux systèmes pour en créer un nouveau. Cette opération met en parallèle et en concurrence tous les processus des deux systèmes et déclare locaux au système résultant toutes les variables et tous les évènements qui étaient locaux à chaque système de départ. Cette opération correspond à la parallèlisation dans les calculs de processus.

Définition 2.8 Assemblage $\otimes \in \mathcal{S} \times \mathcal{S} \to \mathcal{S}$

$$S \otimes S' \quad \triangleq \quad \langle L_S \cup L_{S'}, P_S \cup P_{S'} \rangle$$

$$\mathrm{dom}(\otimes) \quad = \quad \{(S, S') \mid L_S \cap W(S') = L_{S'} \cap W(S) = \emptyset \text{ et } P_S \cap P_{S'} = \emptyset\}$$

Le domaine de définition de l'opérateur signifie que l'assemblage de deux systèmes n'est bien formé que si les systèmes n'écrivent pas sur les locaux de l'autre et qu'ils n'ont pas de processus commun.

L'opération d'assemblage va de paire avec celle d'encapsulation. En effet, lorsque l'on assemble deux systèmes, il arrive souvent qu'il y ait des variables et des évènements qui font office de canal de communication entre les deux systèmes et ces deux là uniquement. On peut changer ces variables et évènements qui étaient globaux pour les deux en des variables et évènements locaux à leur assemblage par une encapsulation.

Cette définition permet de montrer un lemme très simple qui consiste à dire que les variables et évènements lus (respectivement écrits) par l'assemblage de deux systèmes est l'union des variables et évènements lus (resp. écrits) par chacun d'entre-eux.

Lemme 2.1 $\forall S\ S',\ R(S \otimes S') = R(S) \cup R(S')$ et $W(S \otimes S') = W(S) \cup W(S')$

Preuve : $R(S \otimes S') = R(\langle L_S, P_S \rangle \otimes \langle L_{S'}, P_{S'} \rangle) = R(\langle L_S \cup L_{S'}, P_S \cup P_{S'} \rangle) = \bigcup_{p \in P_S \cup P_{S'}} R_p = (\bigcup_{p \in P_S} R_p) \cup (\bigcup_{p' \in P_{S'}} R_{p'}) = R(S) \cup R(S')$. La preuve est similaire pour les variables et évènements écrits. □

2.3 Sémantique Opérationnelle

Nous adoptons une sémantique opérationnelle de traces à la Kahn [Kah74] pour nos systèmes. La sémantique d'un système S est donc un ensemble de séquences d'*états*, qui représente toutes les exécutions possibles d'un système qui contient S. C'est grâce à cette sémantique ouverte que nous pourrons manier la compositionnalité confortablement.

2.3.1 Définition

Un système est un ensemble de processus. L'évolution d'un système dépend donc de l'évolution des processus qui le composent. Comme l'exécution d'un processus est soumis à sa configuration, nous avons besoin de connaître la configuration de chaque processus pour exécuter un système. De plus, on doit disposer de l'ensemble des évènements notifiés ainsi que d'un état mémoire. On divise ces informations en deux parties. La première est appelée la *configuration du système* et elle regroupe la configuration de chaque processus ainsi que les évènements notifiés.

Définition 2.9 Configuration de système $r = (C, E) \in \mathcal{R} \triangleq (\Psi \to \mathcal{C}) \times \mathcal{P}(\mathcal{E})$

Notation : étant donnée une configuration de système $r = (C, E)$, nous utiliserons une notation pointée pour faire référence à chaque élément du couple r. Ainsi, $r.C$ désignera les configurations de processus C et $r.E$ l'ensemble d'évènements E.

L'*état* d'un système est un couple formé d'une configuration du système et d'un état mémoire. La partie contrôle du système est formée par la configuration de chaque processus et des évènements notifiés, et la partie données par l'état mémoire. On distingue ces deux parties car elles vont être traitées différemment lors des analyses : la partie contrôle sera explorée exhaustivement par des techniques de model checking, alors que la partie donnée sera surapprochée par des techniques d'interprétation abstraite.

Définition 2.10 État de système $\sigma = (r, \rho) \in \Sigma \triangleq \mathcal{R} \times \Gamma$

Notation : comme pour les configurations de système, on notera $\sigma.r$ la composante d'un état de système qui représente sa configuration et $\sigma.\rho$ pour sa mémoire. On s'autorise même à écrire directement $\sigma.C$ et $\sigma.E$ pour $\sigma.r.C$ et $\sigma.r.E$ respectivement.

L'évolution d'un système est une succession d'exécutions de processus. Nous choisissons naturellement la structure de séquences d'états pour représenter l'évolution des systèmes. Nous appellerons *traces d'exécution* ces séquences d'états. La sémantique d'un système sera précisément définie par un ensemble de traces d'exécution possibles du système.

Définition 2.11 Trace d'exécution $(t_i)_{i\in\mathbb{N}} \in \mathcal{T} \triangleq \mathbb{N} \to \Sigma$

Définition 2.12 Sémantique opérationnelle d'un système $[\![.]\!] \in \mathcal{S} \to \mathcal{P}(\mathcal{T})$

Soit S un système. Sa sémantique opérationnelle, notée $[\![S]\!]$, est un ensemble de traces d'exécution défini comme suit :

$$[\![S]\!] \triangleq \{t \mid \forall i,\ t_i \xrightarrow{S} t_{i+1}\}$$

La relation $\xrightarrow{S}$ (définie ci-après) permet donc de représenter totalement la sémantique du système S, quels que soient les systèmes avec lesquels il est assemblé. Elle se décompose en trois types de transitions : l'exécution d'un processus, l'évolution de l'environnement extérieur au système, et le changement de configuration des processus suite aux évènements notifiés. Dans certains langages de description, la simulation d'un système se décompose en une hiérarchie de phases distinctes. Pour organiser ces phases, on fait alors appel à un ordonnanceur de processus. Il servira à ordonner l'exécution des processus. Conformément à l'approche SystemC, nous avons choisi un modèle d'ordonnancement *non-préemptif*.

Nous définissons maintenant séparément la relation de transition $\xrightarrow{S}$ d'un système par ses trois composantes, qui sont : l'exécution d'un processus, les transitions Univers, et les transitions de l'ordonnanceur.

Exécution d'un Processus. Pour un système $S = \langle L, P\rangle$, une transition effectuée par le processus $p \in P$ transforme un état σ du système en un état σ' et notifie un ensemble d'évènements E lorsque que les trois conditions suivantes sont réunies :

– l'exécution de p peut se faire depuis la configuration qui lui est associée dans σ vers celle qui lui est associée dans σ'. La mémoire change selon les modifications apportées par cette exécution :

$$(\sigma.C(p), \sigma.\rho) \xrightarrow{p} (\sigma'.C(p), E, \sigma'.\rho) \quad \text{(L1)}$$

– la configuration des processus autres que p ne change pas :

$$\forall p' \neq p,\ \sigma.C(p') = \sigma'.C(p') \quad \text{(L2)}$$

– les évènements émis par l'exécution de p s'ajoutent à ceux de σ pour donner ceux de σ' :

$$\sigma'.E = \sigma.E \cup E \quad \text{(L3)}$$

Remarque : L'exécution d'un processus ne peut qu'ajouter des évènements à l'ensemble des évènements notifiés. Ceux-ci seront tous consommés lors d'une phase de mises à jour provoquée par l'ordonnanceur.

Transition Univers. La modularité de nos analyses suppose qu'un système peut être défini dans un milieu qui lui est inconnu. Nous adoptons donc une sémantique dans l'*Univers* où un système évolue en parallèle avec d'autres systèmes inconnus. C'est ici qu'interviennent les variables locales au système : l'Univers extérieur ne peut pas en modifier les valeurs.

Pour un système $S = \langle L, P \rangle$, une transition *Univers* d'un état σ vers un état σ' est définie par les conditions suivantes :

– les valeurs associées aux variables locales de S n'ont pas changé :

$$\forall x \in L,\ \sigma.\rho(x) = \sigma'.\rho(x) \quad \text{(U1)}$$

– la nature de notification (présent ou absent) d'un évènement local à S reste la même :

$$\forall e \in L,\ e \in \sigma.E \Leftrightarrow e \in \sigma'.E \quad \text{(U2)}$$

– la configuration des processus locaux à S n'a pas changé :

$$\forall p \in P,\ \sigma.C(p) = \sigma'.C(p) \quad \text{(U3)}$$

Ordonnanceur. Dans notre cadre, l'ordonnanceur n'agit que sur les configurations de système et les processus ne sont pas interrompus entre deux configurations successives. C'est lui qui va réveiller les processus et c'est donc par lui que se définit le fait qu'un processus soit sensible ou non à un évènement.

Paramètre 2.13 Ordonnanceur $\xrightarrow{\text{Sched}} \in \mathcal{P}(\mathcal{R} \times \mathcal{R})$

Nous n'avons pas eu besoin dans ce cadre théorique de modéliser plus précisément l'ordonnanceur. Cependant, dans le cas particulier de SystemD, il aura une définition précise. Pour en donner un aperçu ici, l'ordonnanceur peut par exemple choisir un processus en attente d'un évènement e, et le rendre éligible si $e \in \sigma.E$. Auquel cas, e est retiré du stock d'évènements de la configuration du système.

Notation : on note $\sigma \xrightarrow{\text{Sched}} \sigma'$ le fait que σ et σ' sont en relation par une transition ordonnanceur, c'est-à-dire que $\sigma.\rho = \sigma'.\rho$ et que $\sigma.r \xrightarrow{\text{Sched}} \sigma'.r$.

Transitions du Système. Elle se décompose en l'union de trois cas : une transition Locale ou un processus interne au système S s'exécute, une transition Univers où un processus extérieur au système S s'exécute et la transition ordonnanceur.

Définition 2.14 Transition $\dot{\rightarrow} \in \mathcal{S} \rightarrow \mathcal{P}(\Sigma \times \Sigma)$

Soit $S = \langle L, P \rangle$ un système. On pose

$$\xrightarrow{\text{Local}} = \{(\sigma, \sigma') \mid \exists p \in P\ E,\ \text{(L1) et (L2) et (L3)}\}$$

$$\xrightarrow{\text{Univers}} = \{(\sigma, \sigma') \mid \text{(U1) et (U2) et (U3)}\}$$

Alors,

$$\xrightarrow{S} \triangleq \xrightarrow{\text{Local}} \cup \xrightarrow{\text{Univers}} \cup \xrightarrow{\text{Sched}}$$

La *sémantique opérationnelle* d'un système est le système de transitions associé au système. Ce système de transition est un graphe dont les sommets sont les états du système et où les arêtes sont exactement données par une transition du système. De ce graphe, on peut déduire l'ensemble de toutes les traces d'exécution que peut engendrer le système, c'est-à-dire où chaque élément de la trace est en relation avec le précédent par une transition du système : ce sont les chemins du graphe.

2.3.2 Sémantique, Encapsulation et Assemblage

L'encapsulation des variables d'un système restreint les valeurs que peuvent prendre ces variables dans la sémantique du système. En effet, l'Univers ne peut plus agir dessus. Alors qu'une variable globale à un système y est libre de prendre une valeur arbitraire, les variables locales sont contraintes par les processus du système. Ceci s'exprime formellement par la proposition suivante.

Proposition 2.2 (Encapsulation et sémantique opérationnelle)

$$\forall S\ X\ Y, [\![S/X \cup Y]\!] \subseteq [\![S/X]\!]$$

✓ *Cette partie du document a fait l'objet d'un développement* Coq *qui peut se trouver en ligne à l'adresse* `http://nicolas.aih.free.fr/these.html` *et qui contient les définitions jusqu'à ce point ainsi que la preuve de cette proposition. Ici, nous n'en donnons donc que les grandes lignes*

Preuve : il faut montrer que toute trace $t \in S/X \cup Y$ appartient également à S/X. Il suffit donc de montrer que dans une trace quelconque, une transition du système $S/X \cup Y$ entre deux états de la trace est aussi une transition de S/X. Il y a donc trois cas à considérer, un pour chaque façon d'effectuer une transition dans le système $S/X \cup Y$.

– Si c'est une transition Univers, cela signifie que les processus de S n'ont pas été exécutés et que les valeurs des éléments de $X \cup Y$ n'ont pas changé. En particulier, les valeurs des éléments de X n'ont pas changé, et puisque les processus de S n'ont pas été exécutés, alors cela correspond également à une transition Univers de S/X.
– Pour les transitions Locale et Ordonnanceur, la qualité locale ou non des variables et évènements n'intervient pas si bien qu'une transition Locale ou Ordonnanceur de $S/X \cup Y$ est également une transition Locale ou Ordonnanceur de S/X (et même de toute encapsulation sur S).

□

Un système sans aucun processus ni aucune variable locale est totalement ouvert et sa sémantique contient toutes les traces possibles. En effet, ses transitions locales sont vides (aucun processus) et l'Univers peut agir de façon arbitraire sur n'importe quelle variable. Ajouter des processus à un système force certaines variables à prendre des valeurs particulières. Et encapsuler des variables du système les force à s'en tenir aux valeurs que leur ont imposé les processus du système. On comprend donc qu'un système n'est ni plus ni moins qu'une structure qui pose des contraintes sur les états pendant une exécution. Assembler deux systèmes ensemble ajoute donc des contraintes supplémentaires par rapport à les considérer l'un ou l'autre seul. En effet, l'assemblage fait la somme des variables locales et des processus. Une trace d'exécution de deux systèmes assemblés est donc une trace d'exécution de chaque système et réciproquement, comme le montre le théorème suivant.

Théorème 2.3 (Assemblage et sémantique opérationnelle)

$$\forall S\ S',\ [\![S \otimes S']\!] = [\![S]\!] \cap [\![S']\!]$$

√ *Ce théorème peut également se trouver dans le développement* Coq *disponible sur* `http://nicolas.aih.free.fr/these.html` *. Ici aussi, seules les grandes lignes de la démonstration sont décrites.*

Preuve : l'égalité des ensembles sous-entend une double inclusion. Commençons par montrer que $[\![S]\!] \cap [\![S']\!] \subseteq [\![S \otimes S']\!]$. On suppose qu'une certaine trace t appartient à $[\![S]\!] \cap [\![S']\!]$, c'est-à-dire à $[\![S]\!]$ d'une part et à $[\![S']\!]$ d'autre part. On sait donc que chaque couple d'états successifs de la trace t est en relation par les transitions sémantiques de S et de S', et il faut montrer que chaque couple est également en relation par la transition sémantique de $S \otimes S'$. Soit t_i et t_{i+1} un couple quelconque. On considère les trois cas qui font qu'ils sont en relation par une transition de S.
– S'ils sont en relation par une transition Univers de S, c'est que les processus, variables et évènements locaux à S n'ont pas changé de t_i à t_{i+1}. On considère maintenant le fait que t_i et t_{i+1} sont également en relation par une transition de S', ce qui signifie également trois possibilités.
  – Si c'est encore par une transition Univers, c'est que les processus, variables et évènements locaux à S' n'ont pas été modifié de t_i à t_{i+1}, comme ceux de S. Puisque $S \otimes S'$ est l'union des processus, variables et évènements de S et S', comme ces éléments n'ont pas changé de t_i à t_{i+1}, ils sont donc en relation par une transition Univers de $S \otimes S'$.
  – Si c'est une transition Locale à S', c'est qu'il existe une processus p tel que $t_i \xrightarrow{p} t_{i+1}$. Or, $S \otimes S'$ contient les processus de S et de S', et contient donc p. Donc t_i et t_{i+1} sont en relation par une transition Locale d'un processus (p) de $S \otimes S'$.

  - Si c'est une transition Ordonnanceur, de toute façon cette transition ne dépend par du système considéré et t_i et t_{i+1} sont en relation par une transition Ordonnanceur de $S \otimes S'$.
- Si c'est une transition Ordonnanceur ou Locale à S, t_i et t_{i+1} sont en relation par une transition de $S \otimes S'$ pour les même raisons que précédemment dans le cas de S'.

Maintenant, il nous reste à montrer que $[\![S \otimes S']\!] \subseteq [\![S]\!] \cap [\![S']\!]$. De même, on suppose que t appartient à $[\![S \otimes S']\!]$, et donc qu'un certain t_i et un certain t_{i+1} sont en relation par une transition de $S \otimes S'$. Il faut montrer que t_i et t_{i+1} sont en relation par S et par S'. On considère les trois cas qui font qu'ils sont en relation par une transition de $S \otimes S'$.

- Si c'est une transition Univers, cela signifie que les processus, variables et évènements locaux de $S \otimes S'$ n'ont pas changé. En particulier, puisque $S \otimes S'$ est l'union des processus, variables et évènements locaux de S et de S', cela signifie que les processus, variables et évènements locaux de S d'une part et de S' d'autre part n'ont pas changé, si bien que t_i et t_{i+1} sont en relation par une transition Univers de S et en relation par une transition Univers de S'.
- Si c'est une transition Locale, c'est qu'il existe un processus p de $S \otimes S'$ tel que $t_i \xrightarrow{p} t_{i+1}$. Comme les processus de $S \otimes S'$ sont l'union des processus de S et de S', c'est que p appartient soit à S soit à S'. Les deux cas sont semblables, alors considérons par exemple celui où p appartient à S. On sait déjà que t_i et t_{i+1} sont en relation par une transition Locale de S. De plus, S et S' sont assemblables par hypothèse, c'est-à-dire que l'exécution d'un processus de S ne modifie pas les processus, variables et évènements locaux à S'. Donc t_i et t_{i+1} sont en relation par une transition Univers de S'. Le même raisonnement lorsque p appartient à S' peut être conduit pour déduire que t_i et t_{i+1} sont alors en relation par une transition Univers de S.
- Si c'est une transition Ordonnanceur, celle-ci ne dépend pas du système dans lequel on se place, et t_i et t_{i+1} sont donc en relation par une transition Ordonnanceur de S et également en relation par une transition Ordonnanceur de S'.

□

Dans ce chapitre, nous avons défini ce qu'est un système ainsi que sa sémantique. Dans le chapitre suivant, nous allons définir la notion de propriété sur les systèmes. Nous présenterons également une technique de vérification compositionnelle de propriété d'un système et nous expliquerons de quelle manière se fera cette vérification.

CHAPITRE 3

Vérification Compositionnelle

Ce chapitre explique comment nous utilisons des systèmes pour vérifier des systèmes. La méthode s'appuie sur la notion d'automate observateur du model checking qui par le fait qu'ils *observent* (sous-entendu n'interviennent pas sur) les exécutions d'un système, permettent d'énoncer des propriétés dans le même formalisme que le système. La notion de sûreté, ou de vérification, sera encodée dans les états des systèmes. La modularité de l'approche est donnée par des abstractions sûres de systèmes par d'autres, potentiellement plus simples, qui contournent le problème de l'explosion combinatoire du nombre d'états d'un système. Nous reprenons des notions d'assertions et d'assume-guarantee [WL05], d'automates observateurs, de model checking et d'interprétation abstraite [CC77] pour définir un cadre de vérification compositionnelle qui identifie précisément trois algorithmes qui suffisent à vérifier la validité d'une propriété par un système dont des parties peuvent être remplacées par d'autres. Dans ce cadre, les automates observateurs vont avoir un rôle prépondérant lors des analyses puisqu'ils vont permettre d'apporter de la précision aux approximations générées par l'interprétation abstraite. Nous différencions donc deux types de systèmes observateurs : les systèmes qui servent à représenter des propriétés et les systèmes dits *discriminants* qui servent à apporter de la précision aux analyses.

3.1 Sûreté Locale et Propriété

3.1.1 Configuration Sûre

La notion de propriété s'articule autour des configurations des processus. Pendant son exécution, chaque processus a la possibilité de se placer dans une *configuration erreur*. On dispose d'un prédicat qui permet de savoir si une configuration est erronée ou non.

Définition 3.1 Prédicat $P \in \mathit{Prop}$

Le fait qu'une configuration est erronée ou non est une *proposition logique*. Nous choisissons le cadre de la logique classique dans ce document et une proposition sera donc soit vraie soit fausse par axiome. On pourrait donc représenter la validité d'une proposition par un booléen, mais nous préférons garder cet ensemble comme type de données. Nous représentons les propositions par un ensemble noté *Prop*, afin de bien distinguer l'aspect *type de données* de l'aspect *énoncé logique*. Étant donné une proposition P, on note simplement P le fait que la proposition P est vraie, et on note $\neg P$ le fait que la proposition P est fausse.

Paramètre 3.2 Configuration sûre $\mathit{Safe} \in \mathcal{C} \to \mathit{Prop}$

Ainsi, une propriété s'exprime simplement par un processus qui se place en configuration d'erreur lorsque la propriété est violée. Par exemple, un processus qui attend la notification d'un certain évènement e puis qui se déplace dans une configuration erreur si la valeur d'une variable x est inférieure à 100 à ce moment là modélise la propriété qui énonce que x est inférieure à 100 juste après que e a été émis. Un système qui contient un tel processus et dont les traces n'atteignent pas de configuration erreur satisfait automatiquement la propriété. Mais la sémantique d'un système prend en compte tous les systèmes avec lesquels il peut être assemblé. Vérifier qu'aucun processus n'atteint de configuration erreur n'a pas de sens : il est toujours possible qu'il existe un processus en dehors du système et dont on ne sait rien qui peut se mettre en configuration erreur. Il suffit de considérer le système qui n'a aucune variable et aucun évènement locaux, et qui n'est formé que d'un seul processus qui boucle indéfiniment sur une configuration erreur. Ce système est assemblable avec n'importe quel autre et toutes les traces de l'assemblage seront erronées. Plutôt que de vérifier la sûreté de l'Univers de processus tout entier, on se cantonne à vérifier la sûreté d'un système *localement*. On étudie la sûreté locale à un système, qui signifie alors : quel que soit les systèmes avec lesquels il peut être assemblé, une erreur ne peut pas venir du système étudié. La notion de sûreté est toujours paramétrée par un système à vérifier. Nous définissons cette sûreté sur différents ensembles — les états et les traces — en surchargeant le nom de la fonction.

Définition 3.3 État sûr $\mathit{Safe} \in \mathcal{S} \times \Sigma \to \mathit{Prop}$

$$\mathrm{Safe}(S, \sigma) \quad \triangleq \quad \forall p \in P_S,\ \mathrm{Safe}(\sigma.C(p))$$

Définition 3.4 Trace sûre $\mathit{Safe} \in \mathcal{S} \times \mathcal{T} \to \mathit{Prop}$

$$\mathrm{Safe}(S, t) \quad \triangleq \quad \forall i,\ \mathrm{Safe}(S, t_i)$$

Ces définitions conduisent à un lemme qui énonce qu'une trace sûre dans un assemblage est une trace sûre dans chacun des systèmes de l'assemblage.

Lemme 3.1 $\forall S\ S'\ t,\ \mathit{Safe}(S \otimes S', t) \Leftrightarrow \mathit{Safe}(S, t)$ et $\mathit{Safe}(S', t)$

Preuve : $Safe(S \otimes S', t)$ signifie par définition que $\forall i,\ \forall p \in P_{S\otimes S'},\ Safe(t_i.C(p))$. Or, $P_{S\otimes S'} = P_S \cup P_{S'}$ par définition de l'assemblage. Donc on a $\forall i,\ \forall p \in P_S,\ Safe(t_i.C(p))$ qui signifie exactement que $Safe(S, t)$ d'une part, et on a $\forall i,\ \forall p \in P_{S'},\ Safe(t_i.C(p))$ qui signifie exactement que $Safe(S', t)$ d'autre part. □

3.1.2 Propriété des Systèmes

Si un système S est assemblé avec un système S' et que $S \otimes S'$ est sûr (sous-entendu localement), cela signifie qu'aucune trace de $S \otimes S'$ ne conduit un de leurs processus à se retrouver dans une configuration erronée. Si le système S' contient des processus qui peuvent, sous certaines conditions, se placer en configuration erreur, alors le fait d'être une de ces traces ou non représente une propriété des traces, et le système S' représente donc une propriété des ensembles de traces, c'est-à-dire des systèmes. En particulier, si $S \otimes S'$ est sûr, c'est que toutes les traces de S qui sont compatibles avec celles de S' sont sûres, et nous dirons que S valide la propriété S'.

On peut alors imaginer la situation suivante. De la même sorte que S' représente une propriété par des processus qui peuvent se positionner dans une configuration erreur, S peut représenter des hypothèses sur les traces par celles de sa sémantique qui sont sûres. On aimerait donc vérifier que si celles de S sont sûres, alors celles de S' le sont aussi. On obtient une proposition logique qui énonce que sous les hypothèses S (sous-entendu si les traces de S sont toutes sûres), alors on peut déduire S' (sous-entendu les traces de S' sont sûres). Les systèmes peuvent alors jouer le rôle d'hypothèses ou de propriétés des systèmes.

Définition 3.5 Système validant une propriété $\models \in \mathcal{S} \times \mathcal{S} \to Prop$

$$S \models S' \quad \triangleq \quad \forall t \in [\![S \otimes S']\!],\ \mathrm{Safe}(S, t) \Rightarrow \mathrm{Safe}(S', t)$$

Avoir des systèmes pour énoncer des hypothèses ou des propriétés des systèmes offre un cadre de développement et de vérification homogène, très proche du monde de l'ingénierie, contrairement au model checking dont les logiques temporelles sont plutôt réservées à une communauté d'experts. De plus, on bénéficie des mêmes outils pour la simulation et la vérification. Une exécution d'un système S assemblé avec un système S' permet de savoir si cette exécution de S vérifie bien la propriété que décrit S'. Pour cela, il suffit de vérifier que la trace n'est pas sûre pour S' seulement si elle n'est pas sûre non plus pour S. La force du model checking est de proposer des algorithmes statiques qui permettent de pousser ce résultat sur toutes les exécutions de $S \otimes S'$, assurant ainsi que S vérifie la propriété décrite par S', quelle que soit son évolution. Cependant, ceci ne peut se faire que si le nombre d'états de S est fini et relativement petit. Plusieurs techniques permettent de réduire cette explosion du nombre d'états. Nous proposons d'introduire de la modularité en remplaçant les systèmes les uns par les autres, tout en préservant la sûreté.

3.1.3 Comparaison avec les Logiques Temporelles

Encoder les propriétés par des systèmes permet d'exprimer des aspects fonctionnels du fait que les systèmes manipulent les données contenues dans les états, et des aspects temporels du

fait que la sémantique des systèmes est intrinsèquement définie par des séquences d'états. Dès lors, se pose la question de l'expressivité des systèmes en tant que propriétés.

Sans rentrer dans les détails pour l'instant, disons seulement que l'approche par assertions et interprétation abstraite permet de déduire la non-atteignabilité d'états. En particulier, nous pourrons montrer la non-atteignabilité d'un état erreur, c'est-à-dire la propriété d'invariance qui énonce que tous les états sont sûrs. Comme nous le montrerons sur le langage SystemD, l'utilisation de l'approche par assertions et de l'interprétation abstraite nous permettra de vérifier des propriétés de type sûreté [SBB+99], une classe de propriétés très forte pour la vérification de systèmes.

Cependant, la validité définie dans notre cadre ne correspond pas toujours à une propriété de sûreté. Pour le montrer, reprenons la définition de la validité : un système S (hypothèses) valide un système S' (conclusions) par définition lorsque pour toute trace $t \in [\![S \otimes S']\!]$, $\mathit{Safe}(S,t) \Rightarrow \mathit{Safe}(S',t)$. Cela signifie donc que pour vérifier que $S \models S'$, il faut considérer toutes les traces $t \in [\![S \otimes S']\!]$, et vérifier tout d'abord les hypothèses, c'est-à-dire que t est sûre vis-à-vis de S. Si ce n'est pas le cas, la propriété est vérifiée par non respect des hypothèses. Si c'est le cas, c'est-à-dire si aucune hypothèse n'est violée, alors il faut vérifier que t est sûre vis-à-vis de S', c'est-à-dire que toutes les conclusions sont valides. Ceci n'est donc pas une propriété de sûreté. Une conclusion peut-être violée au cours d'une exécution *avant même* qu'une hypothèse ne le soit : cette exécution reste valide. De manière générale, la validité n'est pas une propriété de sûreté.

Les cas particuliers où la validité représente une propriété de sûreté sont les suivants. Parfois, les systèmes à vérifier — ceux du côté des hypothèses — ne contiennent pas d'assertions logiques. Supposons que le langage de description utilisé permette de vérifier cela syntaxiquement. Dans de tels cas, la validité d'un système S (alors sans hypothèse logique) vis-à-vis d'un système S' revient donc à vérifier que pour toute trace $t \in [\![S \otimes S']\!]$, $\mathit{Safe}(S',t)$: c'est une propriété de sûreté.

Vérifier qu'un système est sûr est donc une propriété de sûreté, mais la validité d'un système par rapport à un autre n'en est pas toujours une.

3.2 Modularité

3.2.1 Remplacement

Assembler des systèmes multiplie leurs configurations, rendant la recherche d'état erreur exponentiellement plus longue. Pour contourner ce problème, nous introduisons de la compositionnalité dans la vérification en autorisant le remplacement d'un système par un autre, potentiellement plus simple d'un point de vue combinatoire. Pour que le remplacement s'inscrive dans le contexte de vérification, il faut pouvoir préserver des propriétés vis-à-vis de la sûreté ou non des états, ce qui impose des règles strictes. Pourtant, nous souhaitons rester suffisamment souples pour que le remplacement ait une réelle utilité pratique. Typiquement, un critère de remplacement pourrait être le fait que la sémantique du remplaçant doit contenir au moins toutes les traces de la sémantique du remplacé. Ce critère est trop restrictif car cela

signifie que le remplaçant ne peut pas disposer de ses propres variables locales qui évoluent de façon totalement arbitraire chez le remplacé. On nuance donc la notion de remplacement en la paramétrant par un ensemble I de variables et d'évènements dont la valeur peut changer selon les exécutions d'un système ou d'un autre, et qui contient typiquement les éléments locaux au système remplaçant. Nous imposons également que le remplacement préserve la configuration des processus autres que ceux du remplaçant.

Définition 3.6 Égalité d'états modulo exclusion

$$=_{./.} \in (\mathcal{P}(\mathcal{X} \cup \mathcal{E}) \times \mathcal{P}(\Psi)) \to (\Sigma \times \Sigma) \to \mathit{Prop}$$

$$\sigma =_{I/P} \sigma' \quad \triangleq \quad \begin{cases} & \forall x \notin I,\ \sigma.\rho(x) = \sigma'.\rho(x) \\ \text{et} & \forall e \notin I,\ e \in \sigma.E \Leftrightarrow e \in \sigma'.E \\ \text{et} & \forall p \notin P,\ \sigma.C(p) = \sigma'.C(p) \end{cases}$$

$\sigma =_{I/P} \sigma'$ signifie donc que seules les valeurs des variables de I ont potentiellement changé de σ à σ', seuls les évènements de I n'ont potentiellement pas la même présence de σ à σ', et seuls les processus de P n'ont potentiellement pas la même configuration de σ à σ'. Nous allons donc pouvoir utiliser cette relation pour comparer des traces qui seront les mêmes, à l'exception des informations relatives à un système dont on ne se préoccupe pas.

Clairement, cette relation est une relation d'équivalence (réflexive, symétrique et transitive), et en plus on peut rajouter des variables, évènements et processus à exclure sur des états en préservant le fait qu'ils sont en relation : $\sigma =_{I/P} \sigma' \Rightarrow \sigma =_{I \cup I'/P \cup P'} \sigma'$.

Pour définir qu'une trace d'un système remplace celle d'un autre modulo exclusion, on procède par cas sur chaque transition. Lorsque le système à remplacer exécute une transition Univers, le remplaçant peut effectuer deux transitions Univers. Cela permet d'ajouter des systèmes au remplacé qui pourront se retrouver chez le remplaçant. Lorsque le système à remplacer exécute une transition locale, le système remplaçant peut effectuer un certain nombre de transitions locales (ou aucune) suivies d'exactement une transition Univers (l'identité est une transition Univers) pour mettre en relation des états qui sont égaux à ceux du remplacé modulo exclusion. Enfin, dans le cas d'une transition ordonnanceur, le système remplaçant doit effectuer une transition ordonnanceur qui ne modifie pas le résultat sur les configurations des processus qui ne dépendent pas des évènements exclus (elles peuvent dépendre des autres configurations, comme c'est le cas avec l'écoulement du temps) ; cette action doit également préserver le résultat sur la présence ou non d'évènements en dehors de ceux exclus. Ensuite, le système remplaçant peut effectuer une transition Univers. Ceci permet de garantir que l'exécution du remplaçant va globalement au même rythme que celle du remplacé, quel que soit un assemblage avec un système tiers.

Définition 3.7 Remplacement d'une trace modulo exclusion

$$\prec_{./.} \in (\mathcal{P}(\mathcal{X} \cup \mathcal{E}) \times \mathcal{S}) \to (\mathcal{T} \times \mathcal{T}) \to \mathit{Prop}$$

Étant donnés deux systèmes S et S', un ensemble de variables et d'évènements I, une trace $t \in [\![S]\!]$ et une trace $t' \in [\![S']\!]$, on note $t \prec_{I/S'} t'$ le fait que t' peut remplacer t modulo exclusion de I, en respectant la configuration des processus autres que ceux de S'. La définition se trouve en figure 3.2.1.

$$
t \prec_{I/S'} t' \quad \triangleq \quad \exists (u_n)_{n\in\mathbb{N}} \text{ croissante}, \left\{ \begin{array}{ll}
 & u_0 = 0 \\
\text{et } \forall i, & t_i =_{I/P_{S'}} t'_{u_i} \\
\text{et} & t'_{u_i} \xrightarrow{\text{Univers}(S')} t'_{u_i+1} \xrightarrow{\text{Univers}(S')} t'_{u_{i+1}} \\
 & \quad \text{où } \forall p \notin P_{S'},\ R_p \cap I = \emptyset \Rightarrow t'_{u_i+1}.C(p) = t_{i+1}.C(p) \\
 & \quad \text{si } t_i \xrightarrow{\text{Univers}(S)} t_{i+1} \\
\text{et} & \exists p_1 \ldots p_k \in P_{S'}, \\
 & \quad t'_{u_i} \xrightarrow{p_1} t'_{u_i+1} \xrightarrow{p_2} \ldots \xrightarrow{p_k} t'_{u_{i+1}-1} \xrightarrow{\text{Univers}(S')} t'_{u_{i+1}} \\
 & \quad \text{si } \exists p \in P_S,\ t_i \xrightarrow{p} t_{i+1} \\
\text{et} & t'_{u_i} \xrightarrow{\text{Sched}} t'_{u_i+1} \xrightarrow{\text{Univers}(S')} t'_{u_{i+1}} \\
 & \quad \text{où} \quad \forall p \notin P_{S'},\ R_p \cap I = \emptyset \Rightarrow t'_{u_i+1}.C(p) = t_{i+1}.C(p) \\
 & \quad \text{et} \quad \forall e \notin I,\ e \in t'_{u_i+1}.E \Leftrightarrow e \in t_{i+1}.E \\
 & \quad \text{si } t_i \xrightarrow{\text{Sched}} t_{i+1}
\end{array} \right.
$$

FIG. 3.1 – Définition du remplacement de trace

Le fait qu'un système peut en remplacer un autre modulo des variables et évènements exclus I signifie que pour chaque trace du remplacé, il existe une trace chez le remplaçant qui la remplace modulo les variables et évènements exclus et où seules les configurations des processus du remplaçant et des processus qui lisent des évènements de I peuvent éventuellement être différentes. Finalement, une trace et un remplaçant pour cette trace sont semblables à partir du moment où on n'observe pas quelques éléments particuliers (les variables et évènement exclus ainsi que les processus du système remplaçant) et qu'on autorise un nombre indéfini d'états dans les traces entre deux états égaux modulo exclusion. Pour cette raison, le remplacement de traces est une relation de bisimulation faible [Mil89]. Ainsi, un remplaçant S' pourra effectivement remplacer un système S dans n'importe quel assemblage avec un système S_1 sans restreindre globalement les possibilités d'évolution, à partir du moment où S_1 ne dépend pas du fonctionnement interne de S', symbolisé par les variables et évènements exclus I et la configuration de ses processus. À partir de cela, on distingue deux types de remplacements : le remplacement d'une hypothèse (c'est-à-dire d'un système dont on veut montrer une propriété), et le remplacement d'une conclusion (c'est-à-dire d'un système qui représente une propriété).

Définition 3.8 Remplacement d'une hypothèse $\rhd_{./\Rightarrow} \in \mathcal{P}(\mathcal{X} \cup \mathcal{E}) \to \mathcal{S} \times \mathcal{S} \to \mathit{Prop}$

$$
S \rhd_{I/\Rightarrow} S' \quad \triangleq \quad \forall t \in [\![S]\!],\ \exists t' \in [\![S']\!],\ t \prec_{I/P_{S'}} t' \text{ et } (\text{Safe}(S,t) \Rightarrow \text{Safe}(S',t'))
$$

Définition 3.9 Remplacement d'une conclusion $\rhd_{./\Leftarrow} \in \mathcal{P}(\mathcal{X} \cup \mathcal{E}) \rightarrow \mathcal{S} \times \mathcal{S} \rightarrow Prop$

$$S \rhd_{I/\Leftarrow} S' \quad \triangleq \quad \forall t \in [\![S]\!],\ \exists t' \in [\![S']\!],\ t \prec_{I/P_{S'}} t' \text{ et } (\text{Safe}(S', t') \Rightarrow \text{Safe}(S, t))$$

3.2.2 Remplacement et Validité

On attend du remplacement qu'il permette de montrer des propriétés de validité du système qu'il remplace. Dans le cas d'un remplacement d'hypothèse, il faut s'assurer que toutes les propriétés du remplaçant sont des propriétés du remplacé. Dans le cas d'un remplacement de conclusion, il faut montrer à l'inverse que toutes les propriétés du remplacé sont des propriétés du remplaçant. On retrouve la dichotomie entre les deux côtés d'une implication. Le remplacement joue donc un rôle d'abstraction correcte vis-à-vis de la validité. Pour montrer ce résultat, nous commençons par montrer la cohérence des remplacements d'hypothèse et de conclusion par rapport à l'assemblage.

Lemme 3.2

$$\begin{array}{ll} \forall S_1\ S_1'\ S_2\ I, & R(S_2) \cap I = \emptyset \text{ et } S_1 \rhd_{I/\Rightarrow} S_1' \quad \Longrightarrow \\ & \forall t \in [\![S_1 \otimes S_2]\!],\ \exists t'' \in [\![S_1' \otimes S_2]\!],\ t \prec_{I/P_{S_1'}} t'' \text{ et } \text{Safe}(S_1, t) \Rightarrow \text{Safe}(S_1', t'') \end{array}$$

Preuve : cette preuve est relativement longue. La démonstration complète se trouve en annexe A.1. Ici, nous donnons seulement les intuitions de la preuve par souci de clarté.

Le but de la preuve est de construire une trace $t'' \in [\![S_1' \otimes S_2]\!]$ qui remplace t, variables et évènements de I et processus de S_1' exclus, en sachant que l'on dispose d'une trace $t' \in [\![S_1']\!]$ qui remplace t avec la même exclusion. De plus, il faudra également montrer que la sûreté des processus de $S_1 \otimes S_2$ en t est préservée par la sûreté des processus de $S_1' \otimes S_2$ en t'', sachant que la sûreté des processus de S_1 en t est préservée par la sûreté des processus de S_1' en t'. Pour tout cela, on procède par cas et par récurrence pour construire t''. On fait en sorte de garder les informations de t' pour S_1' et les informations de t pour le reste. L'hypothèse $R(S_2) \cap I = \emptyset$ permet de transporter les mêmes exécutions de S_2 en t vers t'' : en effet, les variables que lit S_2 sont restées les mêmes. Quel que soit le type de transition en t (Univers, locale ou ordonnanceur), il est possible que des variables que lit S_1' soit différentes de t à t'. La transition Univers supplémentaire qu'autorise la définition du remplacement de trace permet de rétablir les valeurs dont S_1' a besoin pour s'exécuter en t'' comme dans t'. Au final, chaque transition dans t'' est une transition de S_1' et de S_2 : si un processus de S_1 s'est exécuté, on a une exécution de S_1' qui simule les mêmes changements, et on est donc dans l'Univers de S_2 (comme S_1' et S_2 sont assemblables, S_1' ne peut pas écrire sur les locaux de S_2). Si S_2 s'est exécuté, on reprend la même exécution qui est Univers pour S_1' car elle l'était pour S_1. De plus, nous n'avons ni rajouté ni enlevé de configurations des processus de S_2 de t à t''. Cette dernière proposition permet de déduire la préservation de la sûreté des processus de S_2. Et comme S_1' préserve la sûreté des processus de S_1, la trace t'' de $S_1' \otimes S_2$ préserve bien la sûreté des processus de $S_1 \otimes S_2$ en t. □

Lemme 3.3

$$\begin{array}{ll} \forall S_1\ S_1'\ S_2\ I, & R(S_1) \cap I = \emptyset \text{ et } S_2 \rhd_{I/\Leftarrow} S_2' \quad \Longrightarrow \\ & \forall t \in [\![S_1 \otimes S_2]\!],\ \exists t'' \in [\![S_1 \otimes S_2']\!],\ t \prec_{I/P_{S_2'}} t'' \text{ et } \text{Safe}(S_2', t'') \Rightarrow \text{Safe}(S_2, t) \end{array}$$

Preuve : on reprend exactement la même construction d'un $t'' \in [\![S_1 \otimes S_2']\!]$ que pour le lemme précédent, avec un certain $t' \in [\![S_1 \otimes S_2]\!]$ (tel que $Safe(S_2', t') \Rightarrow Safe(S_2, t)$ en particulier), et la preuve qui va avec. Seule la partie pour montrer $Safe(S_2', t'') \Rightarrow Safe(S_2, t)$ change (à peine) comme suit. La sûreté des processus de S_1 est la même de t à t'', et la sûreté des processus de S_2' en t'' est préservée par celle des processus de S_2 en t, donc les processus de $S_1 \otimes S_2$ en t préservent bien la sûreté des processus de $S_1 \otimes S_2'$ en t''. □

On ajoute un lemme supplémentaire qui montre que le remplacement d'une trace ne change pas les configurations des processus qui n'ont aucune variable et aucun évènement commun avec celles et ceux exclus.

Lemme 3.4

$$\begin{aligned} \forall S\ p\ t\ t'\ I, \quad & p \notin P_S \text{ et } R_p \cap I = \emptyset \text{ et } t \prec_{I/P_S} t' \quad \Longrightarrow \\ & \forall i,\ \exists i',\ t_i.C(p) = t'_{i'}.C(p) \text{ et } \forall i', \exists i,\ t'_{i'}.C(p) = t_i.C(p) \end{aligned}$$

Preuve : d'après la définition de $\prec$ sur les traces, on sait que $\forall i,\ \exists i', t_i =_{I/P_S} t'_{i'}$, et donc, puisque $p \notin P_S$, que $t_i.C(p) = t'_{i'}.C(p)$. Pour montrer la seconde proposition, il faut vérifier que les états qu'on ajoute dans t' par rapport à t n'ont pas des configurations différentes pour p. Dans le cas d'une transition Univers, on ajoute un seul état qui garde la même configuration en $t'_{i'+1}$ qu'en $t'_{i'}$ pour les processus p' tels que $p' \notin P_S$ et $R_{p'} \cap I = \emptyset$, ce qui est le cas de p. Et cette configuration en $t'_{i'}$ est la même qu'en t_i puisque $t_i =_{I/P_S} t'_{i'}$. Dans le cas d'une transition locale, les états qu'on ajoute sont tous obtenus par des transitions de processus locaux de S, donc seules leur configuration est modifiée. Enfin, dans le cas d'une transition ordonnanceur, encore une fois on ajoute un seul état où pour tout processus p' qui a la même qualité que p — c'est-à-dire tels que $p' \notin P_S$ et $R_{p'} \cap I = \emptyset$ —, p' conserve sa configuration en $t'_{i'+1}$ depuis celle de $t'_{i'}$ qui est également celle en t_i. □

Ce lemme permet de montrer que la sûreté des systèmes qui ne lisent pas de variables ou d'évènements dont la valeur peut changer d'une trace à une autre lors d'un remplacement préserve rigoureusement la sûreté.

Corollaire 3.5

$$\forall S\ S'\ I, \quad P_S \cap P_{S'} = \emptyset \text{ et } R(S) \cap I = \emptyset \text{ et } t \prec_{I/P_{S'}} t' \quad \Longrightarrow \quad \text{Safe}(S,t) \Leftrightarrow \text{Safe}(S,t')$$

Preuve : c'est une application directe du lemme 3.4 précédent. En effet, quel que soit un processus $p \in P_S$, on sait par hypothèse que $p \notin P_{S'}$, et puisque $R(S) \cap I = \emptyset$, alors $R_p \cap I = \emptyset$ ($R(S)$ est l'union des variables et évènements lus de chacun de ses processus). Du coup, on sait que quel que soit un processus de S, ses configurations sont les mêmes en t et en t'. Il y a une configuration erreur d'un processus de S en t si et seulement si il y en a une en t'. □

À présent, nous avons tous les éléments pour énoncer et montrer les théorèmes de correction des remplacements.

Remplacement d'Hypothèse et Validité. Le premier concerne le remplacement d'une hypothèse. Un tel remplacement énonce que valider une propriété sur un système qui en

remplace un autre revient à montrer que le système remplacé valide la même propriété.

Théorème 3.6 (Remplacement d'Hypothèse et Validité)

$$\frac{S_1 \rhd_{I/\Rightarrow} S_1' \qquad S_1' \models S_2}{S_1 \models S_2} \text{ Si } R(S_2) \cap I = \emptyset$$

Preuve : on considère une trace $t \in [\![S_1 \otimes S_2]\!]$ telle que $Safe(S_1, t)$. On doit montrer que $Safe(S_2, t)$. Or, puisque $S_1 \rhd_{I/\Rightarrow} S_1'$ et que $R(S_2) \cap I = \emptyset$, on sait d'après le lemme 3.2 qu'il existe une trace $t' \in [\![S_1' \otimes S_2]\!]$ telle que $t \prec_{I/P_{S_1'}} t'$ et $Safe(S_1, t) \Rightarrow Safe(S_1', t')$. Puisque $P_{S_2} \cap P_{S_1'} = \emptyset$ (car les deux systèmes sont assemblables) et que $t \prec_{I/P_{S_1'}} t'$, on sait de plus d'après le corollaire 3.5 que $Safe(S_2, t) \Leftrightarrow Safe(S_2, t')$. Puisqu'on a effectivement $Safe(S_1, t)$ par hypothèse, alors on a $Safe(S_1', t')$. Or on sait également par hypothèse que $S_1' \models S_2$, c'est-à-dire que $\forall t \in [\![S_1' \otimes S_2]\!]$, $Safe(S_1', t) \Rightarrow Safe(S_2, t)$. Donc en particulier dans le cas de $t' \in [\![S_1' \otimes S_2]\!]$, on a $Safe(S_2, t')$. Enfin, le fait que $Safe(S_2, t) \Leftrightarrow Safe(S_2, t')$ permet de conclure que $Safe(S_2, t)$. □

Remplacement de Conclusion et Validité. Le second théorème concerne le remplacement d'une conclusion. Un tel remplacement énonce que si un système valide une propriété qui en remplace une autre, et que la première propriété remplace la seconde, alors le système valide également la seconde propriété.

Théorème 3.7 (Remplacement de Conclusion et Validité)

$$\frac{S_2 \rhd_{I/\Leftarrow} S_2' \qquad S_1 \models S_2'}{S_1 \models S_2} \text{ Si } R(S_1) \cap I = \emptyset$$

Preuve : on considère une trace $t \in [\![S_1 \otimes S_2]\!]$ telle que $Safe(S_1, t)$. On doit montrer que $Safe(S_2, t)$. Or, puisque $S_2' \rhd_{I/\Leftarrow} S_2$ et que $R(S_1) \cap I = \emptyset$, on sait d'après le lemme 3.3 qu'il existe une trace $t' \in [\![S_1 \otimes S_2']\!]$ telle que $t \prec_{I/P_{S_2'}} t'$ et $Safe(S_2', t') \Rightarrow Safe(S_2, t)$. Puisque $P_{S_1} \cap P_{S_2'} = \emptyset$ (car les deux systèmes sont assemblables) et que $t \prec_{I/P_{S_2'}} t'$, on sait de plus d'après le corollaire 3.5 que $Safe(S_1, t) \Leftrightarrow Safe(S_1, t')$. Puisqu'on a effectivement $Safe(S_1, t)$ par hypothèse, alors on a $Safe(S_1, t')$. Or on sait également par hypothèse que $S_1 \models S_2'$, c'est-à-dire que $\forall t \in [\![S_1 \otimes S_2']\!]$, $Safe(S_1, t) \Rightarrow Safe(S_2', t)$. Donc en particulier dans le cas de $t' \in [\![S_1 \otimes S_2']\!]$, on a $Safe(S_2', t')$. Enfin, le fait que $Safe(S_2', t') \Rightarrow Safe(S_2, t)$ permet de conclure que $Safe(S_2, t)$. □

La compositionnalité de la vérification consiste simplement à utiliser ces deux théorèmes afin de remplacer au choix un système ou une propriété par des systèmes plus simples. Nous développons les possibilités de remplacements et leurs utilisations en pratique.

3.2.3 Vérification par Abstractions et Raffinements

Le remplacement permet une méthodologie de vérification par abstractions successives. Si on souhaite montrer qu'un système composé de plusieurs sous-systèmes vérifie une certaine propriété, la taille du système peut rendre le calcul trop long. Pour montrer que $S_1 \models S$, on peut utiliser un système S'_1 plus simple qui remplace S_1 et montrer que $S'_1 \models S$: c'est le théorème 3.6. Le niveau suivant de la compositionnalité est de pouvoir remplacer *une partie* d'un système par un autre dans une optique de recherche de validité. Pour cela, on montre tout d'abord deux corollaires sur la relation entre assemblage et les remplacement d'hypothèse et de conclusion.

Corollaire 3.8 $\forall S_1\ S'_1\ S_2\ I, \quad R(S_2) \cap I = \emptyset$ et $S_1 \rhd_{I/\Rightarrow} S'_1 \quad \Longrightarrow \quad S_1 \otimes S_2 \rhd_{I/\Rightarrow} S'_1 \otimes S_2$

Preuve : on part d'une certaine trace $t \in [\![S_1 \otimes S_2]\!]$ et on doit montrer qu'il existe une trace $t' \in [\![S'_1 \otimes S_2]\!]$ telle que $t \prec_{I/P_{S'_1 \otimes S_2}} t'$ et $Safe(S_1 \otimes S_2, t) \Rightarrow Safe(S'_1 \otimes S_2, t')$. Puisque $R(S_2) \cap I = \emptyset$, $S_1 \rhd_{I/\Rightarrow} S'_1$ et $t \in [\![S_1 \otimes S_2]\!]$, le lemme 3.2 permet de déduire qu'il existe une trace $t' \in [\![S'_1 \otimes S_2]\!]$ telle que $t \prec_{I/P_{S'_1}} t'$ et $Safe(S_1, t) \Rightarrow Safe(S'_1, t')$. La remarque qui suit la définition de $\prec$ sur les traces permet directement d'avoir $t \prec_{I/P_{S'_1} \cup P_{S_2}} t'$ (on ajoute des processus qui peuvent potentiellement, mais pas obligatoirement, prendre de nouvelles configurations). De plus, par définition de $\otimes$, on a $P_{S'_1} \cup P_{S_2} = P_{S'_1 \otimes S_2}$. Donc on a bien $t \prec_{I/P_{S'_1 \otimes S_2}} t'$. Comme S'_1 et S_2 sont assemblables, ils n'ont pas de processus communs : $P_{S'_1} \cap P_{S_2} = \emptyset$. On sait également par hypothèse que $R(S_2) \cap I = \emptyset$ et on a $t \prec_{I/P_{S'_1}} t'$. Le corollaire 3.5 permet donc de déduire que $Safe(S_2, t) \Leftrightarrow Safe(S_2, t')$. Le fait que $Safe(S_1 \otimes S_2, t)$ implique que $Safe(S_1, t)$ et $Safe(S_2, t)$ d'après le lemme 3.1. Comme on a $Safe(S_1, t) \Rightarrow Safe(S'_1, t')$ et $Safe(S_2, t) \Leftrightarrow Safe(S_2, t')$, on peut déduire que $Safe(S'_1, t')$ et $Safe(S_2, t')$, et donc, toujours d'après le lemme 3.1 $Safe(S'_1 \otimes S_2, t')$. □

Corollaire 3.9 $\forall S_1\ S'_1\ S_2\ I, \quad R(S_2) \cap I = \emptyset$ et $S'_2 \rhd_{I/\Leftarrow} S_2 \quad \Longrightarrow \quad S_1 \otimes S'_2 \rhd_{I/\Leftarrow} S_1 \otimes S_2$

Preuve : la preuve de ce corollaire est similaire à la preuve du corollaire précédent. On part d'une certaine trace $t \in [\![S_1 \otimes S_2]\!]$ et on doit montrer qu'il existe une trace $t' \in [\![S_1 \otimes S'_2]\!]$ telle que $t \prec_{I/P_{S_1 \otimes S'_2}} t'$ et $Safe(S_1 \otimes S'_2, t') \Rightarrow Safe(S_1 \otimes S_2, t)$. Puisque $R(S_2) \cap I = \emptyset$, $S'_2 \rhd_{I/\Leftarrow} S_2$ et $t \in [\![S_1 \otimes S_2]\!]$, le lemme 3.3 permet de déduire qu'il existe une trace $t' \in [\![S_1 \otimes S'_2]\!]$ telle que $t \prec_{I/P_{S'_2}} t'$ et $Safe(S'_2, t') \Rightarrow Safe(S_2, t)$. La remarque qui suit la définition de $\prec$ sur les traces permet directement d'avoir $t \prec_{I/P_{S_1} \cup P_{S'_2}} t'$ (on ajoute des processus qui peuvent potentiellement, mais pas obligatoirement, prendre de nouvelles configurations). De plus, par définition de $\otimes$, on a $P_{S_1} \cup P_{S'_2} = P_{S_1 \otimes S'_2}$. Donc on a bien $t \prec_{I/P_{S_1 \otimes S'_2}} t'$. Comme S_1 et S'_2 sont assemblables, ils n'ont pas de processus communs : $P_{S_1} \cap P_{S'_2} = \emptyset$. On sait également par hypothèse que $R(S_1) \cap I = \emptyset$ et on a $t \prec_{I/P_{S'_2}} t'$. Le corollaire 3.5 permet donc de déduire que $Safe(S_1, t) \Leftrightarrow Safe(S_1, t')$. Le fait que $Safe(S_1 \otimes S'_2, t')$ implique que $Safe(S_1, t')$ et $Safe(S'_2, t')$ d'après le lemme 3.1. Comme on a $Safe(S_1, t) \Leftrightarrow Safe(S_1, t')$ et $Safe(S'_2, t') \Leftrightarrow Safe(S_2, t)$, on peut déduire que $Safe(S_1, t)$ et $Safe(S_2, t)$, et donc, toujours d'après le lemme 3.1 $Safe(S_1 \otimes S_2, t)$. □

C'est grâce à ces lemmes que nous montrons comment utiliser le remplacement dans un assemblage et vis-à-vis de la validité.

Théorème 3.10 (Remplacement d'Hypothèse par Parties)

$$\frac{S_1 \rhd_{I/\Rightarrow} S_1' \qquad S_1' \otimes S_2 \models S}{S_1 \otimes S_2 \models S} \text{ Si } R(S_2) \cap I = R(S) \cap I = \emptyset$$

Preuve : puisque $R(S_2) \cap I = \emptyset$ et $S_1 \rhd_{I/\Rightarrow} S_1'$ alors, d'après le corollaire 3.8, $S_1 \otimes S_2 \rhd_{I/\Rightarrow} S_1' \otimes S_2$. Et puisque $S_1 \otimes S_2 \rhd_{I/\Rightarrow} S_1' \otimes S_2$ et $S_1' \otimes S_2 \models S$ et $R(S) \cap I = \emptyset$, alors $S_1 \otimes S_2 \models S$ d'après le théorème 3.6. □

Théorème 3.11 (Remplacement de Conclusion par Parties)

$$\frac{S_2' \rhd_{I/\Leftarrow} S_2 \qquad S \models S_1 \otimes S_2'}{S \models S_1 \otimes S_2} \text{ Si } R(S_1) \cap I = R(S) \cap I = \emptyset$$

Preuve : puisque $R(S_1) \cap I = \emptyset$ et $S_2' \rhd_{I/\Leftarrow} S_2$ alors, d'après le corollaire 3.9, $S_1 \otimes S_2' \rhd_{I/\Leftarrow} S_1 \otimes S_2$. Et puisque $S \models S_1 \otimes S_2'$ et $R(S) \cap I = \emptyset$, alors $S \models S_1 \otimes S_2$ d'après le théorème 3.6. □

C'est le premier théorème, celui sur le remplacement d'hypothèse qui nous intéresse particulièrement. En effet, les propriétés sont généralement des systèmes déjà assez simples qu'on ne cherchera pas à remplacer en pratique. En revanche, les systèmes que l'on souhaite vérifier sont souvent formés d'un assemblage de plusieurs sous-systèmes. Si on cherche à montrer $S_1 \otimes ... \otimes S_n \models S$, on peut, grâce au théorème 3.10 de remplacement d'hypothèse par parties, remplacer chaque sous-système et, plus intéressant encore, on peut remplacer toute une partie du système par un autre. Il est peut-être possible de déterminer un système S' qui remplace $S_1 \otimes ... \otimes S_i$. Il suffit alors de montrer $S' \otimes S_{i+1} \otimes ... \otimes S_n \models S$. Et on peut continuer ainsi et se débarrasser au fur et à mesure des détails qui n'interviennent pas dans la preuve de la propriété représentée par S jusqu'à obtenir un système S'' qui remplace $S_1 \otimes ... \otimes S_n$. Si $S'' \models S$, on pourra effectivement conclure que $S_1 \otimes ... \otimes S_n \models S$. En effet, le remplacement est une relation transitive, et c'est ce qui fait la compositionnalité.

L'approche réciproque est également possible. Cela consiste à raffiner des systèmes petit à petit. Un système S_1 haut niveau peut suffire à prouver une propriété représentée par un système S. Si S_1 remplace un certain système S_2, on sait que S_2 valide S lui aussi. S_1 joue alors le rôle de *spécification* de S_2. Et on dit que S_2 est un raffinement de S_1. Des étapes de raffinement supplémentaires peuvent être mises en place, mais on peut aussi utiliser un système S_3 qui est un raffinement de S_1 mais qui n'est pas comparable avec S_2 : des systèmes incomparables peuvent satisfaire une même spécification.

L'abstraction (remplacement par un système plus général) consiste à utiliser le théorème 3.10 de remplacement d'hypothèse par parties du bas vers le haut et le raffinement (remplacement

par un système plus spécialisé) à l'utiliser du haut vers le bas. Les techniques d'abstraction et de raffinement sont utilisées dans tout développement logiciel de système dans des approches bottom-up ou top-down. La vérification compositionnelle présentée s'inscrit donc directement dans cette philosophie, ce qui en fait un cadre confortable pour l'ingénieur car elle s'intègre naturellement dans le cycle de développement d'un système.

On peut également assembler des spécifications pour montrer des propriétés sur les systèmes qu'ils spécifient, comme le montre le corollaire suivant.

Corollaire 3.12

$$\frac{S_1 \rhd_{I_1/\Rightarrow} S'_1 \qquad S_2 \rhd_{I_2/\Rightarrow} S'_2 \qquad S'_1 \otimes S'_2 \models S}{S_1 \otimes S_2 \models S} \text{ Si } R(S'_2) \cap I_1 = R(S_1) \cap I_2 = R(S) \cap (I_1 \cup I_2) = \emptyset$$

Preuve : ce corollaire est simplement une double application du théorème 3.10 de remplacement d'hypothèse par parties, comme le montre l'arbre de dérivation suivant où toutes les feuilles sont des hypothèses du corollaire.

$$\frac{S_2 \rhd_{I_2/\Rightarrow} S'_2 \qquad \dfrac{S_1 \rhd_{I_1/\Rightarrow} S'_1 \qquad S'_1 \otimes S'_2 \models S}{S_1 \otimes S'_2 \models S} \text{ car } R(S'_2) \cap I_1 = R(S) \cap I_1 = \emptyset}{S_1 \otimes S_2 \models S} \text{ car } R(S_1) \cap I_2 = R(S) \cap I_2 = \emptyset$$

□

Remplacement et Interprétation Abstraite. Toutefois, il est important de noter notre utilisation du remplacement en pratique. Supposons que l'on travaille sur un système S_1 dont on veut montrer $S_1 \models S$ pour une certaine propriété S. Pour simplifier le problème, on construit un système S_2 qui devra remplacer S_1. Lorsque les systèmes deviennent trop grands, il devient difficile — le calcul est trop long ou consomme trop d'espace mémoire — d'avoir une représentation finie exacte des comportements de S_1 et de S_2. Sans rentrer dans les détails pour le moment, disons simplement que nous utilisons l'interprétation abstraite [CC77] pour obtenir une abstraction $S_1^\sharp$ de S_1 et une abstraction $S_2^\sharp$ de S_2 qui satisfont $[\![S_1]\!] \subseteq [\![S_1^\sharp]\!]$ et $[\![S_2]\!] \subseteq [\![S_2^\sharp]\!]$ par définition et dont on a une représentation finie. En particulier, on sait alors que $S_1 \rhd_{I/\Rightarrow} S_1^\sharp$ et $S_2 \rhd_{I/\Rightarrow} S_2^\sharp$ pour tout I car $S_1^\sharp$ et $S_2^\sharp$ représentent des sur-ensembles respectifs des traces de S_1 et de S_2 . On peut alors calculer une réponse à la question : $S_2^\sharp$ peut-il remplacer $S_1^\sharp$? Si oui et si on arrive à montrer que $S_2^\sharp \models S$, on aura alors montré $S_1^\sharp \models S$. Et puisque $S_1^\sharp$ est une abstraction de S_1, c'est-à-dire qu'il représente plus de comportements, on aura effectivement montré $S_1 \models S$. Mais à aucun moment lors de ce cheminement logique nous n'avons montré que S_2 remplace S_1. Pour obtenir ce résultat, on utilise simplement deux fois la règle de remplacement, comme le montre la dérivation suivante.

Lemme 3.13

$$\dfrac{\dfrac{\pi}{S_1 \rhd_{/\Rightarrow} S_1^\sharp} \qquad \dfrac{\dfrac{\pi_1}{S_1^\sharp \rhd_{I/\Rightarrow} S_2^\sharp} \qquad \dfrac{\pi_2}{S_2^\sharp \models S}}{S_1^\sharp \models S}\ \text{Si } R(S) \cap I = \emptyset}{S_1 \models S}$$

Preuve : ce lemme est une double application explicite du théorème 3.6 de remplacement d'hypothèse.

On remarque en effet que le système S_2 n'apparaît jamais lui-même ; on n'utilise que son abstraction. Pour montrer que $S_1 \models S$, il suffit donc de donner les dérivations de π, π_1 et π_2. Clairement, la dérivation de π doit être assurée par un opérateur d'abstraction des systèmes, qui garantira $[\![S]\!] \subseteq [\![S^\sharp]\!]$, et donc $S \rhd_{\emptyset/\Rightarrow} S^\sharp$ (que l'on note $S \rhd_{/\Rightarrow} S^\sharp$) quel que soit S. Sans cela, notre application de l'interprétation abstraite ne serait pas correcte. Les contributions de nos travaux en plus de ce cadre méthodologique de vérification modulaire consistent à définir un langage de description de système, puis à donner un outil de vérification statique de remplacement d'hypothèse pour montrer π_1, c'est-à-dire que $S_1^\sharp \rhd_{I/\Rightarrow} S_2^\sharp$, et à donner un outil de vérification statique de validité pour montrer π_2, c'est-à-dire que $S_2^\sharp \models S$ (le fait que $S_2^\sharp$ soit un système abstrait ou non ne change pas ce problème). Enfin, nous introduisons notre dernière contribution dans la suite de ce chapitre. Alors que l'interprétation abstraite fait des approximations, nous allons regagner en précision lors des analyses par l'utilisation de *systèmes discriminants*.

3.3 Non Bloquage

Dans la suite de ce chapitre, nous allons définir une forme très forte d'automate observateur : les *systèmes discriminants*. Nous comparerons la sémantique de systèmes avec ou sans discriminant. Cette comparaison soulève un problème classique en description de système : le bloquage. Le bloquage n'apparaît pas explicitement dans la sémantique des systèmes mais il est bien présent, en particulier sous sa forme la plus difficile à appréhender : le *livelock*. Un livelock se forme lorsqu'un système peut continuellement s'exécuter, empêchant les autres de reprendre leur exécution, ce que l'on souhaite exclure du comportement d'un observateur. En effet, l'assemblage ne garantit pas l'absence de livelock. Dans cette section, nous définissons le non bloquage.

Comparaison de configurations. Pour définir une propriété de non bloquage, on commence par supposer que l'on dispose d'une relation d'ordre partiel sur les configurations.

Paramètre 3.10 Comparaison de configuration $<\in \mathcal{C} \times \mathcal{C} \rightarrow \mathit{Prop}$

Intuitivement, $c < c'$ signifie que c est moins libre que c' en terme de contraintes sur les évènements. Naturellement, on note $c \leq c'$ lorsque $c < c'$ ou $c = c'$. On pose un certain nombre d'hypothèses sur le prédicat $<$ qui seront à vérifier sur les langages sur lesquels sera

utilisée la méthodologie — SystemD dans notre cas. Le premier d'entre-eux énonce qu'être en relation par cette comparaison doit préserver la sûreté des configurations.

Hypothèse 3.11 $\forall c\ c',\ c < c' \Rightarrow (\text{Safe}(c) \Leftrightarrow \text{Safe}(c'))$

La seconde hypothèse concerne les exécutions d'un processus.

Hypothèse 3.12 $\forall \sigma\ p\ c,\ \sigma.C(p) < c \Rightarrow \{\sigma' \mid \sigma \xrightarrow{p} \sigma'\} \subseteq \{\sigma' \mid \sigma.C[p \leftarrow c] \xrightarrow{p} \sigma'\} \cup \{\sigma\}$

On signifie par cette hypothèse que si un processus se trouve dans une configuration c moins libre qu'une configuration c', alors les exécutions possibles par p dans la configuration c sont également possibles par p dans c', ou alors le processus ne fait rien, ce qui est notamment le cas lorsque le processus est endormi. On aurait pu décider de dire que si le processus est endormi alors son exécution n'a pas d'image, mais le fait de boucler permet de ne pas bloquer les traces et de pouvoir les prolonger infiniment, tout en conservant exactement les mêmes configurations atteignables, donc en restant correct du point de vue de la sûreté.

Enfin, la dernière hypothèse établit que l'action de l'ordonnanceur sur les configurations ne peut que lever des contraintes. Cela assure que l'action de l'ordonnanceur sur les processus en dehors d'un système (en particulier lorsque celui-ci est discriminant) rapproche un peu plus les processus vers un statut éligible. Cette hypothèse nous permettra de vérifier dans certains cas qu'un nombre fini d'actions de l'ordonnanceur conduira forcément un programme à devenir éligible, évitant ainsi les livelocks.

Hypothèse 3.13 $\forall r\ r',\ r \xrightarrow{\text{Sched}} r' \Rightarrow \forall p,\ r.C(p) \leq r'.C(p)$

Comparaison d'états modulo un système. L'ordre sur les configurations permet de définir un ordre sur les états vis-à-vis d'un système.

Définition 3.14 État moins libre modulo un système $\leq_{\_} \in \mathcal{S} \to \Sigma \times \Sigma \to \mathit{Prop}$

$$\sigma \leq_S \sigma' \quad \triangleq \quad \begin{cases} & \forall x \in R(S) \cup W(S),\ \sigma.\rho(x) = \sigma'.\rho(x) \\ \text{et} & \forall e \in R(S) \cup W(S),\ e \in \sigma.E \Leftrightarrow e \in \sigma'.E \\ \text{et} & \forall p \in P_S,\ \sigma.C(p) \leq \sigma'.C(p) \end{cases}$$

Ainsi, un état est moins libre qu'un autre modulo un système S si toutes les variables et tous les évènements de S ont la même valeur d'un état à l'autre, et que chaque processus de S est moins libre dans le premier état que dans le second. On note $\sigma =_S \sigma'$ lorsque $\sigma \leq_S \sigma'$ et $\sigma' \leq_S \sigma$, c'est-à-dire lorsque σ et σ' ont les mêmes valeurs pour les variables et évènements de S et les mêmes configurations pour les processus de S.

Non bloquage. Le bloquage s'exprime justement en terme de comparaison d'états modulo un système. On note $S' \not\prec S$ le fait que S' ne bloque pas les exécutions de S.

Définition 3.15 Non bloquage $\not\prec \in \mathcal{S} \times \mathcal{S} \to \mathit{Prop}$

$$S' \not\prec S \quad \triangleq \quad \forall \sigma_1\ \sigma_1',\ \ \sigma_1 \xrightarrow{\text{Sched}} \sigma_1' \text{ et } \left(\begin{array}{ll} & \sigma_1 \leq_S \sigma_2 \\ \text{et} & \forall p \in P_{S'},\ \sigma_2.C(p) \in C_p \\ \text{et} & \forall p \notin P_S \cup P_{S'},\ \sigma_2.C(p) = \sigma_1.C(p) \end{array} \right) \implies$$
$$\exists p_1\ ...\ p_i \in P_{S'} \cup \{\text{Sched}\},\ \exists \sigma_2^1\ ...\ \sigma_2^i,$$
$$\sigma_2 \xrightarrow{p_1} \sigma_2^1 \xrightarrow{p_2} ... \xrightarrow{p_i} \sigma_2^i \text{ et } \sigma_1' \leq_S \sigma_2^i$$

Cette définition signifie qu'un système S' est non bloquant pour un système S si quel que soit un état σ_1 qui se réduit par une transition ordonnanceur vers un état σ_1', quel que soit un état σ_2 qui est plus libre que σ_1 pour les configurations des processus de S, alors on peut exécuter à partir de σ_2 un certain nombre de processus de S' et de transitions ordonnanceur et de telle sorte qu'on arrivera à atteindre un état plus libre que σ_1' modulo S. Ceci formalise donc le fait que S' ne bloque pas S puisqu'en exécutant suffisamment souvent S', on peut atteindre les états accessibles depuis S par l'ordonnanceur quelles que soient les configurations des processus de S'. En particulier, cela permet d'exclure un système dont les processus sont en *livelock* alors que les processus d'un autre système attendent un avancement du temps.

Le non bloquage est une propriété indécidable : en effet, il faut analyser exhaustivement toutes les traces d'un assemblage pour s'assurer qu'un système n'en bloque pas un autre. En pratique, nous donnerons un critère statique plus général qui suffit à déduire le non bloquage dans de nombreux cas, sans être trop large pour autant (mais on ne pourra pas éviter les fausses alertes où le critère déduira d'un bloquage potentiel alors qu'il n'y en a pas).

3.4 Système Discriminant

En model checking, un automate observateur est un processus que l'on ajoute de façon particulière au système étudié. En effet, il ne doit pas influencer l'exécution du reste du système ; pour cela on impose généralement qu'un système observateur n'écrive pas sur les variables lues d'un système qu'il observe. Comme nous l'avons dit précédemment, nous utiliserons l'interprétation abstraite pour déduire un sur-ensemble des traces d'un système. Cependant, cette abstraction restera exacte sur les configurations du système. Ainsi, plus il y aura de processus dans un système, plus il y aura de configurations dans le système, et plus il y aura de possibilités de différenciation d'états dans l'abstraction. Ajouter des processus apportera de la précision lors d'une telle analyse par interprétation abstraite. Dans cette section, nous définissons une forme très forte d'automate observateur : les *systèmes discriminants*. Étant donné un système S, un système discriminant pour S ne doit pas changer les informations relatives à S dans toutes les traces de sa sémantique. Cela garantit en particulier que le système discriminant n'interviendra pas quant à la validité de S vis-à-vis de certaines propriétés. Le rôle d'un discriminant est uniquement d'ajouter ses configurations au système afin d'apporter de la précision aux analyses par interprétation abstraite. Typiquement, un discriminant doit pouvoir distinguer les instants cruciaux de l'exécution d'un système vis-à-vis d'une propriété qu'on cherche à montrer.

Un système est discriminant pour un autre s'il n'écrit pas les variables et évènements lus et écrits par l'autre. De plus, il ne doit pas non plus le bloquer, sinon il serait susceptible

de cacher des comportements potentiellement mauvais. Mais un système discriminant n'est pas pour autant totalement déconnecté du système qu'il discrimine : il peut lire la valeur de n'importe quelle variable et se mettre en attente de n'importe quel évènement. Et c'est précisément son utilité : il observe le comportement des autres processus sans influer dessus. De plus, un système discriminant ne doit pas pouvoir déplacer son contrôle dans une configuration d'erreur. Là n'est pas sa fonction de toutes façons.

Définition 3.16 Système Discriminant

$$S_{\mathcal{D}} \text{ discrimine } S \quad \triangleq \quad \left\{ \begin{array}{ll} & \forall t \in [\![S_{\mathcal{D}}]\!],\ \text{Safe}(S_{\mathcal{D}}, t) \\ \text{et} & W(S_{\mathcal{D}}) \cap (R(S) \cup W(S)) = \emptyset \\ \text{et} & S_{\mathcal{D}} \not\sim S \end{array} \right.$$

Un discriminant semble donc ne pas pouvoir intervenir sur le déroulement d'un système qu'il discrimine. Ainsi, il semblerait donc que l'ensemble des traces d'un système avec ou sans discriminant reste le même. En réalité, ceci n'est pas tout à fait vrai. En effet, l'ajout d'un discriminant contraint ses variables et évènements et la configuration de ses processus, même si ceux-ci n'influencent pas les possibilités d'évolution du système discriminé. Ce qui est plus correct, c'est de dire que si une trace est dans la sémantique d'un système S, alors une trace lui ressemblant exactement pour les variables, évènements et processus de S est dans la sémantique du même système avec un discriminant. En revanche, il est possible que des états supplémentaires soient ajoutés, lorsque le discriminant s'exécute. Pour montrer ce résultat fort, nous commençons par définir la notion de trace *étirée* vis-à-vis d'un système.

Étirement de Trace

Définition 3.17 Étirement $\leq_{.} \in \mathcal{S} \to \mathcal{T} \times \mathcal{T} \to \mathit{Prop}$

$$t \leq_S t' \quad \triangleq \quad \exists (u_n)_{n\in\mathbb{N}} \text{ croissante}, \left\{ \begin{array}{ll} & u_0 = 0 \\ \text{et} & \forall i, \left(\begin{array}{ll} & t_i \leq_S t'_{u_i} \\ \text{et} & \forall k,\ u_i \leq k < u_{i+1} - 1 \Rightarrow t'_{k+1} \leq_S t'_k \end{array} \right) \end{array} \right.$$

Cette définition signifie que si t' est un étirement de t modulo S, ce qui se note $t \leq_S t'$, c'est que dans t', on a pu insérer un certain nombre d'états supplémentaires entre ceux de t tels que les états sont de plus en plus libres sur S. Cette définition nous permet de montrer un lemme fondamental, qui énonce que si un système discrimine un autre, alors pour chaque trace du second, il y a une trace dans leur produit qui l'étire. L'idée derrière cette longue démonstration est d'exécuter S normalement, jusqu'à une transition ordonnanceur, et seulement alors d'exécuter les processus du discriminant. Comme il est discriminant, l'hypothèse de non bloquage garantit qu'on finira par retomber sur un état de S comme si on n'avait exécuté que l'ordonnanceur.

Lemme 3.14 $\forall S\ S',\ S' \text{ discrimine } S \Rightarrow \forall t \in [\![S]\!],\ \exists t' \in [\![S \otimes S']\!],\ t \leq_S t'$

Preuve : cette preuve est relativement longue. La démonstration complète se trouve en annexe A.1. Ici, nous donnons seulement les intuitions de la preuve par souci de clarté.

On part d'une trace $t \in [\![S]\!]$ et on construit une trace $t' \in [\![S \otimes S']\!]$ par récurrence et par cas. Dans le cas d'une transition Univers dans t, on garde pour t' les mêmes informations relatives à S entre les deux états, et on ne change pas celles locales à S' : c'est une transition Univers de $S \otimes S'$ qui n'a pas changé les informations relatives à S. Dans le cas d'une exécution d'un processus de S, ses variables et évènements n'ont pas changé par construction de t à t'. Quelle que soit la configuration du processus en question en t, il sera moins contraint ou le même en t' par hypothèse de récurrence. Donc d'après l'hypothèse 3.12, soit il a pu s'exécuter et produire des changements en t, et dans ce cas il pourra faire de même en t' (sans changer les locaux de S' par hypothèse de discrimination), soit il n'a rien fait et on peut représenter cela par une transition Univers identité. Dans le cas d'une transition ordonnanceur, on sait par l'hypothèse de non bloquage qu'on peut exécuter un certain nombre de processus de S' et l'ordonnanceur de telle sorte que l'on retombe sur une configuration moins contrainte pour les processus de S et dont les valeurs des variables et évènements est la même. De plus, l'exécution d'un processus de S' ne modifie pas la valeur des variables et évènements de S (hypothèse de discrimination), donc à plus forte raison elle n'a pas changé la valeur des locaux de S. Cette séquence d'exécutions de processus de S' et d'ordonnanceur est une séquence de transitions Univers de S et d'ordonnanceur. Et au final, l'état obtenu est moins contraint pour les configurations des processus de S par hypothèse de non bloquage. □

Propriété

Le résultat le plus important à propos des discriminants est le suivant, qui énonce que montrer une propriété d'un système avec discriminant suffit à montrer la propriété sans le discriminant.

Théorème 3.15 (Discriminant et Propriété)

$$\frac{S \otimes S_{\mathcal{D}} \models S' \qquad S_{\mathcal{D}} \text{ discrimine } S \otimes S'}{S \models S'}$$

Commentons ce théorème avant de le démontrer. La validité n'est pas naturellement compatible avec l'assemblage. En effet, rien ne prédit dans $S \otimes S_{\mathcal{D}}$ que $S_{\mathcal{D}}$ n'influence pas les exécutions de S de quelques manières que cela soit (écriture sur des variables ou bloquage). De plus, même si $S_{\mathcal{D}}$ n'intervient pas dans les exécutions de S, il peut réduire les exécutions de S'. On doit donc assurer que $S_{\mathcal{D}}$ n'intervient pas sur la vérification de la propriété S' par S, ce qui se fait en étudiant le produit $S \otimes S'$. Si $S_{\mathcal{D}}$ est effectivement discriminant de $S \otimes S'$, on peut utiliser le lemme 3.14 qui montre que les traces d'un discriminé sont effectivement contenues (sous une forme étirée) dans les traces du discriminant. Cela signifie que $S_{\mathcal{D}}$ ne vient pas perturber l'accessibilité des états, en laissant libre les variables et évènements du discriminé et en ne le bloquant pas. Pour montrer ce théorème, on commence par montrer un lemme et son corollaire qui vont assurer la manipulation sûre de traces étirées. Le lemme énonce simplement qu'étant donnés une trace, un étirement de cette trace sur un certain système et un processus de ce système, la sûreté des configurations du processus le long des traces est préservée. Le

corollaire applique simplement le lemme pour obtenir un résultat sur la sûreté et l'assemblage entre une trace et un étirement.

Lemme 3.16 $\forall S\ t\ t',\ t \leq_S t' \Rightarrow \forall p \in P_S,\ ((\forall i,\ \text{Safe}(t_i.C(p))) \Leftrightarrow (\forall i,\ \text{Safe}(t'_i.C(p))))$

Preuve : soient un système S, une trace t et une trace t' telles que $t \leq_S t'$ et un processus $p \in P_S$.
- On commence par montrer que si $\forall i,\ \text{Safe}(t_i.C(p))$, alors quel que soit i', $\text{Safe}(t'_{i'}.C(p))$. D'après la définition de $\leq_S$, on sait que $\forall i',\ \exists i,\ t_i \leq_S t'_{i'}$ car les états supplémentaires insérés dans t' sont en relation par $\leq_S$ avec leur précédent. Donc on sait qu'il existe i tel que $t_i \leq_S t'_{i'}$. Cela signifie donc que $t_i.C(p) \leq t'_{i'}.C(p)$ d'après la définition de $\leq_S$ sur les états et parce que $p \in P_S$. Si $t_i.C(p) = t'_{i'}.C(p)$, puisque $\text{Safe}(t_i.C(p))$ alors on a bien $\text{Safe}(t'_{i'}.C(p))$. Si $t_i.C(p) < t'_{i'}.C(p)$, on a $\text{Safe}(t_i.C(p)) \Leftrightarrow \text{Safe}(t'_{i'}.C(p))$ d'après l'hypothèse 3.11 sur la comparaison de configurations. Et puisque $\text{Safe}(t_i.C(p))$ alors on a bien $\text{Safe}(t'_{i'}.C(p))$.
- À présent, on montre que quel que soit i, $\text{Safe}(t_i.C(p))$ à partir du fait que $\forall i'$, $\text{Safe}(t'_{i'}.C(p))$. D'après la définition de $\leq_S$, on sait que $\forall i,\ \exists i',\ t_i \leq_S t'_{i'}$. On a donc $t_i.C(p) \leq t'_{i'}.C(p)$ d'après la définition de $\leq_S$ sur les états et parce que $p \in P_S$. Si $t_i.C(p) = t'_{i'}.C(p)$, puisque $\text{Safe}(t'_{i'}.C(p))$ alors on a bien $\text{Safe}(t_i.C(p))$. Si $t_i.C(p) < t'_{i'}.C(p)$, on a $\text{Safe}(t_i.C(p)) \Leftrightarrow \text{Safe}(t'_{i'}.C(p))$ d'après l'hypothèse 3.11 sur la comparaison de configurations. Et puisque $\text{Safe}(t'_{i'}.C(p))$ alors on a bien $\text{Safe}(t_i.C(p))$.

□

Corollaire 3.17

$$\forall S\ S'\ t\ t',\ t \leq_{S\otimes S'} t' \Rightarrow ((\text{Safe}(S,t) \Leftrightarrow \text{Safe}(S,t')) \text{ et } (\text{Safe}(S',t) \Leftrightarrow \text{Safe}(S',t')))$$

Preuve : on ne montre qu'une partie de la conjonction, par exemple $\mathit{Safe}(S,t) \Leftrightarrow \mathit{Safe}(S,t')$, l'autre partie se montrant de façon similaire. On montre les deux sens de l'implication.
- On suppose que $\mathit{Safe}(S,t)$. Étant donné un i quelconque, on doit montrer que $\mathit{Safe}(S,t'_i)$, c'est-à-dire que quel que soit $p \in P_S$, $\mathit{Safe}(S,t'_i.C(p))$. Puisque $\mathit{Safe}(S,t)$, c'est que $\forall i,\ \mathit{Safe}(S,t_i)$. En particulier, on sait donc que $\forall i,\ \mathit{Safe}(S,t_i.C(p))$. Comme $p \in P_S$, alors $p \in P_S \cup P_{S'} = P_{S\otimes S'}$. Donc on peut appliquer le lemme 3.16 sur p et avec $t \leq_S t'$ et $\forall i,\ \mathit{Safe}(S,t_i.C(p))$ pour déduire que $\mathit{Safe}(S,t'_i)$.
- On suppose que $\mathit{Safe}(S,t')$. Étant donné un i quelconque, on doit montrer que $\mathit{Safe}(S,t_i)$, c'est-à-dire que quel que soit $p \in P_S$, $\mathit{Safe}(S,t_i.C(p))$. Puisque $\mathit{Safe}(S,t')$, c'est que $\forall i,\ \mathit{Safe}(S,t'_i)$. En particulier, on sait donc que $\forall i,\ \mathit{Safe}(S,t'_i.C(p))$. Comme $p \in P_S$, alors $p \in P_S \cup P_{S'} = P_{S\otimes S'}$. Donc on peut appliquer le lemme 3.16 sur p et avec $t \leq_S t'$ et $\forall i,\ \mathit{Safe}(S,t'_i.C(p))$ pour déduire que $\mathit{Safe}(S,t_i)$.

□

Preuve du théorème de préservation des propriétés par discrimination.

Nos hypothèses nous disent que $S \otimes S_{\mathcal{D}} \models S'$, c'est-à-dire que $\forall t \in [\![S \otimes S_{\mathcal{D}} \otimes S']\!],\ \mathit{Safe}(S \otimes S_{\mathcal{D}},t) \Rightarrow \mathit{Safe}(S',t)$, et aussi que $S_{\mathcal{D}}$ discrimine $S \otimes S'$, c'est-à-dire entre autres que $\forall t \in [\![S_{\mathcal{D}}]\!],\ \mathit{Safe}(S_{\mathcal{D}},t)$. On doit montrer qu'étant donnée une trace $t \in [\![S \otimes S']\!]$ telle que $\mathit{Safe}(S,t)$, on a bien $\mathit{Safe}(S',t)$.

Puisque $S_{\mathcal{D}}$ discrimine $S \otimes S'$ et que $t \in [\![S \otimes S']\!]$, on sait d'après le lemme 3.14 qu'il existe une trace $t' \in [\![S \otimes S_{\mathcal{D}} \otimes S']\!]$ telle que $t \leq_{S\otimes S'} t'$. Puisque $t' \in [\![S \otimes S_{\mathcal{D}} \otimes S']\!]$, on sait par le théorème

2.3 que $t' \in S_{\mathcal{D}}$, et donc par le fait que $S_{\mathcal{D}}$ est discriminant, que $\mathit{Safe}(S_{\mathcal{D}}, t')$. De plus, puisque $t \leq_{S \otimes S'} t'$ et que $\mathit{Safe}(S, t)$ par hypothèse, le corollaire 3.17 permet de déduire que $\mathit{Safe}(S, t')$. Par le lemme 3.1 sur $\mathit{Safe}(S, t')$ et $\mathit{Safe}(S_{\mathcal{D}}, t')$, on déduit que $\mathit{Safe}(S \otimes S_{\mathcal{D}}, t')$. On peut alors utiliser l'hypothèse que $S \otimes S_{\mathcal{D}} \models S'$ pour obtenir une preuve du fait que $\mathit{Safe}(S', t')$. D'après cette dernière proposition et puisque $t \leq_{S \otimes S'} t'$, on peut réutiliser le corollaire 3.17 pour conclure que $\mathit{Safe}(S', t)$. □

Remplacement et Discrimination

Pour finir sur l'étude des systèmes discriminants, nous revenons sur le remplacement et nous montrons comment discrimination et remplacement se combinent dans notre méthodologie de vérification. Les discriminants ne servent pas que lors de la vérification de propriétés. Ils peuvent servir de la même manière lors de la vérification d'un remplacement. Généralement, la méthodologie sera la suivante : on souhaite montrer une propriété S sur un système $S_1 \otimes S_2$. Pour réduire la complexité du problème, on remplace S_1 par un système S'_1, et afin de regagner en précision par rapport aux approximations introduites par l'interprétation abstraite pour cette vérification, on ajoute un système discriminant $S_{\mathcal{D}}$. Enfin, on montre $S'_1 \otimes S_2 \models S$ pour conclure que $S_1 \otimes S_2 \models S$. Cette méthodologie est simplement résumée par l'arbre suivant.

$$\dfrac{S_{\mathcal{D}} \text{ discrimine } S_1 \otimes S_2 \otimes S \qquad \dfrac{S_1 \otimes S_{\mathcal{D}} \rhd_{I/\Rightarrow} S'_1 \qquad S'_1 \otimes S_2 \models S}{S_1 \otimes S_{\mathcal{D}} \otimes S_2 \models S} \text{ Si } R(S_2) \cap I = R(S) \cap I = \emptyset}{S_1 \otimes S_2 \models S}$$

Pour cette dérivation, nous avons utilisé le théorème 3.6 de remplacement d'hypothèse et le théorème 3.15 de préservation des propriétés par discrimination.

Dans la section suivante, nous introduisons la façon de nous vérifions des systèmes, une technologie particulière du fait des systèmes discriminants.

3.5 Model Checking et Interprétation Abstraite

Pour vérifier qu'un système S satisfait une certaine propriété, on peut utiliser les techniques de remplacement, de discrimination et de validité. Chacune de ces techniques reposent sur une exploration des comportements d'un système, ce qui n'est souvent pas décidable.

Vérification statique. Même lorsqu'une propriété d'un système est décidable, il arrive souvent que le nombre d'états du système explose et des techniques de couverture comme le model checking ne sont plus utilisables. En effet, si le model checking a été utilisé avec succès sur des systèmes de petites et moyennes tailles, le passage à des systèmes de grande taille est encore difficile aujourd'hui. De plus, les systèmes que nous étudions sont très haut niveau (TLM [Ghe06]) et les langages auxquels nous nous intéressons autorisent à créer des *boucles*

sur des types de données très grands (par exemple les `integer`) dont la contrepartie théorique (par exemple $\mathbb{Z}$ dans le cas de `integer`) n'assure pas toujours la finitude du nombre d'états des systèmes, si bien qu'on ne peut pas toujours calculer statiquement l'ensemble précis des comportements de l'un de ses programmes. L'interprétation abstraite apporte les outils pour calculer un sur-ensemble des comportements de façon statique. On peut abstraire certaines informations, comme oublier toutes celles qui sont externes au système étudié. Pour les variables qui prennent un nombre indéterminé de valeurs au cours d'une exécution, l'interprétation abstraite offre un cadre qui assure que l'on peut calculer un sur-ensemble de ces valeurs de façon statique. Si au final le sur-ensemble des comportements de $S \otimes S'$ ne contient pas de configuration erreur, on peut déduire logiquement que $S \otimes S'$ n'en contient pas non plus.

Une idée appliquée dans la thèse est de combiner la précision des techniques de model checking et les approximations apportées par les techniques d'interprétation abstraite pour obtenir un sur-ensemble pertinent des comportements d'un système de façon statique. Le model checking sera utilisé pour calculer un *graphe d'états* du système uniquement basé sur ses configurations (une configuration pour chaque processus et l'ensemble des évènements notifiés). L'interprétation abstraite sera utilisée pour décorer ce graphe en associant à chaque configuration du système un environnement abstrait qui accumule les valeurs que chaque variable peut prendre en cette configuration. Le *graphe d'états abstraits* obtenus est une sorte de structure de Kripke où les propositions sont les environnements abstraits.

Vérification interactive. Un phénomène intéressant se dégage de l'utilisation conjointe du model checking et de l'interprétation abstraite. Les algorithmes d'interprétation abstraite travaillent par sur-approximations et différents états seront amenés à fusionner lorsque leur configuration sur le système étudié est la même. D'un autre côté, un discriminant dans un système apporte de la précision en ajoutant ses propres configurations à celles du reste du système. Des états qui seraient fusionnés sans la présence du discriminant vont être différenciés avec la présence de ce dernier. On obtient alors un cadre dans lequel model checking et interprétation abstraite jouent un rôle complémentaire : l'interprétation abstraite permet de perdre de la précision alors que les discriminants par le model checking vont en apporter. Les discriminants pilotent donc les analyses en spécifiant des points de discrimination supplémentaires. De plus, ils sont introduits par l'utilisateur, ce qui permet une mise au point interactive des états à fusionner.

Étude de cas

Nous illustrons ce phénomène sur un cas réduit du système des pompes du chapitre 1 comme décrit dans l'un de nos articles [ACV08b]. Cet exemple étudie une partie du système composé d'un réservoir, d'un environnement qui remplit le réservoir selon un débit constant d'1 volume par unité de temps, d'une pompe qui vide le réservoir et d'un contrôleur, connecté au réservoir et à la pompe, qui active ou désactive cette pompe selon que le volume dépasse un certain niveau ou descend en-dessous d'un autre. Lorsque la pompe est activée, elle vide le réservoir à un débit de 5 volumes par unité de temps vers un environnement extérieur. Le système est représenté figure 3.2.

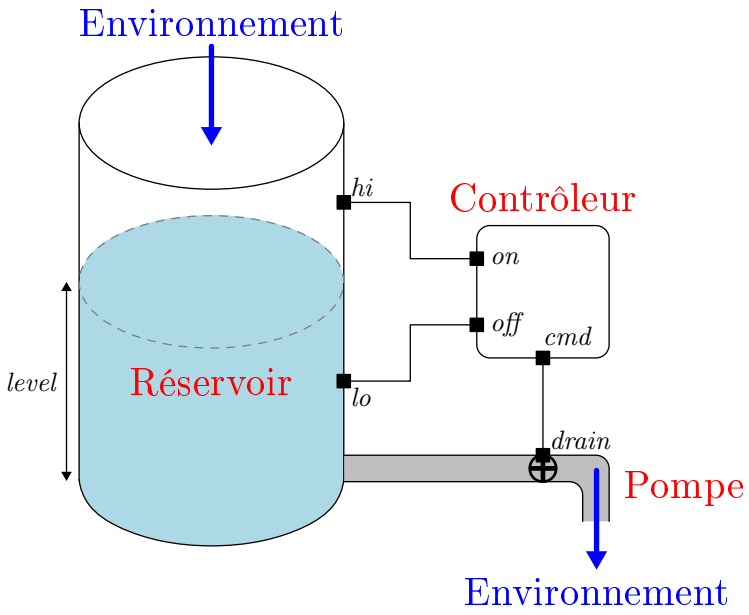

FIG. 3.2 – Un réservoir, un contrôleur et une pompe

On suppose que le système est initialisé avec un volume nul de liquide dans le réservoir et on souhaite étudier l'évolution du niveau du liquide. Les expérimentations conduisent à la conjecture suivante : l'évolution du système se découpe en trois phases. La première est une phase d'initialisation *Init* pendant laquelle le niveau monte jusqu'à atteindre le capteur bas. Lors de la seconde phase, appelée *Fill*, le niveau de liquide monte jusqu'à atteindre le capteur haut. Enfin, pendant la dernière phase appelée *Drain*, le contrôleur active la pompe ce qui provoque une vidange du réservoir car le débit auquel du liquide s'ajoute au réservoir est plus petit que celui auquel il lui en est retiré. Cette phase dure jusqu'à ce que le niveau descende en-dessous du capteur bas et que le contrôleur désactive la pompe de vidange. On repasse alors en phase de remplissage *Fill*. Les phases *Fill* et *Drain* se succèdent ainsi *ad vitam eternam*. Ces observations sont présentées figure 3.3.

Sans rentrer dans les détails, nous montrons de façon intuitive les résultats obtenus sur ce système si on utilise le model checking pour en couvrir l'espace des configurations et l'interprétation abstraite pour représenter le niveau de liquide dans le réservoir avec des intervalles (un domaine simple qui permet des calculs rapides). Le problème suivant apparaît : les configurations des phases *Init* et *Fill* sont les mêmes et les confondre conduit les valeurs du niveau à être fusionnées. On ne différencie plus que deux phases qui alternent, ce qui est une propriété intéressante, mais on perd le fait que le niveau reste plus haut qu'une borne proche du capteur bas après la phase d'initialisation (voir figure 3.4).

Cette imprécision introduite par l'interprétation abstraite sur le niveau de liquide peut être précisée avec un système discriminant et le model checking. Il suffit pour cela de marquer le moment où l'on souhaite différencier le comportement du système. En l'occurrence, dans cet exemple, on souhaite différencier la phase *Init* de la phase *Fill*. On passe de l'une à l'autre exactement au moment où le niveau dépasse le capteur bas. Il suffit donc d'ajouter au système un processus qui observe simplement le passage de cet évènement. Les résultats obtenus alors

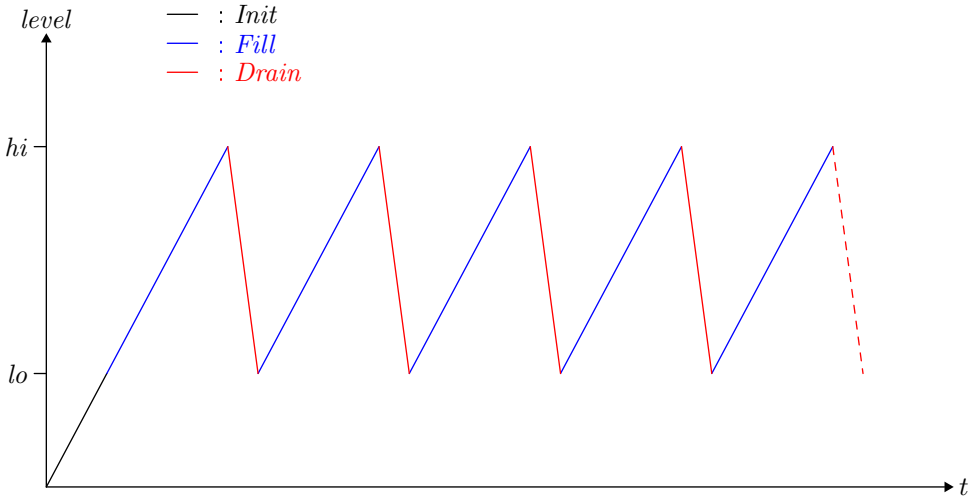

FIG. 3.3 – Observation de l'évolution du système

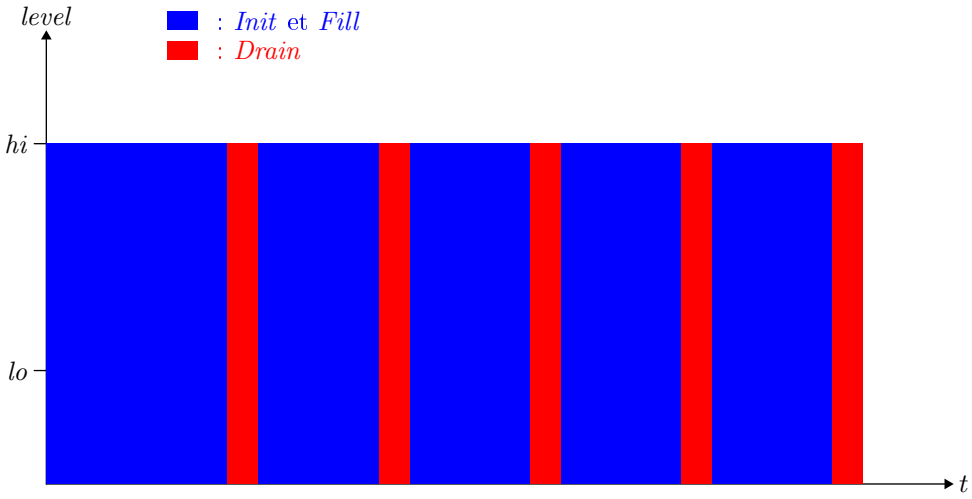

FIG. 3.4 – Résultats avec model checking et interprétation abstraite

sont donnés figure 3.5. On y voit clairement la distinction entre les trois phases *Init*, *Drain* et *Fill*.

Une propriété de ce système vérifiée par notre technologie serait informellement « après une phase d'initialisation, le niveau de liquide reste indéfiniment entre une borne inférieure proche du capteur bas et une borne supérieure proche du capteur haut ». Une telle propriété contient des aspects fonctionnels et temporels. La question se pose donc de connaître l'expressivité des systèmes comme propriétés en comparaison avec les logiques temporelles. Ceci dépend bien sûr du langage de description de système que l'on utilise (puisque les propriétés sont des systèmes), mais du fait de l'utilisation de l'interprétation abstraite, nous étudierons particulièrement les propriétés d'invariance et de sûreté, qui énonce que « tous les états vérifient une bonne propriété » ou encore que « quelque chose de mal n'arrivera pas ».

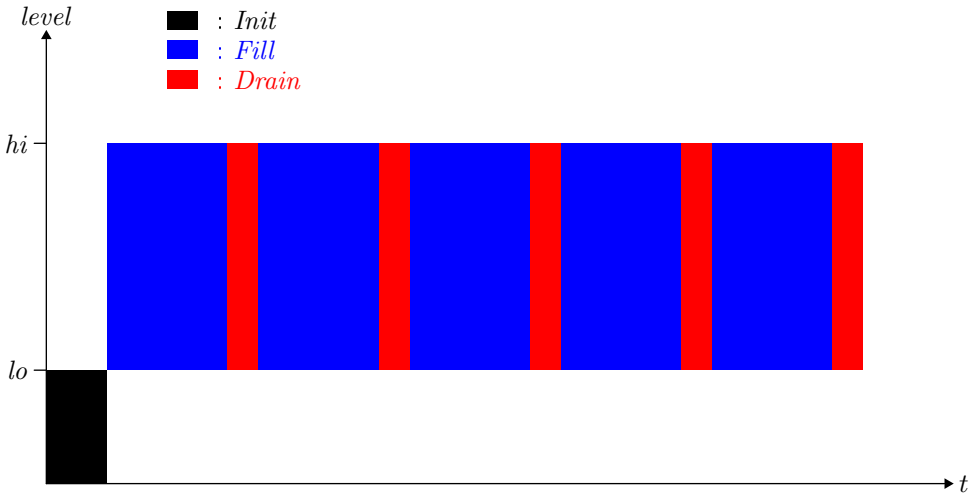

FIG. 3.5 – Résultats avec ajout d'un discriminant

Le chapitre 4 présente SYSTEMD, un langage de description haut niveau à la SYSTEMC qui permet de représenter des systèmes de composants et qui implante les notions de sûreté des configurations, de systèmes discriminants et de remplacement comme définit dans ce chapitre. Ce langage haut niveau se compile dans un langage noyau, KERNELD, qui suffit à représenter de façon simple les systèmes et les processus. KERNELD est décrit dans le chapitre 5 alors que la compilation de SYSTEMD en KERNELD est présentée dans ce même chapitre, en section 5.6. La section 5.7 compare l'expressivité des systèmes SYSTEMD/KERNELD comme propriété avec les logiques temporelles. Le chapitre 6 présente une technique générale qui combine des éléments de model checking et d'interprétation abstraite et qui sera utilisée pour vérifier des programmes KERNELD, puis une instanciation de cette technique avec une analyse avant (et arrière sur les gardes) en 6.4 qui permet de calculer de façon statique un sur-ensemble relativement précis des comportements d'un système décrit en KERNELD. Le chapitre 7 montre comment vérifier qu'un système en valide un autre, qu'un système en remplace un autre ou qu'un système discrimine un autre en s'appuyant notamment sur l'analyse décrite au chapitre 6. Le chapitre 8 qui suit présente les résultats obtenus par notre méthodologie sur l'exemple de la pompe du chapitre 1 en SYSTEMD. Enfin, le chapitre IV présente des travaux pour améliorer les analyses et conclut.

Deuxième partie

SystemD

CHAPITRE 4

Le Langage

SystemD est un langage de descriptions de systèmes de composants. Il permet également de définir des propriétés sur les systèmes et de les vérifier dans un cadre compositionnel par remplacement d'hypothèses (voir le chapitre 3). Les propriétés sont elles-mêmes décrites par des systèmes si bien que le travail de spécification reste très proche du travail de description.

Nous introduisons le langage en reprenant l'exemple des pompes (cf 1.1.2). Nous montrons comment décrire le système complet avec notre la méthodologie de vérification. Ensuite, nous présentons l'ensemble des constructions du langage en les comparant avec celles de SystemC, avant d'en donner une sémantique informelle. Cette sémantique sera définie formellement en passant par la définition de KernelD, le noyau du langage, au chapitre 5 et par la traduction d'un développement SystemD vers KernelD à la section 5.6. KernelD permettra également de discuter en section 5.7 de l'expressivité des propriétés exprimables en SystemD par des systèmes.

4.1 Le Système des Pompes

En SystemD, un système est appelé *module*. Chaque module permet de définir un certain nombre de variables, d'évènements et de processus les manipulant et représentant le comportement du système. On peut déclarer des variables et évènements locaux au module : ceux-ci ne peuvent pas être écrits par d'autres modules mais peuvent être lus, ce qui diffère un peu de la notion classique de localité dans les langages de programmation. Les modules sont également utilisés pour définir quelles sont les variables communes avec les autres modules et donner un environnement initial d'exécution du système.

Le système des pompes s'appuie sur une *interface* et trois composants de base :

- l'interface `Env` représente l'ensemble des modules qui peuvent être connectés en entrée et en sortie au système tout entier. Un `Env` doit pouvoir fournir du liquide à une pompe, et doit pouvoir en recevoir également ;
- un réservoir (module `Tank` qui implante l'interface `Env`) contient un liquide jusqu'à un certain niveau représenté par le champ `level`. Deux capteurs permettent de signaler si le niveau est devenu trop haut (`hi`) ou trop bas (`lo`). Un réservoir est un environnement ;
- un contrôleur (module `Controller`) possède une valeur constante (`value`), une commande de sortie (`cmd`) et deux capteurs (`on` et `off`) qui contrôle la valeur (`value` lorsque le capteur `on` est excité, 0 lorsque c'est `off`) véhiculée sur la sortie `cmd` ;
- une pompe possède un environnement depuis lequel il puise du liquide (`in`), un environnement dans lequel il déverse du liquide (`out`) et une valeur de vidange (`drain`) commandée par un module externe. Son comportement consiste simplement à vider l'environnement `in` dans l'environnement `out` de la quantité `drain`.

La figure 4.1 donne le code SystemD de chacun de ces composants.

Pour construire le système des pompes tout entier (dont un schéma se trouve en figure 4.2), on divise le système en deux parties : une partie haute et une partie basse qui ont pour élément commun une pompe (voir la figure 4.3).

La définition des deux parties présente des similarités : deux pompes sont mises en série avec un réservoir commun. Aux capteurs du réservoir commun sont attachés des capteurs d'un contrôleur qui lui-même envoie sa commande à la seconde pompe. Nous factorisons cette description dans un seul et même module : le module `Pump_Serie` dont le code se trouve en figure 4.4.

La partie haute du système est représentée par un module nommé `Top_Part`. Il contient une instance du module `Pump_Serie`. On fournit un stimulus à la première pompe de cette instance ; sa vitesse de vidange peut se trouver à tout moment comprise entre 0 et 5. On déclare la vitesse de vidange de la seconde pompe à 5 lorsque celle-ci est active. Le code du module `Top_Part` ainsi défini se trouve en figure 4.5.

Cette figure contient également la description de la partie basse du système, représentée par le module `Bottom_Part`. C'est tout simplement deux pompes en série avec réservoir et contrôleur ; la seconde pompe a une vitesse de 10.

Le système des pompes consiste en une partie haute et une partie basse qui ont une pompe commune. Le code du système résultant se trouve en figure 4.6.

Le comportement global du système qu'on cherche à représenter est le suivant : une première pompe remplit le premier réservoir d'une vitesse comprise entre 0 et 5. Une fois que la quantité de ce réservoir atteint une certaine hauteur, le capteur `hi` de celui-ci se déclenche, envoyant un message au capteur `on` du contrôleur correspondant. Le contrôleur active alors une pompe qui vide le premier réservoir à une vitesse de 5 volumes de liquide par unité de temps. Cette pompe s'arrête lorsque le niveau dans le réservoir redescend en-dessous d'un seuil critique, ce qui déclenche le capteur `lo` du réservoir qui envoie un signal au capteur `off` du contrôleur. Le même procédé s'applique au réservoir dans lequel le premier se vide. Lui aussi dispose d'un contrôleur et d'une pompe qui le vide à une vitesse de 10 volumes de liquide par unité de temps. On souhaite montrer qu'après avoir atteint le niveau bas du réservoir la quantité de

Environnement

```
interface Env {
  method add(int);
}
```

Réservoir

```
module Tank : Env {

  int level;
  port_out<bool> hi, lo;

  init() { level = 0; }

  adjust(int amount) : int {
    if (level + amount >= 0) return(amount);
    return(-level);
  }

  add(int amount) {
    level = level + adjust(amount);
    hi.write(level > 900);
    lo.write(level <= 300);
  }

}
```

Contrôleur

```
module Controller {

  port<bool> on, off;
  int value;
  port<int> cmd;

  local: value;

  thread {
    wait(on | off);
    if on.read()  cmd.write(value);
    if off.read() cmd.write(0);
  }
}
```

Pompe

```
module Pump {

  Env in, out;
  port<int> drain;

  thread {
    wait(1s);
    in.add(-drain.read());
    out.add(drain.read());
  }

}
```

FIG. 4.1 – Code SYSTEMD des réservoir, contrôleur et pompe

liquide dans chaque réservoir reste supérieure à une borne proche du niveau bas et inférieure à une borne proche du niveau haut. Le *delta* entre le niveau bas et la borne proche du niveau bas et le *delta* entre le niveau haut et la borne proche du niveau haut viennent de ce que le langage SYSTEMD comprend des constructions synchrones, ce qui induit parfois un délai entre l'écriture d'une valeur et la visibilité de cette écriture pour les composants.

Pour cela, on construit un nouveau module qui contient une instance du système des pompes et auquel on ajoute un programme qui provoque une erreur à l'exécution lorsque la propriété

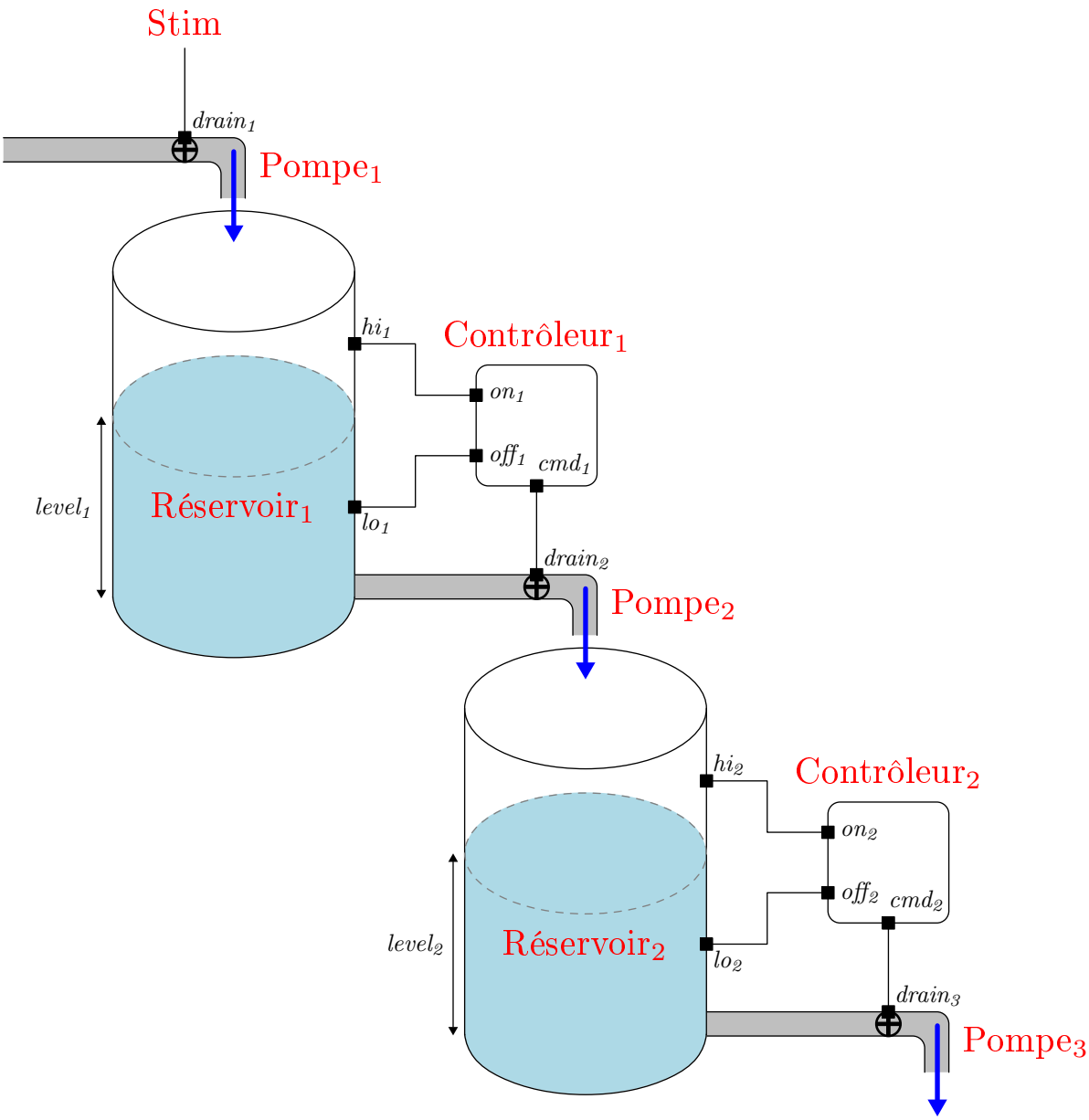

FIG. 4.2 – Le système des pompes

— le fait que les niveaux restent entre deux bornes — est violée (vérification à base d'assertion [FLK03, MMMC05]). Le module `Sys_Prop` obtenu est décrit en figure 4.7 où la propriété a été décomposée en deux : une propriété pour le premier réservoir et une pour le second. De plus, nous ajoutons deux programmes *discriminants* (un par réservoir) qui par leur présence seule apporteront de la précision lors des analyses en explicitant les instants critiques de l'exécution vis-à-vis de la propriété.

En réalité, ce n'est pas exactement ce module que nous utilisons pour la preuve de la propriété ; ou plutôt nous y ajoutons une clause. Nous allons remplacer une partie du système

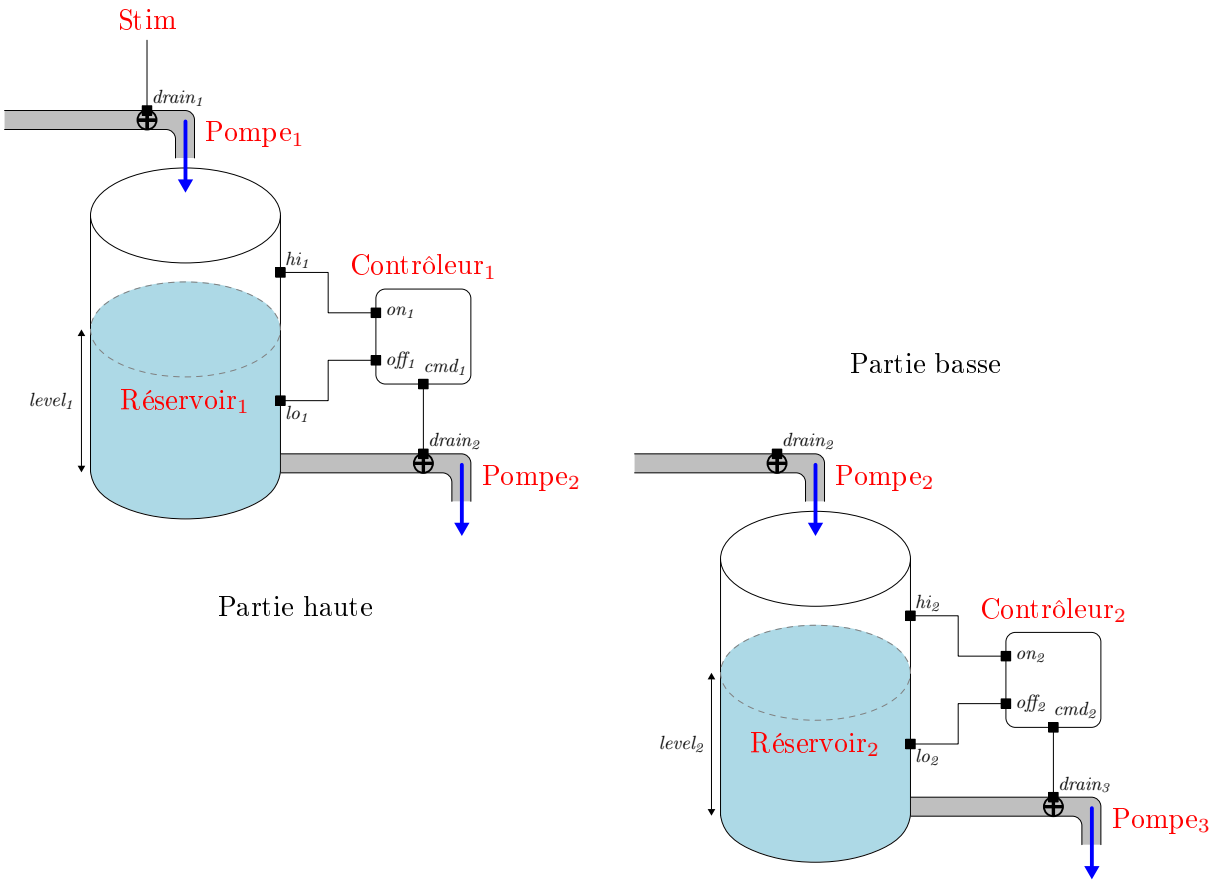

FIG. 4.3 – Découpage du système des pompes

par un autre, plus simple d'un point de vue combinatoire. En effet, la vérification va nécessité d'assembler les parties haute et basse du système, produisant un effet exponentiel sur le nombre d'états du système tout entier. Pour contourner cette explosion et accélérer la vérification, nous proposons par exemple de remplacer la partie haute du système par un module qui fait état des éléments qui ont une influence en dehors de cette partie. On souhaite conserver une approximation du niveau de liquide dans le réservoir, ainsi qu'une approximation de la vitesse de vidange de la pompe au cours du temps. On énonce par là que ces informations suffisent à prouver la propriété qui nous intéresse sur le système des pompes, et que les simplifications effectuées faciliteront sa preuve. Le code du remplacement, ou *abstraction*, est donné en figure 4.8. Sa correction se place dans une logique d'assume-guarantee [HQR98, WL05].

Le module `Sys_Prop` dans lequel on remplace la première partie du système des pompes se trouve en figure 4.9. C'est sur ce module que nous illustrerons les particularités du langage SYSTEMD.

On fait appel au moteur de simulation en déclarant le module à simuler dans une structure qui a pour nom `main_simulation`.

```
main_simulation { SysProp P ; }
```

```
module Pump_Serie {

  Pump p1, p2;
  Controller c;
  Tank common;

  local: p2.drain, c, common;

  Pump_Serie {
    common.init();
    p1.out = common;
    p2.in = common;
    connect<int>(c.cmd, p2.drain);
    connect<bool>(c.on, common.hi);
    connect<bool>(c.off, common.lo);
    c.cmd.old = 0;
    c.on.old = false;
    c.off.old = true;
  }

}
```

FIG. 4.4 – Deux pompes en série avec réservoir et contrôleur

```
module Stim {

  port<int> cmd;

  thread {
    wait(1s);
    cmd.write(rand(0, 5));
  }

}
```

```
module Top_Part {

  Stim stim;
  Pump_Serie ps;

  local: stim, ps.p1;

  Top_Part {
    stim.cmd.old = rand(0, 5);
    p1.drain = stim.cmd;
    c.value = 5;
  }

}
```

```
module Bottom_Part {

  Pump_Serie ps;

  Bottom_Part { c.value = 10; }

}
```

FIG. 4.5 – Code des parties haute et basse du système des pompes

Cette invocation construit le système `P` et l'exécute. Son évolution peut présenter des choix

```
module Pump_System {

  Top_Part tp;
  Bottom_Part bp;

  local: tp.ps.p2;

  Pump_System {
    tp.ps.p2 = bp.ps.p1;
  }

}
```

FIG. 4.6 – Le code du système des pompes en SYSTEMD

indéterministes et la simulation ne donne donc qu'une exécution possible parmi toutes celles que représente le système `P`. En particulier, si des variables sont libres dans le système, la simulation leur donnera des valeurs arbitraires. Lorsqu'un `assert` est violé, la simulation s'interrompt brutalement avec une erreur. Notons également que la simulation ne prend pas en compte les remplacements.

On peut également choisir non pas de simuler, mais de vérifier un système. Il suffit pour cela d'appeler le moteur de vérification.

```
main_analysis { SysProp P ; }
```

Ceci vérifie que les remplacements sont corrects et que quelle que soit une exécution de `P`, une erreur provoquée par une violation d'un `assert` dans une propriété n'arrive jamais.

La sémantique de simulation est définie en section 5.4, et s'appuie sur un langage noyau pour SYSTEMD : KERNELD. Nous présentons la compilation de SYSTEMD en KERNELD dans la section 5.6. Les vérifications par analyses statiques sont présentées en partie III et se basent sur des techniques de model checking et d'interprétation abstraite.

Le reste de ce chapitre consiste à présenter la syntaxe et la grammaire de SYSTEMD, ainsi qu'une sémantique informelle qui facilitera la présentation de KERNELD, de sa sémantique et de son analyse.

4.2 Syntaxe et Grammaire

Le lecteur intéressé pourra trouver une grammaire *in extenso* du langage en annexe A.2. Cette section se contente d'en présenter les grandes lignes qui suffiront à comprendre la philosophie du langage.

SYSTEMD est un langage impératif typé (mais sans objet), avec une notion de polymorphisme par *templates*, dont la syntaxe est proche des classiques du genre impératif

```
module Sys_Prop {

  Pump_System PS;

  property {
    wait(PS.tp.ps.common.lo);
    while (true) {
      assert(290 <= PS.tp.ps.common.level &&
             PS.tp.ps.common.level <= 910);
      wait(1s);
    }
  }

  discriminate {
    wait(PS.tp.ps.common.lo);
    while (true) {
      wait(PS.tp.ps.common.hi);
      wait(PS.tp.ps.common.lo);
    }
  }

  property {
    wait(PS.bp.ps.common.lo);
    while (true) {
      assert(290 <= PS.bp.ps.common.level &&
             PS.bp.common.level <= 910);
      wait(1s);
    }
  }

  discriminate {
    wait(PS.bp.ps.common.lo);
    while (true) {
      wait(PS.bp.ps.common.hi);
      wait(PS.bp.ps.common.lo);
    }
  }

}
```

FIG. 4.7 – Une propriété du système des pompes

(C/C++/SystemC, JAVA, etc). Il contient cependant des constructions matérielles primitives ainsi que des constructions particulières à sa méthodologie de vérification (propriété, discriminant, abstraction).

Un développement SystemD consiste en une succession de définition, puis en l'invocation du moteur de simulation ou du moteur d'analyse. Les définitions peuvent être de trois natures :

Structures (mot-clef `struct`). Ce sont des types de données complexes, constitués de champs et de méthodes pour les manipuler.

```
abstraction Abs_Top_Part(Top_Part tp) {

  int level, out_drain;
  port<bool> lo;
  Env out;

  local: level, out_drain, lo;

  link: level = tp.ps.common.level and
        out_drain = tp.ps.p2.drain.old and
        lo = tp.ps.common.lo and
        out = tp.ps.p2.out;

  Abs_Top_Part {
    level = rand(infty, infty);
    out_drain = 0;
    lo = true;
  }

  thread {
    while (lo) {
      lo = rand();
      level = rand(infty, infty);
      out_drain = 0;
      wait(1s);
    }
    while (true) {
      level = rand(295, 905);
      out_drain = rand(0, 5);
      out.add(out_drain);
      wait(1s);
    }
  }

  discriminate {
    wait(tp.ps.common.lo)
    while (true) {
      wait(tp.ps.common.hi);
      wait(tp.ps.common.lo);
    }
  }

}
```

FIG. 4.8 – Remplacement de la partie haute du système

```
module Sys_Prop {

  Pump_System PS with Abs_Top_Part(ps.tp);

  property {
    wait(PS.tp.ps.common.lo);
    while (true) {
      assert(290 <= PS.tp.ps.common.level &&
             PS.tp.ps.common.level <= 910);
      wait(1s);
    }
  }

  property {
    wait(PS.bp.ps.common.lo);
    while (true) {
      assert(290 <= PS.bp.ps.common.level &&
             PS.bp.common.level <= 910);
      wait(1s);
    }
  }

  discriminate {
    wait(bp.ps.common.lo)
    while (true) {
      wait(bp.ps.common.hi);
      wait(bp.ps.common.lo);
    }
  }

}
```

FIG. 4.9 – Le système des pompes avec un remplacement et une propriété

Module (mot-clef `module`). Ils représentent les systèmes. Ils sont constitués de champs dont les types peuvent être structurés, mais qui peuvent également être propres aux matériels (ce qui n'est pas le cas des structures) : évènements (mot-clef `event`), sous-systèmes (c'est-à-dire d'autres modules), ports (mot-clef `port`, bien que ce type ne soit pas primitif comme nous l'expliciterons lors de la description informelle de la sémantique du langage). Un module peut déclarer local à sa définition — c'est-à-dire réserver son accès en écriture — certains de ses champs. Ils peuvent contenir des définitions de méthodes, mais aussi et surtout de processus (mot-clef `thread`). L'assemblage de deux modules se réalise en les déclarant comme champs d'un troisième module.

Remplacement (mot-clef `abstraction`). Un remplacement prend en argument un module à remplacer et définit le module qui le remplacera. Le corps contient également une construction qui permet d'enregistrer les variables communes entre le remplaçant et le remplacé (mot-clef `link`). Le chapitre 3 introduisait un cadre théorique pour la vérification compositionnelle de systèmes par le remplacement d'hypothèses ou de conclusions. SYSTEMD est une instance de ce cadre où seul le remplacement d'hypothèses est utilisable. Ceci parce que le plus souvent, les conclusions sont des systèmes relativement simples.

En plus de ces trois types de définitions, un développement SystemD peut également définir des interfaces (mot-clef `interface`) et des marcos ; ce sont des commodités de génie logicielle. Une interface définit le prototype des méthodes que doit implanter une structure qui la respecte. Les macros sont des programmes qui permettent de factoriser du code.

Instructions des Programmes. Les méthodes, processus et macros sont des programmes dont les grammaires n'autorisent pas le même type d'instructions. Cependant, ces trois famille de programmes peuvent toutes faire appel aux instructions de programmation impérative classique : déclaration de variables de type structuré, affectation, test, boucle, appel de méthode ou de macro. Les méthodes ont également le droit d'invoquer une instruction de retour de valeur (mot-clef `return`), ce qui n'est pas le cas des processus, qui eux ont cependant le droit de faire appel à des instructions de notification d'évènements (mot-clef `notify`), de mise en attente d'évènements (mot-clef `wait`) et d'assertions (mot-clef `assert`), ce que ne peuvent pas les méthodes. Les macros ont droit à toutes les instructions. Enfin, les propriétés (mot-clef `property`) et les discriminants (mot-clef `discriminate`) sont des processus et peuvent donc s'utiliser à la même place que ces derniers et avec les mêmes instructions.

Types de Base et Expressions. Les types de base sont les entiers, les booléens et les tableaux pour ce qui est des structures classiques de programmation, et les évènements pour le côté matériel. Quant aux expressions, elles sont formées d'expressions arithmétiques et logiques, mais aussi de combinaisons d'évènements pour les statuts d'attente des processus. Un processus peut se mettre en attente qu'un évènement soit notifié, que plusieurs combinaisons d'évènements soient notifiées (`&`), que des combinaisons d'évènements ou d'autres soient notifiées (`|`) ou que des combinaisons d'évènements ne soient pas notifiées (`not`). Des évènements particuliers sont ceux liés au temps, qui est représenté par un entier et une unité (nanoseconde, milliseconde ou seconde).

Maintenant que nous avons présenté la syntaxe et la grammaire du langage SystemD, nous présentons sa sémantique de façon informelle.

4.3 Sémantique

Nous présentons une sémantique informelle du langage afin d'aider la compréhension de la sémantique du noyau du langage et la traduction vers ce noyau. De plus, nous présentons les différences fondamentales du langage avec SystemC.

Chaque module SystemD va représenter un système, donc des variables et des évènements, parmi eux des locaux, et des processus. Une fois le système créé, le moteur de simulation donne une exécution du système selon la sémantique des systèmes au chapitre 2. Il nous restera donc à expliciter la sémantique des instructions des threads SystemD. Le moteur de vérification quant à lui consiste en un algorithme qui calcule un sur-ensemble de toutes les exécutions possibles du système et à s'assurer qu'aucune d'entre elles ne passe par une instruction `assert` dont l'expression pourrait s'évaluer `false`.

4.3.1 Phases d'Exécution

La compilation d'un module donne deux types de processus : les processus comportementaux et les processus de mises à jour. L'exécution d'un module se fait en deux phases distinctes : l'exécution des processus comportementaux puis, lorsque tous les processus comportementaux sont endormis, l'exécution des processus de mises à jour. Lorsque les processus de mises à jour sont à leur tour endormis, le temps de simulation avance si besoin est, puis on reprend la phase comportementale et ainsi de suite.

L'exécution d'un module SystemD se fait sur un environnement qui contient plusieurs informations :
- une valeur pour chaque variable ;
- un ensemble d'évènements notifiés ;
- un statut d'attente pour chaque processus ainsi que l'endroit de son code où son exécution doit reprendre (on appelle cette dernière information le *point de contrôle courant du processus*).

L'exécution se déroule comme suit :

1. Phase comportementale. Tant qu'il existe des processus comportementaux qui ne sont pas en attente d'évènements (on dit qu'ils sont éligibles), on en choisit un de façon indéterministe et on reprend son exécution en son point de contrôle courant. Les évènements notifiés sont ajoutés à l'environnement ;
2. Phase de mise à jour. On exécute tous les processus de mise à jour qui émettent à leur tour des évènements qui sont ajoutés à l'environnement. Cette phase de mise à jour a en particulier pour but de simuler la stabilisation du courant électrique sur les ports ;
3. Phase de réveil. On signale à tous les processus quels évènements ont été notifiés, ce qui les réveille. On vide la liste des évènements notifiés de l'environnement. On réveille également les processus qui sont en attente d'une fraction de temps nulle. Si tous les processus comportementaux sont encore endormis, le temps est avancé de la plus petite fraction qu'un processus attend. Puis on reprend en phase comportementale.

L'exécution des phases 1 à 3 est appelée un *cycle*.

4.3.2 Port

Dans l'exemple des pompes, nous avons utilisé un objet particulier que l'on peut expliciter à présent : le port.

Les ports sont une construction qui modélise une interface de communication synchrone avec les autres modules. Un port est paramétré par le type de données des valeurs qu'il véhicule.

Un port sur un type `t` est en réalité un système formé de deux variables de type `t`, d'un évènement, et d'un processus de mise à jour. La première variable du port représente la valeur du port au début d'une phase comportementale. La seconde représente la valeur qu'on lui affecte pendant cette phase. Le processus de mise à jour va simplement recopier la nouvelle valeur sur l'ancienne et notifier l'évènement associé au port si cette valeur est différente. Se

mettre en attente d'une variable qui est un port signifie se mettre en attente d'un changement de valeur sur ce port, c'est-à-dire de l'évènement associé au port. La compilation effectue cette substitution. Notons au passage l'utilisation du mot-clef `update` dans le code qui définit les ports. C'est un identificateur valide du langage, qui peut se placer à la même position que les mots-clefs `thread` ou `discriminate`, mais dont l'utilisateur ne peut se servir. C'est pourquoi nous ne l'avons pas mentionné dans la grammaire. Enfin, ajoutons que le champ représentant l'ancienne valeur du port est local et ne peut pas être modifié par un autre processus que celui de la mise à jour du port.

Les ports sont très couramment utilisés dans les langages de description comme SystemC. Et comme dans SystemC, nous les définissons dans une bibliothèque que tout développement peut utiliser. Cette bibliothèque contient la définition des ports, mais aussi celle de connexion entre deux ports qui consiste simplement à identifier les variables et évènement de l'un avec ceux de l'autre. Puisque cela repose sur une affectation par adresse, la macro de connexion ne peut être invoquée que dans un constructeur de module. Nous rappelons qu'un champ local d'un module réserve son accès en écriture mais pas en lecture.

```
module port<t> {

  t old, new;
  event modified;

  local: old;

  read() : t { return (p.old); }
  write(t x) { p.new = x; }

  update {
    if (new <> old) notify(modified);
    old = new;
  }

}

function connect<t>(port<t> p1, port<t> p2) { p1 = p2; }
```

4.3.3 Module et Remplacement

Un module représente un système et des contraintes sur l'environnement initial d'exécution. Chaque variable ou évènement que déclare le module donne lieu à une ou des variables ou évènements du système qu'il représente. Une variable d'un type de base dans le module est une variable dans le système. Une variable d'un tableau est autant de variable que la dimension du tableau. La dimension des tableaux est donc une donnée statique en SystemD. En itérant ce schéma, on déplie chaque structure et module en un ensemble de variables et évènements.

Les méthodes sont des macros qui servent à factoriser le code. Elles sont simplement *inlinées* lors la compilation (on exclut donc du langage les fonctions récursives ou mutuellement récursives). Leur sémantique dépend uniquement de la sémantique de leurs instructions. Les threads

et discriminants sont traduits en processus, leur sémantique dépend donc de la sémantique de leurs instructions également.

Au final, un module contient toutes les variables et évènements de ses sous-modules ainsi que leur processus. Un assemblage de modules se fait simplement en déclarant chaque module de l'assemblage dans un module résultat. On peut enregistrer des variables et évènements locaux par l'invocation de la clause `local` dans un module. Le constructeur de module sert à énoncer quelles variables de deux modules représentent le même segment mémoire et à donner des valeurs initiales aux variables.

Les remplacements construisent des modules eux aussi. Mais ils prennent en argument un module supplémentaire qui doit être remplacé. Lorsque le compilateur atteint une clause de remplacement (mot-clef `with`), il ajoute à sa liste de vérifications le fait qu'un certain module doit en remplacer un autre. Cette vérification se fait selon une liste d'identification qui explicite quels variables et évènements doivent garder les mêmes valeurs lors des exécutions d'un module à l'autre.

4.3.4 Instructions

Nous décrivons une sémantique informelle des instructions du langage. Certaines sont classiques, d'autres sont spécifiques à SystemD.

Instructions classiques. Les déclarations sur des types de données, les affectations, la conditionnelle `if {...} else {...}` et la boucle `while` se comportent de façon classique, comme dans la plupart des langages de programmation.

Notification. L'instruction de notification `notify` prend un évènement en argument et l'ajoute à la liste des évènements notifiés. Cette liste est vidée pendant la phase de réveil et sert à réveiller les processus qui en sont en attente. La notification est une des deux instructions qui sert à synchroniser les processus.

Mise en attente. Les instructions de mise en attente `wait` et `halt` sont l'autre mécanisme de synchronisation avec la notification. Un processus qui fait appel à l'instruction `wait` interrompt son exécution pour tout le reste du cycle courant d'exécution. C'est d'ailleurs le seul moyen qu'à un processus pour redonner la main au reste du système. Nous utilisons donc un mode non-préemptif pour l'exécution des processus. Les processus peuvent attendre qu'un évènement soit notifié, mais aussi des combinaisons plus complexes de plusieurs évènements. Il peut attendre que plusieurs combinaisons soient notifiées, qu'une combinaison ou une autre soient notifiées ou encore qu'une combinaison ne le soit pas. Lorsqu'un processus est réveillé et reprend son exécution, il la reprend à l'instruction suivant le `wait` qui l'a endormi. L'instruction `halt` est une mise en attente particulière puisqu'elle interrompt définitivement l'exécution du processus qui ne pourra plus jamais reprendre.

Assertion. L'instruction `assert` pose une contrainte sur l'environnement d'exécution. Comme pour une hypothèse, cette contrainte est donnée par une expression booléenne. Si l'expression est vérifiée dans l'environnement d'exécution qui l'atteint, alors l'exécution du programme continue. Si elle est violée, le système tout entier s'arrête avec un code erreur.

Les principales différences avec SystemC sont pour des besoins de vérification compositionnelle. Cela requiert donc la notion de localité d'une variable ou d'un évènement, la possibilité d'énoncer explicitement qu'un processus est discriminant d'un autre, ou encore la notion de remplacement. De plus, les threads sont automatiquement lancés alors que SystemC fait appel à la liste de sensibilité du thread pour savoir quand il peut être exécuté. D'ailleurs, les possibilités de combinaisons d'évènements sont plus grandes en SystemD puisque SystemC n'autorise pas le mélange des opérations d'attente *ou* (`|`) et *et* (`&`) et qu'il y est impossible de décrire l'absence d'évènement (`not`). D'un point de vue sémantique, SystemD a la particularité de décrire des structures statiques en dehors des constructeurs. Les affectations se font par valeurs (comme sur des enregistrements, y compris pour les tableaux) et il n'est pas possible de manipuler les adresses des objets. On se débarrasse ainsi des pointeurs et de la manipulation mémoire bas niveau pendant une exécution (c'est-à-dire en dehors de la phase d'initialisation des constructeurs de modules) qui ne sont pas utiles pour décrire des systèmes, ce qui simplifie grandement le problème de vérification statique. Enfin, ajoutons que SystemD jouit d'un typage statique fort.

Nous décrivons KernelD dans le chapitre suivant, un langage noyau pour SystemD où toutes ces instructions s'y retrouveront de façon explicite ou implicite. La sémantique de KernelD précisera donc la sémantique formelle des instructions de SystemD. La traduction sera donnée dans la section qui suivra et qui explicite la compilation de SystemD en KernelD.

CHAPITRE 5

KernelD

Ce chapitre introduit KernelD, un langage noyau pour SystemD. Nous explicitons le langage, principalement formé de programmes sous forme de *graphe de flot de contrôle* [Poo95], sa sémantique, puis la traduction d'un développement SystemD en KernelD avant de montrer quel type de propriété le langage KernelD (et donc SystemD) permet de vérifier.

Le langage est bâti sur la base de *programmes* qui s'exécutent de façon à simuler le parallélisme et la concurrence. Afin d'établir des communications et un déroulement cohérent entre-eux, il est usuel d'introduire des *statuts d'attente* comme mécanisme de synchronisation. Les programmes peuvent alors interrompre leur exécution jusqu'à ce que certains *évènements* soient notifiés.

5.1 Expression

Les programmes KernelD manipulent explicitement deux types d'objets : les variables et les évènements.

Variable. On dispose d'un ensemble $\mathcal{X}$ de variables comme on en trouve classiquement dans les langages de programmation. Celles-ci permettent aux programmes qui les manipulent de stocker des informations, des résultats, et même de communiquer avec les autres programmes (mémoire partagée).

Évènement. On dispose également d'un ensemble $\mathcal{E}$ qui représente les évènements. Un évènement est un message que les programmes peuvent s'échanger entre-eux comme moyen de synchronisation.

Paramètre 5.1 Variable $x \in \mathcal{X}$ Évènement $e \in \mathcal{E}$

Point de contrôle. Les programmes seront définis par des graphes de flot de contrôle. Ces graphes contiendront des *points de contrôle* qui servent à repérer l'endroit où se trouve un programme au cours d'une exécution. On se donne donc un ensemble Q de points de contrôle.

Paramètre 5.2 Point de contrôle $q \in Q$

Statut. Toujours au sujet des programmes, ils seront capables d'interrompre leur exécution le temps que certains évènements se produisent. Le statut d'un programme conserve l'ensemble des évènements qui doivent être émis avant que le programme ne puisse reprendre son exécution. Les statuts serviront à construire les configurations des programmes, puisque leur exécution dépend du fait qu'il soit éligible ou non.

Les statuts d'attente sont exprimés en fonction des évènements susceptibles de réveiller les processus et d'une référence discrète au temps représentée par les entiers naturels.

Définition 5.3 Statut $\omega \in \mathcal{W}$

$$\begin{array}{rcl} \omega & ::= & \mathit{ready} \mid \mathit{halt} \\ & \mid & \mathrm{e} \mid \tau \in \mathbb{N} \\ & \mid & \omega \wedge \omega \mid \omega \vee \omega \\ & \mid & \neg\omega \end{array}$$

Ainsi, un programme peut signifier qu'il est prêt à être exécuté (*ready*), qu'il est fini (*halt*), qu'il se met en attente de la notification d'un évènement e, qu'il se met en attente que n fractions de secondes s'écoulent, ou encore qu'il attend des combinaisons de ces évènements de base, comme le fait d'attendre que deux ensembles d'évènements soient notifiés ($\wedge$), qu'un ensemble d'évènements ou un autre soient notifiés ($\vee$) ou encore s'interrompre tant que des évènements sont notifiés ($\neg$).

Valeur. Les programmes permettent de modifier la valeur des variables. L'ensemble des valeurs est laissé ouvert et peut être instancié différemment selon les besoins. On peut utiliser les booléens, les entiers, les flottants, mais la façon dont nous analyserons KernelD restera globalement la même. Cependant, nos exemples reposerons souvent sur les entiers relatifs et les booléens que nous utiliserons comme illustration (les détails de cette instanciation sont donnés en annexe A.3).

Paramètre 5.4 Valeurs $v \in \mathcal{V}$

Nous supposons à disposition un certain nombre d'opérations permises sur ces valeurs, par exemple les opérations arithmétiques et logiques dans le cas des entiers et des booléens. Un exemple d'instanciation qui reprend les opérations de SystemD se trouve en annexe A.3 du document.

Paramètre 5.5 Opération $op \in \mathcal{O}p$

Expression. À partir des valeurs, des variables et des opérations, il est possible de construire des *expressions* KernelD qui suivent fidèlement la structure des expressions SystemD.

Définition 5.6 Expression $exp \in \mathcal{E}xp$

$$exp \quad ::= \quad v \mid x \mid \mathrm{op}(exp, ..., exp)$$

5.2 Programme

Nous représentons un programme par un graphe de flot de contrôle : c'est un point de contrôle initial et et une relation de transition entre des points, étiquetées par des *instructions*.

Définition 5.7 Instruction $inst \in \mathcal{I}$

$$\begin{array}{rcl} \mathrm{inst} & ::= & \varepsilon \\ & \mid & exp? \\ & \mid & x \leftarrow exp \\ & \mid & e! \\ & \mid & \omega \end{array}$$

De façon informelle, nous décrivons la sémantique des instructions :

- ε : instruction *vide*. Elle n'apporte aucune modification ;
- $exp?$: instruction de *garde*. Une transition gardée ne peut être traversée que si la valeur de l'expression qui la garde est "vraie" dans l'état courant ;
- $x \leftarrow exp$: *affectation d'une expression*. Elle modifie l'état d'exécution en associant à la variable x l'évaluation de l'expression exp ;
- $e!$: *notification* de l'évènement e. Ceci indique à l'état d'exécution que l'évènement e est notifié ;
- ω : *mise en attente* des évènements ω. Cette instruction met en pause le programme qui l'a invoquée jusqu'à ce que la combinaison d'évènements ω soit produite.

Définition 5.8 Programme $\kappa = (q_0, T) \in \mathcal{K} \triangleq Q \times \mathcal{P}(Q \times \mathcal{I} \times Q)$

Abus de langage : étant donné un programme $\kappa = (q_0, T)$ on notera souvent $(q, inst, q') \in \kappa$ pour en réalité signifier que $(q, inst, q') \in T$.

Cette structure pour représenter les programmes est indéterministe : étant donné un état d'un programme, il peut exister différentes façons d'atteindre un état suivant. Le choix de considérer une transition plutôt qu'une autre est arbitraire. Cet indéterminisme aura une incidence particulière d'un point de vue génie logiciel pour l'expression de propriété des systèmes.

La figure 5.1 représente un graphe de flot de contrôle de façon graphique. Les points de contrôle sont cerclés et représentés par des entiers. Une flèche dirigée d'un point q vers un point q' et étiquetée par une instruction *inst* signifie que $(q, inst, q') \in \kappa$ et la flèche entrante sur un point signifie que celui-ci est le point de contrôle initial du programme.

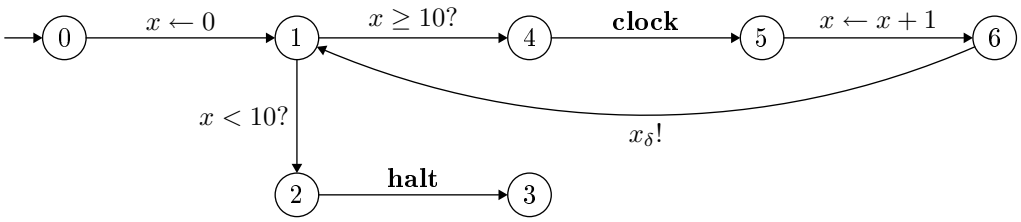

FIG. 5.1 – Un programme KERNELD simplifié

5.3 Module

Les modules sont simplement des ensembles de programmes. Au sein d'un module, les programmes s'exécutent en parallèle et en concurrence sur les variables qu'ils partagent. Les modules vont jouir d'une opération d'encapsulation et d'une opération d'assemblage pour les composer. Afin que chaque module dispose d'une partie qui lui est réservée, nous ajoutons la possibilité de déclarer locaux au module certaines variables et certains évènements : ils ont la propriété de ne pouvoir être modifier que par le module qui les déclare.

Définition 5.9 Module $M = \langle L, K \rangle \in \mathcal{M} \triangleq \mathcal{P}(\mathcal{X} \cup \mathcal{E}) \times \mathcal{P}(\mathcal{K})$

On pourrait définir l'encapsulation et l'assemblage de modules. Mais ces définitions vont suivre fidèlement celles sur les systèmes (cf chapitre 2). D'ailleurs, rappelons que les modules ne sont qu'une façon de représenter des systèmes. On confond donc encapsulation et assemblage de modules avec encapsulation et assemblage de systèmes (sous-entendu, encapsulation et assemblage des systèmes que représentent les modules).

5.4 Sémantique

Les programmes sont une manière de représenter des processus au sens du chapitre 2, et les modules des systèmes. Afin de montrer comment traduire un programme en un processus et un module en un système, nous définissons l'*état* que modifie un module ainsi qu'une sémantique pour l'ensemble des objets composant les modules : expression, instruction, programme.

Notre sémantique est proche de celle de SystemC que l'on peut trouver dans la littérature [Sal03, HP08, MMC+08] (aux différences entre SystemC et SystemD près), mais se rapproche particulièrement des travaux de Ali Habibi et Sofiene Tahar [HT04b], qui donnent une présentation sans passer par un formalisme intermédiaire.

5.4.1 État d'un Module

La notion centrale à la sémantique d'un module est celle d'*état de module*. Les instructions sont des transformateurs d'état. Leur enchaînement sera donc représenté par une succession d'états. C'est ainsi que l'on représente l'exécution d'un module.

Pour définir les états de module, on reprend la définition des états de système du chapitre 2. La seule différence vient de ce qu'on ne considère plus des processus mais des programmes. De plus, on raffine la définition de la *configuration d'un programme*.

L'*état mémoire* est un environnement qui associe une valeur à chaque variable.

Définition 5.10 État mémoire ou Environnement $\rho \in \Gamma \triangleq \mathcal{X} \to \mathcal{V}$

Au cours d'une exécution d'un module, les programmes sont amenés à changer de point de contrôle et de statut. Une configuration est la donnée de ces deux informations.

Définition 5.11 Configuration de programme $c = (q, \omega) \in \mathcal{C} \triangleq Q \times \mathcal{W}$

Notation : étant donnée une configuration $c = (q, \omega)$ on note $c.q$ le point de contrôle q de la configuration c et $c.\omega$ son statut ω.

Définition 5.12 Configuration de module $r = (C, E) \in \mathcal{R} \triangleq (\mathcal{K} \to \mathcal{C}) \times \mathcal{P}(\mathcal{E})$

Définition 5.13 État de module $\sigma = (r, \rho) \in \mathcal{R} \triangleq (\mathcal{K} \to \mathcal{C}) \times \Gamma$

Les états de module contiennent donc les informations suivantes :
- une configuration (c'est-à-dire un point de contrôle et un statut) pour chaque programme ;
- un ensemble d'évènements qui ont été récemment notifiés ;
- une valeur pour chaque variable.

À partir d'un état de module, on peut déduire un état de système. Il suffit pour cela de considérer la correspondance programme → processus. Nous verrons par ailleurs que les programmes se traduisent en des processus. Nous avons volontairement conservé les mêmes notations pour la définition des états de module que pour les états de système pour clairement repérer les notions communes.

5.4.2 Expression

On suppose que l'on dispose d'un prédicat $\phi_{op} \in \mathcal{P}((\mathcal{V} \times ... \times \mathcal{V}) \times \mathcal{V}) \to \mathit{Prop}$ pour chaque opération *op* (voir l'annexe A.3 pour un exemple avec les opérations arithmétiques et booléennes,

ainsi que pour une opération génératrice de nombres aléatoires). $\phi_{op}(v_1, ..., v_n, v)$ signifie que l'expression $op(v_1, ..., v_n)$ peut s'évaluer à v. On suppose également que l'on dispose d'une certaine valeur **true**. La sémantique d'une expression est l'ensemble des valeurs que représente une expression dans un certain environnement.

Définition 5.14 Sémantique d'une expression $[\![.]\!]_. \in \mathcal{E}xp \times \Gamma \to \mathcal{P}(\mathcal{V})$

$$[\![exp]\!]_\rho \triangleq \begin{cases} \{v\} \quad \text{si } exp = v \\ \\ \{\rho(x)\} \quad \text{si } exp = x \\ \\ \{v \mid \exists v_1 \ ... \ v_n,\ v_1 \in [\![exp_1]\!]_\rho \text{ et } ... \text{ et } v_n \in [\![exp_n]\!]_\rho \text{ et } \phi_{op}(v_1, ..., v_n, v)\} \\ \quad \text{si } exp = op(exp_1, ..., exp_n) \end{cases}$$

Définir la sémantique des expressions par une proposition permet d'introduire de l'indéterminisme. On peut alors définir un générateur de valeurs aléatoires qui peut donner différents résultats sur de mêmes arguments.

5.4.3 Instruction

La sémantique des instructions rend compte d'un changement de la mémoire (affectation), d'évènements émis (notification) et potentiellement d'un nouveau statut (mise en attente). De plus, l'exécution d'une instruction peut ne pas avoir d'image lorsque cette instruction est une garde non satisfaite. Tous ces cas ne dépendent que de la mémoire courante.

Définition 5.15 Sémantique d'une instruction $\dot{\to} \in \mathcal{I} \to \mathcal{P}(\Gamma \times (\mathcal{P}(\mathcal{E}) \times \Gamma \times \mathcal{W}))$

$$\rho \xrightarrow{\varepsilon} \emptyset, \rho, ready$$

$$\rho \xrightarrow{exp?} \emptyset, \rho, ready \quad \text{si } \mathbf{true} \in [\![exp]\!]_\rho$$

$$\rho \xrightarrow{x \leftarrow exp} \emptyset, \rho[x \leftarrow v], ready \quad \text{si } v \in [\![exp]\!]_\rho$$

$$\rho \xrightarrow{e!} \{e\}, \rho, ready$$

$$\rho \xrightarrow{\omega} \emptyset, \rho, \omega$$

5.4.4 Programme

Les modules que nous décrivons sont non-préemptifs. Nous définissons donc une *sémantique à grands pas* qui représente l'exécution d'un programme jusqu'à ce qu'il se mette en pause et redonne le contrôle au reste du module (exécution entre deux `wait`). Pour cela, on commence par poser une relation de *sémantique à petits pas* d'un programme qui représente l'exécution d'une instruction d'un programme dont le statut est à *ready* — on dit que le programme est éligible — et à son changement de point de contrôle courant. Le programme choisit de façon indéterministe de transiter depuis son point de contrôle courant à un point de contrôle suivant.

Définition 5.16 Sémantique à petits pas $\dot{\rightarrow}_1 \in \mathcal{K} \rightarrow \mathcal{P}((\mathcal{C} \times \mathcal{P}(\mathcal{E}) \times \Gamma) \times (\mathcal{C} \times \mathcal{P}(\mathcal{E}) \times \Gamma))$

$$(c, E, \rho) \xrightarrow{\kappa}_1 (c, E, \rho) \quad \text{si } c.\omega \neq \mathit{ready}$$

$$((q, \mathit{ready}), E, \rho) \xrightarrow{\kappa}_1 ((q', \omega), E \cup E', \rho') \quad \text{si } \exists \mathit{inst},\ (q, \mathit{inst}, q') \in \kappa \text{ et } \rho \xrightarrow{\mathit{inst}} (E', \rho', \omega)$$

Un grand pas consiste en l'exécution d'une suite de petits pas d'un programme jusqu'à ce qu'il rencontre une instruction de mise en attente et redonne la main au reste du module.

Définition 5.17 Sémantique à grands pas $\dot{\rightarrow} \in \mathcal{K} \rightarrow \mathcal{P}((\mathcal{C} \times \Gamma) \times (\mathcal{C} \times \mathcal{E} \times \Gamma))$

$$\xrightarrow{\kappa} \ \triangleq \ \left\{ ((c, \rho), (c', E, \rho')) \ \middle|\ \begin{array}{l} \exists \rho_1, ..., \rho_n,\ c_1, ..., c_n,\ E_1, ..., E_n, \\ \qquad (c, \emptyset, \rho) \xrightarrow{\kappa}_1 (c_1, E_1, \rho_1) \xrightarrow{\kappa}_1 ... \xrightarrow{\kappa}_1 (c_n, E_n, \rho_n) \xrightarrow{k}_1 (c', E, \rho') \\ \qquad \text{et} \quad c'.\omega \neq \mathit{ready} \end{array} \right\}$$

Les programmes sont en réalité une façon de décrire des processus. Nous donnons cette interprétation par une fonction qui traduit un programme en un processus. Pour cela, nous nous appuyons sur la définition des variables et évènements lus (fonction $R \in \mathcal{K} \rightarrow \mathcal{P}(\mathcal{X} \cup \mathcal{E})$) et écrits (fonction $W \in \mathcal{K} \rightarrow \mathcal{P}(\mathcal{X} \cup \mathcal{E})$) par un programme ainsi que par le calcul de l'ensemble des configurations (fonction $C \in \mathcal{K} \rightarrow \mathcal{P}(\mathcal{C})$) que le programme peut rencontrer. Ces définitions sont détaillées en annexe A.3.

Définition 5.18 Du programme au processus $\Phi \in \mathcal{K} \rightarrow \Psi$

$$\Phi(\kappa) \triangleq \langle R(\kappa), W(\kappa), C(\kappa), \xrightarrow{\kappa} \rangle$$

La sémantique d'un programme est assimilée à la sémantique du processus qui est son image par Φ.

5.4.5 Module

Comme les programmes auxquels on associe un processus, on associe aux modules des systèmes. Ceux-ci sont simplement constitués des programmes traduits en processus.

Définition 5.19 Du module au système $\Phi \in \mathcal{M} \rightarrow \mathcal{S}$

$$\Phi\langle L, K \rangle = \langle L, \bigcup_{\kappa \in K} \Phi(\kappa) \rangle$$

La sémantique d'un module est assimilée à la sémantique du système image du module par Φ. Cette sémantique dépend d'un processus particulier, l'ordonnanceur, qui va gérer les évènements émis et les statuts d'attente des programmes.

$\surd$ *Nous pousserons la traduction entre programme et processus, et module et système, jusqu'à confondre programme avec processus (sous-entendu le processus image par Φ) et module avec système (sous-entendu le système image par Φ).*

5.5 Ordonnanceur

L'ordonnanceur est un processus haut niveau qui décide de réveiller les programmes en attente d'évènements. Avant de voir plus en avant quel est son comportement, explicitons la sémantique des statuts.

5.5.1 Réduction de statut

Les statuts sont *réduits* par les évènements émis au cours de l'exécution du système et du temps qui passe. Ces réductions sont données en terme de systèmes de réécriture, dont les règles s'appliquent à toutes les sous-expressions d'un statut. Le premier système explicite la sémantique des constructeurs.

Définition 5.20 Réduction $\rhd \in \mathcal{W} \to \mathcal{W}$

$$\begin{array}{cc} \mathit{ready} \wedge \omega \rhd \omega & \mathit{halt} \wedge \omega \rhd \mathit{halt} \\ \mathit{ready} \vee \omega \rhd \mathit{ready} & \mathit{halt} \vee \omega \rhd \omega \\ \neg\, \mathit{ready} \rhd \mathit{halt} & \neg\, \mathit{halt} \rhd \mathit{ready} \end{array}$$

Ce système, comme les suivants, satisfait les propriétés d'un système de réécriture de Church-Rosser. La forme normale d'un statut par ces règles existe toujours et est unique. C'est pourquoi ce système est donné comme une fonction pour laquelle on pourra noter $\omega \rhd \omega'$ lorsque $\rhd(\omega) = \omega'$.

La seconde réduction est paramétrée par un ensemble d'évènements. Elle est appelée *réduction positive* car elle se contente de remplacer par *ready* toutes les occurrences d'évènements appartenant à l'ensemble en paramètre dans le statut en argument. Cette fonction permet de réduire dans un statut les évènements qui ont été notifiés au cours de l'exécution d'un module.

Définition 5.21 Réduction positive $\rhd_\cdot \in \mathcal{E} \to \mathcal{W} \to \mathcal{W}$

$$e \rhd_E \mathit{ready} \quad \text{si } e \in E$$

La prochaine réduction n'est pas paramétrée. C'est la *réduction négative*, qui consiste à remplacer les évènements d'un statut par *halt*, pour signifier qu'ils ne se sont pas encore produits.

Définition 5.22 Réduction négative $\rhd^- \in \mathcal{W} \to \mathcal{W}$

$$e \rhd^- \mathit{halt}$$

$$0 \rhd^- \mathit{halt} \qquad \tau \rhd^- \mathit{ready} \quad \text{si } \tau \neq 0$$

Enfin, la dernière réduction est celle associée au temps. Elle est paramétrée par un entier naturel et consiste à réduire toutes les références au temps d'un statut de cette quantité. Si la quantité obtenue est nulle, le statut est réduit à *ready*.

Définition 5.23 Réduction temporelle $\rhd_. \in \mathbb{N} \to \mathcal{W} \to \mathcal{W}$

$$\tau' \rhd_\tau \begin{cases} \mathit{ready} & \text{si } \tau' \leq \tau \\ \tau' - \tau & \text{sinon} \end{cases}$$

$\surd$ *Dans la suite, on considère que les systèmes* $\rhd_E$, $\rhd^-$ *et* $\rhd_\tau$ *utilisent également les règles de réécriture de* $\rhd$.

Les systèmes de réduction positive et négative sont complémentaires et leurs combinaisons permettent de répondre à des questions comme « les évènements ont-ils déjà été produits ? » ou encore « pourront-ils se produire un jour ? ».

En particulier, nous souhaiterons souvent savoir si un statut ω se réduit en *ready* à l'instant précis où les évènements E ont été émis. Supposons que ω ait été réduit positivement selon chacune des variables de l'ensemble E. Tous les évènements notifiés ont donc été réduits à *ready*. On sait alors que les évènements restants dans le statut n'ont pas été émis. Pour répondre à la question « le réveil d'un processus en attente de ω peut-il se faire ? », il faut donc réduire les évènements restants à *halt* (non émis). La sémantique du $\neg$ (les évènements ne doivent pas avoir été émis) transformera ces *halt* en *ready*. Ainsi, la réponse à la question initiale se fait en deux temps : réduction positive selon chaque élément de E puis réduction négative.

Ces définitions s'accompagnent de deux lemmes qui montrent que l'ordre entre les réductions positive et temporelle n'importe pas : le résultat sera le même. On rappelle que les systèmes de réductions peuvent être considérés comme des fonctions.

Lemme 5.1 $\forall \omega\ E\ \tau,\ \rhd_E(\rhd_\tau(\omega)) = \rhd_\tau(\rhd_E(\omega))$

Preuve : on procède par récurrence sur la structure de ω.
- Si $\omega = \mathit{ready}$, alors $\rhd_E(\rhd_\tau(\omega)) = \rhd_E(\rhd_\tau(\mathit{ready})) = \mathit{ready} = \rhd_\tau(\rhd_E(\mathit{ready})) = \rhd_\tau(\rhd_E(\omega))$.
- Si $\omega = \mathit{halt}$, alors $\rhd_E(\rhd_\tau(\omega)) = \rhd_E(\rhd_\tau(\mathit{halt})) = \mathit{halt} = \rhd_\tau(\rhd_E(\mathit{halt})) = \rhd_\tau(\rhd_E(\omega))$.
- Si $\omega = e$, alors si $e \in E$, $\rhd_E(\rhd_\tau(\omega)) = \rhd_E(\rhd_\tau(e)) = \mathit{ready} = \rhd_\tau(\rhd_E(e)) = \rhd_\tau(\rhd_E(\omega))$. Si $e \notin E$, $\rhd_E(\rhd_\tau(\omega)) = \rhd_E(\rhd_\tau(e)) = e = \rhd_\tau(\rhd_E(e)) = \rhd_\tau(\rhd_E(\omega))$.
- Si $\omega = \tau'$, alors $\rhd_E(\rhd_\tau(\omega)) = \rhd_E(\rhd_\tau(e)) = \tau' - \tau = \rhd_\tau(\rhd_E(\tau')) = \rhd_\tau(\rhd_E(\omega))$.
- Si $\omega = \omega_1 \wedge \omega_2$, on sait par hypothèse de récurrence que $\rhd_E(\rhd_\tau(\omega_1)) = \rhd_\tau(\rhd_E(\omega_1))$ et $\rhd_E(\rhd_\tau(\omega_2)) = \rhd_\tau(\rhd_E(\omega_2))$. $\rhd_E(\rhd_\tau(\omega)) = \rhd_E(\rhd_\tau(\omega_1 \wedge \omega_2)) = \rhd_E(\rhd_\tau(\omega_1) \wedge \rhd_\tau(\omega_2)) = \rhd_E(\rhd_\tau(\omega_1)) \wedge \rhd_E(\rhd_\tau(\omega_2))$ car $\rhd_E$ et $\rhd_\tau$ sont des systèmes de réécriture. Donc par hypothèses de récurrence, on a $\rhd_E(\rhd_\tau(\omega)) = \rhd_\tau(\rhd_E(\omega_1)) \wedge \rhd_\tau(\rhd_E(\omega_2)) = \rhd_\tau(\rhd_E(\omega_1 \wedge \omega_2)) = \rhd_\tau(\rhd_E(\omega))$.
- Si $\omega = \omega_1 \vee \omega_2$, on sait par hypothèse de récurrence que $\rhd_E(\rhd_\tau(\omega_1)) = \rhd_\tau(\rhd_E(\omega_1))$ et $\rhd_E(\rhd_\tau(\omega_2)) = \rhd_\tau(\rhd_E(\omega_2))$. $\rhd_E(\rhd_\tau(\omega)) = \rhd_E(\rhd_\tau(\omega_1 \vee \omega_2)) = \rhd_E(\rhd_\tau(\omega_1) \vee \rhd_\tau(\omega_2)) = \rhd_E(\rhd_\tau(\omega_1)) \vee \rhd_E(\rhd_\tau(\omega_2))$ car $\rhd_E$ et $\rhd_\tau$ sont des systèmes de réécriture. Donc par hypothèses de récurrence, on a $\rhd_E(\rhd_\tau(\omega)) = \rhd_\tau(\rhd_E(\omega_1)) \vee \rhd_\tau(\rhd_E(\omega_2)) = \rhd_\tau(\rhd_E(\omega))$.
- Si $\omega = \neg\omega'$, on sait par hypothèse de récurrence que $\rhd_E(\rhd_\tau(\omega')) = \rhd_\tau(\rhd_E(\omega'))$. $\rhd_E(\rhd_\tau(\omega)) = \rhd_E(\rhd_\tau(\neg\omega')) = \rhd_E(\neg \rhd_\tau (\omega')) = \neg \rhd_E (\rhd_\tau(\omega'))$ car $\rhd_E$ et $\rhd_\tau$ sont des systèmes de réécriture. Donc par hypothèses de récurrence, on a $\rhd_E(\rhd_\tau(\omega)) = \neg\rhd_\tau(\rhd_E(\omega')) = \rhd_\tau(\rhd_E(\neg\omega')) = \rhd_\tau(\rhd_E(\omega))$.

□

Lemme 5.2 $\forall \omega_1\ \omega_2\ \omega'_1\ \omega'_2\ E\ \tau,\ \omega_1 \rhd_E \omega'_1$ et $\omega_1 \rhd_\tau \omega_2$ et $\omega_2 \rhd_E \omega'_2 \Rightarrow \omega'_1 \rhd_\tau \omega'_2$

Preuve : on peut réécrire les hypothèses sous forme d'égalité. On a alors $\omega'_1 = \rhd_E(\omega_1)$, $\omega_2 = \rhd_\tau(\omega_1)$ et $\omega'_2 = \rhd_E(\omega_2)$. Par réécriture, on a donc $\omega'_2 = \rhd_E(\rhd_\tau(\omega_1))$. D'après le lemme 5.1, on obtient $\omega'_2 = \rhd_\tau(\rhd_E(\omega_1)) = \rhd_\tau(\omega'_1)$. □

Un dernier lemme montre que deux réductions temporelles successives sont semblables à une seule.

Lemme 5.3 $\forall \tau\ \tau'\ \omega,\ \rhd_\tau(\rhd_{\tau'}(\omega)) = \rhd_{\tau+\tau'}(\omega)$

Preuve : par récurrence sur la structure de ω :
- Si $\omega \in \{ready, halt\} \cup \mathcal{E}$, $\rhd_\tau(\rhd_{\tau'}(\omega)) = \omega = \rhd_{\tau+\tau'}(\omega)$.
- Si $\exists \tau''$, $\omega = \tau''$, alors $\rhd_\tau(\rhd_{\tau'}(\omega)) = \rhd_\tau(\rhd_{\tau'}(\tau'')) = \rhd_\tau(\tau''-\tau') = \tau''-\tau'-\tau = \tau''-(\tau+\tau') = \rhd_{\tau+\tau'}(\tau'') = \rhd_{\tau+\tau'}(\omega)$ (avec le cas particulier qui peut donner *ready* mais qui ne change pas la correction).
- Si $\exists \omega_1\ \omega_2,\ \omega = \omega_1 \wedge \omega_2$, on sait par hypothèses de récurrence que $\rhd_\tau(\rhd_{\tau'}(\omega_1)) = \rhd_{\tau+\tau'}(\omega_1)$ et $\rhd_\tau(\rhd_{\tau'}(\omega_2)) = \rhd_{\tau+\tau'}(\omega_2)$. $\rhd_\tau(\rhd_{\tau'}(\omega)) = \rhd_\tau(\rhd_{\tau'}(\omega_1 \wedge \omega_2)) = \rhd_\tau(\rhd_{\tau'}(\omega_1) \wedge \rhd_{\tau'}(\omega_2)) = \rhd_\tau(\rhd_{\tau'}(\omega_1)) \wedge \rhd_\tau(\rhd_{\tau'}(\omega_2))$ car $\rhd_\tau$ et $\rhd_{\tau'}$ sont des systèmes de réécriture. Par hypothèses de récurrence, on a donc $\rhd_\tau(\rhd_{\tau'}(\omega)) = \rhd_{\tau+\tau'}(\omega_1) \wedge \rhd_{\tau+\tau'}(\omega_2) = \rhd_{\tau+\tau'}(\omega_1 \wedge \omega_2) = \rhd_{\tau+\tau'}(\omega)$.
- Si $\exists \omega_1\ \omega_2,\ \omega = \omega_1 \vee \omega_2$, on fait le même raisonnement que pour $\wedge$ en remplaçant $\wedge$ par $\vee$.
- Si $\exists \omega',\ \omega = \neg\omega'$, on sait par hypothèse de récurrence que $\rhd_\tau(\rhd_{\tau'}(\omega')) = \rhd_{\tau+\tau'}(\omega')$. On a alors $\rhd_\tau(\rhd_{\tau'}(\omega)) = \rhd_\tau(\rhd_{\tau'}(\neg\omega')) = \rhd_\tau(\neg \rhd_{\tau'} (\omega')) = \neg(\rhd_\tau(\rhd_{\tau'}(\omega'))) = \neg(\rhd_{\tau+\tau'}(\omega')) = \rhd_{\tau+\tau'}(\neg\omega') = \rhd_{\tau+\tau'}(\omega)$.

□

5.5.2 Comportement

L'ordonnanceur permet de distinguer plusieurs types de programmes en leur donnant des priorités. Cela peut se faire en utilisant des évènements particuliers, émis par l'ordonnanceur et attendu par les programmes de la priorité concernée.

En KernelD, on sépare deux types de programmes : les programmes *comportementaux* et les programmes de *mises à jour*. Les programmes comportementaux décrivent le comportement des composants du module. Les programmes de mises à jour permettent de simuler la mise à jour des ports par les canaux de communication entre processus. Les programmes d'un même ensemble prioritaire sont tous au même niveau d'un point de vue instant de simulation : ils peuvent s'exécuter dans un ordre complètement arbitraire. Mais ceci n'est pas vrai lorsque l'on mélange les programmes de deux ensembles à priorité différentes. Par exemple, les mises à jour doivent s'effectuer *après* que les programmes comportementaux se soient exécutés. Afin d'introduire cette hiérarchie de priorités entre les programmes, on met l'ordonnanceur à disposition d'un évènement spécial, appelé *upd*, qu'il pourra émettre lorsqu'il le souhaite. Les programmes qui veulent se déclarer programme de mises à jour doivent se mettre en attente, entre autres, de cet évènement.

L'ordonnanceur est toujours éligible. D'ailleurs, au contraire des programmes, on ne lui associe pas de statut. Son comportement se décrit en plusieurs phases disjointes et successives :

- s'il existe un programme éligible, l'ordonnanceur ne fait rien ;
- si aucun programme n'est éligible et que l'évènement *upd* n'est pas présent dans la liste des évènements notifiés, il l'émet et réduit en conséquence les statuts des programmes. On dira qu'il passe en phase de mises à jour, et ça sera le cas dès lors que l'évènement *upd* est notifié. Autrement, lorsque *upd* n'est pas notifié, nous dirons que l'ordonnanceur est en phase comportementale ;
- si aucun programme n'est éligible et que l'ordonnanceur était déjà en phase de mises à jour, il élimine *upd* des évènements notifiés, il réduit le statut de tous les programmes en fonction de la liste des évènements notifiés ainsi obtenue et vide cette liste. Il réduit temporellement le statut de tous les processus selon l'instant le plus court qui apparaît dans l'un de leur statut. Enfin, il repasse en phase comportementale.

Le fait qu'un programme κ soit éligible dans une configuration de module signifie que son statut peut se réduire négativement à *ready*. Nous disons que "κ est éligible dans r" lorsque $r.C(\kappa).\omega \rhd^{-} \mathit{ready}$;

L'instant le plus court apparaissant dans un état est l'instant le plus court apparaissant dans les statuts d'attente de ses programmes. On note $\tau \in \omega$ le fait que la fraction de temps τ apparaît dans le statut ω. Cette relation est définie par les règles suivantes :

$$\tau \in \tau \qquad \tau \in \neg\omega \quad \text{si } \tau \in \omega$$

$$\tau \in \omega_1 \wedge \omega_2,\ \tau \in \omega_1 \vee \omega_2 \quad \text{si } \tau \in \omega_1 \text{ ou } \omega_2$$

L'ensemble $T(\omega) = \{\tau \mid \tau \in \omega\}$ est donc l'ensemble de tous les instants contenus dans le statut ω. De même, l'ensemble $T(r) = \bigcup_{\kappa \in \mathcal{K}} T(r.C(\kappa).\omega)$ est l'ensemble de tous les instants qui apparaissent dans les statuts d'attente courants d'un état r. On note $\tau_{\min}(r)$ le plus petit de ces instants, qui vaut une valeur quelconque (par exemple ∞) lorsqu'aucun instant n'apparaît dans r, et 0 lorsqu'un programme est éligible dans r. En effet, si un programme est éligible, seuls ceux en attente de 0 seconde (δ-*delay* en SystemC) seront réveillés et le temps n'avancera pas.

$$\tau_{\min}(r) \quad \triangleq \quad \begin{cases} 0 & \text{si } \exists\kappa \text{ éligible dans } r \\ \min(T(r)) & \text{sinon} \end{cases}$$

On montre que la réduction positive d'un statut augmente potentiellement l'instant minimum du statut, ce qui peut alors s'étendre à des ensembles de statuts.

Lemme 5.4 $\quad \forall\omega\ E,\ \min(T(\omega)) \leq \min(T(\rhd_E(\omega)))$

Preuve : on procède par récurrence sur la structure de ω.
- Si $\omega \in \{\mathit{ready}, \mathit{halt}\} \cup \{\tau \mid \tau \in \mathbb{N}\} \cup \{e \mid e \notin E\}$, alors $\rhd_E(\omega) = \omega$ et donc $\min(T(\omega)) \leq \min(T(\rhd_E(\omega)))$.
- Si $\exists e \in E,\ \omega = e$, alors $T(\omega) = T(e) = \emptyset = T(\mathit{ready}) = T(\rhd_E(e)) = T(\rhd_E(\omega))$ et donc $\min(T(\omega)) = \min(T(\rhd_E(\omega)))$, d'où $\min(T(\omega)) \leq \min(T(\rhd_E(\omega)))$.
- Si $\exists\omega_1\ \omega_2,\ \omega = \omega_1 \wedge \omega_2$, on sait par hypothèses de récurrence que $\min(T(\omega_1)) \leq \min(T(\rhd_E(\omega_1)))$ et $\min(T(\omega_2)) \leq \min(T(\rhd_E(\omega_2)))$. Par définition on a $T(\omega_1 \wedge \omega_2) = T(\omega_1) \cup T(\omega_2)$. Donc $\min(T(\omega_1 \wedge \omega_2)) = \min(T(\omega_1) \cup T(\omega_2))$. Si $\min(T(\omega_1)) \leq \min(T(\omega_2))$, alors $\min(T(\omega_1) \cup T(\omega_2)) = \min(T(\omega_1))$. Or, par hypothèses de récurrence, $\min(T(\omega_1)) \leq$

$\min(T(\rhd_E(\omega_1)))$ et $\min(T(\omega_1)) \leq \min(T(\omega_2)) \leq \min(T(\rhd_E(\omega_2)))$. Donc $\min(T(\omega_1 \wedge \omega_2)) = \min(T(\omega_1)) \leq \min(T(\rhd_E(\omega_1)) \cup T(\rhd_E(\omega_2))) = \min(T(\rhd_E(\omega_1 \wedge \omega_2)))$. On procède de même lorsque $\min(T(\omega_2)) \leq \min(T(\omega_1))$.
- Si $\exists \omega_1\ \omega_2,\ \omega = \omega_1 \vee \omega_2$, on fait le même raisonnement que pour $\wedge$ en remplaçant $\wedge$ par $\vee$.
- Si $\exists \omega',\ \omega = \neg\omega'$, on sait par hypothèse de récurrence que $\min(T(\omega')) \leq \min(T(\rhd_E(\omega')))$. $\min(T(\omega)) = \min(T(\neg\omega')) = \min(T(\omega')) \leq \min(T(\rhd_E(\omega'))) = \min(T(\rhd_E(\neg\omega'))) = \min(T(\rhd_E(\omega)))$.

□

Lemme 5.5 $\forall W \subseteq \mathcal{W}\ E,\ \min(T(W)) \leq \min(T\{\rhd_E(\omega) \mid \omega \in W\})$

Preuve : on procède par l'absurde en supposant que $\min(T(W)) > \min(T\{\rhd_E(\omega) \mid \omega \in W\})$ et en montrant une contradiction. $\min(T(W)) > \min(T\{\rhd_E(\omega) \mid \omega \in W\}) \Rightarrow \exists \omega \in W$, $\forall \omega' \in W$, $\min(T(\omega')) > \min(T(\rhd_E(\omega)))$. Or on sait d'après le lemme 5.4 que $\min(T(\omega)) \leq \min(T(\rhd_E(\omega)))$ pour tout ω, d'où l'absurdité. □

À présent, définissons formellement le comportement de l'ordonnanceur.

Définition 5.24 Transition Ordonnanceur $\xrightarrow{\text{Sched}} \in \mathcal{P}(\mathcal{R} \times \mathcal{R})$

On pose un certain nombre de notations :
- "r inactif" dénote le fait qu'aucun programme n'est éligible dans r, c'est-à-dire que $\forall \kappa,\ \kappa$ n'est pas éligible dans r ;
- "r est en phase comportementale" signifie que $upd \notin r.E$. "r est en phase de mise à jour" signifie que $upd \in r.E$;
- quelle que soit une réduction de statuts $\rhd$, nous notons "$C \rhd C'$" le fait que les points de contrôle courants de chaque programme restent inchangés de C à C', et que les statuts de tous les programmes ont été réduits selon $\rhd$, c'est-à-dire que $\forall \kappa$, $C(\kappa).q = C'(\kappa).q$ et $C(\kappa).\omega \rhd C'(\kappa).\omega$.

Alors,

$$r \xrightarrow{\text{Sched}} r \quad \text{si } \exists \kappa,\ \kappa \text{ éligible dans } r$$

$$r \xrightarrow{\text{Sched}} r' \quad \begin{array}{ll} & \text{si } r \text{ inactif} \\ \text{et} & r \text{ est en phase comportementale} \\ \text{et} & r'.E = r.E \cup \{upd\} \\ \text{et} & r.C \rhd_{\{upd\}} r'.C \end{array}$$

$$r \xrightarrow{\text{Sched}} r' \quad \begin{array}{ll} & \text{si } r \text{ inactif} \\ \text{et} & r \text{ est en phase de mises à jour} \\ \text{et} & r'.E = \emptyset \\ \text{et} & \exists C,\ r.C \rhd_E C \rhd_\tau r'.C \text{ où } E = r.E - \{upd\} \text{ et } \tau = \tau_{\min}(C) \end{array}$$

Pour résumer, nous montrons en figure 5.2 une représentation graphique de l'évolution de l'exécution d'un module KERNELD. Les transitions de cette figure se lisent comme suit :
- les transitions sont étiquetées par des gardes en numérateur et des actions en dénominateur ;
- une transition gardée par *Éligible ?* ne peut être empruntée que s'il existe un programme éligible dans l'état courant. *¬Éligible ?* est la garde contraire, c'est-à-dire qu'elle ne peut être empruntée que si aucun programme n'est éligible ;

– l'action κ représente l'exécution d'un programme κ quelconque. On rappelle que si le programme n'est pas éligible, il ne fait rien ;
– l'action *upd*! consiste à émettre l'évènement *upd* et à réduire le statut des programmes selon cet évènement ;
– l'action *Réductions* consiste à réduire les statuts des programmes selon les évènements notifiés (sauf *upd*) et selon l'instant le plus court apparaissant dans l'état courant.

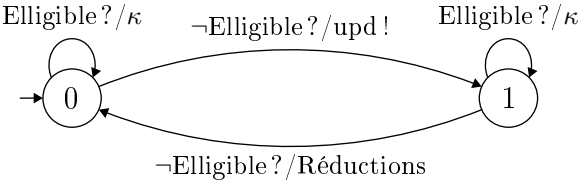

FIG. 5.2 – Exécution en KERNELD

On commence en phase comportementale. Tant que des programmes sont éligibles, ils s'exécutent. Lorsque plus aucun programme n'est éligible, on entre en phase de mises à jour avec la notification de l'évènement *upd* par l'ordonnanceur qui réveille les programmes qui en étaient en attente. Tant que des programmes de mises à jour sont éligibles, ils s'exécutent. Lorsque plus aucun n'est éligible, l'ordonnanceur réduit les statuts de tous les programmes selon les évènements qui ont été émis et selon l'instant le plus court et retourne ensuite en phase comportementale.

5.6 Traduction SYSTEMD → KERNELD

La traduction d'un développement SYSTEMD donne un ou des modules KERNELD, un état initial pour chacun de ces modules, et des directives de simulation ou de vérification. Cette traduction se fait en plusieurs passes. Certaines sont classiques en compilation ; nous les mentionnerons mais ne les détaillerons pas. D'autres sont spécifiques à SYSTEMD et feront l'objet d'explications plus fournies.

√ *Dans la suite, nous ne considérons que des développement* SYSTEMD *valides vis-à-vis de la grammaire présentée en section 4.2.*

La première passe consiste à vérifier qu'aucun appel de fonction dans des threads ou discriminants ne contient d'instructions en dehors de celles admises dans la grammaire des instructions des threads et discriminants.

La seconde passe est celle du typage. SYSTEMD jouit d'un typage statique fort, sans transtypage. Le bon typage d'un développement SYSTEMD est vérifiée lors de la traduction. Cette vérification permet également de s'assurer de l'absence de fonctions ou méthodes récursives ou mutuellement récursives, car le corps de ces fonctions et méthodes seront *inlinées*. De même, SYSTEMD n'autorise pas les structures et modules récursifs ou mutuellement récursifs.

La troisième passe de traduction consiste à vérifier que la qualité locale des variables est respectée. On vérifie donc que les variables locales ne se trouvent pas à gauche d'une affectation

dans une structure ou un module en dehors de la portée de la structure ou du module qui la déclare. De plus, cette passe vérifie également que les discriminants n'écrivent pas sur les variables locale au module qui les contient.

Ensuite, on passe à la phase de traduction d'une structure, d'un module ou d'une abstraction. Nous expliquons les trois cas séparément, même si certaines étapes sont similaires.

Structure

1. La traduction donne comme résultat un ensemble de méthodes et un environnement initial, paramétrés par des noms de variables KernelD qui sont utilisées par les méthodes, et par des types et/ou des variables supplémentaires éventuellement dans le cas d'une structure à templates. Lors d'une déclaration d'une variable de type une structure, les noms de variables KernelD en paramètre sont instanciées par des variables KernelD fraîches ; la correspondance entre variables paramètres et variables d'instance est utilisée pour remplacer chaque variable paramètre par la variable d'instance qui lui est associée dans le corps des méthodes. Il y a autant de variables KernelD pour la structure qu'il y a de variables de types de base dans ses déclarations, et ce récursivement sur ses sous-structures (qui ont déjà été traduites : l'ordre de déclaration a son importance en SystemD). Un tableau est représenté par un nombre d'instance égal à celui de sa dimension. Ces déclarations initialisent la valeur des variables de façon complètement arbitraire dans l'environnement d'exécution (qui est totalement vide au départ).
2. Le cas échéant, les appels de fonctions dans les méthodes sont *inlinés* en suivant un appel par adresse.

Module

1. La traduction d'un module SystemD donne un module KernelD paramétré par des noms de variables KernelD et par des types et/ou des variables supplémentaires dans le cas d'un module à templates. La traduction commence par la création des variables suivant les déclarations, comme pour les structures. De mêmes, les variables des sous-structures et sous-modules deviennent des variables du module qui les contient. Les déclarations initialisent de façon arbitraire les valeurs des variables. Les applications d'abstractions sont ignorées dans le cas d'une simulation. Dans le cas de vérifications, la traduction de l'abstraction est utilisée à la place du module qu'elle remplace ; la liste d'identification de cette abstraction sert à déterminer quelles variables de l'abstraction doivent être utilisées à la place de celles du module à remplacer.
2. Le constructeur du module est ensuite exécuté pour donner des valeurs aux variables et pour partager des segments mémoire. Chaque variable fait partie d'une classe d'équivalence de noms de variables qui sert à déterminer quelles variables sont identifiées. Dans la suite, chaque variable sera remplacée par le représentant de sa classe d'équivalence.
3. Les appels de fonctions et méthodes apparaissant dans les threads et discriminants sont inlinés. Les arguments de ces appels sont passés par adresse.
4. Les threads, discriminants et programmes de mise à jour sont traduits en des programmes KernelD en suivant un algorithme détaillé plus loin. Les threads et programmes de mises à jour sont compris dans une boucle `while (true)` ; les programmes de mises à jour attendent implicitement l'évènement *upd*.

Abstraction La traduction d'une abstraction s'opère comme la traduction d'un module, à ceci près qu'elle ajoute une directive de vérification à l'environnement : « l'abstraction est-elle correcte ? ». Les champs identifiés par la commande `link` seront représentés par une seule et même variable KernelD.

Après toutes ces traductions, on obtient un ensemble de modules KernelD par leur déclaration dans les primitives d'appel du moteur de simulation ou du moteur de vérification. Dans le cas de la simulation, un seul état initial est créé, et il est possible de laisser des variables ouvertes sur l'extérieur. Elles prendront tout au long de la simulation des valeurs arbitraires. Dans le cas de la vérification, on considère tous les états initiaux qu'il était possible d'obtenir en un seul état initial abstrait. Dans la suite, on note $\Sigma_M \subseteq \Sigma$ l'ensemble des états initiaux qu'il est possible d'obtenir par la traduction d'un module M.

À présent, nous détaillons la traduction des threads et discriminants en des programmes KernelD.

$\surd$ *Les traductions qui suivent retournent des programmes* KernelD *qui ont exactement un état puits (qui représenterait l'état de sortie dans des programmes classiques). De tels programmes sont notés* $q_0 \overset{\cdots}{\longrightarrow} q_f$ *où* q_0 *est l'état initial du programme et* q_f *l'état puits.*

En suivant la remarque ci-dessus, nous aurons besoin de connaître les points de contrôle utilisé par un programme.

Définition 5.25 Points de contrôle d'un programme $\mathit{Points} \in \mathcal{K} \to \mathcal{P}(Q)$

$$\text{Points}(\kappa) \triangleq \{q \mid \exists q'\ \mathit{inst},\ (q, \mathit{inst}, q') \text{ ou } (q', \mathit{inst}, q) \in \kappa\}$$

Expression et Statut. Les expressions SystemD se traduisent directement en expression KernelD. De même, les statuts SystemD trouvent également une traduction immédiate en statut KernelD, à ceci près que les variables de type `port` sont remplacées par l'évènement qui leur est associé.

Nous définissons un opérateur $\Pi \in \langle \mathit{thread\text{-}inst} \rangle^* \times \mathcal{P}(Q) \to \mathcal{K}$ qui prend en argument une séquence d'instructions de thread et un ensemble de points de contrôle $Q_\neg$ et qui retourne le programme que représente la séquence d'instructions et qui ne contient pas de points de contrôle dans $Q_\neg$. L'opérateur est défini par récurrence sur la taille de la séquence d'instructions. Nous traduisons donc chacune des instructions, puis la séquence vide et enfin une séquence non vide.

Déclaration. La déclaration d'une variable d'un certain type de données n'a pas de traduction en KernelD. La déclaration au niveau SystemD sert à donner un type à la variable ; c'est une primitive de génie logiciel qui n'a pas de contrepartie KernelD.

Appel de fonction ou de méthode. Les programmes ne contiennent plus d'appel de fonction ni de méthode puisque le code de celles-ci a été inliné.

Affectation. Le programme associé à l'affectation `lv = exp ;` pour une certaine valeur gauche `lv` et une certaine expression `exp` consiste en l'affectation KernelD de la variable KernelD *lv* associée à la variable SystemD `lv` par l'expression KernelD *exp*, image de l'expression SystemD `exp`.

$$\Pi(\texttt{lv = exp ;}, Q_\neg) \quad = \quad q_0 \xrightarrow{lv \leftarrow exp} q_f \qquad \text{où } q_0 \text{ et } q_f \notin Q_\neg$$

Notification. Le programme associé à la notification `notify(e) ;` pour un certain évènement `e` consiste simplement en la notification KernelD de la variable associée à l'évènement.

$$\Pi(\texttt{notify(e) ;}, Q_\neg) \quad = \quad q_0 \xrightarrow{e!} q_f \quad \text{où } q_0 \text{ et } q_f \notin Q_\neg$$

Mise en attente. Comme les statuts SystemD et KernelD se correspondent directement, la traduction des mises en attente est directe. La mise en attente `halt()` consiste à attendre l'évènement spécial *halt*.

$$\Pi(\texttt{wait(}\omega\texttt{) ;}, Q_\neg) \quad = \quad q_0 \xrightarrow{\omega} q_f \quad \text{où } q_0 \text{ et } q_f \notin Q_\neg$$

$$\Pi(\texttt{halt() ;}, Q_\neg) \quad = \quad q_0 \xrightarrow{\textbf{halt}} q_f \quad \text{où } q_0 \text{ et } q_f \notin Q_\neg$$

Conditionnelle. Il y a deux formes de conditionnelles, les deux étant gardées par une expression. La première, de la forme `if (exp) { thread`$_1$ `} else { thread`$_2$ `}` est spécifiée par deux branches : une pour le cas où l'expression s'évalue à vraie, et l'autre pour le cas où l'expression s'évalue à faux. La seconde forme `if (exp) { thread }` est en réalité un cas particulier de la première forme où il n'y a pas d'instruction pour le cas où l'expression s'évalue à faux. La traduction de l'instruction conditionnelle `if (exp) { thread`$_1$ `} else { thread`$_2$ `}` consiste en une transition gardée par la traduction de l'expression `exp` qui mène à la traduction de `thread`$_1$ et une transition gardée par la traduction de l'expression `!exp` (le contraire de `exp`) qui mène à la traduction de `thread`$_2$. À la fin, les deux branches se rejoignent.

$$\Pi(\texttt{if (exp) \{ thread}_1\texttt{ \} else \{ thread}_2\texttt{ \}}, Q_\neg) = \text{voir figure 5.3}$$

où $\quad \Pi(\texttt{thread}_1, Q_\neg) = q_0^1 \xrightarrow{\texttt{thread}_1} q_f^1$

et $\quad \Pi(\texttt{thread}_2, Q_\neg \cup \text{Points}(q_0^1 \xrightarrow{\texttt{thread}_1} q_f^1)) = q_0^2 \xrightarrow{\texttt{thread}_2} q_f^2$

et $\quad \{q_0, q_f\} \cap (Q_\neg \cup \text{Points}(q_0^1 \xrightarrow{\texttt{thread}_1} q_f^1) \cup \text{Points}(q_0^2 \xrightarrow{\texttt{thread}_2} q_f^2)) = \emptyset$

Assertion. L'instruction `assert(exp) ;` se traduit simplement par une conditionnelle qui consiste à continuer le programme si l'expression s'évalue à *vrai* et à se diriger vers un point de contrôle spécial appelé **fail** sinon. Nous reviendrons sur les caractéristiques de **fail** dans la section 5.7.

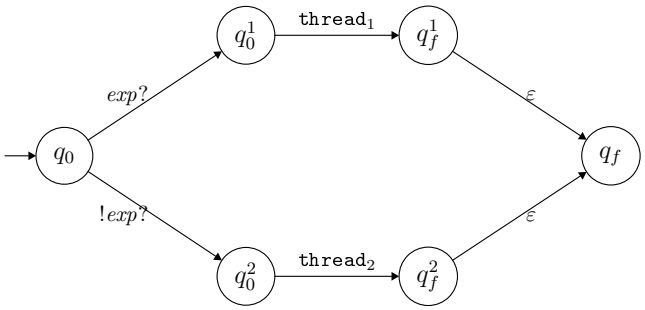

FIG. 5.3 – Traduction de la conditionnelle

Boucle. La traduction de la boucle est une première transition gardée par l'expression de boucle et dirigée vers le corps de la boucle et une transition gardée par l'inverse de l'expression de boucle et dirigée vers un nouvel état.

$$\Pi(\texttt{while (exp) \{ thread \}}, Q_\neg) = \text{voir figure 5.4}$$

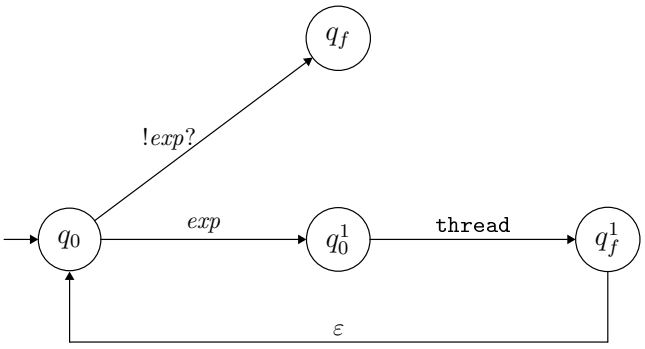

FIG. 5.4 – Traduction de la boucle

$$\begin{array}{ll} \text{où} & \Pi(\texttt{thread}, Q_\neg) = q_0^1 \xrightarrow{\texttt{thread}} q_f^1 \\ \text{et} & \{q_0, q_f\} \cap (Q_\neg \cup \text{Points}(q_0 \xrightarrow{\texttt{thread}} q_f)) = \emptyset \end{array}$$

Instruction vide. L'absence d'instruction se traduit par un programme contenant un point de contrôle initial, un point de contrôle puits, et une seule transition étiqueté par ε depuis le point initial vers le point puits.

$$\Pi(\ , Q_\neg) \quad = \quad q_0 \xrightarrow{\varepsilon} q_f \quad \text{où } q_0 \text{ et } q_f \notin Q_\neg$$

Séquence. Une suite d'instructions $\texttt{inst}_1 ... \texttt{inst}_n$ est traduite en la suite de la traduction de chacune des instructions.

$$\Pi(\texttt{inst}_1 ... \texttt{inst}_n, Q_\neg) \quad = \quad q_0^1 \xrightarrow{\texttt{thread}_1} q_f^1 \xrightarrow{\varepsilon} q_0^2 \ ... \ q_f^{n-1} \xrightarrow{\varepsilon} q_0^n \xrightarrow{\texttt{thread}_n} q_f^n$$

$$
\begin{array}{lll}
\text{où} & \Pi(\texttt{inst}_1, Q_\neg) = q_0^1 \xrightarrow{\texttt{thread}_1} q_f^1 \\
\text{et} & \Pi(\texttt{inst}_2, Q_\neg \cup \text{Points}(q_0^1 \xrightarrow{\texttt{thread}_1} q_f^1)) = q_0^2 \xrightarrow{\texttt{thread}_2} q_f^2 \\
\text{et} & ... \\
\text{et} & \Pi(\texttt{inst}_n, Q_\neg \bigcup_{1 \leq i \leq n-1} \text{Points}(q_0^i \xrightarrow{\texttt{thread}_i} q_f^i)) = q_0^n \xrightarrow{\texttt{thread}_n} q_f^n
\end{array}
$$

5.7 KernelD et les Logiques Temporelles

Cette section s'intéresse à l'expressivité des modules SystemD/KernelD en tant que propriétés logiques. En particulier, nous montrons quels en sont les liens avec les logiques temporelles [Pnu77b, MP81] qui sont le principal moyen d'expression de propriétés sur les systèmes. Comme nous le verrons, les programmes KernelD servent parfois à exprimer des propriétés de type *sûreté* [ADS85, AS86].

5.7.1 Point de Contrôle fail

Nous reprenons la notion de vérification décrite en 3. Celle-ci s'appuie sur un prédicat *Safe* qui permet de distinguer les configurations sûres des configurations erreur. En KernelD, ce prédicat se définit via un point de contrôle particulier, appelé **fail**, dans lequel tout programme à la possibilité de se rendre.

Paramètre 5.26 Point de Contrôle **fail** $\in Q$

C'est ce point de contrôle qui va jouer le rôle de validation d'un état. Le prédicat *Safe* sur les configurations se définit simplement en observant le point de contrôle associé à la configuration.

Définition 5.27 Configuration sûre $Safe \in \mathcal{C} \rightarrow Prop$

$$\text{Safe}(c) \triangleq c.q \neq \textbf{fail}$$

Lorsqu'un programme souhaite signaler une erreur de comportement de son système, il lui suffit de déplacer son contrôle en **fail**. Pour paramétrer ce déplacement par rapport à la mémoire courante, un programme peut utiliser l'instruction de garde qui joue alors le rôle de *propriété de l'état courant*. Par exemple, si l'état courant ne doit pas vérifier qu'une certaine variable x est supérieure à 1000, il suffit de créer une transition étiquetée par la valeur de comparaison entre x et 1000. Selon que cette comparaison donne un résultat vrai ou faux, le programme peut soit se déplacer en **fail**, soit poursuivre son exécution. Nous donnons en figure 5.5 l'exemple d'un programme qui vérifie à chaque fois qu'un certain évènement e est notifié, que la variable x n'est pas supérieure à 1000.

L'expressivité des propriétés d'états dépend donc de l'expressivité des expressions KernelD. Pour exemple, nous avons considéré les expressions arithmétiques et logiques. Dans ce cas, l'ensemble des propriétés d'états que l'on peut formuler en KernelD est donc les propriétés formulables par des prédicats sur des expressions arithmétiques et logiques sur les variables des programmes. Si dans l'état courant, l'expression `2*x*x == y - 10` est vraie, c'est que l'état courant satisfait la propriété $2 \times x^2 = y - 10$.

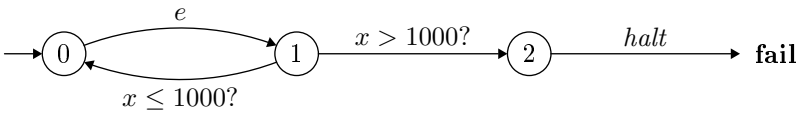

FIG. 5.5 – Une propriété en KERNELD

5.7.2 Sûreté et Validité

Vérifier qu'un module M est sûr va consister à vérifier que **fail** n'est pas atteignable par un des programmes de M, quelle que soit son exécution. On parle de sûreté *locale* puisqu'on ne s'occupe pas du comportement des programmes extérieur à M. En particulier, ceux-ci peuvent se trouver en **fail** sans pour autant que cela ne change la sûreté de M. On reprend les définitions du chapitre 3 et on les adapte des systèmes vers les modules.

Définition 5.28 État sûr $\mathit{Safe} \in \mathcal{M} \times \Sigma \to \mathit{Prop}$

$$\mathrm{Safe}(M, \sigma) \quad \triangleq \quad \forall \kappa \in K_M,\ \mathrm{Safe}(\sigma.C(\kappa).q)$$

Définition 5.29 Trace sûre $\mathit{Safe} \in \mathcal{M} \times \mathcal{T} \to \mathit{Prop}$

$$\mathrm{Safe}(M, t) \quad \triangleq \quad \forall i \in \mathbb{N},\ \mathrm{Safe}(M, t_i)$$

Le fait qu'une trace t est sûre vis-à-vis d'un module M est donc une propriété de la logique temporelle linéaire LTL (c'est une propriété qui ne concerne que cette exécution et pas la façon dont un bout de cette exécution peut brancher vers d'autres traces). C'est exactement la propriété $\mathbf{G}(\mathit{Safe}_\Sigma(M))$ où $\mathit{Safe}_\Sigma \in \mathcal{M} \to \Sigma \to \mathit{Prop}$ est définie par $\mathit{Safe}_\Sigma(M)(\sigma) = \mathit{Safe}(M, \sigma)$ (Safe_Σ currifie simplement les arguments de la fonction $\mathit{Safe} \in \mathcal{M} \times \Sigma \to \mathit{Prop}$).

$$\mathrm{Safe}(M, t) \quad \Leftrightarrow \quad M, t \models \mathbf{G}(\mathit{Safe}_\Sigma(M))$$

Ce résultat s'obtient directement d'après la définition de l'opérateur temporel $\mathbf{G}$ (*always*). Clairement cette propriété est une propriété de sûreté [Sch97], puisqu'elle énonce qu'« aucun état atteignable n'est erroné ». Continuons ce travail de traduction en reprenant la définition de la validité des systèmes et en l'adaptant pour les modules.

Définition 5.30 Module validant une propriété $\models \in \mathcal{M} \times \mathcal{M} \to \mathit{Prop}$

$$M \models M' \quad \triangleq \quad \forall t \in [\![M \otimes M']\!],\ \mathrm{Safe}(M, t) \Rightarrow \mathrm{Safe}(M', t)$$

En suivant la remarque selon laquelle la sûreté locale d'un module est une propriété de logique temporelle, la validité devient la propriété suivante :

$$M \models M' \quad \Leftrightarrow \quad M \otimes M' \models \mathbf{A}\ (\mathbf{G}(\mathit{Safe}_\Sigma(M)) \Rightarrow \mathbf{G}(\mathit{Safe}_\Sigma(M')))$$

Si cette propriété reste une propriété LTL, ce n'est plus une propriété de sûreté car ce n'est plus un seul invariant sur chaque état, mais une relation entre deux propriétés d'invariance sur les états.

Cependant, la vérification de la validité de certains systèmes peut se ramener à une propriété de sûreté. En effet, il peut arriver que le système M à valider ne contienne aucune assertion et ne représente donc pas une hypothèse logique. Ceci peut se vérifier syntaxiquement, en inspectant si M a un programme qui contient **fail**. Lorsqu'aucun programme de M ne contient **fail**, la sûreté des traces de M est automatiquement vérifiée et la formule $\mathbf{G}(Safe_{\Sigma}(M))$ est logiquement équivalente à la proposition toujours vraie **True**. Ainsi, la formule $\mathbf{G}(Safe_{\Sigma}(M)) \Rightarrow \mathbf{G}(Safe_{\Sigma}(M'))$ devient équivalente à $\mathbf{True} \Rightarrow \mathbf{G}(Safe_{\Sigma}(M'))$, elle-même équivalente à $\mathbf{G}(Safe_{\Sigma}(M'))$. Du coup, on obtient une simplification de la propriété décrivant la validité :

$$M \models M' \quad \Leftrightarrow \quad M \otimes M' \models \mathbf{A}\ (\mathbf{G}(Safe_{\Sigma}(M')))$$

La validité est donc une propriété de sûreté sous l'hypothèse que les modules à valider ne peuvent pas échouer, ce qui est bien souvent le cas et ce qui peut parfois se vérifier syntaxiquement.

Comme le contrôle d'un module peut évoluer en fonction des expressions du langage SystemD/KernelD, l'expressivité des formules sur les états de module est exactement celle des expressions du langage. Nous avons choisi pour illustration de pouvoir exprimer les expressions arithmétiques et logiques ; dans cet exemple, les propriétés que représentent les modules SystemD/KernelD sont donc des propriétés de sûreté dont les formules d'états sont les propositions arithmétiques et logiques. La question de la prouvabilité de ces propriétés sera traitée dans la partie III. Disons seulement que la difficulté réside dans la vérification d'une propriété d'état qui sera établie par interprétation abstraite et qui pourra donc produire des verdicts indécis.

Troisième partie

Analyse de SystemD

CHAPITRE 6

Model Checking et Interprétation Abstraite

Vérifier un module KernelD consiste à ajouter un système-propriété au module et à vérifier que le système ainsi complété ne peut pas générer d'erreur. C'est une approche à base d'assertions [FLK03, MMMC05], dans laquelle nous ajoutons de la compositionnalité par des remplacements à la assume-guarantee [HQR98, WL05]. Dans ce chapitre, nous montrons comment calculer de façon statique un sur-ensemble pertinent des comportements possibles d'un module KernelD. Pour cela, nous utilisons le Model Checking [EMC81, QS82] et l'Interprétation Abstraite [CC77, CC92] afin de déterminer un ensemble d'états abstraits atteignables par le système et les transitions entre ces états. La notion d'erreur et de violation d'une propriété sera encodée dans ces états abstraits si bien que la vérification se réduira alors à un problème de non-atteignabilité dans des chemins d'états abstraits [DG02]. Nous décrivons en fin de chapitre une analyse avant (et arrière sur les gardes) d'un module KernelD qui pourra servir à la vérification de propriété, de remplacement et de discrimination, et qui permettra l'application à SystemD de la méthodologie présentée au chapitre 3.

6.1 Pourquoi et Comment ?

La technique la plus répandue pour vérifier des propriétés d'un systèmes de composants est le *model checking*.

Cette technique consiste dans un premier temps à décrire le système étudié, de façon plus ou moins abstraite, comme avec un programme par exemple. On doit pouvoir déduire à partir de la description un certain nombre d'*états* du système, et l'ensemble des *transitions* entre ces états dans le système, c'est-à-dire quand est-ce qu'un état peut être obtenu à partir d'un autre par une action du système. Le résultat obtenu est ce que l'on appelle le *graphe d'états* du système. Une trace d'exécution du système est alors un chemin du graphe d'états.

Dans un second temps, on décrit une propriété temporelle du système. Elle est représentée par une propriété sur les chemins du graphe, comme le fait qu'un état particulier n'est jamais atteignable par exemple. Enfin, un algorithme permet de vérifier de façon automatique et statique que la description satisfait la propriété.

Ceci étant dit, pour que la vérification puisse se faire de façon statique, le système étudié doit comporter un nombre fini d'états. Cela afin que l'ensemble des états puissent être visités en un temps fini. C'est le cas lorsque le système est composé d'un nombre fini de variables dont les valeurs sont un domaine fini (les booléens par exemple).

Le model checking permet donc la vérification de plusieurs processus qui s'exécutent en parallèle et en concurrence, mais où le nombre d'états du système est fini, en utilisant des propriétés sur des chemins d'états.

D'un autre côté, une puissante technique automatique et statique de preuves de programmes est l'*interprétation abstraite*. Cette technique repose sur la notion d'*abstraction de point fixe*. L'idée est d'abstraire les valeurs des variables par des valeurs d'un domaine dans lequel on ne peut fusionner des valeurs différentes qu'un nombre fini de fois avant de trouver comme résultat une valeur déjà calculée. Cette technique a été appliquée avec succès dans le cadre de la preuve de programmes classiques où les programmes s'exécutent en séquence.

L'interprétation abstraite permet de vérifier un programme en attachant à chacun de ses points de contrôle une représentation finie d'un sur-ensemble des valeurs que peuvent prendre chaque variable en ce point.

Dans KernelD, les systèmes sont décrits par des modules. Une façon de faire du model checking d'un module KernelD serait simplement d'utiliser les états de module (une configuration du système et un état mémoire) et la transition sémantique de module pour construire un graphes d'états. Nous ne nous intéressons qu'aux descriptions finies. C'est-à-dire qu'un module est composé d'un nombre fini de programmes et que chaque programme est décrit par un nombre fini de points de contrôle et de transitions entre ces points, et s'appuie donc sur un nombre fini de variables et d'évènements. Ceci assure que le nombre de configurations d'un module est fini, et qu'on peut même déterminer ces informations de façon syntaxique, donc statique. Mais les valeurs que peuvent prendre les variables est quant à lui théoriquement infini, et il n'est pas possible dans le cas général de connaître de façon statique les valeurs que peuvent prendre chaque variable dans chaque configuration, à cause des *boucles* (ou *arcs arrières*) dans les programmes. Le nombre d'états d'un système peut être infini, ou très grand. On obtient donc un cadre dans lequel le model checking classique n'est pas adapté. Et du fait que plusieurs programmes s'exécutent en parallèle et en concurrence, le cadre de l'interprétation abstraite ne l'est pas directement non plus.

Nous allons présenter une technique qui mélange des techniques du model checking et de l'interprétation abstraite, comme on peut en trouver dans le model checking abstrait [CGL94, CC97]. Le model checking abstrait est proche du model checking en ceci qu'il s'intéresse aux propriétés de systèmes où plusieurs processus s'exécutent en parallèle et en concurrence. Et il est proche de l'interprétation abstraite en ceci qu'il permet de réduire des ensembles infinis (ou très grands) d'états des systèmes en fusionnant les états qui partagent des propriétés communes. L'interprétation abstraite introduit classiquement des points de contrôle aux programmes, qui

représentent une information de position, afin de les analyser et de calculer les valeurs que les variables peuvent prendre en ces points. Ici, nous analysons des systèmes. Nous pourrions choisir comme information de position une configuration pour chaque programme du module. En réalité, l'information de position sert surtout à désigner l'ensemble des éléments que l'on souhaite analyser de façon exacte. Nous choisissons de considérer les configurations de module comme information de position. Cela permet d'ajouter aux configurations de programme l'information de notification de chaque évènement. Au cours des analyses, nous resterons donc exacts sur l'ensemble des évènements qui ont été notifiés. Nous proposons de calculer les valeurs des variables de programme en chacune des configurations d'un module.

Afin d'obtenir des analyses assez précises avec des domaines relativement simples, nous ne pouvons pas considérer chaque programme comme une entité évoluant dans un environnement totalement aléatoire. Nous avons donc choisi une approche d'analyse assez proche du cas concret. Les programmes d'un module vont évoluer en parallèle dans l'abstraction, en oubliant les informations qui ne concernent pas le module en question : les configurations des programmes en dehors du module, les évènements et les variables que les programmes du module ne lisent pas et n'écrivent pas.

Dans un premier temps, nous montrons comment obtenir de façon statique un graphe d'états abstraits fini à partir d'une description KernelD, qui représente un sur-ensemble assez précis des états que peuvent donner une exécution réelle. L'abstraction permet de réduire le nombre d'états du système. Lorsque le model checking explose, l'abstraction peut donc parvenir à des résultats en introduisant des approximations. À cela, on combine l'ajout de programmes discriminants dans un module afin de réintroduire de la précision en augmentant le nombre de configurations du module. Ces deux techniques se complètent : l'abstraction réduit le nombre d'états par approximations, et les discriminants augmentent le nombre d'états en précisant le comportement du module.

Dans un second temps, nous pourrons montrer comment déduire des propriétés de la description initiale à partir du graphe d'états abstraits, sans avoir à définir des propriétés de façon ad hoc, comme le nécessite le model checking.

6.2 Graphe d'États

La première étape du model checking d'un système consiste à construire le graphe d'états du système. C'est un graphe dont chaque sommet est un état possible du système au cours de ses exécutions, et où chaque arête représente une transition possible du système d'un état vers un autre. En KernelD, les modules évoluent dans un environnement qui contient toutes les variables possibles. Les états d'un module sont donc l'ensemble Σ des états tout entier. Cet état contient les configurations de tous les programmes, les évènements notifiés et la valeur de chaque variable, que chacun de ces objets soit interne ou externe au module. Et la relation de transition d'un module est exactement sa transition sémantique. Le graphe d'états d'un module représente exactement sa sémantique et réciproquement. En effet, les chemins du graphe sont les traces d'exécution du module.

6.2.1 Graphe d'États Collectés

En pratique, lorsque l'on étudie les comportements d'un système, il devient vite nécessaire de pouvoir conduire les analyses non pas sur une exécution particulière, ni sur toutes les exécutions possibles, mais sur des ensembles particuliers d'exécutions. Par exemple, on peut vouloir considérer une configuration initiale, et au lieu d'étudier le système lorsque son entrée vaut exactement 5, on pourrait souhaiter l'étudier lorsque son entrée est positive. Pour cela, on introduit une sémantique dite *collectrice* du système. En KernelD, cette sémantique collectrice consiste à associer à une position du module — nous avons choisi les configurations du module comme information de position — l'ensemble des états mémoire qui peuvent passer par cette configuration. On peut alors déduire un *graphe d'états collectés* où chaque sommet représente un ensemble d'états et où une arête correspond à une transition possible du module entre des états de deux sommets. De plus, dans un tel graphe, chaque configuration de module est représentée exactement une fois par un sommet. On peut donc représenter cette structure par un graphe où les sommets sont les configurations de module. Des transitions partent de certaines pour arriver à d'autres. Et on attache une information à chacun de ces sommets-configurations : un ensemble d'états mémoire. Cette dernière information peut être vue comme une propriété : un environnement où la variable x_1 a pour valeur v_1 et x_2 a pour valeur v_2 est exactement la propriété qui énonce que la valeur de $x_1 = v_1$ et celle de $x_2 = v_2$. Avec un ensemble d'environnement ρ_1, ρ_2, ..., à partir desquels on construit les propriétés P_1 pour ρ_1, P_2 pour ρ_2, etc, on peut construire la propriété qui énonce que P_1 ou P_2 ou etc. On obtient une sorte de *structure de Kripke*, le graphe de travail du model checking, à ceci près que nos propriétés ne sont pas finiment représentable. En effet, les variables peuvent être en nombre infini, ainsi que les environnements possiblement collectés en une certaine configuration.

Définition 6.1 État collecté $\sigma^* = (r, \rho^*) \in \Sigma^* \triangleq \mathcal{R} \times \mathcal{P}(\Gamma)$

Notation : étant donné un état collecté $\sigma^* = (r, \rho^*)$, nous utiliserons $\sigma^*.r$ pour désigner r et $\sigma^*.\rho^*$ pour désigner ρ^*. On note $\cup^*$ l'opération qui consiste à fusionner les états mémoire de plusieurs ensembles d'états mémoire : $\cup^*\{\rho_1^*, \rho_2^*, ...\} = \rho_1^* \cup \rho_2^* \cup ...$

Définition 6.2 Graphe d'états collectés $\langle F, T \rangle \in (\mathcal{R} \to \mathcal{P}(\Gamma)) \times \mathcal{P}(\mathcal{R} \times \mathcal{R})$.

Un graphe d'états collectés est une fonction F qui associe à chaque configuration un ensemble d'environnements et une relation T de transition entre les configurations.

Pour savoir si un état collecté est accessible depuis un autre, on définit la *transition collectrice* d'un système. Deux états collectés sont liés par une transition collectrice d'un système si l'état de système obtenu par la configuration de l'un et un état mémoire qui lui est associé peut atteindre par une transition concrète du système un état formé par la configuration de l'autre et un état mémoire qui lui est associé.

Définition 6.3 Transition collectrice $\xrightarrow{\cdot}^* \in \mathcal{S} \to \mathcal{P}(\Sigma^* \times \Sigma^*)$

$$\xrightarrow{S^*} \triangleq \{(\sigma_1^*, \sigma_2^*) \mid \exists \rho_1 \in \sigma_1^*.\rho^*,\ \exists \rho_2 \in \sigma_2^*.\rho^*,\ (\sigma_1^*.r, \rho_1) \xrightarrow{S} (\sigma_2^*.r, \rho_2)\}$$

La *sémantique collectrice* d'un module M à partir d'une décoration initiale F_0 est le graphe d'états collectés $\langle F, T\rangle$ où $T = \{(r, r') \mid (r, F(r)) \xrightarrow{M^*} (r', F(r'))\}$ et F est la plus petite solution du système récursif d'équations sémantiques suivant :

$$\textbf{Équations sémantiques :} \quad \forall r \in \mathcal{R},\ F(r) = F_0(r) \cup^* \{\rho^* \mid \exists r',\ (r', F(r')) \xrightarrow{M^*} (r, \rho^*)\}$$

Dans des langages de programmation plus classiques, la relation entre un point de contrôle et le suivant est déjà connu. Dans notre cas, on ne connaît pas a priori les transitions entre deux configurations. En effet, celles-ci dépendent des états mémoire qui leur sont associés.

Un rappel sur les ensembles ordonnés se trouve en annexe A.4. En particulier, on y trouve le théorème de Tarski nous assurant qu'il existe une solution au système ci-dessus. En effet, $\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap\rangle$ est un treillis complet, et pour tout $r \in \mathcal{R}$, la fonction $F(r)$ qui dépend des $F(r')$ est croissante par rapport à $\subseteq$. C'est-à-dire que si les ensembles associés à chaque $F(r')$ grandissent, alors $F(r)$ grandira aussi. Le théorème de Kleene qui se trouve également en annexe énonce que pour résoudre ce système, une méthode correcte procède par itérations successives pour propager les informations nouvelles à chaque étape.

Mais si la solution existe, rien ne garantit qu'elle soit calculable en temps fini. Et même si le temps de calcul peut être fini, il risque d'être beaucoup trop grand en pratique. L'interprétation abstraite pallie à ces deux problèmes en proposant de calculer un sur-ensemble des solutions à l'aide d'une représentation finie des ensembles d'environnements et d'opérations qui assurent qu'on ne peut faire l'union des représentations qu'un nombre fini de fois avant de tomber sur une valeur précédemment calculée (point fixe).

6.2.2 Sémantique Abstraite

La sémantique abstraite d'un module KernelD est un graphe d'états abstraits, c'est-à-dire une représentation finie et calculable en temps fini d'un graphe d'états collectés. Une première abstraction est faite au niveau des configurations : lorsqu'on analyse un module, les configurations des programmes des autres modules n'interviennent pas directement sur son comportement. On se contente de les oublier. Le deuxième niveau d'abstraction concerne les configurations et le temps. Les sommets du graphe d'états abstraits vont représenter des ensembles de configurations. Lorsqu'on oublie les configurations des autres modules, il reste des configurations dont les statuts des programmes peuvent faire référence au temps. Comme on ne sait pas quelle sera le plus court instant des programmes puisqu'on en a oublié, il va falloir toutes les considérer. Pour éviter la quasi-duplication inutile que cela engendrerait, nous fusionnons les configurations dont la seule différence réside dans le temps d'attente de leurs programmes. On peut alors décorer les configurations abstraites du module par des états mémoire abstraits. Chaque état mémoire abstrait représente un ensemble d'états mémoire. Cette décoration est accompagnée par la relation de transition abstraite qui lie les configurations abstraites du module. Notre système d'équations sémantiques reste donc le même à ceci près que l'on considère :

- les configurations restreintes aux programmes du module avec abstraction du temps au lieu des configurations de tous les programmes sans abstraction du temps ;
- un état mémoire abstrait au lieu d'un ensemble d'états mémoire ;
- une transition abstraite du module au lieu d'une transition collectrice.

Configuration restreinte

Définition 6.4 Configuration restreinte d'un module $r_M = (C, E) \in \mathcal{R}_M$

$$\mathcal{R}_M \triangleq (K_M \to \mathcal{C}) \times \mathcal{P}(\mathcal{E}_M)$$

où $\mathcal{E}_M = \mathcal{E} \cap (R(M) \cup W(M))$ est l'ensemble des évènements lus et notifiés par des programmes de M.

Notation : étant donné un module $M = \langle L, K \rangle$, l'ensemble $K \to \mathcal{C}$ des fonctions qui associe une configuration à chaque programme de M est formé d'éléments qui peuvent être vus comme des tableaux à domaine fini lorsque le nombre de programme dans M est fini. Pour distinguer une fonction C restreinte aux programme d'un module d'une fonction totale, on note $C[\kappa]$ la configuration associée au programme κ. Dans la suite, cette notation vaudra pour toutes les fonctions dont le domaine est fini.

On rappelle que la configuration associée à un programme dans une configuration restreinte représente en réalité un ensemble de configurations. Les références au temps ne sont plus fixes, mais représentent des intervalles. Par exemple, 8 représente un ensemble de statuts τ où τ est compris entre 0 et 7 (à 8 la configuration change). De plus, une configuration restreinte représente également les configurations totales où les programmes qui ne sont pas présents dans la configuration restreinte peuvent prendre n'importe quelle configuration. De même, les évènements qui ne sont pas représentés peuvent être présents ou non. Nous définissons une fonction $\mathit{Restrict}^\sharp$ qui permet de passer d'une configuration totale à une configuration restreinte pour un module et une fonction $\mathit{Complete}^\sharp$ qui permet à l'inverse de connaître l'ensemble des configurations totales représentées par une configuration restreinte. Pour la fonction $\mathit{Complete}^\sharp$, les statuts des programmes du module considéré peuvent devenir n'importe quel statut réduit d'une fraction de temps comprise entre 0 et le temps minimal (exclus) apparaissant dans les statuts courants des programmes du module.

Définition 6.5 Restriction d'une configuration $\mathit{Restrict}^\sharp \in \mathcal{R} \to \mathcal{R}_M$

$$\begin{array}{rcll} \mathrm{Restrict}^\sharp(r) & \triangleq & r_M & \text{où} \quad \forall \kappa \in K_M,\ r_M.C[\kappa] = r.C(\kappa) \\ & & & \text{et} \quad \forall e \in \mathcal{E}_M,\ e \in r_M.E \Leftrightarrow e \in r.E \end{array}$$

Définition 6.6 Complétion d'une configuration $\mathit{Complete}^\sharp \in \mathcal{R}_M \to \mathcal{P}(\mathcal{R})$

$$\mathrm{Complete}^\sharp(r_M) \triangleq \left\{ r \;\middle|\; \begin{array}{l} \forall e \in \mathcal{E}_M,\ e \in r.E \Leftrightarrow e \in r_M.E \\ \text{et} \quad \exists \tau,\ 0 \leq \tau < \tau_{\min}(r_M.C) \text{ et } \forall \kappa \in K_M,\ r_M.C[\kappa].\omega \rhd_\tau r.C(\kappa).\omega \end{array} \right\}$$

Équations sémantiques abstraites

Maintenant, passons à la représentation d'un ensemble d'états mémoire. Les représentations utilisées pour abstraire un ensemble d'éléments s'appelle un *domaine abstrait*. Un domaine abstrait $\mathcal{D}$ doit être muni d'un ordre partiel $\sqsubseteq \in \mathcal{D} \times \mathcal{D} \to \mathit{Prop}$, d'opérations de plus petite borne supérieure $\sqcup$ et de plus grande borne inférieure $\sqcap \in \mathcal{P}(\mathcal{D}) \to \mathcal{D}$, d'un plus grand

élément $\top$ et d'un plus petit élément $\bot \in \mathcal{D}$. Si $\mathcal{D}$ permet de représenter des ensembles d'environnements, le problème d'analyse d'un module consiste à déterminer une décoration $F^\sharp \in \mathcal{R}_M \to \mathcal{D}$ satisfaisant le système d'équations sémantiques abstraites suivant :

Équations sémantiques abstraites :

$$\forall r_M \in \mathcal{R}_M,\ F^\sharp[r_M] = F_0^\sharp[r_M] \sqcup \{\rho^\sharp \mid \exists r'_M,\ (r_M, \rho^\sharp) \in [\![M]\!]_1^\sharp(r'_M, F^\sharp[r'_M])\}$$

où $F_0^\sharp$ est une décoration initiale et $[\![M]\!]_1^\sharp$ est la transition abstraite du module M qu'il nous reste à définir. La calculabilité d'une solution à ce système dépend en partie de ce que les configurations d'un module sont finies : on peut donc les énumérer.

Si le domaine $\langle \mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ est un treillis complet, et que la fonction $[\![M]\!]_1^\sharp$ préserve l'accessibilité des configurations et est croissante sur les états mémoire abstraits, on peut encore une fois utiliser le théorème de Tarski et on peut conclure qu'il existe une solution au système d'équations sémantiques abstraites. Le théorème de Kleene nous en donnera le plus petit point fixe. Mais rien n'interdit que le domaine soit infini. Et dans ce cas, il est encore possible que les équations ne se stabilisent pas en temps fini. On peut alors fournir un *opérateur d'élargissement*, noté ∇ et défini en annexe A.4, qui servira à forcer la convergence vers un point fixe (qui ne sera plus obligatoirement le plus petit) en un nombre fini d'étapes, généralement en poussant les valeurs vers des extrêmes. Pour raffiner le point fixe déterminé par élargissement, on s'appuie sur un *opérateur de rétrécissement*, noté $\triangle$ et lui aussi défini en annexe, qui servira justement à diminuer les imprécisions autour des valeurs extrêmes obtenues par élargissement.

Le couple $\langle F^\sharp, T^\sharp \rangle$ où $F^\sharp$ est une solution du système d'équations sémantiques abstraites et $T^\sharp = \{(r_M, r'_M) \mid \exists \rho^\sharp,\ (r'_M, \rho^\sharp) \in [\![M]\!]_1^\sharp(r_M, F^\sharp[r_M])\}$ définit un *graphe d'états abstraits* qui représente un graphe d'états collectés. Pour s'assurer que le graphe d'états abstraits ainsi calculé représente au moins plus de comportements que le graphe d'états collectés d'un module, l'interprétation abstraite propose un critère basé sur une fonction d'abstraction $\alpha \in \mathcal{P}(\Gamma) \to \mathcal{D}$ et sur une fonction de concrétisation $\gamma \in \mathcal{D} \to \mathcal{P}(\Gamma)$ qui doivent satisfaire les propositions suivantes :
– α est croissante : $\forall \rho_1^*\ \rho_2^* \in \mathcal{P}(\Gamma),\ \rho_1^* \subseteq \rho_2^* \Rightarrow \alpha(\rho_1^*) \sqsubseteq \alpha(\rho_2^*)$;
– γ est croissante : $\forall d_1\ d_2 \in \mathcal{D},\ d_1 \sqsubseteq d_2 \Rightarrow \gamma(d_1) \sqsubseteq \gamma(d_2)$;
– α et γ sont compatibles : $\forall \rho^* \in \mathcal{P}(\Gamma)\ d \in \mathcal{D},\ \alpha(\rho^*) \sqsubseteq d \Leftrightarrow \rho^* \subseteq \gamma(d)$
On dit alors que les fonctions α et γ forment une correspondance de Galois entre les ensembles partiellement ordonnés $(\mathcal{P}(\Gamma), \subseteq)$ et $(\mathcal{D}, \sqsubseteq)$, ce que l'on note $(\mathcal{P}(\Gamma), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}, \sqsubseteq)$.

Calcul du graphe d'états abstraits

L'algorithme suivant résume les différentes étapes de calcul du graphe d'états abstraits d'un module.

Algorithme 6.7 (Calcul du graphe d'états abstraits d'un module)

1. Construction des configurations des programmes du module. Syntaxiquement et pour chaque programme, on détermine à chacun de ses points de contrôle quels sont les statuts qu'il est susceptible d'attendre.

2. Les configurations des programmes et les évènements lus et notifiés du module permettent de construire une table (un ensemble fini) des configurations restreintes du module.
3. Construction des configurations restreintes du module. À chaque configuration restreinte, on associe un environnement abstrait initial.
4. Point fixe. On applique un algorithme de calcul de point fixe qui va créer les transitions entre les configurations restreintes et établir quels environnements abstraits peuvent atteindre chacune d'entre elles. Pour faire cette propagation d'informations, il est possible d'utiliser différents types d'itération (Jacobi, chaotique [Nil89]).
5. Résultat. L'algorithme retourne le graphe d'états abstraits ainsi obtenu par point fixe.

Exemple : on considère un module M formé de deux programme κ_1 et κ_2. Le programme κ_1 n'a qu'un seul point de contrôle où il peut être elligible ou en attente d'un évènement e. Le programme κ_2 n'a lui aussi qu'un seul point de contrôle où il peut être elligible ou en attente de 10 unités de temps. Chaque programme ne peut avoir que 2 configurations, si bien que le nombre total de combinaisons de configurations des programmes est de $2 \times 2 = 4$. Le module ne lit et n'écrit qu'un seul évènement : e, qui peut donc être présent ou non (2 possibilités). Si bien qu'en tout, il y a $4 \times 2 = 8$ configurations restreintes possibles pour le module M. Enfin, le module lit ou écrit deux variables : x et y. On représente un graphe d'états abstraits initial pour l'analyse de M en figure 6.1. Toutes les configurations y sont représentées et on leur associe un environnement abstrait initial qui donne une valeur abstraite pour x et une pour y. La configuration où κ_1 est en son point 0 et elligible, et κ_2 en son point 0 et elligible est la configuration restreinte initiale du module.

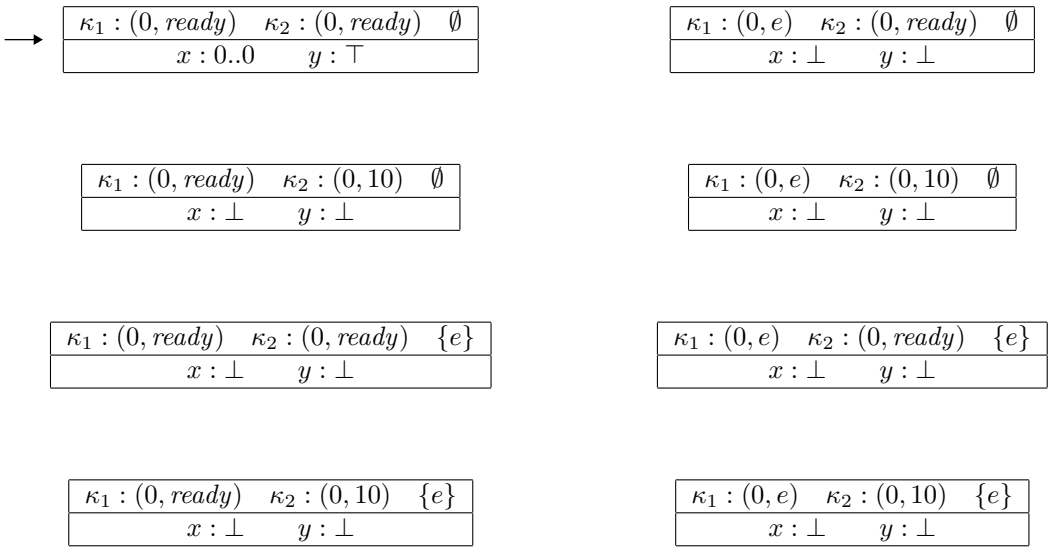

FIG. 6.1 – Exemple de graphe d'états abstraits (avant point fixe)

À partir de ce graphe, l'algorithme 6.7 au point 4 va créer des transitions entre les configurations restreintes de M et calculer un sur-ensemble des valeurs que peuvent prendre x et y en ces configurations. Le résultat est donné en figure 6.2, où seules les configurations atteignables depuis la configuration initiale sont représentées.

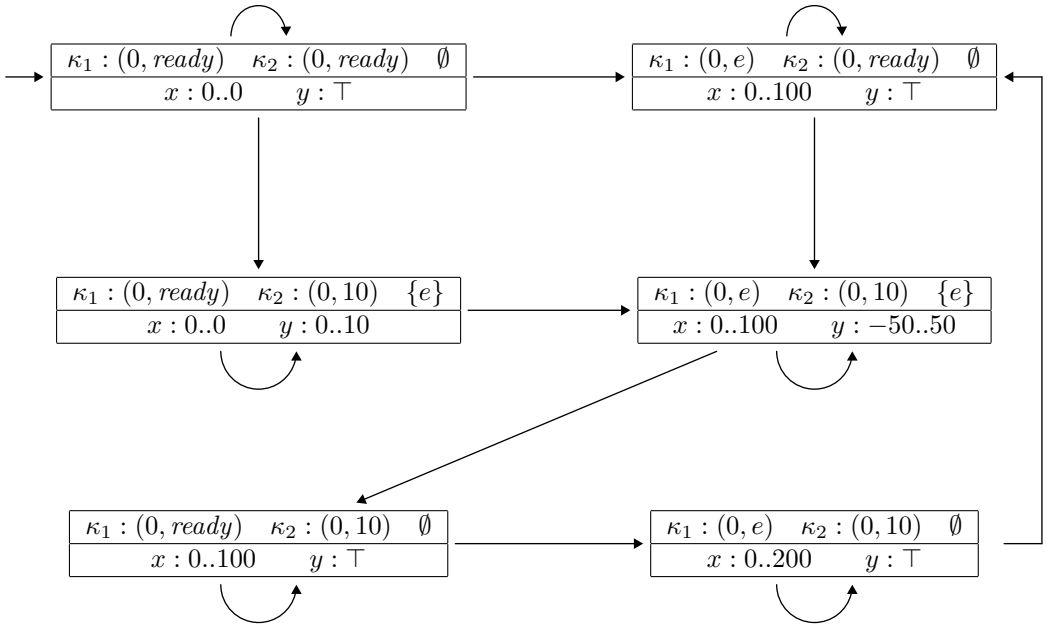

FIG. 6.2 – Exemple de graphe d'états abstraits (après point fixe)

Le déroulement de l'algorithme sera un peu plus détaillé dans le chapitre 8 sur les expérimentations. Ici, nous souhaitons seulement montrer la forme des résultats de l'algorithme. Nous insistons particulièrement sur le fait que le graphe est fini.

Correction

La correction d'une analyse sur un module M est donnée par la proposition suivante, où $\langle F, T \rangle$ est la sémantique collectrice de M à partir de la décoration initiale F_0, et $\langle F^\sharp, T^\sharp \rangle$ est la sémantique abstraite de M à partir de la décoration abstraite initiale $F_0^\sharp$ par l'algorithme 6.7 :

$$\forall r_M \in \mathcal{R}_M,\ F_0^\sharp[r_M] = \alpha(\cup^*\{F_0(r) \mid \text{Restrict}^\sharp(r) = r_M\}) \quad \Longrightarrow$$

$$\left\{ ((r,\rho),\ (r',\rho')) \quad \middle| \quad \begin{array}{ll} & (r,r') \in T \\ \text{et} & \rho \in F(r) \text{ et } \rho' \in F(r') \end{array} \right\}$$

$$\subseteq$$

$$\left\{ ((r,\rho),\ (r',\rho')) \quad \middle| \quad \exists r_M\ r'_M \in \mathcal{R}_M, \quad \begin{array}{ll} & (r_M, r'_M) \in T^\sharp \\ \text{et} & r \in \text{Complete}^\sharp(r_M) \text{ et } r' \in \text{Complete}^\sharp(r'_M) \\ \text{et} & \rho \in \gamma(F^\sharp[r_M]) \text{ et } \rho' \in \gamma(F^\sharp[r'_M]) \end{array} \right\}$$

Cette proposition signifie que toutes les transitions concrètes possibles d'un état vers un autre obtenues à partir d'une décoration initiale sont représentées par la transition abstraite obtenue à partir de l'abstraction de la décoration initiale.

Il nous reste à définir $[\![M]\!]_1^\sharp$ pour obtenir une abstraction correcte des graphes d'états collectés par des graphes d'états abstraits. La précision des analyses dépendra de cette définition et du domaine $\mathcal{D}$. Un domaine très précis permet d'obtenir des résultats très précis mais au prix d'une complexité plus importante. Réciproquement, un domaine trop grossier permet de déterminer rapidement un point fixe mais contre une perte de précision. Enfin, la façon de définir la transition abstraite permet d'obtenir des analyses produisant des types résultats différents. Nous présenterons en particulier un générateur de post-condition avec pré-condition au niveau des gardes.

Nous donnons une vue d'ensemble sur les graphes mis en jeu en figure 6.3, qui fait la relation entre les graphes d'états concrets, les graphes d'états collectés et les graphes d'états abstraits lorsqu'on analyse un certain module M. Une seule variable est étudiée, donc les états mémoire sont représentés par une seule valeur : celle que porte la variable. Un état du système est un couple formé d'une configuration du système (r_0, r_1, etc) et d'un état mémoire, c'est-à-dire une valeur pour la variable. Les valeurs $-5..10$, $0..0$, $\infty..100$ et $-10..\infty$ sont des valeurs du domaine des intervalles, qui est détaillé en annexe A.6 et dont nous nous servons à titre d'exemple. Ce que l'on voit apparaître sur le graphe est un résumé de la problématique et de la solution apportée : le graphe des états concrets contient un nombre infini d'états. Ceux que nous avons symbolisés par « ... » contiennent les états $(r_2, -51)$, $(r_2, -52)$, etc, mais aussi une infinité d'autres états aux configurations différentes (à cause des programmes externes au module). Le graphe des états collectés contient lui aussi un nombre infini d'états : chaque configuration est représentée exactement une fois, mais elles peuvent être en nombre infini. On voit qu'on accumule les valeurs de la variable pour chaque configuration. En r_2, la variable peut prendre les valeurs 100, -50, -51, etc. Le graphe d'états abstraits contient un nombre fini d'états. Ceci s'obtient en filtrant chaque configuration : on ne garde que les informations relatives aux programmes du module M. Par exemple, les configurations complètes r_2 et r_2' ne se distinguent que sur des programmes et évènements en dehors de M. Si on les restreint sur M, elles contiennent les mêmes informations, qu'on a noté r_2^-. On procède ainsi pour les autres configurations et on abstrait les valeurs de la variable par des intervalles qui représentent au moins autant de valeur que concrètement, potentiellement plus.

6.3 Éléments Communs aux Analyses

Cette section présente l'application du model checking et de l'interprétation abstraite au langage SystemD/KernelD. Nous montrons en particulier comment utiliser l'algorithme 6.7 de calcul du graphe d'états abstraits d'un module pour inférer un sur-ensemble des comportements d'un module par une analyse avant. Cette analyse servira alors de base pour des algorithmes de vérification de la validité, du remplacement et de la discrimination comme décrits au chapitre 3.

Mettre en place une analyse, quel que soit le type d'analyse, aura toujours pour but de calculer une transition abstraite d'un module. Le cadre concret nous oblige à construire ce calcul sur les notions de transition Univers, transition ordonnanceur et transition des programmes locaux. De plus, la structure d'un programme se base sur des points de contrôle et sur des transitions entre ces points. Là encore, nous allons pouvoir utiliser l'interprétation abstraite pour calculer une décoration de ces points ; cette méthode est commune à tous les types d'analyse.

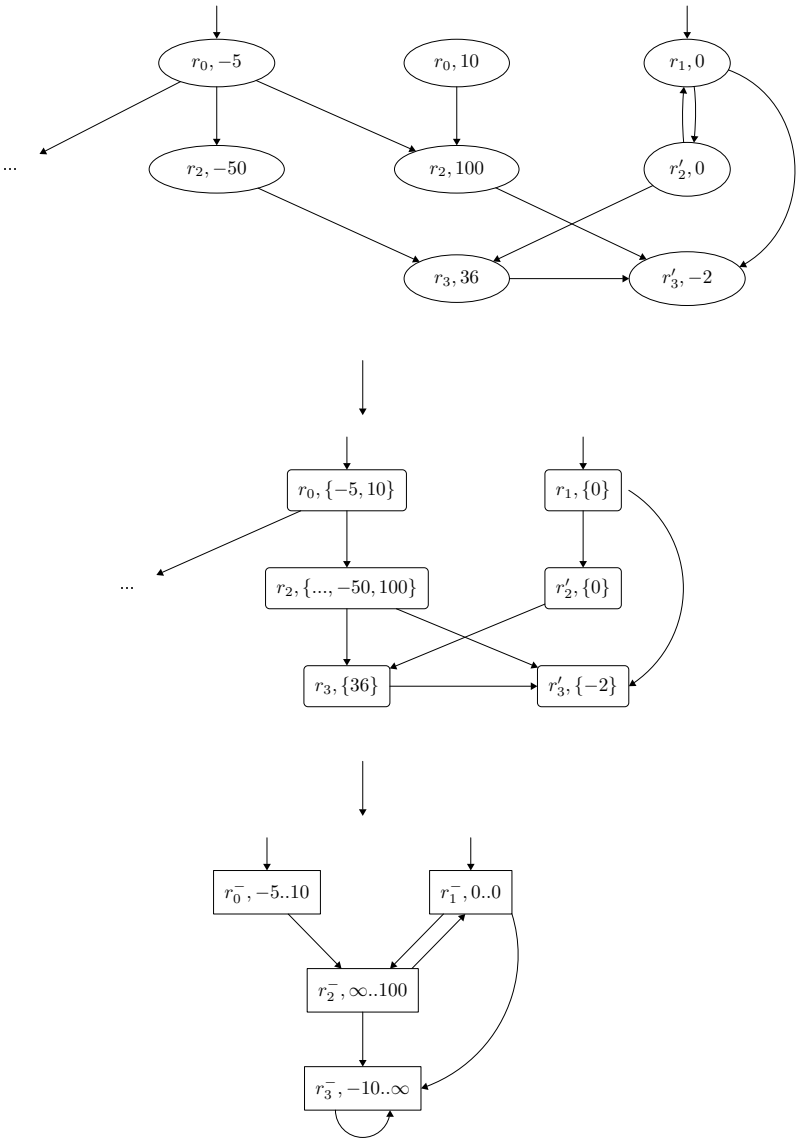

FIG. 6.3 – Exemples de graphes d'états concrets → collectés → abstraits

Dans la suite, on raffine un peu l'abstraction des états mémoire. En réalité, nos analyses ne s'appuient que sur l'abstraction de l'ensemble $\mathcal{V}$ des valeurs des variables. À partir de cette abstraction, on construit les états mémoire abstraits (ou environnements abstraits) comme des fonctions des variables vers un élément du domaine abstrait $\mathcal{D}$ des valeurs. La structure obtenue est un treillis dès lors que le domaine d'abstraction des valeurs en est un (voir l'annexe A.7 pour plus de détails sur la composition de treillis). L'état abstrait d'un module est une configuration restreinte aux variables et évènements lus et écrits par le module, accompagnée d'un état mémoire abstrait.

Définition 6.8 État mémoire abstrait $\rho^\sharp \in \Gamma^\sharp \triangleq \mathcal{X} \to \mathcal{D}$

Définition 6.9 État abstrait d'un module $\sigma^\sharp = (r_M, \rho^\sharp) \in \Sigma^\sharp_M \triangleq \mathcal{R}_M \times \Gamma^\sharp$

Notation : étant donné un état abstrait $\sigma^\sharp = (r_M, \rho^\sharp)$, nous utiliserons $\sigma^\sharp.r_M$ pour désigner r_M et $\sigma^\sharp.\rho^\sharp$ pour désigner $\rho^\sharp$.

Définition 6.10 Graphe d'états abstraits d'un module

$$\langle F^\sharp, T^\sharp \rangle \in \mathcal{G}_M \triangleq (\mathcal{R}_M \to \Gamma^\sharp) \times \mathcal{P}(\mathcal{R}_M \times \mathcal{R}_M)$$

On cherche à définir la fonction $[\![M]\!]^\sharp_1$ de telle sorte qu'elle soit calculable et respecte les conditions nécessaires à ce que les analyses présentées en section 6.2 soient correctes. La sémantique d'un module s'appuie sur trois transitions : l'Univers, l'ordonnanceur et les programmes locaux. La sémantique abstraite doit représenter une accumulation de ces trois transitions. Nous la définissons comme une fonction qui étant donné un état abstrait retourne un ensemble fini d'états abstraits accessibles en une transition abstraite du module à partir de l'argument.

Définition 6.11 Transition abstraite d'un module $[\![.]\!]^\sharp_1 \in \mathcal{M} \to \Sigma^\sharp_M \to \mathcal{P}(\Sigma^\sharp_M)$

$$[\![M]\!]^\sharp_1(\sigma^\sharp) \triangleq \text{Univers}^\sharp(M, \sigma^\sharp) \cup \text{Local}^\sharp(M, \sigma^\sharp) \cup \{(r_M, \sigma^\sharp.\rho^\sharp) \mid r_M \in \text{Sched}^\sharp(\sigma^\sharp.r_M)\}$$

La sémantique abstraite des transitions Univers, ordonnanceur et des programmes se définit à partir de leur sémantique concrète. Si on observe la sémantique concrète de la transition *Local* (cf chapitre 2), on s'aperçoit qu'elle dépend de la sémantique des processus d'un système, et donc des programmes d'un module en KernelD. Les programmes sont définis par des points de contrôle et des transitions entre ces points. On retrouve la même problématique que pour les modules : il est impossible de connaître statiquement quelles valeurs peut prendre une variable en un point donné durant une exécution, à cause des boucles. On résout ce problème de la même manière : on s'appuie sur l'interprétation abstraite en définissant une *sémantique collectrice d'un programme* puis une *sémantique abstraite d'un programme* qui représentera un sur-ensemble des environnements qui peuvent passer par un point de contrôle.

Pendant une analyse, on souhaite garder exacte les informations de configurations des programmes et les évènements notifiés. La sémantique collectrice d'un programme doit donc garder exacte la configuration du programme et les évènements émis. Le grand pas d'un programme se fait par petits pas successifs jusqu'à arriver dans une configuration d'attente. On commence donc par définir la *transition collectrice d'un programme* qui lie des ensembles d'états mémoire distants d'un petit pas d'un programme.

$$\xrightarrow{\cdot}{}^*_1 \in \mathcal{K} \to \mathcal{P}((\mathcal{C} \times \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Gamma)) \times (\mathcal{C} \times \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Gamma)))$$

$$(c, E, \rho^*) \xrightarrow{\kappa^*}_1 (c', E', \rho^{*\prime}) \triangleq \exists \rho \in \rho^*,\ \exists \rho' \in \rho^{*\prime},\ (c, E, \rho) \xrightarrow{\kappa}_1 (c', E', \rho')$$

À partir de cette relation, on peut établir le système d'équations sémantiques à résoudre pour connaître la sémantique collectrice d'un programme κ qui donne une décoration f sur les points de contrôle d'un programme en restant exact sur les évènements émis :

$$\forall c \in \mathcal{C}\ E \in \mathcal{P}(\mathcal{E}),\ f(c, E) = f_0(c, E) \cup^* \{\rho^* \mid \exists c'\ E',\ (c', E', f(c', E')) \xrightarrow{\kappa^*}_1 (c, E, \rho^*)\}$$

où f_0 est une décoration initiale.

Le grand pas collecteur d'un programme à partir d'une configuration c, d'un ensemble d'évènements E et d'un ensemble d'environnements ρ^* s'obtient à partir d'une solution f de ce système sur la décoration initiale f_0 telle que $f_0(c, E) = \rho^*$ et $f_0(c', E') = \emptyset$ dans tous les autres cas, et dans laquelle on ne garde que les configurations d'attente.

$$\stackrel{\cdot}{\longrightarrow}^* \in \mathcal{K} \to \mathcal{P}((\mathcal{C} \times \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Gamma)) \times (\mathcal{C} \times \mathcal{P}(\mathcal{E}) \times \mathcal{P}(\Gamma))$$

$$\stackrel{\cdot}{\longrightarrow}^* \triangleq \{((c, E, \rho^*), (c', E', f(c', E'))) \mid c'.\omega \neq \mathit{ready}\}$$

Remarque : le grand pas concret d'un programme ne prend pas en argument un ensemble d'évènements. Ceci permettait d'assurer structurellement que des évènements peuvent être émis mais qu'ils ne peuvent pas être retirés de l'ensemble des évènements notifiés. Le grand pas collecteur respecte cela puisqu'il s'appuie sur le petit pas concret des programmes. Cependant, pour une question de symétrie, nous choisissons de définir le grand pas collecteur comme une accumulation des évènements émis. Cela permet de conserver le même cadre pour différents types d'analyses, par exemple celles qui vont dans le sens du flot de contrôle et celles qui vont dans le sens inverse.

On passe dans le cadre abstrait en suivant ces définitions. Le *grand pas abstrait* $[\![\kappa]\!]^\sharp$ d'un programme dépend du *petit pas abstrait* $[\![\kappa]\!]^\sharp_1$ du programme qu'il nous faudra définir. Ce petit pas sera une représentation d'un sur-ensemble de toutes les configurations, évènements émis et environnements qui peuvent être obtenus par un petit pas concret.

La décoration abstraite $f^\sharp$ d'un programme κ d'un module M est un point fixe du système d'équations sémantiques abstraites suivant :

$$\forall c \in C(\kappa)\ E \in \mathcal{P}(\mathcal{E}_M),\ f^\sharp[c, E] = f_0^\sharp[c, E] \sqcup \{\rho^\sharp \mid \exists c'\ E',\ (c, E, \rho^\sharp) \in [\![\kappa]\!]^\sharp_1(c', E', f^\sharp(c', E'))\}$$

où $f_0^\sharp$ est une décoration abstraite initiale.

Une solution de ce système devient calculable du fait que les configurations d'un programme et les combinaisons d'évènements lus et écrits par un module sont finies. Les théorèmes de Tarski et de Kleene s'appliquent aux systèmes d'équations sémantiques concrètes et abstraites des programmes de la même manière que pour les modules.

Toujours en considérant une solution du système d'équations sémantiques abstraites, on peut définir le grand pas abstrait d'un programme en ne conservant que les configurations d'attente.

Définition 6.12 Grand pas abstrait $[\![.]\!]^\sharp \in \mathcal{K} \to (\mathcal{C} \times \mathcal{P}(\mathcal{E}_M) \times \Gamma^\sharp) \to \mathcal{P}(\mathcal{C} \times \mathcal{P}(\mathcal{E}_M) \times \Gamma^\sharp)$

$$[\![\kappa]\!]^\sharp(c, E, \rho^\sharp) \triangleq \{(c', E', f^\sharp[c', E']) \mid c'.\omega \neq \mathit{ready}\}$$

où $f^\sharp$ est le grand pas collecteur de κ à partir de $(c, E, \rho^\sharp)$.

La résolution du système d'équations sémantiques abstraites donne une représentation d'un sur-ensemble des états mémoire qui passent par chaque configuration du programme. La cor-

rection de cette analyse est donnée par la proposition suivante :

$$\forall c\ c'\ E\ E'\ \rho^*\ \rho^{*'}\ \rho^\sharp\ \rho^{\sharp'},\quad \rho^* \subseteq \gamma(\rho^\sharp) \text{ et } \rho^{*'} \subseteq \gamma(\rho^{\sharp'}) \text{ et } (c,E,\rho^*) \xrightarrow{\kappa^*} (c',E',\rho^{*'}) \implies (c',E',\rho^{\sharp'}) \in [\![\kappa]\!]^\sharp(c,E,\rho^\sharp)$$

Nous présentons un analyse avant (et arrière sur les gardes). Nous décrivons quel type de résultats elle permet d'obtenir et nous définissons les transitions abstraites Univers, ordonnanceur et petit pas abstrait qui lui sont associées.

6.4 Analyse Avant

L'analyse avant, comme son nom l'indique, propage l'information dans le sens du flot d'exécution d'un module. Elle permet de connaître les informations déduites à partir de conditions initiales. On parle également d'un *calcul de post-conditions*.

6.4.1 Transition Abstraite Univers

Une transition Univers d'un module ne modifie pas les configurations des programmes. Les variables non locales au module peuvent prendre une valeur arbitraire ($\top$ en interprétation abstraite). Les évènements non locaux au module qui n'ont pas encore été notifiés peuvent l'être. Ceci signifie qu'on doit construire l'ensemble de toutes les combinaisons possibles de notification de ces évènements, ce qui reste calculable vu que le nombre d'évènements non locaux lus ou écrits par un module KernelD est fini lorsque le nombre de programmes dans le module est fini et que chacun de ces programmes a un nombre fini de transitions.

Notation : étant donné un environnement abstrait $\rho^\sharp$ et un ensemble de variables $X \subseteq \mathcal{X}$, on note $\rho^\sharp[x \leftarrow d_x]_{\forall x \in X}$ la mise à jour de la variable x de X dans $\rho^\sharp$ par la valeur d_x. La variable x est donc liée dans $\rho^\sharp[x \leftarrow d_x]_{\forall x \in X}$ par le quantificateur $\forall$.

$$\rho^{\sharp'} = \rho^\sharp[x \leftarrow d_x]_{\forall x \in X} \quad \Leftrightarrow \quad \left\{ \begin{array}{ll} & \forall x \in X,\ \rho^{\sharp'}(x) = d_x \\ \text{et} & \forall y \notin X,\ \rho^{\sharp'}(y) = \rho^\sharp(y) \end{array} \right.$$

Définition 6.13 Transition abstraite Univers $\mathit{Univers}^\sharp \in (\mathcal{M} \times \Sigma^\sharp) \to \mathcal{P}(\Sigma^\sharp)$

Soit $M = \langle L, K \rangle$. Alors,

$$\text{Univers}^\sharp(M,\sigma^\sharp) \triangleq \left\{ \sigma^{\sharp'} \ \middle|\ \begin{array}{ll} & \sigma^{\sharp'}.\rho^\sharp = \sigma^\sharp.\rho^\sharp[x \leftarrow \top]_{\forall x \notin L} \\ \text{et} & \forall e \in L,\ e \in \sigma^{\sharp'}.E \Leftrightarrow e \in \sigma^\sharp.E \\ \text{et} & \sigma^{\sharp'}.C = \sigma^\sharp.C \end{array} \right\}$$

6.4.2 Transition Abstraite Ordonnanceur

Le comportement de l'ordonnanceur dépend des configurations de tous les programmes. En particulier, il dépend des configurations des programmes qui ne sont pas dans le module qu'on

analyse. On reprend les trois règles concrètes de l'ordonnanceur et on les adapte à notre cas restreint :

- si un programme quelconque (interne ou externe au module considéré) est elligible, l'ordonnanceur ne fait rien. Cela signifie, en particulier, que si un programme du module analysé est elligible, alors l'ordonnanceur ne fait rien. Mais même si aucun programme du module étudié n'est elligible, il se peut qu'un programme externe au module le soit. Donc de toute façon, on doit considérer qu'il est possible que l'ordonnanceur ne fasse rien ;
- si tous les programmes (internes ou externes) sont endormis, et qu'on est en phase comportementale, l'ordonnanceur peut passer à la phase de mises à jour. Donc si tous les programmes du module analysé sont endormis dans l'état abstrait courant, on peut passer à la phase de mise à jour dans l'abstraction car cela signifie qu'il est possible que tous les programmes soient endormis. Mais il est également possible qu'il y ait des programmes externes au module qui ne sont pas endormis et dans ce cas là, l'ordonnanceur ne doit rien faire ;
- enfin, si tous les programmes sont endormis et qu'on est en phase de mises à jour, l'ordonnanceur peut repasser en phase comportementale après réductions des statuts. Encore une fois, ceci est donc possible lorsque tous les programmes du module étudié sont endormis, même s'il est possible que d'autres programmes soient elligibles, et auquel cas l'ordonnanceur ne fait rien.

Définition 6.14 Transition abstraite Ordonnanceur $Sched^\sharp \in \mathcal{R}_M \to \mathcal{P}(\mathcal{R}_M)$

On pose un certain nombre de notations :

- "κ elligible dans r_M" sera utilisé pour la proposition $r.C[\kappa].\omega \rhd^- ready$;
- "r_M inactif" dénote le fait qu'aucun programme n'est elligible dans r_M, c'est-à-dire que $\forall \kappa \in M,\ \kappa$ n'est pas elligible dans r_M ;
- "r_M est en phase comportementale" signifie que $upd \notin r_M.E$. "r_M est en phase de mise à jour" signifie que $upd \in r_M.E$;
- quelle que soit une réduction de statuts $\rhd$, nous notons "$C \rhd C'$" le fait que les points de contrôle courants de chaque programme de M restent inchangés de C à C', et que les statuts de tous les programmes de M ont été réduits selon $\rhd$, c'est-à-dire que $\forall \kappa \in M$, $C[\kappa].q = C'[\kappa].q$ et $C[\kappa].\omega \rhd C'[\kappa].\omega$.

À partir de ces notations, la transition abstraite ordonnanceur est définie en figure 6.4.2.

6.4.3 Petit Pas Abstrait d'un Programme

Le petit pas abstrait avant consiste simplement à calculer l'ensemble des configurations, des évènements émis et l'état mémoire abstrait à partir d'une configuration particulière. On exécute donc toutes les instructions possibles à partir du point de contrôle courant du programme. Le petit pas abstrait dépend de la sémantique abstraite des instructions de programme.

Nous dirons qu'une configuration de programme c est elligible lorsque $c.\omega \rhd^- ready$ et qu'elle est inactive dans le cas contraire, c'est-à-dire lorsque $c.\omega \not\rhd^- ready$.

$$\mathrm{Sched}^\sharp(r_M) \triangleq \{r_M\} \cup \begin{cases} \emptyset & \text{si } \exists \kappa \in M,\ \kappa \text{ elligible dans } r_M \\ \{r'_M\} & \text{si } r_M \text{ inactif,} \\ & \text{et } r_M \text{ est en phase comportementale} \\ & \text{et } r'_M.E = r_M.E \cup \{upd\} \\ & \text{et } r_M.C \rhd_{\{upd\}} r'_M.C \\ \{r'_M, r''_M\} & \text{si } r_M \text{ inactif,} \\ & \text{et } r_M \text{ est en phase de mises à jour} \\ & \text{et } r'_M.E = r''_M = \emptyset \\ & \text{et } r_M.C \rhd_E r'_M.C \rhd_\tau r''_M.C \\ & \text{où } E = r_M.E - \{upd\} \text{ et } \tau = \tau_{\min}(r'_M.C) \end{cases}$$

FIG. 6.4 – Définition d'une transition abstraite de l'ordonnanceur

Définition 6.15 Petit pas abstrait $[\![.]\!]^\sharp_1 \in \mathcal{K} \to (\mathcal{C} \times \mathcal{P}(\mathcal{E}) \times \Gamma^\sharp) \to \mathcal{P}(\mathcal{C} \times \mathcal{P}(\mathcal{E}) \times \Gamma^\sharp)$

$$[\![\kappa]\!]^\sharp_1(c, E, \rho^\sharp) \triangleq \begin{cases} \{\ (c, E, \rho^\sharp)\ \} & \text{si } c \text{ inactif} \\ \left\{\ ((q', \omega), E \cup E', \rho^{\sharp\prime}) \quad | \quad \exists inst, \quad (q, inst, q') \in \kappa \text{ et } (E', \rho^{\sharp\prime}, \omega) = [\![inst]\!]^\sharp(\rho^\sharp)\ \right\} & \\ \qquad \text{si } c \text{ elligible} & \end{cases}$$

Expressions

La sémantique abstraite des instructions doit prendre en compte la sémantique abstraite des expressions. On suppose que l'on dispose d'une opération abstraite $\phi^\sharp_{op} \in \mathcal{D} \times ... \times \mathcal{D} \to \mathcal{D}$ pour chaque opération *op*. Un exemple de ces opérations sur KERNELD est donné en annexe avec les booléens (section A.5) et les intervalles (section A.6). Ces opérations abstraites doivent seulement satisfaire la proposition suivante :

$$\forall v_1\ ...\ v_n\ v\ d_1\ ...\ d_n, \quad \phi_{op}(v_1, ..., v_n, v) \text{ et } v_1 \in \gamma(d_1) \text{ et } ... \text{ et } v_n \in \gamma(d_n) \implies v \in \gamma(\phi^\sharp_{op}(d_1, ..., d_n))$$

À partir de ces fonctions, on peut définir la *sémantique abstraite avant d'une expression* qui étant donné un environnement abstrait et une expression, retourne la valeur abstraite qui représente l'ensemble des valeurs que peut prendre une expression.

Définition 6.16 Sémantique abstraite avant d'une expression $[\![.]\!]^\sharp_. \in \mathcal{E}xp \times \Gamma^\sharp \to \mathcal{D}$

$$\begin{aligned} [\![v]\!]^\sharp_{\rho^\sharp} &= \alpha(v) \\ [\![x]\!]^\sharp_{\rho^\sharp} &= \rho^\sharp(x) \\ [\![\mathrm{op}(exp_1, ..., exp_n)]\!]^\sharp_{\rho^\sharp} &= \phi^\sharp_{\mathrm{op}}([\![exp_1]\!]^\sharp_{\rho^\sharp}, ..., [\![exp_n]\!]^\sharp_{\rho^\sharp}) \end{aligned}$$

De plus, afin d'établir un sur-ensemble des contraintes que pose une conditionnelle, on se donne une sémantique abstraite arrière pour les opérations. Celle-ci consiste en une fonction $\overleftarrow{\phi}^\sharp_{op} \in \mathcal{D}^n \times \mathcal{D} \to \mathcal{D}^n$ pour chaque opération op, qui pour chaque n-uplet d'arguments et étant donné une valeur maximale de retour, raffine la valeur que les arguments doivent prendre pour que l'opération retourne un résultat compris dans la valeur maximale. Un exemple sur les booléens et un sur les intervalles se trouvent aux annexes A.5 et A.6 du manuscrit. La seule règle que doit respecter cette fonction est que les arguments raffinés doivent au moins contenir les solutions correctes que contiennent les arguments non raffinés.

$$\left.\begin{array}{ll} \forall d_1 \ ... \ d_n \ d \ d'_1 \ ... \ d'_n \ v_1 \ ... \ v_n \ v & \\ \quad \overleftarrow{\phi}^\sharp_{\text{op}}(d_1, ..., d_n, d) = (d'_1, ..., d'_n) & \\ \text{et} \quad v_1 \in \gamma(d_1) \text{ et ... et } v_n \in \gamma(d_n) \text{ et } v \in \gamma(d) & \\ \text{et} \quad \phi_{\text{op}}(v_1, ..., v_n, v) & \end{array}\right\} \Longrightarrow v_1 \in \gamma(d'_1) \text{ et ... et } v_n \in \gamma(d'_n)$$

Ces fonctions permettent de définir la *sémantique abstraite arrière d'une expression* qui retourne un ensemble de contraintes sur ses arguments sous forme d'un environnement abstrait.

Définition 6.17 Sémantique abstraite arrière d'une expression

$$\overleftarrow{[\![. : .]\!]}^\sharp_. \in \mathcal{E}xp \times \mathcal{D} \times \Gamma^\sharp \to \Gamma^\sharp$$

$$\begin{array}{rcl} \overleftarrow{[\![v : d]\!]}^\sharp_{\rho^\sharp} & = & \begin{cases} \rho^\sharp & \text{si } \alpha(v) \sqcap d \neq \bot_{\mathcal{D}} \\ \bot & \text{sinon} \end{cases} \\ \overleftarrow{[\![x : d]\!]}^\sharp_{\rho^\sharp} & = & \rho^\sharp[x \leftarrow \rho^\sharp(x) \sqcap d] \\ \overleftarrow{[\![\text{op}(exp_1, ..., exp_n) : d]\!]}^\sharp_{\rho^\sharp} & = & \rho^\sharp \bigsqcap_{i=1}^n \overleftarrow{[\![exp_i : d_i]\!]}^\sharp_{\rho^\sharp} \\ \multicolumn{3}{c}{\text{où } \overleftarrow{\phi}^\sharp_{\text{op}}([\![exp_1]\!]^\sharp_{\rho^\sharp}, ..., [\![exp_n]\!]^\sharp_{\rho^\sharp}, d) = (d_1, ..., d_n)} \end{array}$$

Instructions

La sémantique abstraite des instructions est très proche de leur sémantique concrète. Un cas particulier est à mentionner : lorsque l'instruction est une garde, on peut continuer l'analyse en se restreignant à l'environnement abstrait contraint par le fait que la garde doit s'évaluer à $\mathbf{true}^\sharp$ car ce sont les seules valeurs qui permettent de traverser la garde.

Définition 6.18 Sémantique abstraite d'une instruction

$$[\![.]\!]^\sharp \in \mathcal{I} \to \Gamma^\sharp \to (\mathcal{P}(\mathcal{E}) \times \Gamma^\sharp \times \mathcal{W})$$

$$[\![inst]\!]^\sharp(\rho^\sharp) \triangleq \begin{cases} (\emptyset, \rho^\sharp, ready) & \text{si } inst = \varepsilon \\ (\emptyset, \overleftarrow{[\![exp : \mathbf{true}^\sharp]\!]}^\sharp_{\rho^\sharp}, ready) & \text{si } inst = exp? \\ (\emptyset, \rho^\sharp[x \leftarrow [\![exp]\!]^\sharp_{\rho^\sharp}], ready)) & \text{si } inst = x \leftarrow exp \\ (\{e\}, \rho^\sharp, ready) & \text{si } inst = e! \\ (\emptyset, \rho^\sharp, \omega) & \text{si } inst = \omega \end{cases}$$

6.4.4 Correction de l'Analyse Avant

La correction de l'analyse consiste à confronter les sémantiques concrète et abstraite pour montrer que la sémantique abstraite capture bien toutes les transitions possibles à partir de ses arguments. Nous allons manipuler des états concrets et des états abstraits formés d'une configuration restreinte et d'un environnement abstrait. La correspondance entre états concrets et états abstraits est définie par les fonctions d'abstraction et de concrétisation suivantes, où $\alpha_\Gamma \in \mathcal{P}(\Gamma) \to \Gamma^\sharp$ est la fonction d'abstraction des environnements concrets et $\gamma_\Gamma \in \Gamma^\sharp \to \mathcal{P}(\Gamma)$ est la fonction de concrétisation des environnements abstraits.

Tout d'abord, étant donné une configuration r, on pose $\Gamma_r \in \mathcal{P}(\Sigma) \to \mathcal{P}(\Gamma)$ pour l'ensemble des environnements associés à r dans un certain ensemble d'états, ce qui se définit par$\Gamma_r(S) = \{\rho \mid (r,\rho) \in S\}$.

α et γ se définissent alors comme suit :

$\alpha \in \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma^\sharp)$
$\alpha(S) \quad = \quad \{(r_M, \rho^\sharp) \mid \rho^\sharp = \alpha_\Gamma(\{\Gamma_r(S) \mid r_M = \text{Restrict}^\sharp(r)\})\}$

$\gamma \in \mathcal{P}(\Sigma^\sharp) \to \mathcal{P}(\Sigma)$
$\gamma(S^\sharp) \quad = \quad \{(r,\rho) \mid \exists (r_M, \rho^\sharp) \in S^\sharp,\ r \in \text{Complete}^\sharp(r_M) \text{ et } \rho \in \gamma_\Gamma(\rho^\sharp)\}$

Formellement, la correction de l'analyse se résume à démontrer que :

$$\text{(G)} \quad \forall \sigma\ \sigma'\ \sigma^\sharp,\ \sigma \in \gamma(\sigma^\sharp) \text{ et } \sigma \xrightarrow{M} \sigma' \Rightarrow \exists \sigma^{\sharp\prime} \in [\![M]\!]^\sharp_1(\sigma^\sharp),\ \sigma' \in \gamma(\sigma^{\sharp\prime})$$

C'est-à-dire que toute transition concrète d'un module à partir d'un état est prise en compte dans la transition abstraite du module. Or, on a $\sigma \xrightarrow{M} \sigma'$ si σ et σ' sont en relation par une transition Univers de M, par une transition locale de M ou par une transition ordonnanceur. La transition abstraite $[\![M]\!]^\sharp$ est exactement construite par l'union des transitions Univers abstraite de M, par les transitions locales abstraites et M et par la transition abstraite ordonnanceur. Donc si chaque transition concrète est correctement représentée par sa transition abstraite, la proposition (G) est vérifiée.

$$\text{(U)} \quad \forall \sigma\ \sigma'\ \sigma^\sharp,\ \sigma \in \gamma(\sigma^\sharp) \text{ et } \sigma \xrightarrow{\text{Univers}(M)} \sigma' \Rightarrow \exists \sigma^{\sharp\prime} \in \text{Univers}^\sharp(M, \sigma^\sharp),\ \sigma' \in \gamma(\sigma^{\sharp\prime})$$

$$\text{(L)} \quad \forall \sigma\ \sigma'\ \sigma^\sharp,\ \sigma \in \gamma(\sigma^\sharp) \text{ et } \sigma \xrightarrow{\text{Local}(M)} \sigma' \Rightarrow \exists \sigma^{\sharp\prime} \in \text{Local}^\sharp(M, \sigma^\sharp),\ \sigma' \in \gamma(\sigma^{\sharp\prime})$$

$$\text{(S)} \quad \forall \sigma\ \sigma'\ \sigma^\sharp,\ \left\{ \begin{array}{l} \sigma \in \gamma(\sigma^\sharp) \text{ et } \sigma.r \xrightarrow{\text{Sched}} \sigma'.r \text{ et } \sigma.\rho = \sigma'.\rho \quad \Rightarrow \\ \exists \sigma^{\sharp\prime},\ \sigma^{\sharp\prime}.r_M \in \text{Sched}^\sharp(\sigma^\sharp.r_M) \text{ et } \sigma^\sharp.\rho^\sharp = \sigma^{\sharp\prime}.\rho^\sharp \text{ et } \sigma' \in \gamma(\sigma^{\sharp\prime}) \end{array} \right.$$

Lemme 6.1 (U) et (L) et (S) $\Rightarrow$ (G)

Preuve : nos hypothèses sont que $\sigma \in \gamma(\sigma^\sharp)$ et $\sigma \xrightarrow{M} \sigma'$. On doit montrer que $\exists \sigma^{\sharp\prime} \in [\![M]\!]^\sharp_1(\sigma^\sharp),\ \sigma' \in \gamma(\sigma^{\sharp\prime})$. Pour cela, on procède par cas sur $\sigma \xrightarrow{M} \sigma'$.

- Si $\sigma \xrightarrow{\text{Univers}(M)} \sigma'$ et puisque $\sigma \in \gamma(\sigma^\sharp)$ par hypothèse, on peut utiliser la proposition (U) pour déduire que $\exists \sigma^{\sharp'} \in \text{Univers}^\sharp(M, \sigma^\sharp),\ \sigma' \in \sigma^{\sharp'}$. Puisque $\text{Univers}^\sharp(M, \sigma^\sharp) \subseteq [\![M]\!]_1^\sharp(\sigma^\sharp)$ par définition de $[\![M]\!]_1^\sharp(\sigma^\sharp)$ (c'est l'union de trois ensembles, dont $\text{Univers}^\sharp(M, \sigma^\sharp)$), on a bien $\exists \sigma^{\sharp'} \in [\![M]\!]_1^\sharp(\sigma^\sharp),\ \sigma' \in \gamma(\sigma^{\sharp'})$.
- Le cas où $\sigma \xrightarrow{\text{Local}(M)} \sigma'$ est exactement le même que le précédent, avec Local à la place d'Univers. Puisque $\sigma \in \gamma(\sigma^\sharp)$ par hypothèse, on peut utiliser la proposition (L) pour déduire que $\exists \sigma^{\sharp'} \in \text{Local}^\sharp(M, \sigma^\sharp),\ \sigma' \in \sigma^{\sharp'}$. Puisque $\text{Local}^\sharp(M, \sigma^\sharp) \subseteq [\![M]\!]_1^\sharp(\sigma^\sharp)$ par définition de $[\![M]\!]_1^\sharp(\sigma^\sharp)$ (c'est l'union de trois ensembles, dont $\text{Local}^\sharp(M, \sigma^\sharp)$), on a bien $\exists \sigma^{\sharp'} \in [\![M]\!]_1^\sharp(\sigma^\sharp),\ \sigma' \in \gamma(\sigma^{\sharp'})$.
- Le dernier cas est celui où $\sigma.r \xrightarrow{\text{Sched}} \sigma'.r$ et $\sigma.\rho = \sigma'.\rho$. Avec $\sigma \in \gamma(\sigma^\sharp)$ par hypothèse, on déduit grâce à la proposition (S) que $\exists \sigma^{\sharp'},\ \sigma^{\sharp'}.r_M \in \text{Sched}^\sharp(\sigma^\sharp.r_M)$ et $\sigma^\sharp.\rho^\sharp = \sigma^{\sharp'}.\rho^\sharp$ et $\sigma' \in \gamma(\sigma^{\sharp'})$. Or, $\{(r_M, \sigma^\sharp.\rho^\sharp) \mid r_M \in \text{Sched}^\sharp(\sigma^{\sharp'}.r_M)\} \subseteq [\![M]\!]_1^\sharp(\sigma^\sharp)$ par définition de $[\![M]\!]_1^\sharp(\sigma^\sharp)$. Donc on a bien $\exists \sigma^{\sharp'} \in [\![M]\!]_1^\sharp(\sigma^\sharp),\ \sigma' \in \gamma(\sigma^{\sharp'})$.

□

Il nous reste à montrer chacune des propositions (U), (L) et (S).

Lemme 6.2

$$\forall \sigma\ \sigma'\ \sigma^\sharp,\ \sigma \in \gamma(\sigma^\sharp) \text{ et } \sigma \xrightarrow{\text{Univers}(M)} \sigma' \Rightarrow \exists \sigma^{\sharp'} \in \text{Univers}^\sharp(M, \sigma^\sharp),\ \sigma' \in \gamma(\sigma^{\sharp'}) \quad \text{(U)}$$

Preuve : on part du fait que $\sigma \in \gamma(\sigma^\sharp)$ et $\sigma \xrightarrow{\text{Univers}(M)} \sigma'$. On construit $\sigma^{\sharp'}$ de la façon suivante :

$$\sigma^{\sharp'}.C = \sigma^\sharp.C \qquad \forall e \in \mathcal{E}_M,\ e \in \sigma^{\sharp'}.E \Leftrightarrow e \in \sigma'.E \qquad \sigma^{\sharp'}.\rho^\sharp = \sigma^\sharp.\rho^\sharp[x \leftarrow \top]_{\forall x \notin L_M}$$

Il suffit alors de montrer que $\sigma^{\sharp'} \in \text{Univers}^\sharp(M, \sigma^\sharp)$ et $\sigma' \in \gamma(\sigma^{\sharp'})$.

- On a $\sigma^{\sharp'}.C = \sigma^\sharp.C$ et $\sigma^{\sharp'}.\rho^\sharp = \sigma^\sharp.\rho^\sharp[x \leftarrow \top]_{\forall x \in L_M}$ par définition de $\sigma^{\sharp'}$. D'après la définition de $\text{Univers}^\sharp$, il ne nous reste plus qu'à montrer $\forall e \in L_M \subseteq \mathcal{E}_M,\ e \in \sigma^{\sharp'}.E \Leftrightarrow e \in \sigma^\sharp.E$ pour conclure que $\sigma^{\sharp'} \in \text{Univers}^\sharp(M, \sigma^\sharp)$. Or $\sigma \xrightarrow{\text{Univers}(M)} \sigma'$, donc $\forall e \in L_M,\ e \in \sigma.E \Leftrightarrow e \in \sigma'.E \Leftrightarrow e \in \sigma^{\sharp'}.E$ par les définitions de $\xrightarrow{\text{Univers}(M)}$ et de $\sigma^{\sharp'}$. De plus, $\sigma \in \gamma(\sigma^\sharp)$. Cela signifie en particulier que $\sigma.r \in \mathit{Complete}^\sharp(\sigma^\sharp.r_M)$, et donc que $\forall e \in L_M \subseteq \mathcal{E}_M,\ e \in \sigma.E \Leftrightarrow \sigma^\sharp.E$ par définition de $\mathit{Complete}^\sharp$. Donc, on a bien $\forall e \in L_M,\ e \in \sigma^\sharp.E \Leftrightarrow \sigma^{\sharp'}.E$.
- Montrer $\sigma' \in \gamma(\sigma^{\sharp'})$, c'est montrer que $\sigma'.r \in \mathit{Complete}^\sharp(\sigma^{\sharp'}.r_M)$ et $\sigma'.\rho \in \gamma_\Gamma(\sigma^{\sharp'}.\rho^\sharp)$.
  - Le fait que $\sigma'.r \in \mathit{Complete}^\sharp(\sigma^{\sharp'}.r_M)$ est équivalent à $\forall e \in \mathcal{E}_M,\ e \in \sigma^{\sharp'}.E \Leftrightarrow e \in \sigma'.E$ (qui est vrai par construction de $\sigma^{\sharp'}$) et $\exists \tau,\ 0 \leq \tau \leq \tau_{\min}(\sigma^{\sharp'}.C)$ et $\forall \kappa \in K_M,\ \sigma^{\sharp'}.C[\kappa].\omega \rhd_\tau \sigma'.C(\kappa).\omega$. Or, on sait que $\sigma \in \gamma(\sigma^\sharp)$, donc que $\exists \tau,\ 0 \leq \tau \leq \tau_{\min}(\sigma^\sharp.C)$ et $\forall \kappa \in K_M,\ \sigma^\sharp.C[\kappa].\omega \rhd_\tau \sigma.C(\kappa).\omega$. La preuve s'achève en remplaçant dans cette dernière proposition σ par σ' car puisque $\sigma \xrightarrow{\text{Univers}(M)} \sigma'$, alors $\forall \kappa \in K_M,\ \sigma.C(\kappa) = \sigma'.C(\kappa)$, et on peut également y remplacer $\sigma^\sharp$ par $\sigma^{\sharp'}$ car $\sigma^{\sharp'}.C = \sigma^\sharp.C$ par définition de $\sigma^{\sharp'}$.
  - Pour montrer que $\sigma'.\rho \in \gamma_\Gamma(\sigma^{\sharp'}.\rho^\sharp)$, il faut montrer que $\forall x,\ \sigma'.\rho(x) \in \gamma_\mathcal{D}(\sigma^{\sharp'}.\rho^\sharp(x))$. On distingue deux cas : celui où $x \in L_M$ et celui où $x \notin L_M$. On sait que $\sigma \xrightarrow{\text{Univers}(M)} \sigma'$, et donc que $\forall x \in L_M,\ \sigma.\rho(x) = \sigma'.\rho(x)$. Puisque $\sigma \in \gamma(\sigma^\sharp)$, alors $\forall x,\ \sigma.\rho(x) \in \gamma_\mathcal{D}(\sigma^\sharp.\rho^\sharp(x))$. Et comme $\forall x \in L_M,\ \sigma.\rho(x) = \sigma'.\rho(x)$, on a $\forall x \in L_M,\ \sigma'.\rho(x) \in \gamma_\mathcal{D}(\sigma^\sharp.\rho^\sharp(x))$. Or,

$\sigma^{\sharp'} \in \text{Univers}^\sharp(M, \sigma^\sharp)$, on sait donc que $\forall x \in L_M,\ \sigma^\sharp.\rho^\sharp(x) = \sigma^{\sharp'}.\rho^\sharp(x)$. On en conclut que $\forall x \in L_M,\ \sigma'.\rho(x) \in \gamma_{\mathcal{D}}(\sigma^{\sharp'}.\rho^\sharp(x))$. Pour le cas où $x \notin L_M$, on sait que $\sigma^{\sharp'}.\rho^\sharp(x) = \top$ par définition de $\sigma^{\sharp'}$. Par définition de $\top$, on sait automatiquement que $\forall x \notin L_M,\ \sigma'.\rho(x) \in \gamma_{\mathcal{D}}(\top) = \gamma_{\mathcal{D}}(\sigma^{\sharp'}.\rho^\sharp(x))$.

□

Lemme 6.3

$$\begin{array}{l} \forall \sigma\ \sigma'\ \sigma^\sharp,\quad \sigma \in \gamma(\sigma^\sharp) \text{ et } \sigma.r \xrightarrow{\text{Sched}} \sigma'.r \text{ et } \sigma.\rho = \sigma'.\rho \quad \Rightarrow \\ \qquad\qquad \exists \sigma^{\sharp'},\ \sigma^{\sharp'}.r_M \in \text{Sched}^\sharp(\sigma^\sharp.r_M) \text{ et } \sigma^\sharp.\rho^\sharp = \sigma^{\sharp'}.\rho^\sharp \text{ et } \sigma' \in \gamma(\sigma^{\sharp'}) \end{array} \qquad \text{(S)}$$

Preuve : on part du fait que $\sigma \in \gamma(\sigma^\sharp)$, $\sigma.r \xrightarrow{\text{Sched}} \sigma'.r$ et $\sigma.\rho = \sigma'.\rho$. On construit $\sigma^{\sharp'}$ de la façon suivante : si $\exists \kappa,\ \kappa$ elligible dans $\sigma.r$, alors $\sigma^{\sharp'}.r_M = \sigma^\sharp.r_M$. Sinon si $\sigma.r$ est inactif et en phase comportementale, alors $\sigma^{\sharp'}.E = \sigma^\sharp.E \cup \{upd\}$ et $\sigma^\sharp.C \rhd_{\{upd\}} \sigma^{\sharp'}.C$. Sinon, c'est que $\sigma.r$ est inactif et en phase de mises à jour. Dans ce cas, on pose $\sigma^{\sharp'}.E = \emptyset$, $E = \sigma^\sharp.E - \{upd\}$, $E' = \sigma.E - \{upd\}$, $C_M = \rhd_E(\sigma^\sharp.C)$, $C = \rhd_{E'}(\sigma.C)$, $\tau = \tau_{\min}(C_M)$ et $\tau' = \tau_{\min}(C)$. Si $\tau' = \tau$, alors on pose et $\sigma^{\sharp'}.C = \rhd_\tau(C_M)$, sinon on pose $\sigma^{\sharp'}.C = C_M$. Dans tous les cas, on pose $\sigma^{\sharp'}.\rho^\sharp = \sigma^\sharp.\rho^\sharp$. On peut alors montrer que $\sigma^{\sharp'}.r_M \in \text{Sched}^\sharp(\sigma^\sharp.r_M)$, $\sigma^{\sharp'}.\rho^\sharp = \sigma^\sharp.\rho^\sharp$ (directement satisfait par construction de $\sigma^{\sharp'}$) et $\sigma' \in \gamma(\sigma^{\sharp'})$. On commence par montrer $\sigma^{\sharp'}.r_M \in \text{Sched}^\sharp(\sigma^\sharp.r_M)$, ce que l'on fait en procédant par cas sur la nature de σ.

- Si $\exists \kappa,\ \kappa$ elligible dans $\sigma.C(\kappa)$, alors $\sigma^{\sharp'}.r_M = \sigma^\sharp.r_M \in \text{Sched}^\sharp(\sigma^\sharp.r_M)$ par définition de $\sigma^{\sharp'}$ et $\text{Sched}^\sharp(\sigma^\sharp.r_M)$.
- Si $\sigma.r$ est inactif et en phase comportementale, alors $\sigma^\sharp.r_M$ est inactif et en phase comportementale car $\sigma \in \gamma(\sigma^\sharp)$, donc $\sigma.r \in \text{Complete}^\sharp(\sigma^\sharp.r_M)$, donc $\forall e \in \mathcal{E}_M,\ e \in \sigma.E \Leftrightarrow e \in \sigma^\sharp.E$ et les statuts des programmes dans $\sigma^\sharp$ se réduisent en ceux de σ (si σ n'a pas de programme elligible, alors $\sigma^\sharp$ non plus). Alors, $\sigma^{\sharp'}.r_M \in \text{Sched}^\sharp(\sigma^\sharp.r_M)$ par définition de $\sigma^{\sharp'}$ et $\text{Sched}^\sharp(\sigma^\sharp.r_M)$ dans ce cas.
- Enfin, si $\sigma.r$ est inactif et en phase de mises à jour, alors $\sigma^\sharp.r_M$ est inactif et en phase de mises à jour (comme pour le cas précédent). Et de même $\sigma^{\sharp'}.r_M \in \text{Sched}^\sharp(\sigma^\sharp.r_M)$ par définition de $\sigma^{\sharp'}$ et $\text{Sched}^\sharp(\sigma^\sharp.r_M)$ dans ce cas.

Pour finir, nous devons montrer que $\sigma' \in \gamma(\sigma^{\sharp'})$, ce qui signifie qu'il faut montrer $\sigma'.r \in \text{Complete}^\sharp(\sigma^{\sharp'}.r_M)$ et $\sigma'.\rho \in \gamma_\Gamma(\sigma^{\sharp'}.\rho^\sharp)$. Cette dernière proposition est vérifiée puisque $\sigma \in \gamma(\sigma^\sharp)$, et donc que $\sigma.\rho \in \gamma_\Gamma(\sigma^\sharp.\rho^\sharp)$. Or, $\sigma.\rho = \sigma'.\rho$ et $\sigma^\sharp.\rho^\sharp = \sigma^{\sharp'}.\rho^\sharp$ par hypothèse et définition. Donc $\sigma'.\rho \in \gamma_\Gamma(\sigma^{\sharp'}.\rho^\sharp)$. Quant à la proposition $\sigma'.r \in \text{Complete}^\sharp(\sigma^{\sharp'}.r_M)$, elle se montre en décomposant la définition de $\sigma^{\sharp'}$.

- Si $\exists \kappa,\ \kappa$ est elligible dans $\sigma.r$, alors $\sigma'.r = \sigma.r$ et $\sigma^{\sharp'}.r_M = \sigma^\sharp.r_M$. Or, $\sigma \in \gamma(\sigma^\sharp)$, donc $\sigma.r \in \text{Complete}^\sharp(\sigma^\sharp.r_M)$. Donc par réécriture, on a bien $\sigma'.r \in \text{Complete}^\sharp(\sigma^{\sharp'}.r_M)$.
- Si $\sigma.r$ est inactif et en phase comportementale, alors $\sigma'.E = \sigma.E \cup \{upd\}$, et $\sigma.C \rhd_{\{upd\}} \sigma'.C$. De plus $\sigma^{\sharp'}.E = \sigma^\sharp.E \cup \{upd\}$ et $\sigma^\sharp.C \rhd_{\{upd\}} \sigma^{\sharp'}.C$ d'après la définition de $\sigma^\sharp$. De plus, on a également $\sigma'.E = \sigma.E \cup \{upd\}$ et $\sigma.C \rhd_{\{upd\}} \sigma'.C$ puisque $\sigma.r \xrightarrow{\text{Sched}} \sigma'.r$. $\sigma \in \gamma(\sigma^\sharp) \Rightarrow \sigma.r \in \text{Complete}^\sharp(\sigma^\sharp) \Rightarrow \forall e \in \mathcal{E}_M,\ e \in \sigma.E \Leftrightarrow e \in \sigma^\sharp.E$. Donc $\forall e \in \mathcal{E}_M,\ e \in \sigma.E \cup \{upd\} \Leftrightarrow e \in \sigma^\sharp.E \cup \{upd\}$. Par réécriture, on obtient que $\forall e \in \mathcal{E}_M,\ e \in \sigma'.E \Leftrightarrow e \in \sigma^{\sharp'}.E$. Toujours d'après $\sigma.r \in \text{Complete}^\sharp(\sigma^\sharp)$, on sait que $\exists \tau,\ 0 \leq \tau < \tau_{\min}(\sigma^\sharp.C)$ et $\sigma^\sharp.C \rhd_\tau \sigma.C$. D'après $\sigma.C \rhd_{\{upd\}} \sigma'.C$ et $\sigma^\sharp.C \rhd_{\{upd\}} \sigma^{\sharp'}.C$ montrés précédemment et d'après le lemme 5.2, on en déduit que $\exists \tau,\ 0 \leq \tau < \tau_{\min}(\sigma^\sharp.C)$ et $\sigma^{\sharp'}.C \rhd_\tau \sigma'.C$. D'après le lemme 5.5, puisque $\sigma^\sharp.C \rhd_E \sigma^{\sharp'}.C$, alors $\tau_{\min}(\sigma^\sharp.C) \leq \tau_{\min}(\sigma^{\sharp'}.C)$. Donc $\exists \tau,\ 0 \leq \tau < \tau_{\min}(\sigma^{\sharp'}.C)$ et

$\sigma^{\sharp'}.C \rhd_\tau \sigma'.C$. Ceci associé au fait que $\forall e \in \mathcal{E}_M,\ e \in \sigma'.E \Leftrightarrow e \in \sigma^{\sharp'}.E$ permet de conclure que $\sigma'.r \in \text{Complete}^\sharp(\sigma^{\sharp'})$ par définition.

- Si $\sigma.r$ est inactif et en phase de mises à jour, alors $\sigma'.E = \emptyset$ et il existe un C tel que $\sigma.C \rhd_{E'} C \rhd_{\tau'} \sigma'.C$ avec $E' = \sigma.E - \{upd\}$ et $\tau' = \tau_{\min}(C)$ car $\sigma \xrightarrow{\text{Sched}} \sigma'$. De plus on a posé $\sigma^{\sharp'}.E = \emptyset$, $E = \sigma^\sharp.E - \{upd\}$ et $\tau = \tau_{\min}(C_M)$ d'après la définition de $\sigma^{\sharp'}$. Déjà, on peut déduire que $\sigma'.E = \emptyset = \sigma^{\sharp'}.E$, et donc que $\forall e \in \mathcal{E}_M,\ e \in \sigma'.E \Leftrightarrow e \in \sigma^{\sharp'}.E$. Il nous reste encore à montrer que $\exists \tau,\ 0 \leq \tau < \tau_{\min}(\sigma^{\sharp'}.C)$ et $\forall \kappa \in K_M,\ \sigma^{\sharp'}.C[\kappa].\omega \rhd_\tau \sigma'.C(\kappa).\omega$. Pour cela, on commence par énoncer que comme $\sigma \in \gamma(\sigma^\sharp)$, on sait que $\forall e \in \mathcal{E}_M,\ e \in \sigma.E \Leftrightarrow e \in \sigma^\sharp.E$, donc que $\forall e \in \mathcal{E}_M,\ e \in \sigma.E - \{upd\} \Leftrightarrow e \in \sigma^\sharp.E - \{upd\}$ ou encore que $\forall e \in \mathcal{E}_M,\ e \in E \Leftrightarrow e \in E'$. Comme seuls les évènements de $\mathcal{E}_M$ interviennent sur les statuts des programmes de M (par définition de $\mathcal{E}_M$ et des évènements lus), on déduit que $\forall \kappa \in K_M,\ \rhd_E(\sigma^\sharp.C[\kappa].\omega) = \rhd_{E'}(\sigma^\sharp.C[\kappa].\omega)$. Ensuite, on procède encore une fois par cas en suivant la définition de $\sigma^{\sharp'}$.
  - Si $\tau = \tau'$, alors puisque $\sigma \in \gamma(\sigma^\sharp)$, c'est que $\forall \kappa \in K_M,\ \sigma^\sharp.C[\kappa].\omega \rhd_0 \sigma.C(\kappa).\omega$ et $\forall \kappa \in K_M,\ \sigma^\sharp.C[\kappa].\omega = \sigma.C(\kappa).\omega$. Sinon les minimums de $\sigma.C$ et $\sigma^\sharp.C$ seraient différents et τ et τ' le seraient également. Or, dans le cas où $\tau = \tau'$, on a $\forall \kappa \in K_M,\ \sigma^{\sharp'}.C = \rhd_\tau(\rhd_E(\sigma^\sharp.C))$. Donc on a $\forall \kappa \in K_M,\ \sigma^{\sharp'}.C[\kappa].\omega = \rhd_\tau(\rhd_{E'}(\sigma.C(\kappa).\omega)) = \sigma'.C(\kappa).\omega$, ou encore $\forall \kappa \in K_M,\ \sigma^{\sharp'}.C[\kappa].\omega \rhd_0 \sigma'.C(\kappa).\omega$. Donc il existe bien τ tel que $0 \leq \tau < \tau_{\min}(\sigma^{\sharp'}.C)$ et $\forall \kappa \in K_M,\ \sigma^{\sharp'}.C[\kappa].\omega \rhd_\tau \sigma'.C(\kappa).\omega$: il suffit de prendre $\tau = 0$. En effet, $\sigma^{\sharp'}.C$ s'obtient en réduisant le statut de chaque programme de $\sigma^\sharp.C$ par E puis par le minimum du résultat. Il n'y a donc plus de 0 possible dans un statut d'un programme de $\sigma^{\sharp'}.C$, et donc $0 < \tau_{\min}(\sigma^{\sharp'}.C)$.
  - Si $\tau \neq \tau'$, $\sigma^{\sharp'}.C = \rhd_E(\sigma^\sharp.C)$. On sait donc que $\forall \kappa \in K_M,\ \sigma^\sharp.C[\kappa].\omega \rhd_E \sigma^{\sharp'}.C[\kappa].\omega$. On a également un certain C' tel que $\forall \kappa \in K_M,\ \sigma.C(\kappa).\omega \rhd_E C'(\kappa).\omega \rhd_{\tau'} \sigma'.C(\kappa).\omega$. Et comme $\sigma \in \gamma(\sigma^\sharp)$, on a un τ'' tel que $0 \leq \tau'' < \tau$ et $\forall \kappa \in K_M,\ \sigma^\sharp.C[\kappa].\omega \rhd_{\tau''} \sigma.C(\kappa).\omega$. En utilisant le lemme 5.2, on en déduit $\forall \kappa \in K_M,\ \sigma^{\sharp'}.C[\kappa].\omega \rhd_{\tau''} C'(\kappa).\omega$ et donc $\forall \kappa \in K_M,\ \sigma^{\sharp'}.C[\kappa].\omega \rhd_{\tau'+\tau''} \sigma'.C(\kappa).\omega$ avec $\sigma^\sharp.C \rhd_{\tau''} \sigma.C$, donc $\tau_{\min}(\sigma^{\sharp'}.C) \geq \tau_{\min}(\sigma^\sharp.C)$ (lemme 5.4) $> \tau'' + \tau_{\min}(\sigma.C) = \tau'' + \tau'$.

□

Comme pour la sémantique abstraite des modules dont le calcul du point fixe repose sur leur transition abstraite, la sémantique abstraite des programme est une accumulation de résultats calculés à partir d'une transition abstraite simple. On doit vérifier que la sémantique abstraite des instructions capturent toutes les possibilités d'exécution. Les instructions s'appuient sur les expressions alors nous devons montrer avant tout la correction de la sémantique abstraite des expressions.

Lemme 6.4 $\quad \forall exp\ \rho\ \rho^\sharp,\ \rho \in \gamma(\rho^\sharp) \Rightarrow [\![exp]\!]_\rho \subseteq \gamma_\mathcal{D}([\![exp]\!]^\sharp_{\rho^\sharp})$

Preuve : par récurrence sur *exp*.

- Si $\exists v,\ exp = v$, alors $[\![exp]\!]_\rho = \{v\}$ et $[\![exp]\!]^\sharp_{\rho^\sharp} = \alpha_\mathcal{D}(v)$. Or, puisque $\alpha_\mathcal{D}$ et $\gamma_\mathcal{D}$ forment une connexion de Galois, on a bien $\{v\} \subseteq \gamma_\mathcal{D}(\alpha_\mathcal{D}(\{v\}))$.
- Si $\exists x,\ exp = x$, alors $[\![exp]\!]_\rho = \{\rho(x)\}$ et $[\![exp]\!]^\sharp_{\rho^\sharp} = \rho^\sharp(x)$. Par hypothèse, on a $\rho \in \gamma(\rho^\sharp)$ et donc $\rho(x) \in \gamma_\mathcal{D}(\rho^\sharp(x))$.
- Si $\exists op\ exp_1\ ...\ exp_n,\ exp = op(exp_1, ..., exp_n)$, c'est que $\exists v_1 \in [\![exp_1]\!]_\rho\ ...\ v_n \in [\![exp_n]\!]_\rho$ et $v \in [\![exp]\!]_\rho$ tels que $\phi_{op}(v_1, ..., v_n, v)$. De plus, par hypothèse de récurrence, on a $\forall i,\ v_i \in$

$[\![exp_i]\!]_\rho \subseteq \gamma_{\mathcal{D}}([\![exp_i]\!]^\sharp_{\rho^\sharp})$. Donc, par définition de $\phi^\sharp_{op}$, on a $v \in \phi^\sharp_{op}([\![exp_1]\!]^\sharp_{\rho^\sharp}, ..., [\![exp_n]\!]^\sharp_{\rho^\sharp}) = [\![exp]\!]^\sharp_{\rho^\sharp}$.

□

Lemme 6.5 $\forall exp\ \rho\ \rho^\sharp\ v\ d,\ \rho \in \gamma(\rho^\sharp)$ et $v \in [\![exp]\!]_\rho$ et $v \in \gamma_{\mathcal{D}}(d) \Rightarrow \rho \in \gamma(\overleftarrow{[\![exp : d]\!]}^\sharp_{\rho^\sharp})$

Preuve : par récurrence sur *exp*.

- Si $\exists v',\ exp = v'$, alors forcément $v' = v$ car $v \in [\![exp]\!]_\rho = \{v'\}$. $\overleftarrow{[\![exp : d]\!]}^\sharp_{\rho^\sharp} = \overleftarrow{[\![v : d]\!]}^\sharp_{\rho^\sharp}$. $v \in \gamma_{\mathcal{D}}(d) \Leftrightarrow \alpha_{\mathcal{D}}(v) \sqsubseteq d$ car $\alpha_{\mathcal{D}}$ et $\gamma_{\mathcal{D}}$ forment une connexion de Galois. Donc $\alpha_{\mathcal{D}}(v) \sqcap d = \alpha_{\mathcal{D}}(v) \neq \bot_{\mathcal{D}}$. Donc $\overleftarrow{[\![exp : d]\!]}^\sharp_{\rho^\sharp} = \rho^\sharp$ et on a bien $\rho \in \gamma(\rho^\sharp) = \gamma(\overleftarrow{[\![exp : d]\!]}^\sharp_{\rho^\sharp})$.
- Si $\exists x,\ exp = x$, alors $\rho(x) = v$ car $v \in [\![exp]\!]_\rho = \{\rho(x)\}$. $\overleftarrow{[\![exp : d]\!]}^\sharp_{\rho^\sharp} = \rho^\sharp[x \leftarrow \rho^\sharp(x) \sqcap d]$. Donc $\forall y \neq x,\ \rho(x) \in \gamma_{\mathcal{D}}(\rho^\sharp(y)) = \gamma_{\mathcal{D}}(\overleftarrow{[\![exp : d]\!]}^\sharp_{\rho^\sharp}(y))$. $\rho \in \gamma(\rho^\sharp) \Rightarrow v = \rho(x) \in \gamma_{\mathcal{D}}(\rho^\sharp(x)) \Leftrightarrow \alpha_{\mathcal{D}}(v) \sqsubseteq \rho^\sharp(x)$ car $\alpha_{\mathcal{D}}$ et $\gamma_{\mathcal{D}}$ forment une connexion de Galois. De plus, $v \in \gamma_{\mathcal{D}}(d) \Leftrightarrow \alpha_{\mathcal{D}}(v) \sqsubseteq d \Leftrightarrow \alpha_{\mathcal{D}}(v) \sqsubseteq \rho^\sharp(x) \sqcap d = \overleftarrow{[\![exp : d]\!]}^\sharp_{\rho^\sharp}(x)$. D'où $\rho(x) = v \in \gamma(\overleftarrow{[\![exp : d]\!]}^\sharp_{\rho^\sharp})$.
- Si $\exists op\ exp_1\ ...\ exp_n,\ exp = op(exp_1, ..., exp_n)$, c'est que $\exists v_1 \in [\![exp_1]\!]_\rho\ ...\ v_n \in [\![exp_n]\!]_\rho$ tels que $\phi_{op}(v_1, ..., v_n, v)$. $\overleftarrow{[\![exp : d]\!]}^\sharp_{\rho^\sharp} = \rho^\sharp \bigsqcap_{i=1}^n \overleftarrow{[\![exp_i : d_i]\!]}^\sharp_{\rho^\sharp}$ où $(d_1, ..., d_n) = \overleftarrow{\phi}^\sharp_{op}([\![exp_1]\!]^\sharp_{\rho^\sharp}, ..., [\![exp_1]\!]^\sharp_{\rho^\sharp}, d)$. De plus, par hypothèse de récurrence, on a $\forall i,\ v_i \in \gamma_{\mathcal{D}}(d_i) \Rightarrow \rho \in \gamma(\overleftarrow{[\![exp_i : d_i]\!]}^\sharp_{\rho^\sharp})$. Or, $\forall i,\ v_i \in [\![exp_i]\!]_\rho \Rightarrow v_i \in \gamma_{\mathcal{D}}([\![exp_i]\!]^\sharp_{\rho^\sharp})$ d'après le lemme 6.4. Donc, d'après la définition de $\overleftarrow{\phi}^\sharp_{op}$, on a bien $\forall i,\ v_i \in \gamma_{\mathcal{D}}(d_i)$. Donc on sait que $\forall i,\ \rho \in \gamma(\overleftarrow{[\![exp_i : d_i]\!]}^\sharp_{\rho^\sharp})$, ce qui est équivalent à $\forall i,\ \alpha(\rho) \sqsubseteq \overleftarrow{[\![exp_i : d_i]\!]}^\sharp_{\rho^\sharp}$ car α et γ forment une connexion de Galois. Donc $\alpha(\rho) \sqsubseteq \bigsqcap_{i=1}^n \overleftarrow{[\![exp_i : d_i]\!]}^\sharp_{\rho^\sharp}$. Ajoutons à cela le fait que $\rho \in \gamma(\rho^\sharp) \Leftrightarrow \alpha(\rho) \sqsubseteq \rho^\sharp$, et on obtient que $\alpha(\rho) \sqsubseteq \rho^\sharp \bigsqcap_{i=1}^n \overleftarrow{[\![exp_i : d_i]\!]}^\sharp_{\rho^\sharp}$, ou encore $\rho \in \gamma(\rho^\sharp \bigsqcap_{i=1}^n \overleftarrow{[\![exp_i : d_i]\!]}^\sharp_{\rho^\sharp}) = \gamma(\overleftarrow{[\![exp : d]\!]}^\sharp_{\rho^\sharp})$

□

Lemme 6.6

$$\forall inst\ \rho\ E\ \rho'\ \omega^\sharp_\rho,\ \rho \in \gamma(\rho^\sharp) \text{ et } \rho \xrightarrow{inst} E, \rho', \omega \Rightarrow \exists \rho^{\sharp\prime},\ [\![inst]\!]^\sharp(\rho^\sharp) = (E, \rho^{\sharp\prime}, \omega) \text{ et } \rho' \in \gamma(\rho^{\sharp\prime})$$

Preuve : par cas sur la nature de *inst*.

- Si $inst = \varepsilon$, alors $\rho \xrightarrow{inst} \emptyset, \rho, ready$ et $[\![inst]\!]^\sharp(\rho^\sharp) = (\emptyset, \rho^\sharp, ready)$ avec $\rho \in \gamma(\rho^\sharp)$ par hypothèse.
- Si $\exists exp,\ inst = exp?$, alors $\rho \xrightarrow{inst} \emptyset, \rho, ready$ si $\mathbf{true} \in [\![exp]\!]_\rho$ et $[\![inst]\!]^\sharp(\rho^\sharp) = (\emptyset, \overleftarrow{[\![exp : \mathbf{true}^\sharp]\!]}^\sharp_{\rho^\sharp}, ready)$. D'après le lemme 6.5, on a bien $\rho \in \gamma(\overleftarrow{[\![exp : \mathbf{true}^\sharp]\!]}^\sharp_{\rho^\sharp})$ lorsque $\mathbf{true} \in [\![exp]\!]_\rho$.
- Si $\exists x\ exp,\ inst = x \leftarrow exp$, alors $\rho \xrightarrow{inst} \emptyset, \rho[x \leftarrow v], ready$ avec $v \in [\![exp]\!]_\rho$ et $[\![inst]\!]^\sharp(\rho^\sharp) = (\emptyset, \rho^\sharp[x \leftarrow [\![exp]\!]^\sharp_{\rho^\sharp}], ready)$. Il faut donc montrer que $\rho[x \leftarrow v] \in \gamma(\rho^\sharp[x \leftarrow [\![exp]\!]^\sharp_{\rho^\sharp}])$ où $v \in [\![exp]\!]_\rho$. $\forall y \neq x,\ \rho[x \leftarrow v](y) = \rho(y)$ et $\rho^\sharp[x \leftarrow [\![exp]\!]^\sharp_{\rho^\sharp}](y) = \rho^\sharp(y)$. Or, $\rho \in \gamma(\rho^\sharp)$, donc $\forall y,\ \rho(y) \in \gamma_{\mathcal{D}}(\rho^\sharp(y))$. Enfin, $\rho[x \leftarrow v](x) = v \in [\![exp]\!]_\rho$ et $\rho^\sharp[x \leftarrow [\![exp]\!]^\sharp_{\rho^\sharp}](x) = [\![exp]\!]^\sharp_{\rho^\sharp}$. D'après le lemme 6.4, $v \in \gamma_{\mathcal{D}}([\![exp]\!]^\sharp_{\rho^\sharp})$ et donc $\rho(x) \in \gamma_{\mathcal{D}}(\rho^\sharp(x))$.

– Si $\exists e,\ inst = e!$, alors $\rho \xrightarrow{inst} \{e\}, \rho, ready$ et $[\![inst]\!]^\sharp(\rho^\sharp) = (\{e\}, \rho^\sharp, ready)$ avec $\rho \in \gamma(\rho^\sharp)$ par hypothèse.
– Si $\exists \omega,\ inst = \omega$, alors $\rho \xrightarrow{inst} \emptyset, \rho, \omega$ et $[\![inst]\!]^\sharp(\rho^\sharp) = (\emptyset, \rho^\sharp, \omega)$ avec $\rho \in \gamma(\rho^\sharp)$ par hypothèse.

□

Ces quelques preuves closent ce chapitre et montre que l'on peut obtenir une représentation finie d'un sur-ensemble des comportements d'un module. Un exemple du déroulement de cette analyse sera donnée au chapitre 8. Mais tout d'abord, nous montrons dans le chapitre suivant comment nous servir du résultat d'une analyse pour vérifier des propriétés et la correction d'un remplacement ou d'une discrimination.

CHAPITRE 7

Algorithmes de Vérification

Ce chapitre montre comment vérifier sur SystemD les trois briques qui constituent la vérification compositionnelle de programmes : la vérification qu'un module satisfait une propriété décrite elle aussi par un module, la vérification qu'un module discrimine un autre (non bloquage) et enfin la vérification qu'un module remplace correctement un autre. Pour cela, nous nous servons du résultat principal du chapitre précédent : l'algorithme 6.7 de calcul du graphe d'états abstraits d'un module qui donne une représentation finie d'un sur-ensemble des exécutions du module.

7.1 Propriété

D'après la définition 3.5 du chapitre 3, le fait qu'un système décrit en SystemD valide une propriété consiste à s'assurer que pour toute trace de l'assemblage du système avec la propriété, si la trace est sûre vis-à-vis des programmes du système (c'et-à-dire qu'une configuration où un programme du système à son point de contrôle en **fail** n'est pas atteignable), alors la trace est sûre vis-à-vis des programmes de la propriété. Nous commençons par montrer un cas simple où le système ne contient pas d'assertion, avant d'énoncer un algorithme général de vérification de la validité.

7.1.1 Exemple

Si le système ne contient pas d'assertion, la validité du système vis-à-vis d'une propriété consiste à vérifier si oui ou non **fail** est atteignable par un programme de la propriété. En réalité, on ne s'intéresse pas vraiment à savoir si un état erreur est atteignable, on veut

surtout vérifier qu'aucun ne l'est. Ces deux propositions qui semblent équivalentes sont en fait traitées différemment dans une problématique de vérification. L'une consiste à vérifier que quelque chose arrive (il existe une trace où **fail** est atteignable), alors que l'autre consiste à vérifier que quelque chose n'arrive pas (**fail** n'est jamais atteignable, quelle que soit la trace). Comme l'interprétation abstraite calcule un sur-ensemble des comportements, c'est la seconde proposition la plus adaptée à cet outil. Si l'analyse d'un module montre qu'une configuration contenant **fail** n'est pas atteignable dans l'abstraction, c'est que **fail** n'est pas atteignable quelle que soit l'exécution concrète du module. Comme on reste exact sur les informations de configurations des programmes locaux à un module, nous n'aurons aucun mal à assurer qu'aucun de ces programmes ne se trouve en **fail**.

Pour exemple, considérons le programme de la figure 7.1 que nous nommons κ.

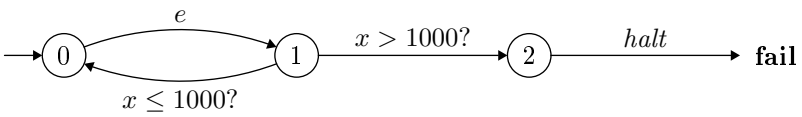

FIG. 7.1 – Une propriété en KERNELD

Informellement, ce programme représente la propriété fonctionnelle et temporelle : « lorsque e est notifié, la valeur de x est inférieure à 1000 ». On construit le module $M = \langle \emptyset, \{\kappa\} \rangle$. Ce module va jouer le rôle de propriété. Analyser le module M seul n'a pas d'intérêt, puisque M représente une propriété, et ne vaut donc que lorsqu'il est composé avec un module à vérifier. En effet, analyser M conduira certainement à un état erreur : aucune variable n'y est locale, en particulier x qui peut donc prendre à tout moment n'importe quelle valeur, donc une valeur supérieure à 1000. Remarquons cependant que M est observateur pour x : il n'en change jamais la valeur et se contente de l'observer. M semble donc tout à fait adapté à l'étude de l'évolution de la valeur de x au cours des exécutions d'un module la modifiant. Considérons le module $M' = \langle \{x\}, \{\kappa'\} \rangle$ où κ' est représenté figure 7.2. Pour le vérifier, on analyse l'assemblage $M \otimes M'$.

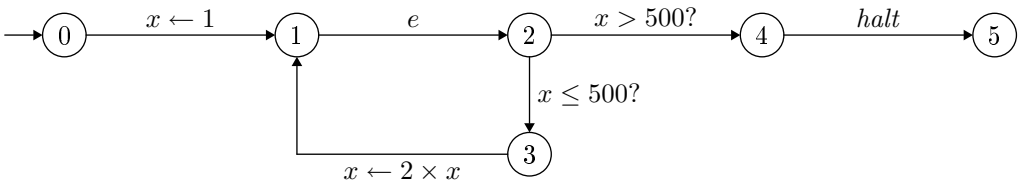

FIG. 7.2 – Un programme sur x

On présente en figure 7.3 le résultat de l'analyse de $M \otimes M'$ où initialement les programmes sont elligibles et placés en leur point de contrôle initial et où x peut avoir n'importe quelle valeur. L'analyse est conduite selon l'algorithme 6.7 de calcul du graphe d'états abstraits décrit aux sections précédentes. Le graphe d'états abstraits a été simplifié en considérant les remarques suivantes :

– nous n'indiquons par les statuts des programmes : ils se déduisent du graphe de flot de contrôle des programmes. Par exemple, κ' attend obligatoirement e (ou est elligible) en 2 ;

- les transitions de l'Univers (sous-entendu des programmes en dehors des modules M et M') ont été supprimées. Leur action mène simplement à boucler sur l'état courant ;
- les transitions de l'ordonnanceur ont également été supprimées : elles se contentent de réduire les statuts et de donner la main à un programme, des informations également superflues lorsque notre intérêt se porte sur le point de contrôle courant des programmes et la valeur de la variable x. Dans le même esprit, on ne mentionne pas l'évènement *upd* que personne n'attend de toutes façons (il n'y a pas de port dans cet exemple) ;
- les transitions des programmes qui n'ont pas la main ont également été supprimées. Elles aussi consistent seulement en une boucle sur l'état courant ;
- les configurations non accessibles ne sont pas mentionnées ;
- enfin, un état contient le point de contrôle courant de κ puis de κ' ainsi que l'intervalle de valeurs possibles pour x en cette configuration.

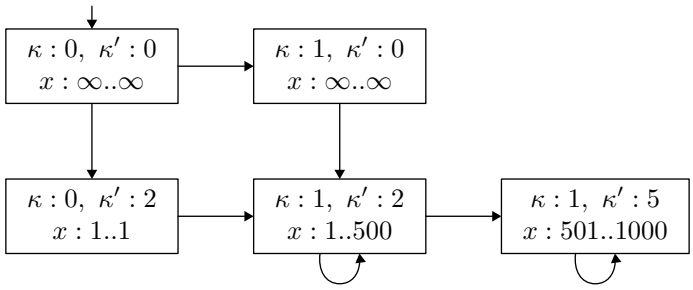

FIG. 7.3 – Analyse de $M \otimes M'$

Le contrôle du système tout entier se déplace comme suit : tout d'abord, n'importe quel programme peut s'exécuter. Ce sera soit κ (ce qui aboutit à la configuration $1, 0$) soit κ' (ce qui conduit à la configuration $0, 2$). Ensuite, celui des deux qui ne s'est pas exécuté aura la main et on atteindra la configuration $1, 2$. Dans tous les cas, cela mène x à prendre la valeur 1. Les programmes attendent ensuite tout deux l'émission de l'évènement e par l'environnement extérieur (par une transition Univers donc). Lorsque l'évènement est émis, la valeur de x double. Cette opération d'attente de e et de valeur doublée pour x continue tant que cette valeur est inférieure ou égale à 500. La valeur de x en début de cette boucle peut donc prendre une valeur entre 1 et 500. Puisque sa valeur double dans la boucle, x peut prendre des valeurs entre 2 et 1000 en sortie de boucle. Lorsque sa valeur est supérieure (strictement) à 500, elle sort et κ' se déplace au point 5, où x peut valoir entre 501 et 1000.

Au final, aucun état où κ est dans **fail** n'est accessible à partir de l'état initial dans ce graphe abstrait. Cela signifie donc qu'aucun état concret où κ est dans **fail** n'est accessible à partir des états concrets initiaux contenus dans l'état abstrait initial, et cela assure donc que x reste inférieure ou égale à 1000 à tout instant d'une exécution de $M \otimes M'$. Lorsque κ' est en 5 nous atteignons un cas limite pour la propriété, car il est possible que x vaille exactement 1000.

7.1.2 Vérification de la Validité

Dans les cas où le système à valider ne contient pas d'assertion, après l'analyse qui décore les configurations du module et de sa propriété, la correction du système consiste simplement

à s'assurer qu'il n'existe pas de chemin partant de l'état initial jusqu'à un état erreur de la propriété sans passer par une contradiction (une variable qui n'a pas de valeur, ou $\perp$ en interprétation abstraite).

Dans le cas général, la définition de la validité (voir section 3.1) nous dit que le système à valider peut contenir des assertions qui sont alors vues comme des hypothèses logiques. Dans ce cas, la validité du système vis-à-vis de sa propriété consiste à vérifier que pour chaque trace, si le système ne provoque pas d'erreur sur la trace et qu'elle ne contient pas de contradiction, alors le propriété ne doit pas non plus contenir d'erreur sur cette trace. Si le système est représenté par un module M et la propriété par un module M', $M \models M'$ est vérifié si la formule suivante de la logique temporelle linéaire LTL [Pnu77a, MP81] est vérifiée sur le graphe d'états abstraits de $M \otimes M'$:

$$\mathbf{A}\ (\mathbf{G}\ (\mathrm{Safe}_M \wedge \mathrm{NoBot}) \Rightarrow \mathbf{G}\ \mathrm{Safe}_{M'})$$

où Safe_M, NoBot et $\mathit{Safe}_{M'}$ sont des propriétés des états abstraits définies comme suit :
$\mathrm{Safe}_M(\sigma^\sharp) = \forall \kappa \in K_M,\ \sigma^\sharp.C[\kappa] \neq \mathbf{fail}$
$\mathrm{NoBot}(\sigma^\sharp) = \forall x,\ \sigma^\sharp.\rho^\sharp(x) \neq \perp$
$\mathrm{Safe}_{M'}(\sigma^\sharp) = \forall \kappa \in K_{M'},\ \sigma^\sharp.C[\kappa] \neq \mathbf{fail}$

Cette formule suit très fidèlement la définition de la validité, à ceci près qu'elle se place dans un contexte d'interprétation abstraite.

Équité. Cependant, il y a un détail supplémentaire que nous ajoutons pour vérification. Le graphe d'états abstraits d'un module contient des transitions Univers qui bouclent sur chacun de ses états. Cela signifie en théorie que l'Univers peut s'exécuter indéfiniment, sans jamais laisser la main à un programme local du module ou à l'ordonnanceur. En fait, cela n'est pas propre à l'Univers : les programmes endormis provoquent également des boucles, ainsi que l'ordonnanceur s'il s'exécute en cours de phase. Pour éviter cette situation dégénérée où les traces sont ainsi bloquées, on suppose une hypothèse d'*équité* [SBB+99] sur notre langage, si bien que l'Univers, les programmes et l'ordonnanceur doivent toujours s'exécuter à un moment ou à un autre.

L'algorithme suivant résume la vérification de $M \models M'$.

Algorithme 7.1 (Vérification de la Validité)

1. Analyse de $M \otimes M'$. On applique l'algorithme 6.7 de calcul du graphe d'états abstraits de $M \otimes M'$. Celui-ci représente un sur-ensemble des comportements du système.
2. On retourne le résultat de la vérification de la formule $\mathbf{A}\ (\mathbf{G}\ (\mathrm{Safe}_M \wedge \mathrm{NoBot}) \Rightarrow \mathbf{G}\ \mathrm{Safe}_{M'})$ par un model checker de LTL.

7.2 Remplacement

Le remplacement est une technique que nous avons mise au point pour contourner le problème de l'explosion combinatoire du nombre d'états d'un système. De ce point de vue, un travail

de réduction du nombre d'états a déjà été effectué en partie grâce à la fusion des états par interprétation abstraite. Cependant, afin de pouvoir traiter des systèmes toujours plus grands, toujours plus complexes, le remplacement introduit un degré d'abstraction supplémentaire.

Cette section reprend donc la méthodologie développé en section 3.2 sur la définition du remplacement. Nous y avons distingué deux types de remplacement : le remplacement d'un système à valider (dont la contrepartie logique est l'hypothèse) et le remplacement d'un système qui valide (dont la contrepartie logique est la conclusion). Dans SystemD, nous avons choisi de ne mettre en place que le remplacement d'hypothèse et pas le remplacement de conclusion, parce que les propriétés sont généralement des systèmes assez simples avec relativement peu de configurations et qui ne nécessitent pas de simplifications combinatoires. Nous donnons dans cette section un algorithme de vérification du remplacement d'hypothèse.

Nous rappelons tout d'abord la méthodologie employée. Le remplacement consiste utiliser un système à la place d'un autre pour valider une propriété du second. Pour cette vérification, nous comparons le graphe d'états abstraits de chacun des systèmes, c'est-à-dire que nous comparons les abstractions des deux systèmes. Si bien que quel que soit le résultat, on ne peut rien conclure sur la comparaison des deux systèmes originaux. En effet, l'abstraction d'un système est un sur-ensemble des comportements du système. La méthodologie consiste à vérifier qu'un système peut être remplacé par l'abstraction d'un autre, ce qui reste correct dès lors que l'on reste dans le monde abstrait lorsqu'on utilise le système remplaçant. Pour savoir si M' peut remplacer M, on commence par construire les graphes d'états abstraits de $M^\sharp$ et de $M'^\sharp$ de M et de M'. À cette étape, on sait que la sémantique concrète de $M^\sharp$ contient toutes les traces de la sémantique concrète de M. Si à chacune des traces de $M^\sharp$, on arrive à faire correspondre une trace de $M'^\sharp$ qui la remplace, on saura que à partir d'une trace de M, on peut construire une trace de $M'^\sharp$ qui la remplace.

Tout d'abord, nous allons énoncer un critère qui suffit à montrer que l'abstraction d'un module peut remplacer celle d'un autre. Nous donnerons la preuve de cette implication et enfin nous donnerons un algorithme qui permet de vérifier que le critère est respecté étant donné deux modules et une interface.

7.2.1 Critère de Remplacement

Le travail va se faire sur les graphes d'états abstraits des modules. Le critère à vérifier pour assurer que l'abstraction $M'^\sharp$ d'un module M' peut remplacer l'abstraction $M^\sharp$ d'un module M sans considérer un ensemble I de variables et d'évènements exclus est le suivant : il faut pouvoir projeter chaque configuration de $M^\sharp$ dans une configuration de $M'^\sharp$ telle que la valeur abstraite d'une variable en dehors de I en $M^\sharp$ est plus petite (ou plus fine, ou plus précise) quelle celle en $M'^\sharp$. Les évènements en dehors de I doivent avoir la même valeur de notification : ils sont présents dans une configuration de $M^\sharp$ si et seulement si ils le sont dans leur configuration image en $M'^\sharp$. De plus, si aucun programme de M est en **fail**, aucun programme de M' ne doit l'être dans la configuration image. Ceci vérifie l'inclusion des états en dehors de variables et d'évènements exclus et des configurations des programmes de M'. Ensuite, en suivant la définition de remplacement d'une trace, nous avons également des vérifications à effectuer au niveau des transitions. Une transition Univers entre deux configurations dans $M^\sharp$ doit

correspondre à deux transitions Univers entre les configurations images respectives dans $M'^\sharp$. Pour le cas d'une transition locale, nous sommes un peu plus restrictif que dans la définition puisque nous imposons qu'elle corresponde à une transition locale exactement suivie d'une transition Univers. Enfin, la transition de l'ordonnanceur entre des configurations dans $M^\sharp$ doit correspondre à une transition ordonnanceur *de même type* suivie d'une transition Univers. Par "de même type", nous signifions que si l'ordonnanceur ne fait rien dans $M^\sharp$, alors il ne doit rien faire non plus dans $M'^\sharp$. S'il effectue un changement de phase, alors il doit effectuer le même changement de phase où, dans le cas d'un passage de la phase de mises à jour à la phase comportement, l'avancement du temps est le même.

Pour exprimer notre critère, nous supposons que le graphe d'états abstraits d'un module a ses transitions étiquetées soit par *Univers* si la transition correspond à une transition Univers, soit par *Prog* si la transition est effectuée par l'exécution d'un programme local, soit par *SchedNothing* si la transition se fait par une exécution de l'ordonnanceur qui ne fait rien, soit par *SchedPhase* si la transition se fait par une exécution de l'ordonnanceur qui passe de la phase comportementale à la phase de mises à jour, soit par $\mathit{SchedPhase}(\tau)$ si la transition se fait par une exécution de l'ordonnanceur qui passe de la phase de mises à jour à la phase comportementale avec avancement du temps de τ fraction. On désigne par *label* un élément de l'ensemble $\{\mathit{Univers},\ \mathit{Prog},\ \mathit{SchedNothing},\ \mathit{SchedPhase}\} \cup \{\mathit{SchedPhase}(\tau) \mid \tau \in \mathbb{N}\}$.

Le but du critère de remplacement d'un module M par un module M' étant donné un ensemble I de variables et d'évènements exclus est de déterminer une fonction $\pi \in \mathcal{R}_M \to \mathcal{R}_{M'}$ qui satisfait trois conditions. Pour cela, on suppose que $\langle F^\sharp, T^\sharp \rangle$ est le graphe d'états abstraits de M et $\langle F^{\sharp'}, T^{\sharp'} \rangle$ celui de M'.

La première condition concerne les valeurs des variables et la présence ou non des évènements :

$$(1) \quad \forall r_M \in \mathcal{R}_M,\ \left\{ \begin{array}{ll} & \forall x \in I,\ F^\sharp[r_M](x) \sqsubseteq F^{\sharp'}[\pi[r_M]](x) \\ \text{et} & \forall e \in \mathcal{E}_M \cap I,\ e \in r_M.E \Leftrightarrow e \in \pi[r_M].E \end{array} \right.$$

La seconde condition concerne la préservation de la sûreté :

$$(2) \quad \forall r_M \in \mathcal{R}_M,\ \mathrm{Safe}(r_M) \Rightarrow \mathrm{Safe}(\pi[r_M])$$

La fonction *Safe* prend normalement un module en argument, mais comme on considère des configurations restreintes, on sait automatiquement de quel module il s'agit.

La troisième et dernière condition concerne le respect des transitions :

$$(3) \quad \forall r_M\ r'_M \in \mathcal{R}_M\ \mathrm{label},\ (r_M, \mathrm{label}, r'_M) \in T^\sharp \Rightarrow \exists r_{M'},\ \left\{ \begin{array}{ll} & (\pi[r_M], \mathrm{label}, r_{M'}) \in T^{\sharp'} \\ \text{et} & (r_{M'}, \mathrm{Univers}, \pi[r'_M]) \in T^{\sharp'} \end{array} \right.$$

Le critère de remplacement établi qu'il existe une projection qui respecte les trois conditions.

Définition 7.2 Critère de Remplacement $\rightsquigarrow_\cdot \in \mathcal{M} \times \mathcal{M} \times \mathcal{P}(\mathcal{X} \cup \mathcal{E}) \to \mathit{Prop}$

Le critère de remplacement de M par M' étant donné un ensemble I de variables et d'évènements exclus se note $M' \rightsquigarrow_I M$ et est défini comme suit :

$$M' \rightsquigarrow_I M \quad \triangleq \quad \exists \pi \in \mathcal{R}_M \to \mathcal{R}_{M'},\ (1) \text{ et } (2) \text{ et } (3)$$

La preuve suivante montre la correction du critère vis-à-vis des abstractions des modules, c'est-à-dire que le critère est une condition suffisante pour effectivement remplacer l'abstraction d'un module par l'abstraction d'un autre. On note $M^\sharp$ l'abstraction d'un module M (pour être totalement rigoureux, c'est la concrétisation de son abstraction).

Proposition 7.1 (Correction du Critère de Remplacement)

$$\forall M\ M'\ I,\ M' \rightsquigarrow_I M \Rightarrow M^\sharp \rhd_{I/\Rightarrow} M^{\sharp'}$$

Preuve : la preuve est assez immédiate vu que le critère suit fidèlement la définition du remplacement. Il faut juste se rappeler que les états d'un graphe d'états abstraits sont abstraits, et que leur complétion représente un ensemble d'états où les programmes en dehors du module peuvent prendre n'importe quelle configuration.

7.2.2 Algorithme de Vérification du Critère

Nous proposons un algorithme qui permet de vérifier s'il existe bien une projection qui satisfait le critère de remplacement d'un module par un autre modulo une interface. Cet algorithme suppose que l'on dispose de configurations initiales des modules. Ceci parce que nous traiterons toujours les modules à partir de configurations initiales, et ensuite parce que cela réduit considérablement l'ensemble des possibilités de projection. L'algorithme procède en deux temps. Le premier temps met en place une heuristique afin de réduire rapidement les possibilités de projection. Cette heuristique consiste à associer à chaque configuration du remplacé un ensemble de configurations du remplaçant qui peuvent satisfaire le critère. Un tableau *Tab* contient cet ensemble de contraintes pour chaque configuration du remplacé. Il est raffiné au fur et à mesure de l'algorithme selon les transitions des graphes d'états abstraits : si *Tab* associe à une configuration r la configuration $Tab[r] = \{r'\}$, cela signifie que la seule possibilité de projection de r est r'. Dans ce cas, on sait également que les configurations qui s'obtiennent à partir de r par une exécution d'un programme doivent avoir pour image par projection une configuration qui s'obtient à partir de r' par une exécution d'un programme puis d'une transition Univers. On procède de même pour les transitions ordonnanceur (notons qu'étant donnée une action de l'ordonnanceur, une configuration se déplacera vers une unique autre). Dans le cas des programmes, on fait l'union des possibilités.

On décrit l'algorithme par le point fixe d'une fonction qui prend en argument une pile de configurations à traiter et un tableau qui associe à chaque configuration du remplacé un ensemble de configurations du remplaçant. On itère une fonction de mise à jour des contraintes tant qu'il reste des configurations à traiter, c'est-à-dire tant que la pile n'est pas vide. L'algorithme vérifie que le critère de remplacement d'un module M par un module M' avec un ensemble I de variables et évènements exclus est respecté. On suppose que $\langle F^\sharp, T^\sharp \rangle$ est le graphe d'états abstraits de M et $\langle F^{\sharp'}, T^{\sharp'} \rangle$ celui de M'.

Étant données une configuration $r_M \in \mathcal{R}_M$ et une configuration $r_{M'} \in \mathcal{R}_{M'}$, on note $r_M \sqsubseteq_I r_{M'}$ le prédicat suivant, qui établit que les conditions de projection (1) et (2) (valeurs des variables non exclues plus grossières, évènements non exclus pareillement émis et sûreté préservée) sont satisfaites :

$$r_M \sqsubseteq_I r_{M'} \quad = \quad \left\{ \begin{array}{ll} & \forall x \notin I,\ F^\sharp[r_M].\rho^\sharp(x) \sqsubseteq F^{\sharp'}[r'_{M'}].\rho^\sharp(x) \\ \text{et} & \forall e \notin I,\ e \in (r'_M.E \Leftrightarrow e \in r'_{M'}.E) \text{ ou } e \notin \mathcal{E}_{M'} \\ \text{et} & \text{Safe}(r'_M) \Rightarrow \text{Safe}(r'_{M'}) \end{array} \right.$$

L'algorithme commence par vérifier que les environnements abstraits associés aux configurations initiales de M et de M' satisfont bien ce prédicat.

Nous définissons l'ensemble des configurations de M' candidates pour la projection d'une configuration de M sachant de quelle configuration elle provient et étant donné un tableau de contraintes :

$$\text{Can} \in (\mathcal{R}_M \to \mathcal{P}(\mathcal{R}_{M'})) \times \mathcal{R}_M \times \mathcal{R}_M \to \mathcal{P}(\mathcal{R}_{M'})$$

$$\text{Can}(\text{Tab}, r_M, r'_M) = \left\{ r'_{M'} \quad | \quad \exists \text{label},\ \exists r^1_{M'} \in \text{Tab}[r_M] \text{ et } r^2_{M'}, \begin{array}{ll} & (r_M, \text{label}, r'_M) \in T^\sharp \\ \text{et} & (r^1_{M'}, \text{label},\ r^2_{M'}) \in T^{\sharp'} \\ \text{et} & (r^2_{M'}, \text{Univers},\ r'_{M'}) \in T^{\sharp'} \\ \text{et} & r'_M \sqsubseteq_I r'_{M'} \end{array} \right\}$$

Au cours de l'algorithme, on met à jour l'ensemble de configuration de M' qui seront associées à une configuration de M avec l'intersection des configurations précédemment associées et des configurations candidates. La fonction f à itérer sélectionne la configuration en tête de pile et met ainsi à jour l'ensemble des contraintes associées à chaque configuration qui la suit. Les autres configurations gardent leurs contraintes. On ajoute à la pile toutes les configurations dont les contraintes ont ainsi été modifiées.

$$f(\text{Stack}, \text{Tab}) \quad = \quad \left\{ \begin{array}{ll} (\text{Stack}, \text{Tab}) & \text{si } \textit{Stack} \text{ est vide} \\ (\text{Stack}', \text{Tab}') & \text{sinon} \end{array} \right.$$

où $r_M = \text{Top}(\text{Stack})$

$$\forall r'_M,\ \text{Tab}'[r'_M] = \left\{ \begin{array}{ll} \text{Tab}[r'_M] \cap \text{Can}(\text{Tab}, r_M, r'_M) & \text{si } \exists \textit{label},\ (r_M, \textit{label},\ r'_M) \in T^\sharp \\ \text{Tab}[r'_M] & \text{sinon} \end{array} \right.$$

$$\text{Stack}' = \text{Push}(\text{Pop}(\text{Stack}), \{r'_M \mid \text{Tab}'[r'_M] \neq \text{Tab}[r'_M]\})$$

On itère la fonction f jusqu'à obtention d'un point fixe, et à partir d'une pile qui contient la configuration initiale de M et d'un tableau qui associe à la configuration initiale de M la configuration initiale de M' et qui associe aux autres configurations de M l'ensemble de toutes les configurations de M'. La première partie de l'algorithme s'achève avec le calcul du tableau en point fixe, noté $\textit{Tab}_1$, qui restreint considérablement les possibilités.

$$\text{Tab}_1 = f^i(\text{Stack}_0, \text{Tab}_0) \quad \text{où} \left\{ \begin{array}{l} f^i(\text{Stack}_0, \text{Tab}_0) = f^{i+1}(\text{Stack}_0, \text{Tab}_0) \\ \text{Stack}_0 = \text{Push}(r_{M_0}, \text{EmptyStack}) \\ \text{Tab}_0[r_{M_0}] = r_{M'_0} \\ \forall r_M \neq r_{M_0},\ \text{Tab}_0[r_M] = \mathcal{R}_{M'} \end{array} \right.$$

Pour savoir si la projection existe, il ne nous reste plus qu'à enquêter sur toutes les combinaisons d'images possibles d'après $\textit{Tab}_1$. C'est là que s'exprime l'intérêt de la première partie de l'algorithme : essayer toutes les combinaisons possibles sans restriction est bien trop lourd. Le critère de remplacement du module M par le module M' avec un ensemble I de variables et évènements exclus est respecté si :

$$\exists \pi \in \mathcal{R}_M \to \mathcal{R}_{M'},\ \left\{ \begin{array}{ll} & \forall r_M,\ \pi[r_M] \in \text{Tab}_1[r_M] \\ \text{et} & \forall r'_M\ \textit{label},\ (r_M, \textit{label}, r'_M) \in T^\sharp \Rightarrow (\pi[r_M], \textit{label}, \pi[r'_M]) \in T^{\sharp'} \end{array} \right.$$

L'algorithme peut s'arrêter dès qu'il trouve une solution, peu importe laquelle. Notons d'ailleurs que s'il existe un r_M qui ne peut pas avoir d'image, c'est-à-dire tel que $\textit{Tab}_1[r_M] = \emptyset$,

l'algorithme s'arrête immédiatement en déterminant qu'il n'existe pas de projection satisfaisante.

Pour résumer, l'algorithme de vérification de remplacement d'un module M par un module M' avec un ensemble I de variables et évènements exclus retourne un booléen en procédant comme suit :

Algorithme 7.3 (Vérification du Remplacement)

1. Analyse de M et de M'. On applique l'algorithme 6.7 de calcul du graphe d'états abstraits (fini) sur M et sur M'.
2. Initialisation du point fixe. Une pile contient la première contrainte : l'image de la configuration initiale de M est la configuration initiale de M'. L'algorithme s'arrête en retournant **faux** si ces configurations ne respectent pas les conditions sur les environnements abstraits qui leur sont associés (prédicat $\sqsubseteq_I$).
3. Boucle d'itération. Si la pile est vide, aller en 6.
4. On retire la première configuration de la pile et pour chacun de ses voisins dans le graphe d'états abstraits de M, on recherche quelles sont les configurations de M' qui peuvent leur correspondre selon les transitions du graphe d'états abstraits de M' et les configurations de M' déjà associées à la configuration de M qui était en tête de pile.
5. On ajoute à la pile toutes les configurations de M qui ont été mises à jour à l'étape précédente et on retourne en 2. Fin de la boucle d'itération.
6. S'il existe un chemin correctement labélisé en suivant les contraintes calculées précédemment, alors retourner **vrai**. Sinon retourner **faux**.

Correction de l'Algorithme. Le déroulement de l'algorithme est une image fidèle des conditions du critère de remplacement. En effet, l'invariant de point fixe est le fait qu'à chaque instant, les configurations de M' associées à celle de M sont un sur-ensemble des configurations qui peuvent satisfaire le critère. Pour s'en convaincre, il suffit d'observer comment ont été déduites ces configurations : la construction respecte rigoureusement les conditions de remplacement (valeurs des variables, présence des évènements, sûreté, labélisation des chemins).

Terminaison de l'Algorithme. Le variant est le couple formé de la taille de la pile et du tableau de contraintes : le tableau de contraintes est un tableau d'ensembles où chacun ne peut que décroître selon l'ordre bien fondé $\supseteq$ (de plus petit élément $\emptyset$). Si lors d'une itération aucun ensemble n'a décru, c'est la taille de la pile qui décroît puisque cela signifie qu'aucune contrainte supplémentaire n'a été ajoutée, et donc qu'on a dépilé sans rempiler.

7.3 Discrimination

La définition d'un système discriminant (voir section 3.4) s'articule autour de trois conditions. Ainsi, un module $M_{\mathcal{D}}$ discrimine un module M si :

- $M_{\mathcal{D}}$ est sûre : $\forall t \in [\![M_{\mathcal{D}}]\!],\ \mathit{Safe}(M_{\mathcal{D}}, t)$;
- $M_{\mathcal{D}}$ n'intervient pas sur les variables et évènements de M : $W(M_{\mathcal{D}}) \cap (R(M) \cup W(M)) = \emptyset$;
- $M_{\mathcal{D}}$ ne bloque pas les exécutions de M : $M_{\mathcal{D}} \not\prec M$.

Parmi ces conditions, c'est le fait que $M_{\mathcal{D}}$ n'intervient pas sur les variables et évènements de M qui est la plus simple à vérifier : c'est purement syntaxique.

Pour vérifier que $M_{\mathcal{D}}$ est sûre, plutôt que de construire le graphe d'états abstraits et de s'assurer que **fail** n'est pas accessible, on utilise un critère syntaxique : aucun programme de $M_{\mathcal{D}}$ ne doit contenir le point de contrôle **fail** ; automatiquement, cela signifie que **fail** n'est pas atteignable. Nous faisons ce choix car un discriminant n'a pas vocation à exprimer des propriétés ou des hypothèses de validité, mais à apporter de la précision aux analyses en ajoutant ses configurations au système discriminé.

Il nous reste donc une condition à vérifier : le non bloquage. Là encore, nous n'allons pas donner un critère qui respecte rigoureusement la définition, mais un critère plus général qui suffit à ce que la condition de non bloquage soit vérifiée. D'après la sémantique de KernelD, un programme ne peut en bloquer un autre qu'au niveau de l'ordonnanceur. C'est le seul moment d'une exécution où un programme qui n'écrit pas sur les variables ou évènements d'un autre peut encore l'influencer parce que la réduction d'un statut par l'ordonnanceur dépend du statut des autres programmes. En réalité, ceci n'est vrai que dans un cas particulier : la réduction temporelle qui dépend du temps minimal apparaissant dans un statut.

Comparaison de configuration. On définit la comparaison de configuration laissée en paramètre en section 3.3 sur le non bloquage et on montre que les hypothèses que nous avions posées sont effectivement vraies avec cette définition. Étant données deux configurations c et c', le fait que $c < c'$ doit signifier que c' est moins contraint que c au niveau des évènements, qu'il se situe à moins de réduction de *ready*.

Définition 7.4 Comparaison de configuration $<\in \mathcal{C} \times \mathcal{C} \to \mathit{Prop}$

$$(q, \omega) < (q', \omega') \quad \triangleq \quad q = q' \text{ et } \omega < \omega'$$

Définition 7.5 Comparaison de statut $<\in \mathcal{W} \times \mathcal{W} \to \mathit{Prop}$

$$\omega < \omega' \quad \triangleq \quad \begin{cases} & \omega \not\rhd^{-} \mathit{ready} \\ \text{et} & \omega \neq \omega' \\ \text{et} & \begin{cases} & \exists E,\ \omega \rhd_E \omega' \\ \text{ou} & \exists E,\ \exists \tau,\ \omega' = \rhd_\tau(\rhd_E(\omega)) \end{cases} \end{cases}$$

La première hypothèse que doit vérifier cette définition est le fait que la sûreté des configurations reste la même lorsque deux configurations sont comparables : $\forall c\ c',\ c < c' \Rightarrow (\mathrm{Safe}(c) \Leftrightarrow \mathrm{Safe}(c'))$. Ce qui est clairement vrai puisque le fait que $c < c'$ signifie que c et c' sont dans le même point de contrôle.

La seconde hypothèse concerne une relation entre les exécutions possibles depuis deux configurations comparables : $\forall \sigma\ p\ c,\ \sigma.C(p) < c \Rightarrow \{\sigma' \mid \sigma \xrightarrow{p} \sigma'\} \subseteq \{\sigma' \mid \sigma.C[p \leftarrow c] \xrightarrow{p} \sigma'\} \cup \{\sigma\}$.

Cette condition signifie que les exécutions possibles depuis une configuration c plus petite que c' sont les mêmes que celles possibles depuis c' ou alors consiste en l'identité (le programme ne fait rien). Cette hypothèse est clairement vérifiée par la sémantique KernelD. En effet, si $c < c'$, alors $c.\omega \not\rhd^{-} \mathit{ready}$. Donc un programme en c ne pourra que boucler sur l'état courant.

Enfin, la troisième et dernière hypothèse énonce que l'ordonnanceur doit obligatoirement changer en une configuration plus grande (ou ne rien faire) les configurations des programmes. Cette hypothèse est également vérifiée par l'ordonnanceur KernelD étant donné que l'ordonnanceur ne modifie pas le point de contrôle d'un programme, qu'il ne fait rien en milieu de phase, et qu'il réduit les statuts de tous les programmes selon des évènements lorsque l'on passe de la phase comportementale à la phase de mises à jour, et qu'il réduit les statuts de tous les programmes selon des évènements et le temps lorsque l'on passe de la phase de mises à jour à la phase comportementale.

Critère de non bloquage. Notre critère de non bloquage consiste à énoncer que si les programmes de M' ne notifient pas les évènements lus par ceux de M, et que si un programme de M peut se mettre en attente d'un temps non nul, alors aucun programme de M' ne peut rentrer dans une boucle qui ne contient que des attentes sans temps (ou à temps 0) et des références à des évènements locaux à M', alors M' ne bloque pas M. En effet, plusieurs cas peuvent se présenter, dans lesquels M' ne change pas la présence ou non d'évènements lus par M car ses programmes ne notifient pas les évènements lus par ceux de M :

- Si les programmes de M sont en attente de statuts sans référence au temps ou avec une référence nulle, alors les statuts des programmes de M' n'interviennent pas sur la réduction de ceux de M d'après la définition de l'ordonnanceur.
- Si un programme de M se met en attente d'une fraction de temps non nulle, on sait qu'après un certain nombre d'exécutions des programmes de M', ces derniers seront tous en attente soit d'une fraction de temps non nulle, soit d'évènements de M'. S'ils sont en attente d'une fraction de temps non nulle, le temps avancera et réduira strictement les statuts des programmes de M qui sont en attente d'une fraction de temps. Si un programme de M' est en attente d'un évènement non local à M', alors soit c'est un évènement non local à M et il est possible qu'il ne soit pas notifié par l'Univers, et l'ordonnanceur pourra avancer le temps. Si c'est un évènement local à M alors que les programmes de M sont bloqués, l'évènement sera retiré de la liste des évènements notifiés après la prochaine exécution de l'ordonnanceur et il ne pourra plus être notifié puisque les programmes de M sont bloqués. Et dans ce cas, l'ordonnanceur pourra avancer le temps lors de son exécution suivante.

Si ce critère n'est pas souple, il a le mérite d'être statiquement facilement vérifiable : il suffit de procéder à une analyse syntaxique sur les statuts des programmes de M et les boucles et les statuts des programmes de M'.

Une autre solution que nous avons en perspective serait d'établir un critère moins restrictif à partir du graphe d'états abstraits de $M \otimes M'$ lorsque l'on dispose de celui-ci, pour l'analyse de la validité ou du remplacement par exemple (construire le graphe d'états abstraits uniquement pour la vérification de la discrimination est trop coûteux). À partir du graphe d'états abstraits, une analyse peut permettre d'établir s'il y a des boucles dans l'exécution de $M \otimes M'$ où des programmes de M' s'exécutent et où tous les programmes de M sont en attente, dont certains d'une fraction de temps. Si les programmes de M' n'étaient pas là, le temps pourrait avancer et certains programmes de M pourraient peut-être reprendre leur exécution.

Ceci conclut ce chapitre technique dans lequel nous avons fait le lien entre la partie théorique et le langage SystemD/KernelD en ce qui concerne la validité d'une propriété d'un système, le remplacement et la discrimination. Ces techniques sont illustrées dans le prochain chapitre où nous appliquons les différents algorithmes au système des pompes.

CHAPITRE 8

Expérimentations

Dans ce chapitre, nous montrons l'intérêt de la technologie développée dans le document en étudiant le système des pompes en SystemD. Dans un premier temps, nous considérons une version simplifiée du système avec propriété mais sans remplacement, et nous déroulons l'algorithme 7.1 de vérification de la validité pour montrer comment il parvient à vérifier la propriété. Nous ajoutons alors un remplacement et procédons de même avec l'algorithme 7.3 de vérification du remplacement en montrant le gain apporté sur le nombre de configurations du système. Nous avons présenté les mécanismes utilisés pour l'obtention de tels résultats dans un article [ACV08b], et une partie de la méthodologie de vérification dans un autre [ACV08a].

8.1 Pompes et Propriétés

Nous rappelons le schéma du système des pompes en figure 8.1. Le nom des principaux programmes du système sont inscrits en rouge. Exemple : le système comprend un programme qui joue le rôle de stimulus, appelé `Stim`, un programme qui gère le comportement du premier contrôleur, appelé $\mathtt{c}_1$, etc. Le code SystemD pour ce système se trouve en section 4.1.

8.1.1 Configurations

Le système simplifié contient les éléments suivants :
- un système formé de deux réservoirs connectés par une pompe commune. Le second réservoir dispose également d'une pompe qui le vide. Les pompes sont connectées à des contrôleurs eux-mêmes reliés à leur réservoir d'entrée. La pompe commune a un débit de 5 unités et la pompe sur le second réservoir à un débit de 10 unités. De plus une pompe externe joue le

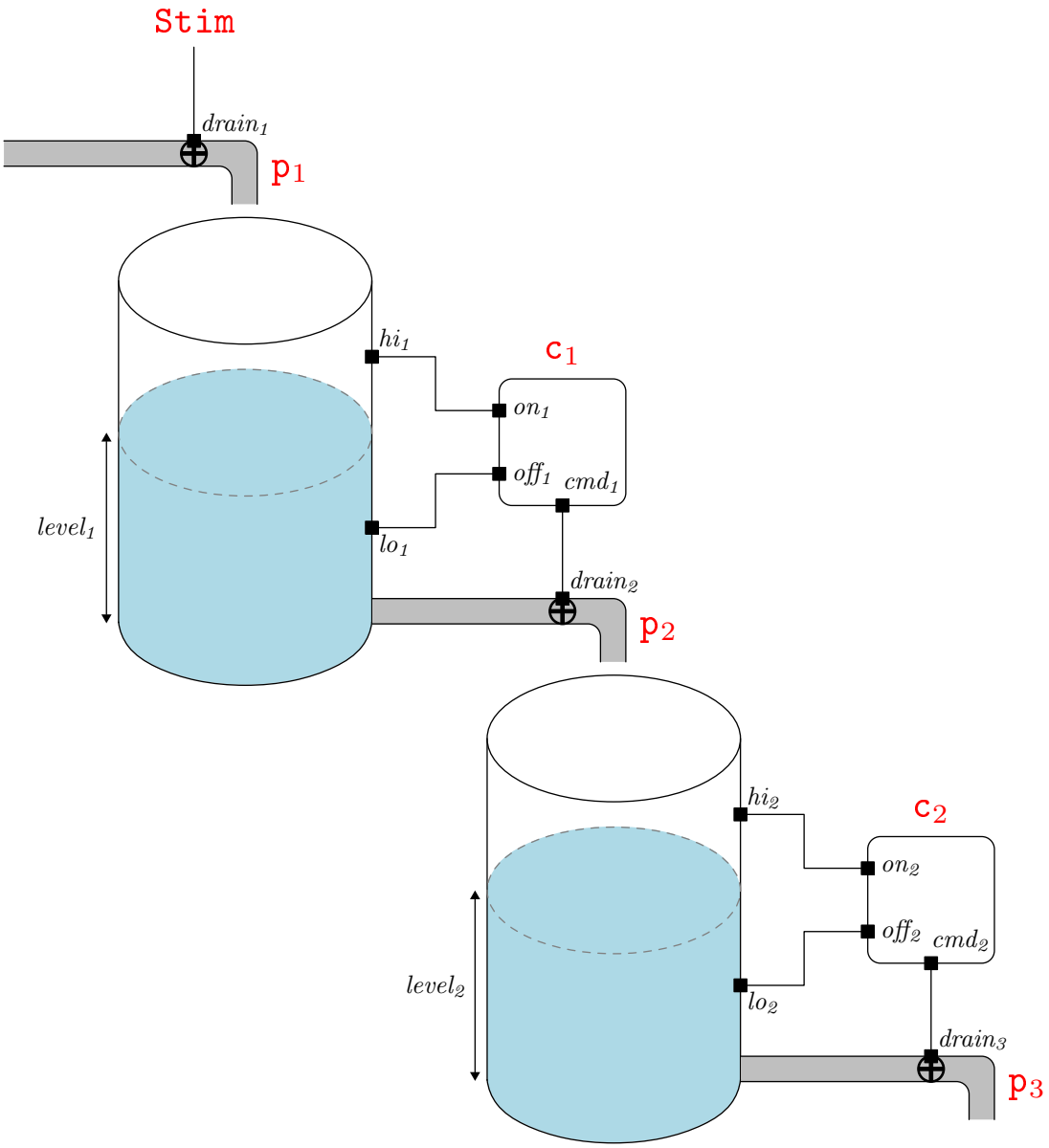

FIG. 8.1 – Le système des pompes

rôle de stimulus en versant du liquide à une vitesse fluctuant de façon indéterminée entre 0 et 5.

- une propriété de ce système sous forme de deux programmes qui vérifient que les liquides dans les réservoirs restent compris entre deux bornes une fois que le liquide dépasse le capteur bas du réservoir.
- deux discriminants, un concernant le premier réservoir et un pour le second, afin de différencier les phases importantes de l'exécution du système pour aider l'algorithme à prouver la propriété.

L'analyse de ce système commence avec la traduction du module SystemD en un module KernelD. Le module obtenu est formé de :

- 21 variables pour représenter les niveaux dans les réservoirs, les valeurs sur les ports, etc ;
- 7 évènements tous implicites (utilisés par les ports) ;
- 11 programmes comportementaux dont une propriété et un discriminant pour chaque réservoir ;
- 7 programmes de mises à jour (implicites et contenus dans les ports).

Nous allons utiliser l'algorithme 7.1 de vérification de la validité par itération chaotique sur ce module. Faisons tout d'abord un point sur l'ensemble des configurations du système. Il faut pour cela en énumérer les programmes avec leurs points de synchronisation (les endroits où le programme se met en attente) et les statuts d'attente qu'ils peuvent prendre en ces points.

- Du côté des programmes comportementaux, on en trouve trois qui concernent les pompes. Ils contiennent chacun un seul point de synchronisation où ils sont soit en attente qu'une seconde passe, soit éligibles. Cela fait donc $2^3 = 8$ combinaisons de configurations pour ces programmes.
- On trouve aussi deux programmes comportementaux qui concernent les contrôleurs qui ne contiennent eux aussi qu'un seul point de synchronisation. En ce point de synchronisation, deux statuts d'attente sont possibles (en attente de `on` ou de `off`, ou éligibles). Ce qui donne $2^2 = 4$ combinaisons de configurations des programmes associés aux contrôleurs.
- Il y a également un programme comportemental qui concerne le stimulus et qui ne contient qu'un seul point de synchronisation, où il a deux statuts possibles : en attente qu'une seconde s'écoule ou éligible. Il y a donc 2 configurations possibles au total pour ce programme.
- Les programmes propriété contiennent quant à eux deux points de synchronisation où ils peuvent attendre deux statuts pour chaque (en attente de l'excitation du capteur `lo` ou éligible pour le premier point et en attente d'une seconde ou éligible pour le deuxième) et un point d'erreur. Il y a donc $5^2 = 25$ configurations possibles pour ces deux programmes réunis.
- Les deux programmes discriminants contiennent chacun trois points de synchronisation où ils ont deux statuts en chaque point (éligible ou en attente d'un évènement, `lo` ou `hi`). Ceci nous fait donc $2^6 = 64$ configurations pour ces deux programmes.
- Les programmes de mises à jour ne contiennent qu'un seul point de synchronisation (le point de contrôle initial) et deux statuts possibles en ce point : en attente de l'évènement de mises à jour *upd* ou éligible. Il y a 7 programmes de mises à jour ce qui donne $2^7 = 128$ combinaisons possibles de configurations pour ces programmes.

Les possibilités de configurations des programmes sont donc au nombre de $8 \times 4 \times 2 \times 25 \times 64 \times 128 = 13.107.200$. On multiplie ce résultat par le nombre possible de valeurs de notification pour les 7 évènements du système, ce qui donne $13.107.200 \times 2^7 = 1.677.721.600$ de configurations possibles. Ceci étant dit, l'immense majorité des configurations ne sont pas accessibles depuis la configuration initiale du module dans laquelle les évènements ne sont pas encore notifiés. De plus, de nombreuses combinaisons de notification des évènements ne sont pas atteignables elles non plus. Notons qu'en model checking classique, ce nombre de configurations devrait encore être multiplié par le nombre de valeurs potentielles pour chaque variable du module, qui est indéterminé pour l'instant et qui peut être très grand (on s'attend à ce que les liquides dans les réservoirs aient un niveau entre 0 et un nombre proche de 900, mais rien n'indique que ce nombre ne va pas être dépassé...).

Dans le cas d'une itération chaotique comme nous allons utiliser, nous n'avons pas besoin de construire l'ensemble de toutes les configurations a priori. On commence simplement à partir d'une configuration initiale et on exécute le module dans l'abstraction, si bien qu'on ne considère que les configurations que l'on obtient au fur et à mesure du déroulement de l'algorithme. On part d'une configuration où les contrôleurs sont en attente et où le stimulus et les programmes associés aux pompes sont éligibles. De plus, les propriétés et les discriminants attendent que l'évènement associé au capteur bas de leur réservoir respectif soit notifié. On associe à chaque configuration du système un environnement où les valeurs des variables booléennes sont exactes et où les valeurs des variables entières sont abstraites par des intervalles (défini en annexe A.6). Le résultat est un treillis comme fonction d'un ensemble vers un treillis (voir l'annexe A.7 pour les combinaisons de treillis). Pour l'exemple, on ne présente les résultats que sur quelques variables et évènements qui concernent le premier réservoir. On schématise les états abstraits comme sur la figure 8.2 qui représente la configuration initiale du système que nous appellerons *PumpsReady*, ainsi que son environnement abstrait initial.

PumpsReady

Stim : *ready* $\quad$ p_1 : *ready* $\quad$ c_1 : $on_1 \mid off_1$ $\quad$ p_2 : *ready*	lo_1 : **true** $\quad$ hi_1 : **false**
$Prop_1$: lo_1 $\quad$ $disc_1$: $(0, lo_1)$ $\quad$ updates : *upd*	$drain_1$: 0..5 $\quad$ $drain_2$: 0..0
Évènements : $\emptyset$	$level_1$: 0..0

FIG. 8.2 – La configuration *PumpsReady*

La figure contient les informations suivantes : sur fond gris se trouve la configuration du système, c'est-à-dire un statut et un point de contrôle (omis lorsqu'il se déduit du statut) pour chaque programme du système (mais nous nous contentons de quelques uns d'entre eux) et la liste des évènements émis. Nous identifions les configurations des programmes de mises à jour sous le mot `updates`. En effet, ils ont tous le même type de point de contrôle (un seul où ils sont éligibles ou attendent *upd*) et ils ont tous des écritures et des lectures disjointes si bien que nous ferons comme s'ils étaient exécutés *en même temps*. Collé à la configuration et sur fond blanc se trouve l'environnement abstrait associé. C'est une liste d'éléments (un seul dans le cas de cette figure) qui contiennent un booléen pour chaque variable à valeur booléenne et un intervalle pour chaque variable à valeur entière.

Nous rappelons ci-dessous le code de la propriété sur la partie haute du système. Elle consiste à énoncer qu'une fois que le capteur bas a été atteint par le liquide, le niveau reste compris entre deux bornes proches des capteurs bas et haut pendant tout le reste de l'évolution du système.

```
property {
  wait(lo1);
  while (true) {
    assert(290 <= level1 && level1 <= 910);
    wait(1s);
  }
}
```

Pour aider l'algorithme de vérification à inférer cette propriété, nous avons ajouté au système un discriminant qui, par l'apport de ses configurations, va amener plus de précision dans l'analyse. Son code est le suivant :

```
discriminate {
  wait(lo1);
  while (true) {
    wait(hi1);
    wait(lo1);
  }
}
```

8.1.2 Déroulement de l'Algorithme

Phase d'Initialisation

À partir de la configuration *PumpsReady* et de son environnement abstrait associé, voici les différentes possibilités d'évolution du système :
- une transition Univers mène à boucler sur la même configuration avec le même environnement. En effet, aucune variable n'est ouverte dans cet exemple ;
- une transition ordonnanceur bouclera également puisqu'il y a des programmes éligibles ;
- les transitions des programmes locaux qui ne sont pas éligibles (contrôleur, propriété, discriminant et programmes de mises à jour) mènent aussi à boucler ;
- l'exécution de chaque programme éligible (`Stim`, $\mathtt{p}_1$ et $\mathtt{p}_2$) mène à une nouvelle configuration.

L'exécution de `Stim` affecte à la variable $drain_1$ une valeur comprise entre 0 et 5, ce qui ne change donc pas l'environnement. La configuration r_1 obtenue est la même que *PumpsReady* à ceci près que `Stim` est en attente d'une seconde. L'exécution de $\mathtt{p}_1$ consiste à verser un volume de liquide compris entre 0 et 5 dans le premier réservoir. Comme le niveau était de 0 dans ce réservoir, il devient compris entre 0 et 5. La configuration r_2 obtenue est la même que *PumpsReady* mais où $\mathtt{p}_1$ est en attente d'une seconde. Enfin, l'exécution de $\mathtt{p}_2$ ne changera pas l'environnement étant donné que la pompe prendra 0 volume de liquide du premier réservoir pour en déposer 0 dans le second. La configuration r_3 obtenue est la même que *PumpsReady* mais où $\mathtt{p}_2$ est en attente d'une seconde. Les configurations r_1, r_2 et r_3 ainsi que les transitions du graphe d'états abstraits que cela crée sont représentées en figure 8.3, où les transitions bouclantes qui ne changent rien ont été omises (Univers, ordonnanceur et programmes inactifs).

À partir de la configuration, r_1, les programmes $\mathtt{p}_1$ et $\mathtt{p}_2$ peuvent s'exécuter dans un ordre quelconque. À partir de la configuration r_2, ce sont les programmes `Stim` et $\mathtt{p}_2$ qui peuvent s'exécuter, et dans la configuration r_3, ce sont `Stim` et $\mathtt{p}_2$. En fait, à ce stade, les programmes `Stim` et $\mathtt{p}_2$ ne changent rien : les informations apportées par `Stim` sont déjà prises en compte (ouverture de la première pompe d'une valeur entre 0 et 5) et la pompe de comportement $\mathtt{p}_2$ n'est pas encore ouverte. Du coup, toutes les imbrications possibles d'exécution de `Stim`, $\mathtt{p}_1$ et $\mathtt{p}_2$ mènent à la même configuration, appelée *EndBehaviour*, où tous les programmes sont en attente et où la phase comportementale (ou *phase d'évaluation* en SystemC) se termine. On y fusionne les environnements abstraits. Ces nouveaux résultats sont donnés en figure 8.4.

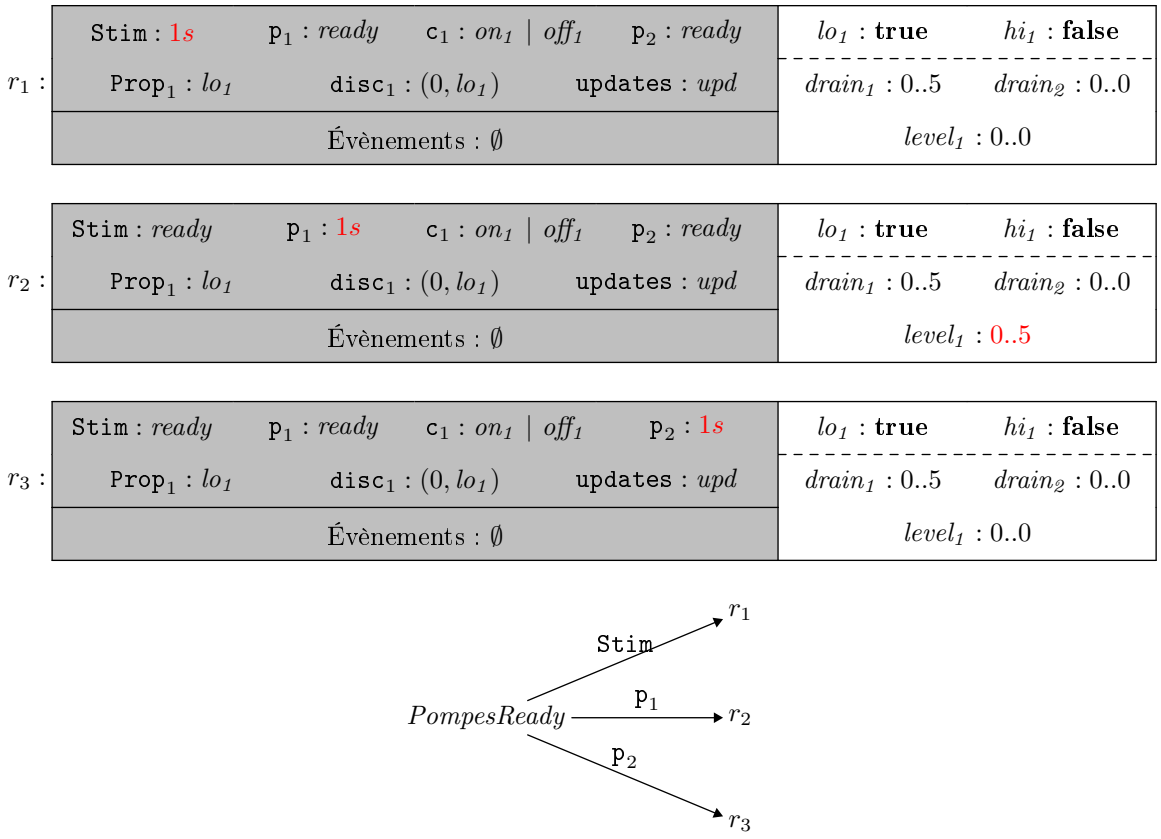

FIG. 8.3 – Configurations intermédiaires

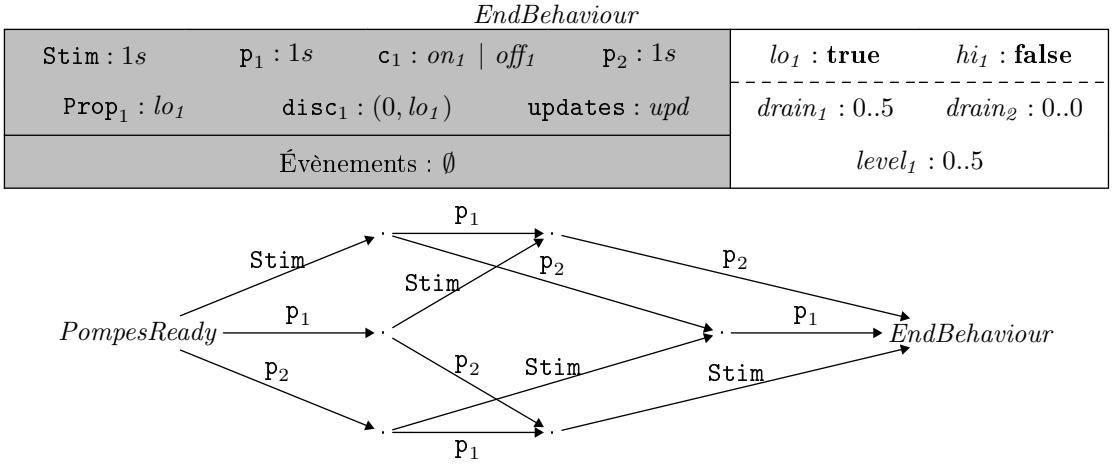

FIG. 8.4 – Premiers résultats de fin de phase comportementale

L'étape suivante va consister à passer en phase de mises à jour. Pour cela, on ajoute l'évènement *upd* à la liste des évènements notifiés et on réduit les statuts des programmes selon cet évènement, ce qui va avoir pour effet de réveiller les programmes de mises à jour. Un seul

signal a potentiellement changé durant la phase comportementale précédente : celui entre le stimulus et la première pompe dont on sait seulement que sa valeur est comprise entre 0 et 5. Deux configurations seront donc créées, l'une avec l'évènement associé qui est notifié et l'autre sans. Mais ceci n'a pas d'incidence (aucun programme n'attend cet évènement) et la réduction suivante va relancer le stimulus et les pompes et on revient à la configuration *PumpsReady*. Cet enchaînement est schématisé en figure 8.5, où l'environnement abstrait a été retiré par souci de clarté lorsqu'il n'est pas modifié (pendant la phase de mises à jour). Les modifications entre une configuration et sa configuration précédente sont reportées en rouge. Notons que la valeur de $level_1$ passe à $0..\infty$ dans la configuration *PumpsReady*, ce qui résulte de l'élargissement de l'ancienne valeur 0..0 par la nouvelle valeur 0..5.

À partir de là, les capteurs bas et haut du réservoir vont être sollicités et l'exécution de la phase comportementale va provoquer plusieurs évolutions possibles. Une première possibilité est le cas où la valeur de lo_1 vaut **true** (c'est-à-dire lorsque le niveau est en dessous de 300) et la valeur de hi_1 vaut **false** (c'est-à-dire lorsque le niveau est en dessous de 900). L'algorithme établit que la valeur de $level_1$ est alors comprise entre 0 et 300. La seconde possibilité est le cas où la valeur de lo_1 vaut **false** (c'est-à-dire lorsque le niveau est au dessus — strictement — de 300) et la valeur de hi_1 vaut **false** (c'est-à-dire lorsque le niveau est en dessous de 900). L'algorithme établit que la valeur de $level_1$ est alors comprise entre 301 et 900. Enfin, le dernier cas est celui où la valeur de lo_1 vaut **false** (c'est-à-dire lorsque le niveau est au dessus — strictement — de 300) et la valeur de hi_1 vaut **true** (c'est-à-dire lorsque le niveau est au dessus — strictement — de 900). L'algorithme établit que la valeur de $level_1$ est alors comprise entre 901 et $+\infty$. Parmi ces trois possibilité, une seule permet de revenir à la configuration *PumpsReady*. En effet, lorsque lo_1 passe à **false**, l'évènement de changement de valeur qui lui est associé va être notifié, ce qui va réveiller les programmes $\texttt{Prop}_1$ et $\texttt{disc}_1$ qui vont changer changer de configuration puis de point de contrôle. Seul le cas où lo_1 vaut **true** et où $level_1$ est compris entre 0 et 300 pourra revenir à la configuration *PumpsReady*. Après rétrécissement, on obtient donc les environnements en figure 8.6 pour *PumpsReady* et *EndBehaviour*, qui représentent la première partie de l'évolution du système, appelée phase *Init*, lorsque le niveau monte jusqu'à atteindre le capteur bas.

Phase de Remplissage et de Vidange

L'environnement de *EndBehaviour* où la valeur de lo_1 est à **false** va évoluer vers une nouvelle configuration. En effet, ce changement de valeur pour lo_1 va notifier un évènement si bien que le discriminant va changer de configuration ; on ne reviendra plus dans un état de la phase *Init*. La propriété va devenir active et vérifie à chaque seconde que le niveau dans le réservoir est bien compris entre 290 et 910. Après point fixe, deux phases se distinguent.

Dans la première phase, appelée *Fill*, le discriminant attend un changement perçu par le capteur haut hi_1. Durant cette phase, la pompe de vidange est inactive et le niveau de liquide monte dans le réservoir à une vitesse comprise entre 0 et 5. Dans cette phase, l'évolution du système revient toujours à un moment ou à un autre dans la configuration de fin de phase comportementale où tous les programmes sont en attente, juste avant que l'ordonnanceur ne réveille les programmes de mises à jour. Et dans cette configuration, deux cas se présentent. Tout d'abord le cas principal où le liquide monte en restant en dessous du capteur haut, de

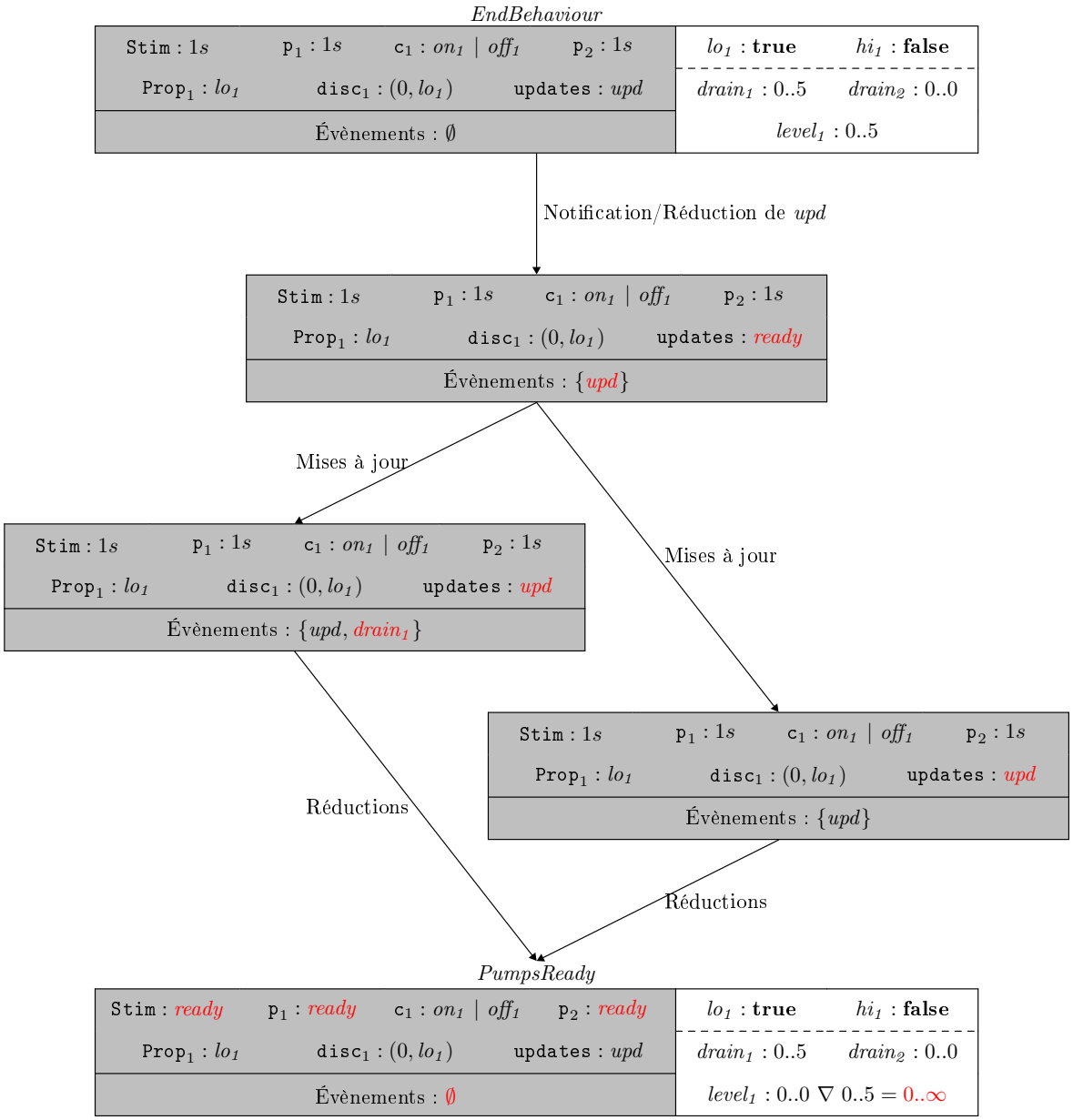

FIG. 8.5 – Premiers résultats sur la phase de mises à jour

sorte qu'il n'est pas sollicité. Le niveau est alors compris entre 301 et 900, et l'évolution peut dans la plupart des cas revenir à cette même configuration un cycle plus tard. Et puis il y a le cas où le niveau est passé juste au dessus, compris entre 901 et 905, de telle sorte que le capteur haut a été sollicité, ce qui va réveiller et faire changer de configuration le discriminant lors de la phase de mises à jour suivante. On change alors de phase. La configuration de fin de phase comportementale pour *Fill* et ses environnements après point fixe sont représentés figure 8.7.

La seconde et dernière phase, appelée *Drain*, suit la phase *Fill* lorsque le niveau a dépassé le

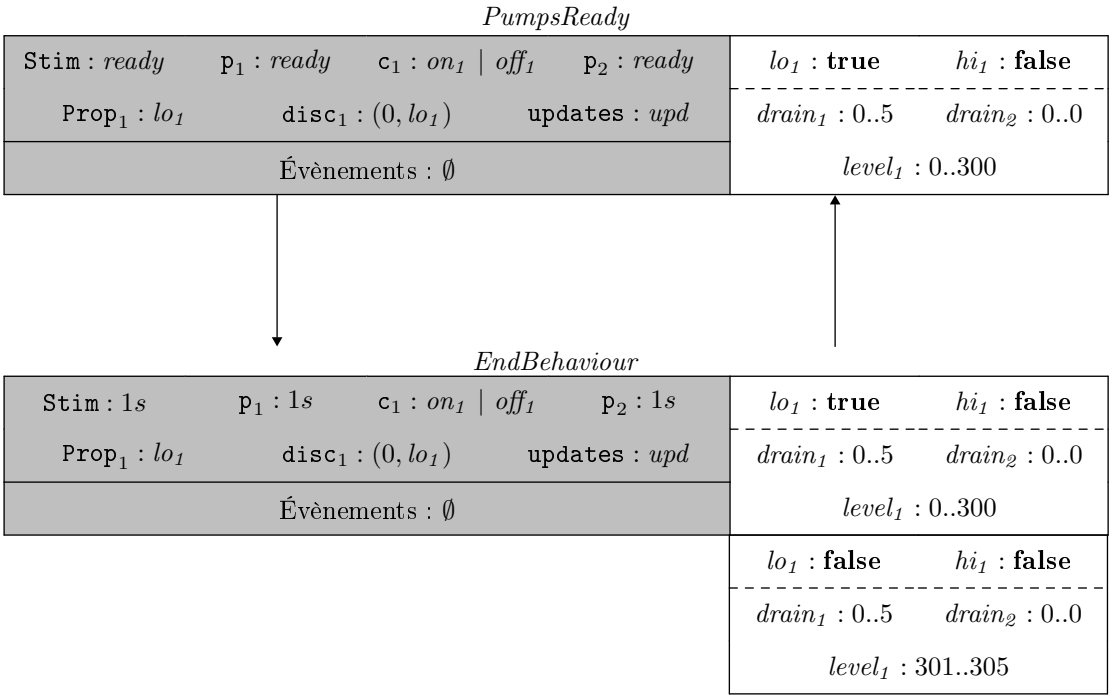

FIG. 8.6 – Phase *Init* après élargissement et rétrécissement

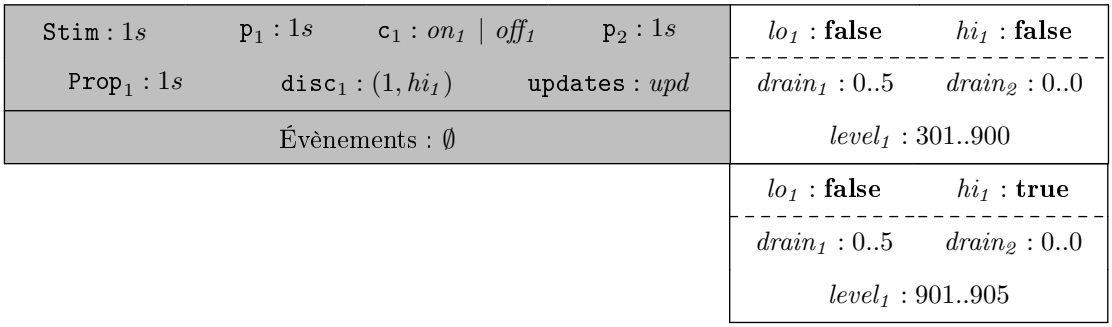

FIG. 8.7 – Configuration principale de la phase *Fill* après élargissement et rétrécissement

capteur haut. Dans ce cas, la notification de changement de valeur pour ce capteur va réveiller le contrôleur qui va ouvrir la pompe pour qu'elle vide le réservoir à une vitesse de 5. De plus, le discriminant va changer de configuration. Durant cette phase comme dans la précédente, l'évolution du système revient toujours dans la configuration de fin de phase comportementale où tous les programmes sont en attente. Deux cas se présentent. Il y a un cas où le niveau descend ou stagne continuellement, et est compris entre 301 et 900. Et puis il y a le cas où le niveau descend en dessous du capteur bas, provoquant un changement de valeur qui conduira à revenir en phase de remplissage *Fill* puisque la pompe sera désactivée par le contrôleur et que le discriminant reviendra dans sa configuration précédente. La configuration de fin de phase comportementale pour *Drain* et ses environnements après point fixe sont représentés figure 8.8.

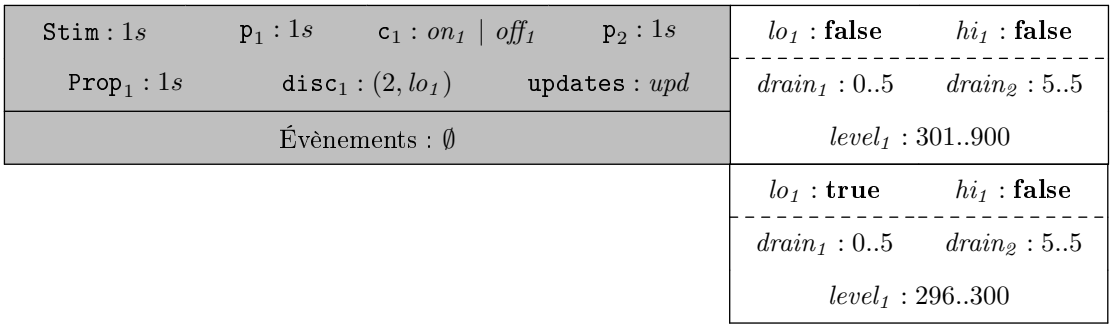

FIG. 8.8 – Configuration principale de la phase *Drain* après élargissement et rétrécissement

La figure 8.9 représente l'évolution globale de la partie haute du système des pompes avec le premier réservoir, entre les phases *Init*, *Fill* et *Drain*, et comme déterminée par l'algorithme 7.1 de vérification de la validité. La partie basse avec le second réservoir se comporte de la même façon à ceci près les phase de remplissage et de vidange sont implicitement dépendante de l'évolution de la partie haute. L'algorithme établit que **fail** n'est pas accessible ce qui garantit que les propriétés sont vérifiées par le système : après la phase d'initialisation de chaque réservoir, le niveau liquide reste compris entre 290 et 910.

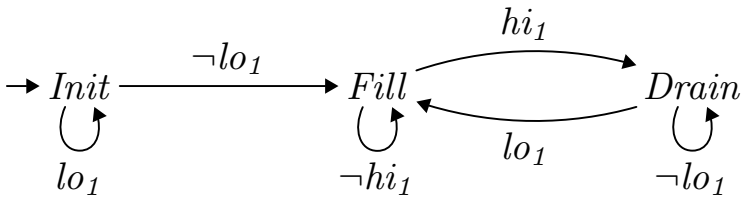

FIG. 8.9 – Résultat du calcul du graphe d'états abstraits sur la partie haute du système

Sans discriminant. La différence entre la phase *Fill* et la phase *Drain* se fait au niveau de la configuration du discriminant. Sans le discriminant, les deux phases se confondent et l'évolution du système contient alors deux phases : la phase d'initialisation et la phase de remplissage et de vidange confondues, ces deux phases restant distinctes par la configuration de la propriété. La conséquence de la fusion des phases *Fill* et *Drain* est catastrophique pour la preuve de la propriété. En effet, les vitesses de vidange en chaque phase vont être fusionnées et sera donc comprise entre 0 et 5. On perd toute précision sur les instants où la pompe est inactive et les instants où elle vide le réservoir ; c'est cette précision que venait apporter le discriminant en changeant de configuration aux instants précis de changement de vitesse de vidange, tout cela très simplement en trois lignes de code. Cette perte de précision de la vitesse de vidange induit une perte de précision sur le niveau de liquide dans le réservoir : à tout moment, il est possible que 5 unités soient ajoutées ou retirées du réservoir. Le point fixe inférera une valeur totalement imprécise ($\infty..\infty$) sur ce niveau, ce qui conduira la propriété à pouvoir se déplacer en **fail**.

8.2 Remplacement

On considère le remplacement de la partie haute du système des pompes comme décrit en figure 4.8 de la section 4.1. Le remplacement met en jeu trois éléments : le système à remplacer (la partie haute du système des pompes), le système remplaçant (constitué dans notre exemple d'un seul programme), et un discriminant pour la partie haute qui sert à apporter de la précision pour inférer la correction du remplacement. Nous allons appliquer l'algorithme 7.3 de vérification du remplacement pour vérifier que le remplaçant peut effectivement remplacer la partie haute du système des pompes auquel on a ajouté un programme supplémentaire : le discriminant. Par sa nature discriminante, ceci assurera que le remplaçant peut effectivement remplacer le système des pompes sans discriminant, comme nous l'avons montré dans le chapitre 3 théorique. Ensuite, nous montrons comment l'algorithme 7.1 de vérification de la validité assure que le système des pompes avec remplacement est correct (ce qui implique qu'il est correct sans remplacement), puis quelle réduction de complexité le remplacement a amené.

8.2.1 Vérification du Remplacement

L'algorithme de remplacement consiste à déterminer si pour chaque configuration du remplacé, il existe une configuration du remplaçant qui prend en compte au moins autant de comportements, et dont les évolutions sont au moins les mêmes. On peut pour cela ne pas considérer les variables locales du remplaçant (à condition que les systèmes avec lesquels on l'assemble ne les lise pas).

On commence par construire le graphe d'états abstraits des deux systèmes mis en jeu. Celui de la partie haute du système des pompes avec discriminant a été développé dans la section précédente. Celui du système remplaçant est très simple : il n'est constitué que d'un seul programme qui contient deux points de contrôle où il peut adopter deux statuts pour chaque (éligible ou en attente d'une seconde). Nous appelons le premier point de contrôle (c'est-à-dire le `wait` situé le plus haut dans le code) 0 et le second 1. Le graphe d'états abstraits du remplaçant est donné en figure 8.10 avec les environnements associés à ses configurations. On simplifie le graphe comme dans la section précédente par souci de lisibilité, en ne reportant que les configurations de fin de phase comportementale.

On distingue deux phases particulières. La première, appelée *Init*, lorsque le programme du remplaçant est en son point de contrôle 0, lo_1 vaut alors obligatoirement **true** même si cette variable est externe pour le remplaçant et peut donc évoluer de manière totalement indéterminée. La valeur du niveau dans le réservoir est quelconque. Une fois que l'Univers fait passer la valeur de lo_1 à **false**, le programme du remplaçant change de configuration et passe en 1 pour ne jamais revenir à la configuration précédente, et boucle sur un environnement où le niveau dans le réservoir est compris entre 295 et 905. C'est ce que nous appelons la phase *Main*.

La vérification est alors assez simple. La phase *Init* du remplaçant correspond naturellement à la phase d'initialisation de la partie haute du système des pompes, lorsque le capteur lo_1 n'a pas encore été atteint. La valeur de $drain_2$ est la même dans toutes les configurations de la

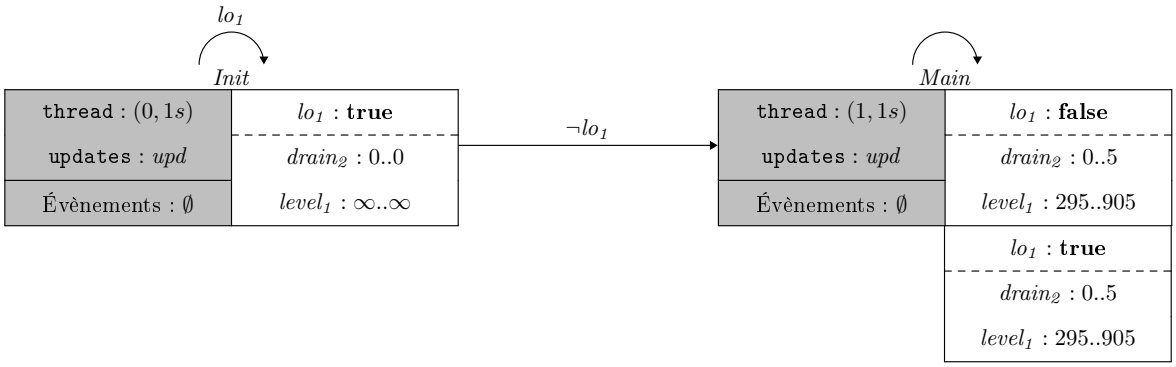

FIG. 8.10 – Résultat du calcul du graphe d'états abstraits sur le système remplaçant

phase d'initialisation de la partie haute que dans les configurations de la phase d'initialisation du remplaçant. En réalité nous n'avions pas besoin d'être aussi précis. La valeur de lo_1 est la même et les valeurs de $level_1$ dans la partie haute (entre 0 et 300) sont effectivement comprises dans celles du remplaçant (entre $-\infty$ et $+\infty$). Lorsqu'un programme de la partie haute s'exécute, le programme remplaçant peut s'exécuter, quitte à ne rien faire s'il n'est pas éligible. L'action de la pompe d'entrée qui consistait à remplir le réservoir est abstraite par le fait que le niveau dans le réservoir est quelconque. L'action du contrôleur est totalement abstraite pendant cette partie puisqu'il n'agit pas et l'action de la pompe de vidange est inutile (elle vide le réservoir de 0 volume de liquide). Les phases *Fill* et *Drain* de la partie haute sont représentées chez le remplaçant par la phase *Main*. En effet, lo_1 y prend une valeur quelconque, $drain_2$ a une valeur comprise entre 0 et 5 chez le remplaçant alors qu'il avait une valeur précisément soit de 0 soit de 5 dans la partie haute. Enfin, alors que le niveau était compris entre 296 et 905 dans la partie haute, phases *Fill* et *Drain* confondues, elle est comprise entre 295 et 905 chez le remplaçant, ce qui contient effectivement les valeurs de la partie haute. Remarquons que ce résultat ne fonctionne qu'avec le discriminant. Sans sa présence, comme nous l'avions décrit dans la section précédente, l'algorithme ne parvient plus à déterminer de valeurs précises pour le niveau de liquide dans la partie haute. Le plongement des états abstraits de la partie haute vers les états abstraits du remplaçant comme l'infère l'algorithme de vérification du remplacement est représenté en figure 8.11.

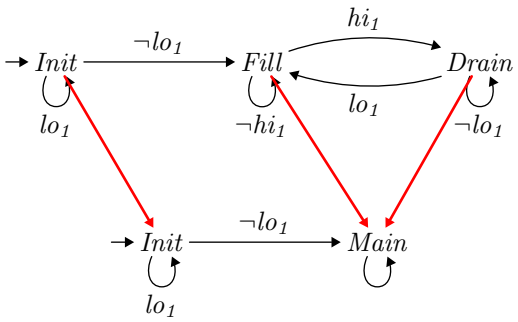

FIG. 8.11 – Plongement des états abstraits de la partie haute dans ceux du remplaçant

Finalement, la logique de remplacement a été la suivante : la vérification de la propriété sur la partie haute du système des pompes a nécessité l'ajout d'un discriminant. Une fois que cette propriété a été prouvée, on a remplacé le système par un système plus simple qui contient les informations nécessaires à la preuve de la propriété sur la partie haute, mais aussi les informations de la partie haute nécessaire à la preuve de la propriété de la partie basse. Nous avons abstrait la partie haute du système des pompes (par notre analyse par interprétation abstraite), apporter de la précision avec un discriminant, puis abstrait à nouveau la partie haute avec un remplacement.

8.2.2 Validité après Remplacement

On reprend l'algorithme 7.1 de vérification de la validité et la façon dont nous avons montré son déroulement sur le système des pompes complet sans remplacement, sauf que nous considérons ici le remplacement de la partie haute. Le code de la partie haute explicitait précisément l'évolution de cette partie. Le remplacement est une abstraction de cette évolution qui suffit à montrer la propriété recherchée sur le système tout entier. Le résultat est grosso modo le produit synchrone des graphes d'états abstraits du remplaçant et de la partie basse.

Quatre phases se distinguent donc. La première est celle où le réservoir de la partie haute se remplit progressivement jusqu'à atteindre son capteur bas. Cela correspond à la phase d'initialisation du remplaçant. La partie basse est alors totalement inactive. La seconde phase correspond à celle où le niveau dans le réservoir de la partie haute oscille entre deux bornes proches de son capteur bas et de son capteur haut, où la vitesse de vidange de la pompe commune entre les parties basse et haute est comprise entre 0 et 5 et où le réservoir de la partie basse peut donc se remplir jusqu'à atteindre son capteur bas. C'est la phase d'initialisation de la partie basse. Notons que pendant cette phase, seule la propriété de la partie haute est active. Durant la troisième et la quatrième phase, le remplaçant a le même comportement que précédemment : le niveau dans le réservoir haut est compris entre 295 et 905 et la vitesse de vidange de la pompe commune est compris entre 0 et 5. Pendant la troisième phase, le niveau dans le réservoir de la partie basse monte jusqu'à atteindre le capteur haut. L'algorithme infère que le niveau de liquide dans le réservoir bas pendant cette phase est compris entre 301 et 910 de la même manière que pour la partie basse (mais la vitesse de vidange de la pompe reliée au réservoir de la partie basse est de 10 au lieu de 5). C'est la phase de remplissage de la partie basse. Enfin, durant la quatrième et dernière, le contrôleur active la pompe. Cette phase correspond donc aux instants où la pompe de sortie du système vide le réservoir de la partie basse à une vitesse de 10. Le niveau pendant cette phase est compris entre 291 et 900. C'est la phase de vidange de la partie basse. Les configurations des différentes phases, leur environnement associé par le calcul du graphe d'états abstraits du système et leur relation pendant l'évolution du système sont représentés figure 8.12 (même simplifications qu'auparavant : seules les configurations de fin de phases comportementales sont indiquées).

D'un point de vue de la validité le constat est clair : aucune configuration où les programmes propriété ont leur contrôle en **fail** n'est atteignable, ce qui signifie que les propriétés sont vérifiées.

8.2.3 Réduction du Nombre de Configurations

Comme les nombres de configurations de deux systèmes se multiplient lorsqu'on les assemble, le gain d'un remplacement vis-à-vis du nombre de configurations d'un système s'évalue par le rapport des nombres de configurations du remplacé et du remplaçant. En pratique, c'est vis-à-vis du nombre de configurations accessibles que la réduction s'exprime, mais ce nombre est plus difficile à obtenir que le nombre de configurations totales, qui représente tout de même une approximation intéressante.

La partie haute du système des pompes est formée d'un stimulus (un programme de deux configurations), de deux pompes (deux programmes de deux configurations chacun), d'un contrôleur (un programme de deux configurations) et de quatre programmes de mises à jour (deux configurations pour chaque). De plus, il comprend quatre évènements. Ajoutons le programme discriminant le concernant qui possède six configurations. La partie haute avec discriminant est donc formée de $2 \times 2^2 \times 2 \times 2^4 \times 2^4 \times 6 = 24576$ configurations.

Le remplaçant quant à lui ne contient qu'un seul programme de cinq configurations, un programme de mises à jour (deux configurations) et un évènement. Le nombre de configuration totale pour le système remplaçant est donc de $5 \times 2 \times 2 = 20$.

Le facteur de réduction en nombre de configurations théorique apporté par le remplacement est donc de $\frac{24576}{2 \times 2 \times 5} = 1228.8$. Cela signifie qu'il y a 1228.8 fois moins de configurations dans un système où la partie haute avec discriminant a été remplacée. On comprend vite l'utilité du remplacement, en particulier dans les systèmes comprenant beaucoup de composant. Le remplacement de deux parties peut rapidement réduire le nombre de configurations d'un système d'un facteur de l'ordre du million.

Évidemment, l'exemple des pompes se prête bien au jeu du remplacement. Plusieurs réserves sont à émettre. Tout d'abord, la réduction concerne la partie haute avec discriminant, ce dernier étant formé de six configurations. Un discriminant est mis en place vis-à-vis d'une propriété. On pourrait imaginer vouloir montrer une propriété qui ne nécessite pas la mise en place d'un discriminant pour la partie haute. Le facteur de réduction apporté par le remplacement ne serait alors plus que de $\frac{1228.8}{6} = 204.8$, ce qui reste somme toute une réduction appréciable. De plus, le remplacement à un coût : celui de la vérification que le remplacement est correct. Comme on cherche à plonger chaque configuration du remplacé dans une configuration du remplaçant, le remplacement gagne à ce que le remplaçant contienne un très petit nombre de configuration vis-à-vis du remplacé (mais après tout, tel est son but). Ce dernier argument favorise tout particulièrement l'utilisation du remplacement d'autant plus que le système dans lequel il utilisé est formé de l'assemblage d'un grand nombre de composants. Afin d'évaluer plus précisément les bénéfices apportés par notre technique de remplacement, nous devons poursuivre les expérimentations avec des systèmes de natures (peu ou beaucoup d'interactions entre composants) et de tailles différentes, nous y reviendrons dans la conclusion du document.

C'est sur ces dernières considérations que nous fermons ce chapitre dans lequel nous avons pu observer les algorithmes de vérification de la validité et de vérification du remplacement

en action sur le système des pompes. Nous avons également montré quel était le facteur de réduction du nombre de configurations apporté par le remplacement dans cet exemple, bien que cela s'accompagne de contraintes qu'il nous faut étudier encore afin de cadrer plus précisément son utilisation de manière plus générale en pratique. Dans le chapitre conclusion qui suit, nous revenons sur cette question et nous présentons quelques perspectives de la méthodologie et des techniques développées tout au long du document.

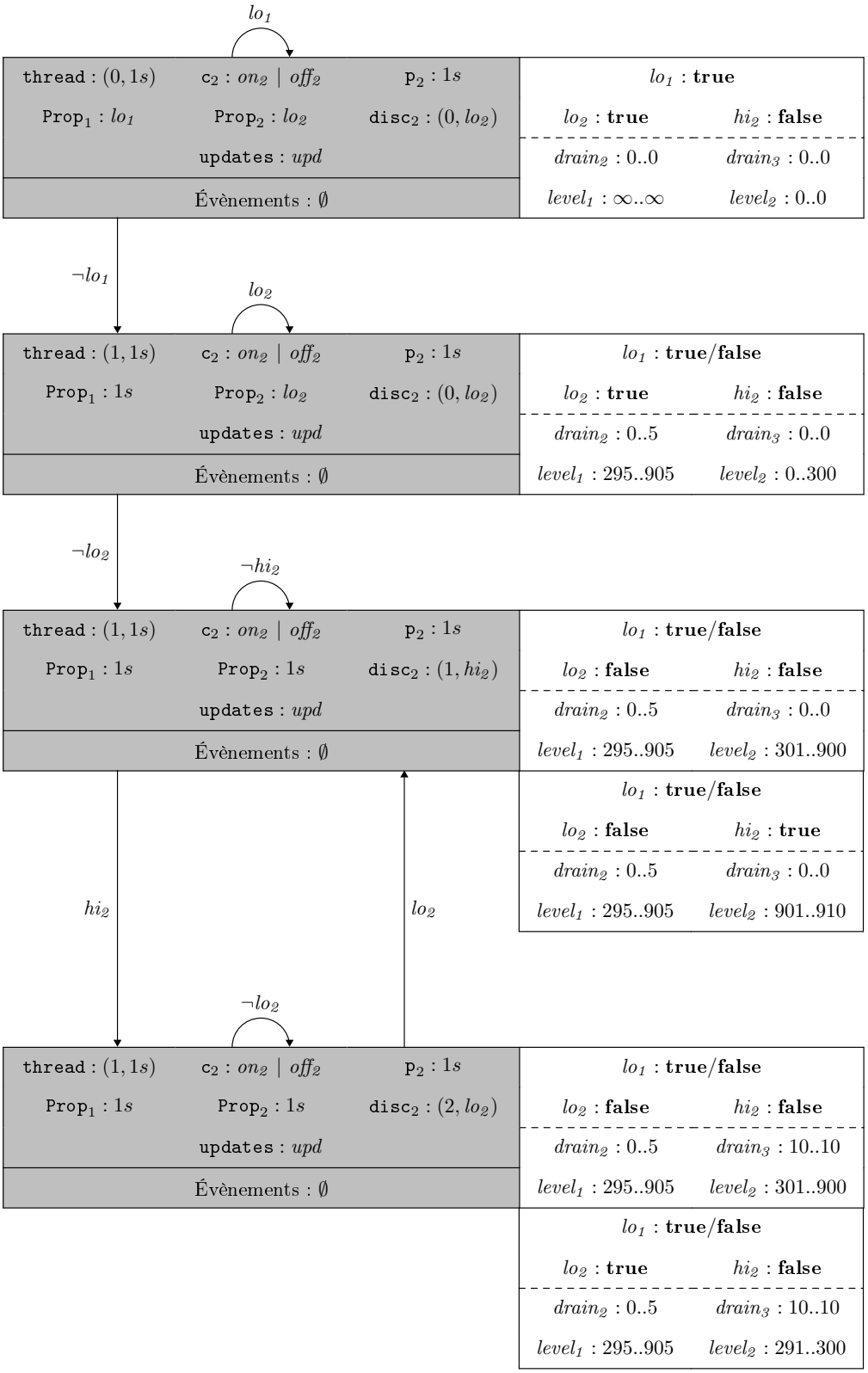

FIG. 8.12 – Graphe d'états abstraits du système des pompes avec remplacement

Quatrième partie

Perspectives et Conclusion

Perspectives et Conclusion

Dans ce document, nous avons présenté une méthodologie de vérification de description de système d'un haut niveau algorithmique et mélangeant des aspects synchrones et asynchrones. Les propriétés des systèmes sont aussi des systèmes, ce qui facilite grandement les liens entre le travail de description et le travail de spécification lors du développement d'un système. De plus, la vérification d'une propriété par un système est automatique, bien que l'utilisateur puisse intervenir, toujours dans le même langage, pour guider l'algorithme de vérification lorsque celui-ci ne parvient pas à établir la validité. Afin de contourner l'explosion combinatoire en nombre d'états d'un système, problème qui apparaît lorsque plusieurs composants sont assemblés, nous avons mis en place une technique de remplacement de systèmes. Cette méthodologie a donné lieu à un langage de description et de spécification appelé KernelD qui est le noyau sur lequel nous avons développé nos algorithmes d'analyses. Dans un souci de génie logiciel, nous avons augmenté les constructions de KernelD pour créer le langage SystemD, adapté au développement et à la vérification de descriptions de système de composants et qui ne cache pas ses ressemblances avec le standard SystemC.

Les travaux que nous avons effectués sont une base et nous travaillons encore sur de nombreux aspects afin de pouvoir développer un outil qui pourrait être utilisé en industrie pour décrire, spécifier et vérifier des systèmes. La finalité serait de pouvoir vérifier un système de très grande taille. C'est cette finalité qui nous guide vers les aspects que nous devons améliorer ou éclaircir.

Réduction par Remplacement. Parmi ces aspects, nous souhaitons étudier sur plusieurs exemples réels le gain apporté par le remplacement, du point de vue de la réduction du nombre de configurations accessibles (en théorie nous avons considéré le nombre de configurations totales), mais aussi du point de vue du temps de calcul des algorithmes. Le temps de calcul des algorithmes est évidemment lié au nombre de configurations accessibles mais pas seulement. En effet, pour vérifier qu'un système peut effectivement en remplacer un autre, certaines topologies du graphe d'états abstraits du remplaçant permettent des temps de calcul plus rapide que d'autres. Avec cette étude, nous espérons cibler plus précisément les cas où un remplacement s'avère bénéfique et réduit effectivement les temps de calcul.

Remplacement Automatique. Une voie que nous voulons explorer est celle d'un calcul automatique d'un système remplaçant. Le remplacement permet d'abstraire un système en oubliant des informations, comme la valeur de certaines variables et de certains évènements qui n'interviennent pas sur les systèmes extérieurs. Plutôt que de construire entièrement un système remplaçant, on pourrait imaginer simplifier un système, par exemple en choisissant certaines variables et certains évènements pour lesquels on souhaite garder de la précision. Une autre façon de faire du remplacement automatique, que l'on pourrait tout à fait combiner avec la première, vient de l'observation suivante : lorsqu'un système modélise une propriété, c'est que les traces de ce système sont aussi des traces sûres de la propriété. Une propriété peut donc jouer le rôle de remplaçant des systèmes qu'elle valide. Cependant, la propriété n'est en général pas suffisante en elle-même puisqu'elle fait référence aux variables internes du système observé. On peut alors chercher la plus petite partie (ou simplification) qu'il faut ajouter à la propriété pour avoir un remplacement cohérent. Avec ce remplacement particulier d'un système par une de ses propriétés, on retrouve un contexte à la logique de Hoare où la spécification d'un programme est utilisée à la place du programme quand il s'agit de montrer des propriétés d'un développement qui le contient. Dans notre cas, la spécification est elle-même un programme.

Vivacité, Contre-Exemples. L'interprétation abstraite permet d'obtenir un sur-ensemble des états atteignables par une exécution d'un système. Nous utilisons cette technique pour montrer qu'un état erreur n'est pas atteignable. Ceci caractérise des propriétés de sûreté. Un autre type de propriétés importantes souvent rencontrées lors de la vérification de système est les propriétés de vivacité, qui permettent d'énoncer que certains états sont toujours atteignables. Nous seulement il n'y a aucune construction KernelD qui permet d'exprimer qu'un état doit être atteignable, mais en plus l'interprétation abstraite comme nous l'utilisons ne permet pas de montrer que de tels états seront bien atteignables. Pour autant, nous pouvons tout de même effectué un travail qui aidera l'ingénieur. Comme l'interprétation abstraite permet de calculer un sur-ensemble des états atteignables, si l'analyse d'un système montre qu'un état que l'on souhaite atteindre n'est pas atteignable dans l'abstraction, c'est qu'il ne l'est pas non plus dans le cas concret. Cela signifie que l'on peut montrer la non validité d'une propriété de vivacité, et que l'on peut extirper une trace du modèle abstrait qui ne satisfait pas la propriété. Cette trace ne représente peut-être pas une trace que le système peut produire en réalité, mais cela fournit tout de même à l'ingénieur des informations supplémentaires pour comprendre les exécutions du système.

Langage et Génie Logiciel. Les remarques précédentes nous conduiront certainement à modifier le langage SystemD et peut-être KernelD également. Mais même sans ces considérations, nous aimerions travailler avec des ingénieurs en développement de système pour adapter au mieux le langage. Cela pourrait signifier revoir complètement sa structure, pour coller mieux à l'approche *composants et assemblages* de la théorie sous-jacente plutôt qu'à l'approche *orienté objets* de SystemC dont nous ne nous servons pas.

C'est sur ces perspectives que se clôt ce document qui, nous l'espérons, apporte une certaine originalité pertinente sur la vérification formelle de description de système de composants.

Bibliographie

[16605] IEEE Standard 1666-2005. IEEE Standard SystemC Language Reference Manual, 2005.

[ABG+00] Yael Abarbanel, Ilan Beer, Leonid Glushovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs : Automatic Generation of Simulation Checkers from Formal Specifications. In *CAV '00 : Proceedings of the 12th International Conference on Computer Aided Verification*, pages 538–542, London, UK, 2000. Springer-Verlag.

[Acc04] Accellera. Property Specification Language Reference Manual Rev 1.1, 2004.

[ACV08a] Nicolas Ayache, Loïc Correnson, and Franck Védrine. Scenarios for validating SystemC descriptions. In *ICC'08 : Proceedings of the 12th WSEAS international conference on Circuits*, pages 468–473, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).

[ACV08b] Nicolas Ayache, Loïc Correnson, and Franck Védrine. Verifying SystemC with Scenario. In *2nd International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2008)*, eWiC, Leeds, UK, 2008. British Computer Society.

[ADS85] Bowen Alpern, Alan J. Demers, and Fred B. Schneider. Safety Without Stuttering. Technical report, Ithaca, NY, USA, 1985.

[AJLW92] Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors. *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, August 1991, Davis, California, USA*. IEEE Computer Society, 1992.

[AS86] Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. Technical report, Ithaca, NY, USA, 1986.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 2003. ACM Press.

[BCG87] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Characterizing Kripke Structures in Temporal Logic. In *TAPSOFT '87/CAAP '87 : Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 1 : Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming*, pages 256–270, London, UK, 1987. Springer-Verlag.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier : Technology transfer of formal methods inside Microsoft. In *IFM*, pages 1–20. Springer, 2004.

[BDK+04] Matthias Brill, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut Wittke. Live Sequence Charts : An Introduction to Lines, Arrows, and Strange Boxes in the Context of Formal Verification. In *SoftSpez Final Report*, volume 3147 of *Lecture Notes in Computer Science*, pages 374–399. Springer, 2004.

[BE07] Cecile Braunstein and Emmanuelle Encrenaz. Using CTL formulae as component abstraction in a design and verification flow. In *ACSD '07 : Proceedings of the Seventh International Conference on Application of Concurrency to System Design*, pages 80–89, Washington, DC, USA, 2007. IEEE Computer Society.

[Bei90] Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., 1990.

[BENS07] Martin Braun, Hans Eveking, Volker Nimbler, and Martin Schikel. CandoGen - A Property-Based Model Generator. In *Proceedings of the University Booth at DATE 2007*, 2007.

[Bir67] Garrett Birkhoff. Lattice theory (3rd edition). American Mathematical Society, 1967.

[BJR01] Grady Booch, Ivar Jacobson, and Jim Rumbaugh. OMG Unified Modeling Language Specification, September 2001.

[Boa96] Ariane 5 Inquiry Board. ARIANE 5 – Flight 501 Failure. Technical report, European Space Agency, 1996.

[Bry92] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3) :293–318, 1992.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC79] P. Cousot and R. Cousot. Constructive Versions of Tarski's Fixed Point Theorems. *Pacific Journal of Mathematics*, 81(1) :43–57, 1979.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4) :511–547, August 1992.

[CC97] P. Cousot and R. Cousot. Parallel Combination of Abstract Interpretation and Model-Based Automatic Analysis of Software. In R. Cleaveland and D. Jackson, editors, *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software, AAS'97*, pages 91–98, Paris, France, January 1997. ACM Press, New York, New York, United States.

[CC99] P. Cousot and R. Cousot. Refining Model Checking by Abstract Interpretation. *Automated Software Engineering*, 6(1) :69–95, 1999.

[CG03] Lukai Cai and Daniel Gajski. Transaction Level Modeling In System Level Design. Technical report, University of California, March 2003.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5) :752–794, September 2003.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5) :1512–1542, 1994.

[CI03] S. Swan C.N. Ip. A tutorial introduction on the new SystemC verification standard. Technical report, 2003. White Paper.

[CMC08] L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *Proc. of the Sixth Asian Symposium on Programming Languages and Systems (APLAS'08)*, volume 5356 of *LNCS*, pages 3–18, Bangalore, India, December 2008. Springer.

[Coq] L'assistant à la démonstration Coq. `http://coq.inria.fr/`.

[CP04] Magali Contensin and Laurence Pierre. Model-Checking Systems with Unbounded Variables without Abstraction. In *Algebraic Methodology and Software Technology, 10th International Conference, AMAST*, volume 3116 of *Lecture Notes in Computer Science*, pages 87–111. Springer, July 2004.

[DG02] Rolf Drechsler and Daniel Große. Reachability Analysis for Formal Verification of SystemC. In *DSD '02 : Proceedings of the Euromicro Symposium on Digital Systems Design*, page 337, Washington, DC, USA, 2002. IEEE Computer Society.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8) :453–457, 1975.

[Dow] Gilles Dowek. *Théorie des Types.* notes de cours de master 2, `http://www.lix.polytechnique.fr/~dowek/Cours/theories_des_types.ps.gz`.

[EH83] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited : on branching versus linear time (preliminary report). In *POPL '83 : Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 127–140, New York, NY, USA, 1983. ACM.

[EI90] Institute O. Electrical and Electronics E. (IEEE). IEEE Standard Glossary of Software Engineering Terminology, 1990. `http://standards.ieee.org/reading/ieee/std_public/description/se/610.12-1990_desc.html`.

[EMC81] Emerson Edmund M. Clarke. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Logics of Programs*, volume 131. LNCS, 1981.

[ES96] Andreas Ermedahl and Mikael Sjodin. Interval Analysis of C-variables using Abstract Interpretation, 1996.

[FLK03] Harry Foster, David Lacey, and Adam Krolnik. *Assertion-Based Design.* Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[FM92] K. Forsberg and H. Mooz. The relationship of systems engineering to the project cycle. volume 4, pages 36–43, September 1992.

[FM04] Jean-Christophe Filliâtre and Claude Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, November 2004. Springer-Verlag.

[FMS98] Bernd Finkbeiner, Zohar Manna, and Henny Sipma. Deductive Verification of Modular Systems. In *COMPOS'97 : Revised Lectures from the International Symposium on Compositionality : The Significant Difference*, pages 239–275, London, UK, 1998. Springer-Verlag.

[Fra] L'analyseur Frama-C. `http://frama-c.cea.fr/`.

[GAO92] GAO. Patriot Missile Defense : Software Problem Led to System Failure at Dhahran, Saudi Arabia. Technical report, U.S. General Accouting Office, 1992.

[Ghe06] Frank Ghenassia. *Transaction-Level Modeling with SystemC : TLM Concepts and Applications for Embedded Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[GHT04] Amjad Gawanmeh, Ali Habibi, and Sofiène Tahar. Enabling SystemC Verification using Abstract State Machines. In *Languages for Formal Specification and Verification*, pages 649–661, 2004.

[GKOT00] Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors. *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings*, volume 1912 of *Lecture Notes in Computer Science.* Springer, 2000.

[GTMH07] Amjad Gawanmeh, Sofiène Tahar, Haja Moinudeen, and Ali Habibi. A Design for Verification Approach Using an Embedding of PSL in AsmL. *Journal of Circuits, Systems, and Computers*, 16(6) :859–881, 2007.

[HJM+02] Tom Henzinger, Ranjit Jhala, Rupak Majumdar, George Necula, Gregoire Sutre, and Westley Weimer. Temporal-Safety Proofs for Systems Code. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, pages 526–538. Lecture Notes in Computer Science 2404, Springer-Verlag, January 2002.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HM03] David Harel and Rami Marelly. *Come, Let's Play : Scenario-Based Programming Using LSC's and the Play-Engine.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[Hoa83] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1) :53–56, 1983.

[HOL] L'assistant à la démonstration HOL. `http://hol.sourceforge.net/`.

[HP08] Claude Helmstetter and Olivier Ponsini. A Comparison of Two SystemC/TLM Semantics for Formal Verification. In *Proceedings of the 6th ACM-IEEE International Conference on Formal Methods and Models for Codesign MEMOCODE'2008*, June 2008.

[HQR98] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You Assume, We Guarantee : Methodology and Case Studies. In *CAV '98 : Proceedings of the 10th International Conference on Computer Aided Verification*, pages 440–451, London, UK, 1998. Springer-Verlag.

[HT04a] A. Habibi and S. Tahar. Towards an efficient assertion based verification of SystemC designs. In *HLDVT '04 : Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, pages 19–22, Washington, DC, USA, 2004. IEEE Computer Society.

[HT04b] Ali Habibi and Sofiene Tahar. SystemC Semantics in Fixpoint. Technical report, Concordia University, Department of Electrical and Computer Engineering, December 2004.

[Hym03] Charles Hymans. Design and Implementation of an Abstract Interpreter for VHDL. In *CHARME*, volume 2860 of *Lecture Notes in Computer Science*, pages 263–269. Springer, 2003.

[Isa] L'assistant à la démonstration Isabelle. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`.

[Jea] Bertrand Jeannet. L'interpréteur abstrait NBac. `http://pop-art.inrialpes.fr/people/bjeannet/nbac/`.

[Kah74] Gilles Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.

[KEP06] Daniel Karlsson, Petru Eles, and Zebo Peng. Formal verification of SystemC designs using a Petri-net based representation. In *DATE '06 : Proceedings of the conference on Design, automation and test in Europe*, pages 1228–1233, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[Kle03] Stephen Cole Kleene. *Mathematical logic.* 1967, réédité en 2003.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1) :11–44, 1995.

[Luc] Le langage Lucid Synchrone. `http://www.lri.fr/~pouzet/lucid-synchrone/`.

[MA00] B. Marre and A. Arnould. Test Sequences Generation from LUSTRE Descriptions : GATEL. September 2000. Proceedings of ASE-00 : The 15th IEEE Conference on Automated Software Engineering, Grenoble, France, IEEE CS Press.

[MCF+98] Zohar Manna, Michael A. Colon, Bernd Finkbeiner, Henny B. Sipma, and Tomas E. Uribe. Abstraction and Modular Verification of Infinite-State Reactive Systems. In *Requirements Targeting Software and Systems Engineering (RTSE), LNCS*. Springer-Verlag, 1998.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[MFMS06] K.L. Man, A. Fedeli, M. Mercaldi, and M.P. Schellekens. SystemCFL : An Infrastructure for a TLM Formal Verification Proposal (with an overview on a tool set for practical formal verification of SystemC descriptions). In *Proceedings of IEEE East-West Design & Test Workshop EWDTW, Sochi, Russia*, September 2006.

[Mil89] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Mil99] Robin Milner. *Communicating and Mobile Systems : the Pi-Calculus.* Cambridge University Press, June 1999.

[Min06] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1) :31–100, 2006.

[MMC+08] F. Maraninchi, M. Moy, J. Cornet, L. Maillet-Contoz, C. Helmstetter, and C. Traulsen. SystemC/TLM Semantics for Heterogeneous System-on-Chip Validation. In *2008 Joint IEEE-NEWCAS and TAISA Conference*, Montreal, June 2008.

[MMMC05] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy : A Toolbox for the Analysis of Systems-on-a-Chip at the Transactional Level. In *International Conference on Application of Concurrency to System Design*, June 2005.

[Moo79] Ramon E. Moore. Methods and Applications of Interval Analysis. In *Studies in Applied Mathematics, SIAM*, 1979.

[MP81] Z Manna and Amir Pnueli. Verification of concurrent programs, Part I : The temporal framework. Technical report, Stanford, CA, USA, 1981.

[MPMU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA Tool for Certification of JAVA/JAVACARD Programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2) :89–106, 2004.

[Mye79] Glenford J. Myers. *The Art of Software Testing.* Wiley, 1979.

[NHR92] F. Lagnier N. Halbwachs and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language Lustre. Special Issue on the Specification and Analysis of Real-Time Systems. IEEE Transactions on Software Engineering, 1992.

[Nil89] Ulf Nilsson. Towards a Framework for the Abstract Interpretation of Logic Programs. In *PLILP '88 : Proceedings of the 1st International Workshop on Programming Language Implementation and Logic Programming*, pages 68–82, London, UK, 1989. Springer-Verlag.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS.* Springer, 2002.

[NWR05] P. Mouy N. Williams, B. Marre and M. Roger. PathCrawler : Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In *5th European Dependable Computing Conference*, pages 281–292, Budapest, Hungary, April 2005. Springer-Verlag.

[OMAB08] Yann Oddos, Katell Morin-Allory, and Dominique Borrione. Assertion-Based Design with Horus. In *MEMOCODE*, pages 75–76. IEEE Computer Society, 2008.

[PF08] Laurence Pierre and Luca Ferro. A Tractable and Fast Method for Monitoring SystemC TLM Specifications. *IEEE Trans. Comput.*, 57(10) :1346–1356, 2008.

[Pnu77a] Amir Pnueli. The temporal logic of programs. In *SFCS '77 : Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[Pnu77b] Amir Pnueli. The temporal logic of programs. In *SFCS '77 : Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[Poo95] Joseph Poole. *A method to determine a basis set of paths to perform program testing.* U.S. Department of Commerce/National Institute of Standards and Technology, NISTIR 5737, November 1995.

[Pri67] A. Prior. *Past, Present and Future.* Oxford University Press, 1967.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[Rev01] Nathalie Revol. Introduction to interval arithmetic. Technical report, INRIA, 2001.

[RF] William M. Goble Rainer Faller. Open IEC 61508 Certification of Products. `http://www.exida.com/articles/IEc%2061508%20Cerftification.pdf`.

[Ric53] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2) :358–366, 1953.

[Sal03] Ashraf Salem. Formal Semantics of Synchronous SystemC. In *DATE '03 : Proceedings of the conference on Design, Automation and Test in Europe*, page 10376, Washington, DC, USA, 2003. IEEE Computer Society.

[SBB+99] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels : techniques et outils du model-checking.* Vuibert, April 1999.

[Sch97] Fred B. Schneider. *On concurrent programming.* Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[Sch02] Bernd S. W. Schröder. *Ordered Sets : an introduction.* Springer, 2002.

[SMV] Le model checker SMV. `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[SPI] Le model checker SPIN. `http://spinroot.com/spin/whatispin.html`.

[Sysa] Le langage SystemVerilog. `http://www.systemverilog.org/`.

[Sysb] Open SystemC Initiative. `http://www.systemc.org/home`.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5 :285–309, 1955.

[Tec] Esterel Technologies. Le langage Esterel. `http://www.esterel-technologies.com/`.

[TLN+89] Donald E. Thomas, Elizabeth D. Lagnese, John A. Nestor, Jayanth V. Rajan, Robert L. Blackburn, and Robert A. Walker. *Algorithmic and Register-Transfer Level Synthesis : The System Architect's Workbench.* Kluwer Academic Publishers, Norwell, MA, USA, 1989.

[UPP] Le model checker UPPAAL. `http://www.uppaal.com`.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, pages 238–266, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[Var07] Moshe Y. Vardi. Formal techniques for SystemC verification. In *DAC '07 : Proceedings of the 44th annual Design Automation Conference*, pages 188–192, New York, NY, USA, 2007. ACM.

[Ver] Le langage Verilog. `http://www.verilog.com/`.

[VHD] VHDL Analysis and Standardization Group. `http://www.eda.org/vhdl-200x/`.

[WL05] Dong Wang and Jeremy Levitt. Automatic assume guarantee analysis for assertion-based formal verification. In *ASP-DAC '05 : Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 561–566, New York, NY, USA, 2005. ACM.

ANNEXE A

Annexe

A.1 Preuves

Lemme 3.2

$$\forall S_1\ S_1'\ S_2\ I,\quad R(S_2)\cap I=\emptyset \text{ et } S_1 \rhd_{I/\Rightarrow} S_1' \quad\Longrightarrow$$
$$\forall t\in[\![S_1\otimes S_2]\!],\ \exists t''\in[\![S_1'\otimes S_2]\!],\ t\prec_{I/P_{S_1'}} t'' \text{ et } \mathrm{Safe}(S_1,t)\Rightarrow \mathrm{Safe}(S_1',t'')$$

Preuve : on suppose que l'on dispose d'une certaine trace $t\in[\![S_1\otimes S_2]\!]$. Nous allons construire une trace $t''\in[\![S_1'\otimes S_2]\!]$ qui satisfait les conditions de remplacement. Pour cela, on commence par utiliser le théorème 2.3 d'assemblage et sémantique opérationnelle pour déduire que $t\in[\![S_1]\!]$ à partir du fait que $t\in[\![S_1\otimes S_2]\!]$. On peut alors appliquer le remplacement de S_1 par S_1' pour obtenir une trace $t'\in[\![S_1']\!]$ telle que $t\prec_{I/P_{S_1'}} t'$ et $\mathit{Safe}(S_1,t)\Rightarrow\mathit{Safe}(S_1',t')$; on appelle u' la suite qui fait la relation entre les états de t et t'. On construit une suite u'' qui fait la relation entre les états de t et de t'' en même temps que l'on construit t'', par récurrence, et on montre que par cette construction, on a bien $t''\in[\![S_1'\otimes S_2]\!]$, $t\prec_{I/S_1'} t''$ et $\mathit{Safe}(S_1)\Rightarrow\mathit{Safe}(S_1',t'')$. En réalité, entre t' et t'', seuls les états intermédiaires par rapport à t seront légèrement modifiés pour intégrer la sémantique de S_2. On pose $\forall i,\ u_i''=u_i'$ (en particulier $u_0''=u_0'=0$) et $t''_{u_i''}=t'_{u_i'}$. Étant donné un certain i, on procède par cas sur $t_i\xrightarrow{S_1\otimes S_2}t_{i+1}$ pour définir $t''_{u''_{i+1}}$ à partir de $t''_{u_i''}$.

– Si $t_i\xrightarrow{\mathrm{Univers}(S_1\otimes S_2)}t_{i+1}$, c'est donc en particulier que $t_i\xrightarrow{\mathrm{Univers}(S_1)}t_{i+1}$ et $t_i\xrightarrow{\mathrm{Univers}(S_2)}t_{i+1}$ par définition de la transition Univers d'un système (la valeur des variables et évènements locaux et la configuration des processus de S_1 et S_2 n'ont pas changé de t_i à t_{i+1}). Puisque $t_i\xrightarrow{\mathrm{Univers}(S_1)}t_{i+1}$, alors par définition de t' pour laquelle on a $t\prec_{I/S_1'} t'$, on sait que

$t'_{u'_i} \xrightarrow{\text{Univers}(S'_1)} t'_{u'_i+1} \xrightarrow{\text{Univers}(S'_1)} t'_{u'_{i+1}}$. On définit $t''_{u''_i+1}$ comme suit. Pour tout ce qui est valeur des variables et des évènements, on garde celles en $t''_{u''_i}$: $\forall x,\ t''_{u''_i+1}.\rho(x) = t''_{u''_i}.\rho(x)$ et $\forall e,\ e \in t''_{u''_i+1}.E \Leftrightarrow e \in t''_{u''_i}.E$. Pour ce qui est configuration des processus, on prend celles en $t'_{u'_i+1}$: $\forall p,\ t''_{u''_i+1}.C(p) = t'_{u'_i+1}.C(p)$. On a effectivement que $\forall p,\ R_p \cap I = \emptyset \Rightarrow t''_{u''_i+1}.C(p) = t'_{u'_i+1}.C(p) = t_{i+1}.C(p)$ qui est une condition nécessaire pour le bon prolongement de t'' pour que $t \prec_{I/P_{S'_1}} t''$ dans le cas d'une transition Univers de S'_1. La configuration des processus de S'_1 en $t''_{u''_i+1}$ est la même qu'en $t'_{u'_i+1}$, qui est la même en $t'_{u'_i}$ (transition Univers de S'_1) qui est la même en $t''_{u''_i}$ (par définition de t''). La configuration des processus de S_2 est la même en $t''_{u''_i+1}$ qu'en $t'_{u'_i+1}$ qui est la même en t_{i+1} (car $R(S_2) \cap I = \emptyset$, donc $\forall p \in P_{S_2},\ R_p \cap I = \emptyset$) qui est la même en t_i (transition Univers de S_2) qui est la même en $t'_{u'_i}$ (par définition de t') qui est la même en $t''_{u''_i}$ (par définition de t''). De plus, la valeur de toutes les variables et de tous les évènements, donc en particulier ceux qui sont locaux à S'_1 ou à S_2, est la même en $t''_{u''_i+1}$ qu'en $t''_{u''_i}$ (par définition de $t''_{u''_i+1}$). Donc on a bien $t''_{u''_i} \xrightarrow{\text{Univers}(S'_1 \otimes S_2)} t''_{u''_i+1}$. La configuration des processus de S_2 en $t''_{u''_{i+1}}$ est la même qu'en $t'_{u'_{i+1}}$ (par définition de $t''_{u''_{i+1}}$), qui est la même en t_{i+1} (par définition de t' et parce que S'_1 et S_2 n'ont pas de processus communs car ils sont assemblables), qui est la même en $t'_{u'_i+1}$ (parce que $R(S_2) \cap I = \emptyset$), qui est la même en $t''_{u''_i+1}$. La configuration des processus de S'_1 est la même en $t''_{u''_{i+1}}$ qu'en $t'_{u'_{i+1}}$ (définition de t''), qui est la même en $t'_{u'_i+1}$ (transition Univers de S'_1), qui est la même en $t''_{u''_i+1}$ (définition de $t''_{u''_i+1}$). La valeur des variables et évènements locaux de S_2 est la même en $t''_{u''_{i+1}}$ qu'en $t'_{u'_{i+1}}$ (définition de t''), qui est la même en t_{i+1} (définition de t'), qui est la même en t_i (transition Univers de S_2), qui est la même en $t'_{u'_i}$ (définition de t'), qui est la même en $t''_{u''_i}$ (définition de t'') qui est la même en $t''_{u''_i+1}$ (définition de $t''_{u''_i+1}$). La valeur des variables et évènements locaux de S'_1 est la même en $t''_{u''_{i+1}}$ qu'en $t'_{u'_{i+1}}$ (définition de t''), qui est la même en $t'_{u'_i+1}$ (transition Univers de S'_1), qui est la même en $t'_{u'_i}$ (transition Univers de S'_1), qui est la même en $t''_{u''_i}$ (définition de t'') qui est la même en $t''_{u''_i+1}$ (définition de $t''_{u''_i+1}$). Donc $t''_{u''_i+1} \xrightarrow{\text{Univers}(S'_1)} t''_{u''_{i+1}}$ et $t''_{u''_i+1} \xrightarrow{\text{Univers}(S_2)} t''_{u''_{i+1}}$ donc $t''_{u''_i+1} \xrightarrow{\text{Univers}(S'_1 \otimes S_2)} t''_{u''_{i+1}}$. Au final on obtient l'enchaînement d'états suivant : $t''_{u''_i} \xrightarrow{\text{Univers}(S'_1 \otimes S_2)} t''_{u''_i+1} \xrightarrow{\text{Univers}(S'_1 \otimes S_2)} t''_{u''_{i+1}}$ qui prolonge correctement t'' pour que $t'' \in [\![S'_1 \otimes S_2]\!]$ et $t \prec_{I/P_{S'_1}} t''$.

– Si $\exists p \in P_{S_1}$, $t_i \xrightarrow{p} t_{i+1}$, c'est que $t_i \xrightarrow{\text{Univers}(S_2)} t_{i+1}$ et qu'il existe p_1, ..., p_k tels que $t'_{u'_i} \xrightarrow{p_1} t'_{u'_i+1} \xrightarrow{p_2} ... \xrightarrow{p_k} t'_{u'_{i+1}-1} \xrightarrow{\text{Univers}(S'_1)} t'_{u'_{i+1}}$. Dans ce cas, on prend exactement $t''_{u''_i+1} = t'_{u'_i+1}$, ..., $t''_{u''_{i+1}-1} = t'_{u'_{i+1}-1}$. Ainsi, on sait d'ores et déjà que les transitions entre chacun de ces états sont des transitions de S'_1. De plus, S'_1 n'écrit pas les variables et évènements locaux de S_2 car ils sont assemblables, donc chaque transition d'un processus de S'_1 est une transition Univers de S_2. Enfin, on sait donc que la valeur des variables et évènements locaux de S_2 et la configuration de ses processus sont les mêmes en $t''_{u''_{i+1}-1}$ qu'en $t''_{u''_i}$, qui sont les mêmes en $t'_{u'_i}$, qui sont les mêmes en t_i (définition de $t'_{u'_i}$) qui sont les mêmes en t_{i+1} (puisque $t_i \xrightarrow{\text{Univers}(S_2)} t_{i+1}$) qui sont les mêmes en $t'_{u'_{i+1}}$ (définition de $t'_{u'_{i+1}}$) qui sont les mêmes

en $t''_{u''_{i+1}}$ (définition de $t''_{u''_{i+1}}$). Donc $t''_{u''_{i+1}-1} \xrightarrow{\text{Univers}(S_2)} t''_{u''_{i+1}}$. On peut donc déduire que $t''_{u''_i} \xrightarrow{S'_1 \otimes S_2} t'_{u'_i+1} \xrightarrow{S'_1 \otimes S_2} ... \xrightarrow{S'_1 \otimes S_2} t'_{u'_{i+1}-1} \xrightarrow{S'_1 \otimes S_2} t'_{u'_{i+1}}$, ce qui prolonge correctement t'' par une série de processus de S'_1 suivie d'une transition Univers (et donc c'est une prolongation correcte pour que $t \prec_{I/P_{S'_1}} t''$ dans le cas d'une exécution de S_1) et telle que $t'' \in [\![S'_1 \otimes S_2]\!]$

– Si $\exists p \in P_{S_2}$, $t_i \xrightarrow{p} t_{i+1}$, c'est que $t_i \xrightarrow{\text{Univers}(S_1)} t_{i+1}$ et donc $t'_{u'_i} \xrightarrow{\text{Univers}(S'_1)} t'_{u'_i+1} \xrightarrow{\text{Univers}(S'_1)} t'_{u'_{i+1}}$. On définit $t''_{u''_i+1}$ de la manière suivante. La valeur des variables et évènements non locaux à S'_1 est celle de t_{i+1}, ainsi que la configuration des processus en dehors de S'_1 : $\forall x \notin L_{S'_1}$, $t''_{u''_i+1}.\rho(x) = t_{i+1}.\rho(x)$, $\forall e \notin L_{S'_1}$, $e \in t''_{u''_i+1}.E \Leftrightarrow e \in t_{i+1}.E$ et $\forall p \notin P_{S'_1}$, $t''_{u''_i+1}.C(p) = t_{i+1}.C(p)$. D'ailleurs, cette dernière proposition est une condition pour le prolongement correct de t'' pour que $t \prec_{I/P_{S'_1}}$ dans le cas d'une transition Univers de S_1. Pour les informations locales à S'_1 (valeur des variables et évènements et configuration des processus), on prend celles de $t''_{u''_i}$: $\forall x \in L_{S'_1}$, $t''_{u''_i+1}.\rho(x) = t''_{u''_i}.\rho(x)$, $\forall e \in L_{S'_1}$, $e \in t''_{u''_i+1}.E \Leftrightarrow e \in t''_{u''i}.E$ et $\forall p \in P_{S'_1}$, $t''_{u''_i+1}.C(p) = t''_{u''i}.C(p)$. Directement, on a $t''_{u''_i} \xrightarrow{\text{Univers}(S'_1)} t''_{u''_i+1}$ (les locaux et les processus de S'_1 n'ont pas changé par définition). De plus, on sait que $R(S_2) \cap I = \emptyset$ d'après les hypothèses du lemme. Donc les variables et évènements qui n'ont pas les mêmes valeurs de t_i à $t''_{u''_i}$ ne changent pas le comportement des processus de S_2. Comme S_2 n'écrit pas sur les locaux de S'_1 car et que les variables et évènements autres que ceux de S'_1 ont la même valeur de $t''_{u''_i}$ à $t''_{u''_i+1}$, on a $t''_{u''_i} \xrightarrow{p} t''_{u''_i+1}$ et donc $t''_{u''_i} \xrightarrow{S'_1 \otimes S_2} t''_{u''_i+1}$. La valeur des variables et évènements locaux et la configuration des processus de S_2 est la même en $t''_{u''_i+1}$ qu'en t_{i+1} (par définition de $t''_{u''_i+1}$), qui est la même en $t'_{u'_{i+1}}$ (définition de $t'_{u'_{i+1}}$) qui est la même en $t''_{u''_{i+1}}$ (définition de $t''_{u''_{i+1}}$). Donc $t'_{u'_{i+1}} \xrightarrow{\text{Univers}(S_2)} t''_{u''_{i+1}}$. De plus, la valeur des variables et évènements locaux et la configuration des processus de S'_1 est la même en $t''_{u''_{i+1}}$ qu'en $t'_{u'_{i+1}}$ (définition de $t''_{u''_{i+1}}$), qui est la même en $t'_{u'_i}$ (accessibilité par deux transitions Univers de S'_1), qui est la même en $t''_{u''_i}$ (définition de $t''_{u''_i}$) qui est la même en $t''_{u''_i+1}$ (définition de $t''_{u''_i+1}$). Donc $t''_{u''_i+1} \xrightarrow{\text{Univers}(S'_1)} t''_{u''_{i+1}}$. Nous avons donc montré dans ce point que $t''_{u''_i} \xrightarrow{S'_1 \otimes S_2} t''_{u''_i+1} \xrightarrow{S'_1 \otimes S_2} t''_{u''_{i+1}}$, qui prolonge correctement t'' par deux transitions Univers de S'_1 (c'est donc une prolongation correcte pour que $t \prec_{I/P_{S'_1}} t''$ dans le cas d'une transition Univers de S_1) et telle que $t'' \in [\![S'_1 \otimes S_2]\!]$.

– Si $t_i \xrightarrow{\text{Sched}} t_{i+1}$, c'est que $t'_{u'_i} \xrightarrow{\text{Sched}} t'_{u'_i+1} \xrightarrow{\text{Univers}(S'_1)} t'_{u'_{i+1}}$. On pose $t''_{u''_i+1} = t'_{u'_i+1}$. Comme $t'_{u'_i} \xrightarrow{\text{Sched}} t'_{u'_i+1}$, on sait que $t''_{u''_i} \xrightarrow{\text{Sched}} t''_{u''_i+1}$ et donc que $t''_{u''_i} \xrightarrow{S'_1 \otimes S_2} t''_{u''_i+1}$. Puisque $R(S_2) \cap I = \emptyset$ par hypothèse, on sait que $\forall p \in P_{S_2}, R_p \cap I = \emptyset$. Donc on sait que $\forall p \in P_{S_2}, t''_{u''_i+1}.C(p) = t'_{u'_i+1}.C(p) = t_{i+1}.C(p) = t'_{u'_{i+1}}.C(p) = t''_{u''_{i+1}}.C(p)$ (par définition de t' et t''). Donc la configuration des processus de S_2 n'a pas changé de $t''_{u''_i+1}$ à $t''_{u''_{i+1}}$. L'ordonnanceur ne change pas la valeur des variables, donc la valeur des variables locales de S_2 est la même de $t''_{u''_i}$ à $t''_{u''_i+1}$. De plus, la valeur des variables locales de S_2 est la même de $t''_{u''_i}$ à $t'_{u'_i}$, qui est la même en t_i, qui est la même en t_{i+1} (transition ordonnanceur), qui est la

même en $t'_{u'_{i+1}}$ (définition de t') et qui est la même en $t''_{u''_{i+1}}$ (définition de t''). Donc la valeur des variables locales de S_2 est la même en $t''_{u''_i+1}$ et en $t''_{u''_{i+1}}$. Comme $R(S_2) \cap I = \emptyset$, on sait en particulier que les évènements locaux de S_2 ne sont pas dans I. Donc par définition de t', $\forall e \in L_{S_2}, e \in t'_{u'_i+1}.E \Leftrightarrow e \in t_{i+1}.E$. Donc les évènements locaux de S_2 n'ont pas changé et ont la même présence de t_{i+1} à $t''_{u''_i+1}$ (car $t''_{u''_i+1} = t'_{u'_i+1}$). Or, Les évènements en dehors de I ont la même présence de t_{i+1} à $t'_{u'_{i+1}}$ (définition de $t'_{u'_{i+1}}$) qui ont la même présence que dans $t''_{u''_{i+1}}$ (définition de $t''_{u''_{i+1}}$). Donc les évènements locaux de S_2 n'ont pas changé de présence de à $t''_{u''_i+1}$ à $t''_{u''_{i+1}}$. Nous avons donc réussi à montrer que la valeur des variables et évènements locaux de S_2 ainsi que la configuration de ses processus n'ont pas changé de $t''_{u''_i+1}$ à $t''_{u''_{i+1}}$, c'est donc que $t''_{u''_i+1} \xrightarrow{\text{Univers}(S_2)} t''_{u''_{i+1}}$. On a bien $t''_{u''_i} \xrightarrow{\text{Sched}} t''_{u''_i+1} \xrightarrow{S'_1 \otimes S_2} t''_{u''_{i+1}}$, qui prolonge correctement t'' par l'exécution de l'ordonnanceur suivie suivie d'une transition Univers de S'_1 (et donc c'est une prolongation correcte pour que $t \prec_{I/P_{S'_1}} t''$ dans le cas d'une exécution de l'ordonnanceur) et telle que $t'' \in [\![S'_1 \otimes S_2]\!]$.

Pour finir cette démonstration, on doit montrer que $Safe(S_1, t) \Rightarrow Safe(S'_1, t'')$. Si on a $Safe(S_1, t)$ on sait du fait que S'_1 remplace S_1 et par définition du remplacement que $Safe(S'_1, t')$. Or, la configuration des processus de S'_1 est la même point à point de t' à t'' d'après la construction ci-dessus de t''. Donc la sûreté des états en S'_1 est préservée et on a bien $Safe(S'_1, t'')$. □

Lemme 3.14 $\forall S\ S',\ S'$ observe $S \Rightarrow \forall t \in [\![S]\!],\ \exists t' \in [\![S \otimes S']\!],\ t \leq_S t'$

Preuve : on part du fait que $(L_{S_\mathcal{O}} \cup W(S)) \cap (R(S) \cup W(S)) = \emptyset$ et $S' \not\prec S$ (le fait que toutes les traces de $S_\mathcal{O}$ soient sûres n'intervient pas). On considère une trace $t \in [\![S]\!]$ et on définit une suite $(u_n)_{n\in\mathbb{N}}$ croissante qui fait la relation entre les indices de t et d'une trace t' dont on montre qu'elle est dans $[\![S \otimes S']\!]$ et qu'elle étire t sur S. Pour cela, on procède par récurrence et par cas sur la structure de t. On pose $u_0 = 0$ (obligatoire d'après la définition de l'étirement) et $t'_0 = t_0$; on a bien $t_0 \leq_S t_0 = t'_0$ (on n'a pas changé d'état, donc en particulier on n'a pas changé les informations relatives à S). Maintenant, on considère un certain i tel que $t_i \leq_S t'_{u_i}$. On décompose $t_i \xrightarrow{S} t_{i+1}$.

- Si $t_i \xrightarrow{\text{Univers}(S)} t_{i+1}$, on pose $u_{i+1} = u_i + 1$ et on définit $t'_{u_{i+1}}$ comme suit. On garde les mêmes valeurs pour les variables et évènements de S qu'en t_{i+1} : $\forall x \in R(S) \cup W(S),\ t'_{u_{i+1}}.\rho(x) = t_{i+1}.\rho(x)$ et $\forall e \in R(S) \cup W(S),\ e \in t'_{u_{i+1}}.E \Leftrightarrow e \in t_{i+1}.E$. On garde les mêmes valeurs pour les variables et évènements locaux de S' qu'en t'_{u_i} : $\forall x \in L_{S'},\ t'_{u_{i+1}}.\rho(x) = t'_{u_i}.\rho(x)$ et $\forall e \in L_{S'},\ e \in t'_{u_{i+1}}.E \Leftrightarrow e \in t'_{u_i}.E$. Enfin, les configurations des processus de S et S' restent les mêmes qu'en t'_{u_i} : $\forall p \in P_S \cup P_{S'},\ t'_{u_{i+1}}.C(p) = t'_{u_i}.C(p)$. On a bien $t'_{u_i} \xrightarrow{S \otimes S'} t'_{u_i+1} = t'_{u_{i+1}}$ car les valeurs des les variables et évènements de S (à plus forte raison, ses locaux) sont les mêmes qu'en t_{i+1}, donc les mêmes qu'en t_i (transition Univers de S) et donc les mêmes qu'en t'_{u_i} (car $t_i \leq_S t'_{u_i}$). Les variables et évènements locaux de S' (disjoints de ceux de S car S' observe S) restent les mêmes qu'en t'_{u_i} par définition. Enfin, les processus de $S \otimes S'$, qui sont ceux de S et de S' par définition de l'assemblage, restent dans la même configuration qu'en t'_{u_i}. Donc on a $t'_{u_i} \xrightarrow{\text{Univers}(S \otimes S')} t'_{u_{i+1}}$. De plus, on a bien $t_{i+1} \leq_S t'_{u_{i+1}}$ car par définition, les variables et évènements de S gardent les mêmes valeurs entre t_{i+1} et $t'_{u_{i+1}}$. Les processus

de S en $t'_{u_{i+1}}$ sont dans les mêmes configurations qu'en t'_{u_i}, elles-mêmes plus libres qu'en t_i, elles-mêmes égales à celles en t_{i+1}.

- Si $t_i \xrightarrow{p} t_{i+1}$ avec $p \in S$, on pose $u_{i+1} = u_i + 1$ et on définit $t'_{u_{i+1}}$ en considérant deux cas : celui où $t_i.C(p) = t'_{u_i}.C(p)$ et celui où $t_i.C(p) < t'_{u_i}.C(p)$.
  - Si $t_i.C(p) = t'_{u_i}.C(p)$, alors on se trouve dans un cas où les valeurs des variables et évènements de S et la configuration de p sont les mêmes. Donc on peut exécuter p de la même manière. On pose $\forall x \in R(S) \cup W(S),\ t'_{u_{i+1}}.\rho(x) = t_{i+1}.\rho(x), \forall e \in R(S) \cup W(S),\ e \in t'_{u_{i+1}}.E \Leftrightarrow e \in t_{i+1}.E$ et $t'_{u_{i+1}}.C(p) = t_{i+1}.C(p)$, c'est-à-dire qu'on reprend les effets de l'exécution de p pour ce qui le concerne. Pour le reste, on garde les mêmes valeurs : $\forall x \notin R(S) \cup W(S),\ t'_{u_{i+1}}.\rho(x) = t'_{u_i}.\rho(x), \forall e \notin R(S) \cup W(S),\ e \in t'_{u_{i+1}}.E \Leftrightarrow e \in t'_{u_i}.E$ et $\forall p' \neq p,\ t'_{u_{i+1}}.C(p) = t'_{u_i}.C(p)$. On sait donc que t'_{u_i} et $t'_{u_{i+1}}$ sont en relation par une transition de S (c'est une transition locale par p). De plus, comme $L_{S'} \cap R(S) \cup W(S) = \emptyset$ car S' est observateur de S et par la définition de $t'_{u_{i+1}}$, les valeurs des variables et évènements locaux de S' et les configurations de ses processus n'ont pas changé de t'_{u_i} et $t'_{u_{i+1}}$. Donc t'_{u_i} et $t'_{u_{i+1}}$ sont en relation par une transition Univers de S', et on a bien $t'_{u_i} \xrightarrow{S \otimes S'} t'_{u_{i+1}}$ par le théorème 2.3. Les valeurs des variables et évènements de S sont les mêmes de entre t_{i+1} et $t'_{u_{i+1}}$ par définition de $t'_{u_{i+1}}$. Toujours par définition de $t'_{u_{i+1}}$, la configuration de p est la même qu'en t_{i+1}. Il reste à montrer que la configuration des autres processus est plus libre en $t'_{u_{i+1}}$ qu'en t_{i+1} pour avoir $t_{i+1} \leq_S t'_{u_{i+1}}$. Or, pour ces processus, la configuration en $t'_{u_{i+1}}$ est la même qu'en t'_{u_i}, qui est plus libre qu'en t_i par hypothèse de récurrence, et celle en t_i est la même qu'en t_{i+1} puisque $t_i \xrightarrow{p} t_{i+1}$.
  - Si $t_i.C(p) < t'_{u_i}.C(p)$, alors soit $t_i.C[p \leftarrow t'_{u_i}.C(p)] \xrightarrow{p} t_{i+1}$, soit $t_{i+1} = t_i$ d'après l'hypothèse 3.12 sur $<$. Si $t_{i+1} = t_i$, alors on pose $t'_{u_{i+1}} = t'_{u_i}$ et on a directement $t'_{u_i} \xrightarrow{S \otimes S'} t'_{u_{i+1}}$ par une transition Univers qui ne fait rien, et $t_{i+1} \leq_S t'_{u_{i+1}}$ par simple réécriture et parce que $t_i \leq_S t'_{u_i}$. Si $t_i.C[p \leftarrow t'_{u_i}.C(p)] \xrightarrow{p} t_{i+1}$, on peut reprendre la même définition et les mêmes arguments que pour le point précédent. En effet, seule sa configuration a changé dans les informations relatives à p, or on sait que ce changement de configurations ne change pas les états atteignables par une exécution du processus.
- Si $t_i \xrightarrow{\text{Sched}} t_{i+1}$, on va chercher à appliquer la définition du non bloquage. Pour cela, on commence par faire une transition Univers de $S \otimes S'$ sur t'_{u_i} pour remettre tous les processus en dehors de S et de S' sur la même configuration qu'en t_i. Clairement, ce nouvel état est égal au précédent modulo S puisqu'on n'en a changé aucune information relative. On peut alors appliquer le fait que S' est non bloquant pour S pour en déduire qu'il existe un certain j et j exécutions de processus de S' ou de l'ordonnanceur qui mènent à un état moins libre que t_{i+1} modulo S. C'est à cet état là que l'on définit $t'_{u_{i+1}}$. On pose donc $u_{i+1} = u_i + 1 + j$ (le 1 concerne la première transition Univers). Cette séquence d'exécution est bien dans S' puisqu'elle consiste exécutions de en processus de S' et en transitions ordonnanceur. Elle est bien dans S puisque S' n'écrit aucune information relative à S, et que nous avons donc une séquence de transitions Univers et ordonnanceur pour ce dernier. Enfin, les transitions des processus de S' ne changent pas les informations relatives à S, donc deux états de cette séquence qui se suivent par une transition d'un processus de S' restent égaux modulo S, et dans le cas des transitions ordonnanceur, on sait par hypothèse 3.13 qu'un état est moins libre modulo n'importe quel système que son image par l'ordonnanceur.

□

A.2 SystemD

La définition de la syntaxe et de la grammaire de SystemD repose sur les conventions suivantes :
- les mots et symboles-clefs du langage sont écrits dans une police de style *machine à écrire*. Exemple : `module` ;
- les règles de la grammaire sont indiquées entre parenthèses angulaires et en italique. Exemple : ⟨*module*⟩.

À part ces deux règles, on utilise les conventions classiques pour les expressions rationnelles.

Un développement SystemD est une suite de définition de structures, de modules, de remplacements, d'interface et de fonctions statiques ainsi que l'invocation soit du moteur de simulation, soit du moteur de vérification. Les structures décrivent des types de données complexes, les modules représentent des systèmes, les remplacements consistent à simplifier le code de certains modules par des modules plus simples, les interfaces permettent d'abstraire du code dans une certaine mesure, et les fonctions statiques sont des macros qui n'appartiennent à aucune structure ni aucun module en particulier.

$$\langle \textit{SystemD} \rangle \quad ::= \quad (\langle \textit{structure} \rangle \mid \langle \textit{module} \rangle \mid \langle \textit{abstraction} \rangle \mid \langle \textit{interface} \rangle \mid \langle \textit{function} \rangle)^+ \; (\langle \textit{simulation} \rangle \mid \langle \textit{verification} \rangle)$$

La simulation se fait sur exactement un module. Un module donné en argument du moteur de simulation est *compilé* en un système qui sera alors passé au moteur de simulation. Si l'on souhaite observer une exécution d'un système, il faut donc le représenter par un module au préalable.

$$\langle \textit{simulation} \rangle \quad ::= \quad \texttt{main(}\ \langle \textit{instanciation} \rangle\ \texttt{)}$$

Contrairement à la simulation, le moteur de vérification peut prendre plusieurs modules en argument. Chacun d'entre-eux donnera lieu à un système ; ils seront ensuite vérifiés un à un.

$$\langle \textit{verification} \rangle \quad ::= \quad \texttt{analyse \{}\ \langle \textit{instanciation} \rangle^+\ \texttt{\}}$$

Le reste de la syntaxe/grammaire repose en partie sur une façon de nommer les objets du développement. Un nom valide est une succession de lettres, de chiffres et de quelques symboles spéciaux.

$$\langle \textit{letter} \rangle \quad ::= \quad \texttt{A-Z} \mid \texttt{a-z} \qquad \langle \textit{identifier} \rangle \quad ::= \quad \langle \textit{letter} \rangle\ (\langle \textit{letter} \rangle \mid \texttt{0-9} \mid \texttt{\_})^*\ \texttt{'}^*$$

Les arguments passés aux moteurs de simulation et de vérification consistent simplement en des déclarations d'identificateurs auxquels on donne un type représentant un module.

$$\langle \textit{instanciation} \rangle \quad ::= \quad \langle \textit{module-type} \rangle\ \langle \textit{identifier} \rangle\ \texttt{;}$$

Une structure est un type de données complexe qui est représenté par un ensemble de *champs* (les objets que la structure manipule) et de leur type de données, et d'un ensemble de *méthodes*

qui sont des fonctions de manipulation des champs. Les structures SystemD sont proches des structures à la C mais leur syntaxe ressemble plutôt à des classes en programmation orientée objet — sans les notions de hiérarchie et de protection, d'où l'absence de constructeur. Pour que les structures soient assez génériques, nous ajoutons des *templates* qui permettent à une structure de prendre en argument une constante ou un type de données, dans une syntaxe proche des langages de programmation classiques qui les implantent également (C++, Java, etc). Ceci offre des possibilités de polymorphisme pour le développement de systèmes. Enfin, une structure peut implanter une interface ; la structure doit donner une définition pour chaque méthode dont le prototype apparaît dans l'interface.

⟨*structure*⟩ ::= `struct[<`⟨*templates*⟩`>]` ⟨*identifier*⟩ [`:` ⟨*identifier*⟩] `{`
⟨*data-decl*⟩*
⟨*method*⟩*
`}`

Un module représente un système. Il doit donc définir les choses suivantes :
- la déclaration ainsi que le type des objets qu'il lit ou écrit. Ce peut être des données classiques, comme une variable porteuse d'une valeur entière, des structures, un évènement, mais aussi d'autres modules. On parle alors de sous-modules. Chaque variable et évènement d'un sous-module est une variable ou un évènement du module qui le contient ;
- l'enregistrement des variables et évènements qui seront locaux au module ;
- un *constructeur* du module sur lequel nous reviendrons ensuite. Si aucun constructeur n'est spécifié, le constructeur vide est utilisé par défaut ;
- un ensemble de *programmes* qui contient entre autres les processus du système que représente le module.

Les modules comme les structures peuvent se reposer sur la notion de templates.

⟨*module*⟩ ::= `module[<`⟨*templates*⟩`>]` ⟨*identifier*⟩ [`:` ⟨*identifier*⟩] `{`
⟨*decl*⟩*
[`local:` (⟨*left-value*⟩,)*⟨*left-value*⟩ `;`]
[⟨*constructor*⟩]
⟨*programs*⟩
`}`

Un remplacement, ou abstraction, prend en argument un module à remplacer. Le reste de l'abstraction consiste à définir un module remplaçant. On peut y définir quelles sont les variables et évènements que le remplacé et le remplaçant ont en commun (mot-clef `link`).

⟨*abstraction*⟩ ::= `abstraction[<`⟨*templates*⟩`>]` ⟨*identifier*⟩`(`⟨*module-type*⟩ ⟨*identifier*⟩`)` `{`
⟨*decl*⟩*
[`local:` (⟨*left-value*⟩,)*⟨*left-value*⟩ `;`]
[`link:` (⟨*link-clause*⟩ `and`)*⟨*link-clause*⟩ `;`]
[⟨*constructor*⟩]
⟨*programs*⟩
`}`

⟨*link-clause*⟩ ::= ⟨*left-value*⟩ `=` ⟨*left-value*⟩

Une interface permet de généraliser des structures ou des modules en imposant un certain nombre de fonctions ou méthodes à implanter. C'est lors de la compilation que l'on vérifie que chaque structure ou module respecte son interface.

$$\begin{array}{rcl} \langle \textit{interface} \rangle & ::= & \texttt{interface}\ \langle \textit{identifier} \rangle\ \texttt{\{} \\ & & \quad (\langle \textit{identifier} \rangle \texttt{(} \langle \textit{args} \rangle \texttt{)}\ [\texttt{:}\ \langle \textit{datatype} \rangle]\ \texttt{;})^{+} \\ & & \texttt{\}} \end{array}$$

À présent, nous rentrons dans le détail des règles qui composent la définition de structure, de module, de remplacement ou d'interface.

Les déclarations permettent de nommer un certains nombre d'objets qui ont des structures bien distinctes. On y trouve :
- les types de base : entiers (`int`) et booléens (`bool`) ;
- les tableaux notés classiquement avec des crochets `[ ]` pour leur dimension ;
- les types de données plus complexes, représentés par des structures (instanciées pour celles à templates) ;
- le type des évènements : `event` ;
- des modules qui représentent les sous-composants du module qui les déclare (instanciés pour ceux à templates).

Certains de ces types sont propres aux systèmes : évènements et modules. On distingue donc deux types de déclarations : un type de données est tout sauf un évènement ou un module. Un type de manière générale peut être n'importe laquelle des constructions ci-dessus. Notons que les identificateurs sont utilisés aussi bien pour nommer une structure qu'un module. La différence sera faite à la compilation.

Remarque : les ports ne sont pas un type primitif du langage ; nous les définissons plus loin.

$$\begin{array}{rcl} \langle \textit{base-type} \rangle & ::= & \texttt{int} \mid \texttt{bool} \end{array}$$

$$\begin{array}{rcl} \langle \textit{datatype} \rangle & ::= & \langle \textit{base-type} \rangle \\ & \mid & \langle \textit{identifier} \rangle [\texttt{<} \langle \textit{temps-app} \rangle \texttt{>}] \\ & \mid & \langle \textit{datatype} \rangle \texttt{[} \langle \textit{integer} \rangle \texttt{]} \end{array}$$

$$\begin{array}{rcl} \langle \textit{module-type} \rangle & ::= & \langle \textit{identifier} \rangle [\texttt{<} \langle \textit{temps-app} \rangle \texttt{>}] \end{array}$$

$$\begin{array}{rcl} \langle \textit{type} \rangle & ::= & \langle \textit{base-type} \rangle \\ & \mid & \langle \textit{type} \rangle \texttt{[} \langle \textit{integer} \rangle \texttt{]} \\ & \mid & \texttt{event} \\ & \mid & \langle \textit{module-type} \rangle \end{array}$$

Cette différence entre les types se retrouvent dans les déclarations : déclaration de type de données, et déclaration de type général. Dans le cas de la déclaration d'un sous-module, on peut remplacer ce dernier par un autre par l'utilisation d'une clause `with`.

$$\langle \textit{data-decl} \rangle ::= \langle \textit{datatype} \rangle\ (\langle \textit{identifier} \rangle \texttt{,})^{*}\ \langle \textit{identifier} \rangle\ \texttt{;}$$

$$\begin{array}{rcl} \langle \textit{decl} \rangle & ::= & \langle \textit{type} \rangle\ (\langle \textit{identifier} \rangle \texttt{,})^{*}\ \langle \textit{identifier} \rangle\ \texttt{;} \\ & & \langle \textit{module-type} \rangle\ (\langle \textit{identifier} \rangle\ [\langle \textit{with-clause} \rangle] \texttt{,})^{*}\ \langle \textit{identifier} \rangle\ [\langle \textit{with-clause} \rangle]\ \texttt{;} \end{array}$$

Les clauses `with` sont une succession d'applications de remplacement. L'application d'un remplacement est simplement donnée par le nom du remplacement (éventuellement instancié s'il possède des templates) appliqué au module à remplacer. La définition du remplacement explicite comment construire le module à utiliser à la place de l'argument.

⟨*with-clause*⟩ ::= `with` (⟨*abs-app*⟩ `and`)* ⟨*abs-app*⟩
⟨*abs-app*⟩ ::= ⟨*identifier*⟩[`<`⟨*temps-app*⟩`>`](⟨*left-value*⟩)

Les templates peuvent être une constante d'un type de données ou un type de données quelconque désigné par un identificateur. L'application de templates est donc une série d'expressions qui représentent chacune un type de données ou dont le type est un type de données.

⟨*templates*⟩ ::= (⟨*template*⟩`,`)* ⟨*template*⟩
⟨*template*⟩ ::= ⟨*datatype*⟩ ⟨*identifier*⟩ | ⟨*identifier*⟩
⟨*temps-app*⟩ ::= (⟨*temp-app*⟩`,`)* ⟨*temp-app*⟩
⟨*temp-app*⟩ ::= ⟨*exp*⟩ | ⟨*datatype*⟩

Un programme d'un module peut être de deux natures : soit c'est une méthode qui n'est en fait qu'une macro destinée à factoriser du code, soit c'est un *processus* qui représente un comportement. On trouve aussi les constructeurs de module qui sont des programmes particuliers et que l'on traite à part.

⟨*programs*⟩ ::= (⟨*method*⟩ | ⟨*thread*⟩)*

Qu'un programme soit une méthode ou un processus, il peut déclarer des variables qui lui sont locales ; cette fois le mot *local* est utilisé dans son sens classique : la portée d'une telle variable est restreinte au code du programme qui la déclare. Le type des variables locales aux programmes doit être un type de données. On exclut donc les évènements, les ports et les composants qui servent à représenter des notions de système.

Le constructeur d'un module est formé du nom du module et d'un *corps*. Ce corps permet de confondre des variables ou des évènements et d'initialiser l'environnement d'exécution du module. Le corps est une suite d'instructions de constructeur.

⟨*constructor*⟩ ::= ⟨*identifier*⟩ `{` ⟨*constructor-inst*⟩* `}`

Une instruction peut être utilisée par tous les programmes, les constructeurs et les fonctions : l'affectation.

⟨*aff*⟩ ::= ⟨*left-value*⟩ `=` ⟨*exp*⟩ `;`

Le corps d'un constructeur est formé d'instructions de déclaration, d'affectation et d'appels de fonctions. Autour de ces instructions de base, on ajoute des instructions de contrôle que sont la conditionnelle et la boucle. L'affectation dans un constructeur se fait par adresse ; c'est le seul endroit de SystemD où cela sera le cas.

⟨*constructor-inst*⟩ ::= ⟨*data-decl*⟩ | ⟨*aff*⟩
| ⟨*function-call*⟩
| `if (`⟨*exp*⟩`) {` ⟨*constructor-inst*⟩* `}` [`else {` ⟨*constructor-inst*⟩* `}`]
| `while (`⟨*exp*⟩`) {` ⟨*constructor-inst*⟩* `}`

Une méthode porte un nom, prend des arguments qui servent à factoriser encore plus le code et définit un corps. Certaines méthodes effectuent un calcul. Lorsque c'est le cas, on doit mentionner le type de retour de ce calcul. Ce doit être un type de données ; les méthodes ne retournent donc pas d'évènement ni de module. Mais elles peuvent retourner des structures. Il se peut également que des méthodes ne retournent pas de valeurs mais modifient directement la valeur de certaines variables par effets de bord. Dans ce cas, la méthode ne doit pas mentionner de type de retour. Le corps de la méthode est lui aussi une suite d'instructions, mais différentes de celles des constructeurs.

⟨*method*⟩ ::= ⟨*identifier*⟩`(`⟨*args*⟩`)` [`:` ⟨*datatype*⟩] `{` ⟨*method-inst*⟩* `}`

⟨*args*⟩ ::= [(⟨*datatype*⟩ ⟨*identifier*⟩`,`)* ⟨*datatype*⟩ ⟨*identifier*⟩]

Les instructions autorisées dans le corps d'une méthode sont les mêmes que pour un constructeur, avec en plus l'instruction d'arrêt de la méthode avec une valeur de retour. Notons que contrairement au cas *constructeur* où l'affectation se fait par adresse, l'affectation dans une méthode se fait par valeur.

⟨*method-inst*⟩ ::= ⟨*data-decl*⟩ | ⟨*aff*⟩
| ⟨*function-call*⟩
| `if (`⟨*exp*⟩`) {` ⟨*method-inst*⟩* `}` [`else {` ⟨*method-inst*⟩* `}`]
| `while (`⟨*exp*⟩`) {` ⟨*method-inst*⟩* `}`
| `return(`⟨*exp*⟩`) ;`

Les processus ne prennent pas d'argument et sont seulement définis par leur corps. Les processus peuvent être discriminants, c'est-à-dire qu'ils n'écrivent pas sur les variables et évènements du module dont ils dépendent et ne le bloque pas. Les processus peuvent également être des propriétés, qui ne doivent pas violer des assertions si le système lui-même n'en viole pas.

⟨*thread*⟩ ::= (`thread` | `discriminate` | `property`) `{` ⟨*thread-inst*⟩* `}`

Les instructions des processus sont encore différentes de celles des constructeurs et des méthodes. Aux déclarations, affectations (par valeur), conditionnelles, boucles et appels de fonctions, on ajoute des instructions propres aux systèmes : notifications d'évènements et mise en attente du processus (ou blocage définitif). De plus, ils peuvent faire appel à une instruction de vérification de propriété (`assert`).

⟨*thread-inst*⟩ ::= ⟨*data-decl*⟩ | ⟨*aff*⟩
| ⟨*function-call*⟩
| `notify(`⟨*identifier*⟩`) ;`
| `wait(`⟨*status*⟩`) ;` | `halt() ;`
| `assert(`⟨*exp*⟩`) ;`
| `if (`⟨*exp*⟩`) {` ⟨*thread-inst*⟩* `}` [`else {` ⟨*thread-inst*⟩* `}`]
| `while (`⟨*exp*⟩`) {` ⟨*thread-inst*⟩* `}`

Les fonctions sont des macros statiques qui ne concernent pas directement un module ou une structure, qui peuvent être templatées, qui peuvent prendre des arguments de n'importe quel

type et faire appel à n'importe quelle instruction parmi celles des constructeurs, des méthodes et des processus.

⟨*function*⟩ ::= `function` ⟨*identifier*⟩[`<`⟨*templates*⟩`>`](⟨*function-args*⟩) [`:` ⟨*datatype*⟩] `{`
⟨*function-inst*⟩*
`}`

⟨*function-args*⟩ ::= [(⟨*type*⟩ ⟨*identifier*⟩,)* ⟨*type*⟩ ⟨*identifier*⟩]

⟨*function-inst*⟩ ::= ⟨*data-decl*⟩ | ⟨*aff*⟩
| ⟨*function-call*⟩
| `notify(`⟨*identifier*⟩`) ;`
| `wait(`⟨*status*⟩`) ;` | `halt() ;`
| `assert(`⟨*exp*⟩`) ;`
| `return(`⟨*exp*⟩`) ;`
| `if (`⟨*exp*⟩`) {` ⟨*function-inst*⟩* `}` [`else {` ⟨*function-inst*⟩* `}`]
| `while (`⟨*exp*⟩`) {` ⟨*function-inst*⟩* `}`

⟨*function-call*⟩ ::= ⟨*identifier*⟩[`<`⟨*temps-app*⟩`>`](⟨*exp*⟩`,`)* ⟨*exp*⟩) `;`

Les statuts d'attente des processus sont des combinaisons d'évènements. Un processus peut se mettre en attente qu'un évènement soit notifié, que plusieurs combinaisons d'évènements soient notifiées (`&`), que des combinaisons d'évènements ou d'autres soient notifiées (`|`) ou que des combinaisons d'évènements ne soient pas notifiées (`not`). Des évènements particuliers sont ceux liés au temps, qui est représenté par un entier et une unité (nanoseconde, milliseconde ou seconde). Se mettre en attente de `4s` signifie d'attendre que 4 secondes se soient écoulées.

⟨*status*⟩ ::= ⟨*identifier*⟩ | ⟨*integer*⟩⟨*time-unit*⟩
| ⟨*status*⟩ `&` ⟨*status*⟩
| ⟨*status*⟩ `|` ⟨*status*⟩
| `not(`⟨*status*⟩`)`

⟨*time-unit*⟩ ::= `s` | `ms` | `ns`

Enfin, comme nous l'avons vu dans certaines règles précédentes, les instructions utilisent des *expressions*. Celles-ci sont des combinaisons de constantes entières et booléennes, d'opérations sur ces constantes, de méthodes définies par l'utilisateur, mais aussi la désignation des variables. On peut désigner une variable en faisant directement appel à son identificateur quand elle est dans la portée immédiate de l'instruction qui l'appelle, ou en utilisant le mot-clef `this` dans le cas d'une structure ou d'un module. Cela peut également être un champ d'une structure ou d'un module ou encore la case d'un tableau. On réunit ces désignations sous le nom de *valeur gauche* car ce sont les seules expressions auxquelles on peut affecter une valeur.

⟨*left-value*⟩ ::= ⟨*identifier*⟩ | `this`
| ⟨*left-value*⟩`.`⟨*identifier*⟩
| ⟨*left-value*⟩`[`⟨*exp*⟩`]`

Parmi les expressions, remarquons la fonction `rand` qui choisit de façon indéterministe soit un booléen si aucun argument ne lui est appliqué, soit une valeur entre deux bornes, pour lesquelles on peut utiliser la valeur spéciale `infty` qui représente l'infini. Si en pratique les

valeurs du langage sont bornées, en théorie elles ne le sont pas, ce qui rend la valeur `infty` intéressante pour la vérification.

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle integer \rangle \mid \langle bool \rangle \\
& \mid & \langle left\text{-}value \rangle \\
& \mid & \texttt{(}\langle exp \rangle\texttt{)} \\
& \mid & \langle exp \rangle\ \langle binop \rangle\ \langle exp \rangle \\
& \mid & \langle unop \rangle\ \langle exp \rangle \\
& \mid & \texttt{infty} \\
& \mid & \texttt{rand(}[\langle exp \rangle\texttt{,}\langle exp \rangle\texttt{]}\texttt{)} \\
& \mid & \langle function\text{-}call \rangle \mid [\langle left\text{-}value \rangle\texttt{.}]\langle method\text{-}call \rangle
\end{array}
$$

$$\langle method\text{-}call \rangle ::= \langle identifier \rangle\texttt{(}[(\langle exp \rangle\texttt{,})^*\ \langle exp \rangle\texttt{]}\texttt{)}$$

$$\langle binop \rangle ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{==} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \qquad \langle unop \rangle ::= \texttt{-} \mid \texttt{!}$$

$$\langle integer \rangle ::= \mathbb{Z} \qquad \langle bool \rangle ::= \texttt{true} \mid \texttt{false}$$

A.3 KernelD

A.3.1 Opérations

On considère les entiers et les booléens comme exemple de valeurs pour KernelD.

$$v \in \mathcal{V} \triangleq \mathbb{Z} \cup \mathbb{B}$$

Les expressions KernelD s'articulent sur un ensemble d'opération $\mathcal{O}p$. Pour l'exemple, on reprend les opérations utilsiées en SystemD. On pose donc

$$op \in \mathcal{O}p \triangleq \{+, -, \times, /, ==, <, \leq, \&\&, ||, !, random_{\mathbb{Z}}, random_{\mathbb{B}}\}$$

où $random_{\mathbb{Z}}$ est une opération aléatoire génératrice d'entiers et $random_{\mathbb{B}}$ est génératrice de booléens.

À part pour les opérations *random*, la sémantique des opérations est la fonction qu'elles réalisent.

$$\forall op \in \{+, -, \times, /, ==, <, \leq, \&\&, ||\},\ \phi_{op}(v_1, v_2, v_1\ op\ v_2)$$
$$\phi_!(v, !v)$$

Pour la sémantique de $random_{\mathbb{Z}}$, on autorise l'utilisation du symbole spécial ∞.

$$\forall z \in \mathbb{Z},\ \phi_{random_{\mathbb{Z}}}(\infty, \infty, z)$$
$$\forall z\ z' \in \mathbb{Z},\ \phi_{random_{\mathbb{Z}}}(\infty, z, z') \quad \text{si } z \geq z'$$
$$\forall z\ z' \in \mathbb{Z},\ \phi_{random_{\mathbb{Z}}}(z, \infty, z') \quad \text{si } z \leq z'$$
$$\forall z_1\ z_2\ z \in \mathbb{Z},\ \phi_{random_{\mathbb{Z}}}(z_1, z_2, z) \quad \text{si } z_1 \leq z \leq z_2$$
$$\forall b \in \mathbb{B}, \phi_{random_{\mathbb{B}}}(b)$$

A.3.2 Variables et Évènements lus et écrits

Les variables et évènements lus et écrits par les programmes dépendent directement des instructions du programme. L'affectation et la notification sont des instructions d'écriture, les expressions et les mises en attente des instructions de lecture.

Définition A.1 Évènements d'un statut $R \in \mathcal{W} \to \mathcal{P}(\mathcal{E})$

$$R(\omega) \triangleq \begin{array}{ll} \emptyset & \text{si } \omega = \textit{ready} \text{ ou } \textit{halt} \\ \{e\} & \text{si } \omega = e \\ R(\omega_1) \cup R(\omega_2) & \text{si } \omega = \omega_1 \wedge \omega_2 \text{ ou } \omega_1 \vee \omega_2 \\ R(\omega') & \text{si } \omega = \neg\omega' \end{array}$$

Définition A.2 Variables d'une expression $R \in \mathcal{E}xp \to \mathcal{P}(\mathcal{X})$

$$R(exp) \triangleq \begin{array}{ll} \emptyset & \text{si } \exists v \in \mathcal{V},\ exp = v \\ \{x\} & \text{si } exp = x \\ \bigcup_{i=1}^{n} R(exp_i) & \text{si } \exists op \in \mathcal{O}p,\ exp = op(exp_1, ..., exp_n) \end{array}$$

Définition A.3 Variables et évènements lus par une instruction $R \in \mathcal{I} \to \mathcal{P}(\mathcal{X} \cup \mathcal{E})$

$$R(inst) \triangleq \begin{array}{ll} \emptyset & \text{si } inst = \varepsilon \\ R(exp) & \text{si } inst = exp? \\ R(exp) & \text{si } inst = x \leftarrow exp \\ \emptyset & \text{si } inst = e! \\ R(\omega) & \text{si } inst = \omega \end{array}$$

Définition A.4 Variable ou évènement écrit par une instruction $W \in \mathcal{I} \to \mathcal{P}(\mathcal{X} \cup \mathcal{E})$

$$R(inst) \triangleq \begin{array}{ll} \emptyset & \text{si } inst = \varepsilon \\ \emptyset & \text{si } inst = exp? \\ \{x\} & \text{si } inst = x \leftarrow exp \\ \{e\} & \text{si } inst = e! \\ \emptyset & \text{si } inst = \omega \end{array}$$

Définition A.5 Variables et évènements lus par un programme $R \in \mathcal{K} \to \mathcal{P}(\mathcal{X} \cup \mathcal{E})$

$$R(\kappa) \triangleq \bigcup R(inst) \quad \text{tel que } \exists q\ q',\ (q, inst, q') \in \kappa$$

Définition A.6 Variables et évènements écrits par un programme $W \in \mathcal{K} \to \mathcal{P}(\mathcal{X} \cup \mathcal{E})$

$$W(\kappa) \triangleq \bigcup W(inst) \quad \text{tel que } \exists q\ q',\ (q, inst, q') \in \kappa$$

Les variables et évènements lus ou écrits par définition par un programme sont donc calculables en temps fini si le programme a une représentation finie. Il suffit de parcourir entièrement le graphe de contrôle du programme pour les connaître. De plus, ce sont effectivement des sur-ensembles des variables et évènements lus ou écrits sémantiquement. En effet, seules les instructions d'affectation et de notification peuvent écrire les variables (pour la première) et

les évènements (pour la seconde). Les variables dont dépend un programme sont clairement celles contenues dans les expressions des affectations et des gardes. Enfin, la configuration d'un programme dépend des évènements qu'il est susceptible d'attendre (sémantique de l'ordonnanceur).

A.3.3 Configurations d'un Programme

Les configurations d'un programme sont ses points de synchronisation (points de contrôle d'où vient une mise en attente) associés aux statuts que le programme peut y attendre. Ces statuts ne sont pas seulement ceux qui apparaissent comme étiquette des transitions, mais également leurs sous-statuts qui peuvent venir de la réduction du statut par certains évènements.

Définition A.7 Sous-statuts d'un statut $Sub \in \mathcal{W} \to \mathcal{P}(\mathcal{W})$

$$\begin{array}{rcll} \mathrm{Sub}(\omega) & \triangleq & \omega & \text{si } \omega = \textit{ready} \text{ ou } \textit{halt} \\ & & \{\textit{ready}, e\} & \text{si } \omega = e \\ & & \{\omega'_1 \wedge \omega'_2 \mid \omega'_i \in \mathrm{Sub}(\omega_i)\} & \text{si } \omega = \omega_1 \wedge \omega_2 \\ & & \{\omega'_1 \vee \omega'_2 \mid \omega'_i \in \mathrm{Sub}(\omega_i)\} & \text{si } \omega = \omega_1 \vee \omega_2 \\ & & \{\textit{ready}\} \cup \{\neg\omega'' \mid \omega'' \in \mathrm{Sub}(\omega')\} & \text{si } \omega = \neg\omega' \end{array}$$

Définition A.8 Configurations d'un programme $C \in \mathcal{K} \to \mathcal{C}$

$$C(\kappa) \triangleq \{(q', \omega') \mid \exists q\ \omega,\ (q, \omega, q') \in \kappa \text{ et } \omega' \in \mathrm{Sub}(\omega)\}$$

Au cours des exécutions d'un système, un programme du système ne peut redonner la main au reste du système que s'il se trouve dans une de ces configurations. En effet, la sémantique à grand pas d'un programme énonce que le programme s'arrête lorsqu'il rencontre une instruction d'attente. De plus, clairement, les réductions transforment un statut en l'un de ses sous-statuts (systèmes de réécriture).

A.4 Ensembles (Partiellement) Ordonnés

Nous rappelons des définitions et résultats sur les ensembles ordonnés. Des présentations plus complètes peuvent se trouver dans la littérature [Sch02, NNH99].

Définition A.9 Un *ensemble partiellement ordonné* (ou *poset*) est un ensemble muni d'une d'ordre (une relation réflexive, antisymétrique et transitive). On note $\langle E, \sqsubseteq \rangle$ un poset E avec $\sqsubseteq$ pour relation d'ordre.

$$\begin{array}{rcl} \langle E, \sqsubseteq \rangle & \Leftrightarrow & \forall e \in E,\ e \sqsubseteq e \\ & & \forall e\ e' \in E,\ e \sqsubseteq e' \text{ et } e' \sqsubseteq e \Rightarrow e = e' \\ & & \forall e_1\ e_2\ e_3 \in E,\ e_1 \sqsubseteq e_2 \text{ et } e_2 \sqsubseteq e_3 \Rightarrow e_1 \sqsubseteq e_3 \end{array}$$

Définition A.10 Soit $\langle E, \sqsubseteq \rangle$ un poset et $F \subseteq E$ un sous-ensemble de E. Une *borne supérieure* (respectivement *inférieure*) de F est un élément de e plus grand (resp. plus petit) que tous les éléments de F. La *plus petite borne supérieure* (respectivement la *plus grande borne inférieure*) de F, notée $\sqcup F$ (resp. $\sqcap F$), est une borne supérieure (resp. inférieure) de F plus petite (resp. grande) que toutes les autres bornes supérieures (resp. inférieures) de F.

$$\begin{array}{l} \forall x \in F,\ x \sqsubseteq \sqcup F \\ \forall x \in F,\ \sqcap F \sqsubseteq x \\ \forall b \in E,\ \forall x \in F,\ x \sqsubseteq b \Rightarrow \sqcup F \sqsubseteq b \\ \forall b \in E,\ \forall x \in F,\ b \sqsubseteq x \Rightarrow b \sqsubseteq \sqcap F \end{array}$$

Définition A.11 Un poset $\langle E, \sqsubseteq \rangle$ est un *treillis* si et seulement si tout couple d'élément de E admet une plus petite borne supérieure et une plus grande borne inférieure. On note $\langle E, \sqsubseteq, \sqcup, \sqcap \rangle$ un treillis.

$$\langle E, \sqsubseteq, \sqcup, \sqcap \rangle \quad \Leftrightarrow \quad \forall e\ e' \in E,\ \exists b\ b' \in E,\ b = e \sqcup e' \text{ et } b' = e \sqcap e'$$

Définition A.12 Un poset $\langle E, \sqsubseteq \rangle$ est un *treillis complet* si et seulement si toute partie de E admet une plus petite borne supérieure et une plus grande borne supérieure. On note $\langle E, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ un treillis complet où $\top = \sqcup E$ est le plus grand élément de E et $\bot = \sqcup \emptyset$ son plus petit élément.

$$\langle E, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle \quad \Leftrightarrow \quad \forall F \sqsubseteq E,\ \exists b\ b' \in E,\ b = \sqcup F \text{ et } b' = \sqcap F$$

Définition A.13 Une *chaîne croissante* d'un poset $\langle E, \sqsubseteq \rangle$ est une partie de E totalement ordonnée par $\sqsubseteq$ et qui possède une borne inférieure.

$$\begin{array}{rcl} F \text{ est une chaîne de } E & \Leftrightarrow & \exists b \in E,\ \forall x \in F,\ b \sqsubseteq x \\ & & \forall x\ y \in F,\ x \sqsubseteq y \text{ ou } y \sqsubseteq x \end{array}$$

Définition A.14 Un *cpo* est un poset qui a un plus petit élément et dont toute chaîne croissante admet une borne supérieure. On note $\langle E, \sqsubseteq, \bot \rangle$ un cpo.

$$\begin{array}{rcl} \langle E, \sqsubseteq, \bot \rangle & \Leftrightarrow & \forall e \in E,\ \bot \sqsubseteq e \\ & & \text{pour toute chaîne } F \text{ de } E, \exists b \in E,\ \forall x \in F,\ x \sqsubseteq b \end{array}$$

Définition A.15 Étant donné un poset $\langle E, \sqsubseteq \rangle$, une fonction $f \in E \to E$ est *croissante* si et seulement si elle préserve l'ordre $\sqsubseteq$.

$$f \text{ croissante} \quad \Leftrightarrow \quad \forall e\ e' \in E,\ e \sqsubseteq e' \Rightarrow f(e) \sqsubseteq f(e')$$

Définition A.16 Étant donné un poset $\langle E, \sqsubseteq \rangle$, une fonction $f \in E \to E$ est *continue* si et seulement si elle est croissante et qu'elle préserve les bornes supérieures des chaînes croissante.

$$f \text{ est continue} \quad \Leftrightarrow \quad \text{pour toute chaîne croissante } F, \sqcup\{f(x) \mid x \in F\} = f(\sqcup F)$$

Définition A.17 Soient $\langle E, \sqsubset \rangle$ un poset et $f \in E \to E$ une fonction. Un *point fixe* de f est un élément $x \in E$ tel que $f(x) = x$. Le plus petit point fixe de f est un point fixe plus petit que tous les autres points fixes de la fonction. On note $lfp(f)$ le plus petit point fixe de f.

$$f(lfp(f)) = lfp(f) \qquad \forall x \in E,\ f(x) = x \Rightarrow lfp(f) \sqsubseteq x$$

Théorème A.1 (Tarski-Davis) Soit E un treillis complet et $f \in E \to E$ une fonction continue sur E. Alors, f admet un plus petit point fixe.

$$E \text{ est un treillis complet et } f \text{ est continue sur } E \implies \exists x \in E,\ x = lfp(f)$$

Des articles [Tar55, CC79] et livres [Sch02] détaillent ce théorème ainsi que sa preuve.

Théorème A.2 (Point fixe de Kleene) Soit E un treillis complet de plus petit élément $\bot$ et $f \in E \to E$ une fonction continue sur E. Le point fixe de f peut d'obtenir en considérant les plus petites bornes supérieures des itérés de f à partir de $\bot$.

$$lfp(f) \quad = \quad \sqcup\{f^i(\bot) \mid i \in \mathbb{N}\} \qquad \text{où } \forall x,\ f^0(x) = x \text{ et } f^{i+1}(x) = f(f^i(x))$$

Le lecteur intéressé par ce théorème et sa preuve peut se référer à la littérature [Bir67].

Afin d'accélérer — voire de réduire à un temps fini — le calcul d'un point fixe, on utilise un opérateur d'élargissement, puis un opérateur de rétrécissement qui raffinera le résultat [CC77, CC92].

Définition A.18 Étant donné un domaine $\mathcal{D}$, un *opérateur d'élargissement* $\nabla \in \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ est une fonction qui statisfait les propriétés suivantes :
- $\forall d_1\ d_2 \in \mathcal{D},\ d_1 \sqcup d_2 \sqsubseteq d_1 \nabla d_2$;
- pour toute chaîne croissante $d_0 d_1 ...$, la chaîne $s_0 s_1 ...$ telle que $s_0 = d_0$ et $\forall i,\ s_{i+1} = s_i \nabla d_{i+1}$ est ultimement stationnaire, c'est-à-dire que $\exists n_0,\ \forall i > n_0,\ s_i = s_{n_0}$.

Définition A.19 Étant donné un domaine $\mathcal{D}$, un *opérateur de rétrécissement* $\triangle \in \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ est une fonction qui satisfait les deux propriétés suivantes :
- $\forall d_1\ d_2 \in \mathcal{D},\ d_1 \sqsubseteq d_2 \Rightarrow d_1 \sqsubseteq d_1 \triangle d_2 \sqsubseteq d_2$;
- pour toute chaîne décroissante $d_0 d_1 ...$, la chaîne $s_0 s_1 ...$ telle que $s_0 = d_0$ et $\forall i,\ s_{i+1} = s_i \triangle d_{i+1}$ est ultimement stationnaire, c'est-à-dire que $\exists n_0,\ \forall i > n_0,\ s_i = s_{n_0}$.

A.5 Treillis des Booléens

Cette section reprend des définitions et résultats connus sur le domaine des booléens [NNH99, ES96].

Le treillis des booléens est noté $\mathbb{B}^\sharp$. Comme l'ensemble des booléens est fini, on se contente de lui ajouter un élément pour représenter l'ensemble tout entier et un élément pour représenter l'absence de valeur.

$$\mathbb{B}^\sharp \quad \triangleq \quad \{\bot,\ \mathbf{true}^\sharp,\ \mathbf{false}^\sharp,\ \top\}$$

Pour ce qui est de l'ordre $\sqsubseteq_{\mathbb{B}^\sharp}$, de la plus petite borne supérieure $\sqcup_{\mathbb{B}^\sharp}$ et de la plus grande borne inférieure $\sqcap_{\mathbb{B}^\sharp}$, le *diagramme de Hasse* du treillis en figure A.1 suffit à tous les représenter de façon concise.

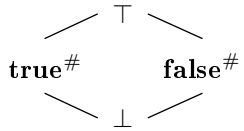

FIG. A.1 – Diagramme de Hasse du treillis des booléens

Pour définir une connexion de Galois entre $\mathbb{B}^\sharp$ et l'ensemble $\mathbb{B}$ des booléens, on s'appuie sur les fonctions d'abstraction $\alpha_{\mathbb{B}^\sharp}$ et de concrétisation $\gamma_{\mathbb{B}^\sharp}$ suivantes :

$$\begin{array}{lcllcl}
\alpha_{\mathbb{B}^\sharp}(\mathbb{B}) & = & \top_{\mathbb{B}^\sharp} & \gamma_{\mathbb{B}^\sharp}(\top_{\mathbb{B}^\sharp}) & = & \mathbb{B} \\
\alpha_{\mathbb{B}^\sharp}(\{\mathbf{true}\}) & = & \mathbf{true}^\sharp & \gamma_{\mathbb{B}^\sharp}(\mathbf{true}^\sharp) & = & \{\mathbf{true}\} \\
\alpha_{\mathbb{B}^\sharp}(\{\mathbf{false}\}) & = & \mathbf{false}^\sharp & \gamma_{\mathbb{B}^\sharp}(\mathbf{false}^\sharp) & = & \{\mathbf{false}\} \\
\alpha_{\mathbb{B}^\sharp}(\emptyset) & = & \bot_{\mathbb{B}^\sharp} & \gamma_{\mathbb{B}^\sharp}(\bot_{\mathbb{B}^\sharp}) & = & \emptyset
\end{array}$$

Avec ces définitions, on a $(\mathbb{B}, \subseteq) \underset{\alpha_{\mathbb{B}^\sharp}}{\overset{\gamma_{\mathbb{B}^\sharp}}{\leftrightarrows}} (\mathbb{B}^\sharp, \sqsubseteq_{\mathbb{B}^\sharp})$.

Comme le domaine est fini et très petit en hauteur, on utilise la plus petite borne supérieure et la plus grande borne inférieure respectivement pour les opérateurs d'élargissement et de rétrécissement.

Les opérations booléennes sont définies comme suit dans l'abstraction, où étant donné un symbole d'opération op, on note $op^\sharp$ sa fonction abstraite associée au lieu de $\phi^\sharp_{op}$:

$$b_1^\sharp \;\&\&^\sharp\; b_2^\sharp \;=\; \begin{cases} \bot & \text{si } b_1^\sharp \text{ ou } b_2^\sharp = \bot \\ \mathbf{false}^\sharp & \text{sinon et si } b_1^\sharp \text{ ou } b_2^\sharp = \mathbf{false}^\sharp \\ \mathbf{true}^\sharp & \text{sinon et si } b_1^\sharp = b_2^\sharp = \mathbf{true}^\sharp \\ \top & \text{sinon} \end{cases}$$

$$b_1^\sharp \;||^\sharp\; b_2^\sharp \;=\; \begin{cases} \bot & \text{si } b_1^\sharp \text{ ou } b_2^\sharp = \bot \\ \mathbf{true}^\sharp & \text{sinon et si } b_1^\sharp \text{ ou } b_2^\sharp = \mathbf{true}^\sharp \\ \mathbf{false}^\sharp & \text{sinon et si } b_1^\sharp = b_2^\sharp = \mathbf{false}^\sharp \\ \top & \text{sinon} \end{cases}$$

$$!^\sharp b^\sharp \;=\; \begin{cases} \mathbf{true}^\sharp & \text{si } b^\sharp = \mathbf{true}^\sharp \\ \mathbf{false}^\sharp & \text{si } b^\sharp = \mathbf{false}^\sharp \\ b^\sharp & \text{sinon} \end{cases}$$

Les opérations booléennes arrières prennent en plus un booléen abstrait dont on souhaite qu'il soit le résultat de l'opération. Le résultat de ces opérations est une valeur raffinée pour chaque argument de l'opération. On note $b_1^\sharp \;\overleftarrow{op}^\sharp\; b_2^\sharp : b^\sharp$ la sémantique abstraite arrière de l'opération

op.

$$
\begin{array}{lcl}
b_1^\sharp \overleftarrow{\&\&}^\sharp b_2^\sharp : \mathbf{true}^\sharp & = & (\mathbf{true}^\sharp,\ \mathbf{true}^\sharp) \\
b_1^\sharp \overleftarrow{\&\&}^\sharp b_2^\sharp : \mathbf{false}^\sharp & = & \begin{cases} (\bot, \bot) & \text{si } b_1^\sharp = b_2^\sharp = \mathbf{true}^\sharp \\ (b_1^\sharp, b_2^\sharp) & \text{sinon} \end{cases} \\
b_1^\sharp \overleftarrow{||}^\sharp b_2^\sharp : \mathbf{true}^\sharp & = & \begin{cases} (\bot, \bot) & \text{si } b_1^\sharp = b_2^\sharp = \mathbf{false}^\sharp \\ (b_1^\sharp, b_2^\sharp) & \text{sinon} \end{cases} \\
b_1^\sharp \overleftarrow{||}^\sharp b_2^\sharp : \mathbf{false}^\sharp & = & (\mathbf{false}^\sharp,\ \mathbf{false}^\sharp) \\
\overleftarrow{!}^\sharp b^\sharp : \mathbf{true}^\sharp & = & \mathbf{false}^\sharp \\
\overleftarrow{!}^\sharp b^\sharp : \mathbf{false}^\sharp & = & \mathbf{true}^\sharp
\end{array}
$$

A.6 Treillis des Intervalles

Cette section reprend des définitions et résultats connus sur le domaine des intervalles [CC77, Moo79, Rev01, ES96].

L'ensemble des intervalles est noté $\mathbb{I}$. Un intervalle est un couple formé avec des entiers ou avec le symbole spécial ∞. On pose $\mathbb{Z}^\infty = \mathbb{Z} \cup \{-\infty, +\infty\}$ et nous étendons la relation d'ordre $\leq$ définie sur $\mathbb{Z}$ à $\pm\infty$, ce qui donne la relation notée $\leq_\infty$ définie comme suit :

$$\forall z\ z' \in \mathbb{Z},\ z \leq_\infty z' \triangleq z \leq z'$$

$$\forall z \in \mathbb{Z}^\infty,\ z \leq_\infty +\infty \text{ et } -\infty \leq_\infty z$$

Abus de langage : lorsque ceci n'est pas ambigu, nous utiliserons $\leq$ à la place de $\leq_\infty$. Ceci vaudra également pour tous les opérateurs que nous allons définir pour lesquels le sous-script ne sera indiqué que lorsqu'il y aura possibilité d'ambiguïté.

Un intervalle est alors soit vide, noté $\emptyset_\mathbb{I}$, soit bien formé : sa borne gauche est inférieure à sa borne droite. Un intervalle qui ne respecte pas cette condition est considéré vide.

$$
\begin{array}{rcll}
\mathbb{I} & \triangleq & \emptyset & \\
 & | & z..z' & \text{où } z, z' \in \mathbb{Z}^\infty \text{ et } z \leq z'
\end{array}
$$

Remarque : ces règles imposent que dans un intervalle bien formé, le symbole ∞ ne peut apparaître comme borne gauche qu'avec le signe $-$, et qu'il ne peut apparaître comme borne droite qu'avec le signe $+$. Nous omettrons donc souvent le signe en suivant cette remarque.

Ordre La relation d'ordre sur les intervalles, notée $\subseteq_\mathbb{I}$, est définie par les règles suivantes :

$$\forall I,\ \emptyset \subseteq I$$

$$a..b \subseteq c..d \quad \Leftrightarrow \quad c \leq a \text{ et } b \leq d$$

Plus petite borne supérieure Elle représente l'union large des intervalles, c'est-à-dire l'intervalle entre la plus petite et la plus grande des valeurs qui apparaissent dans les arguments. On la note $\cup_{\mathbb{I}}$.

$$\forall I,\ \emptyset \cup I \triangleq I$$

$$a..b \cup c..d \triangleq z..z' \quad \text{où } z = \min_{\leq}\{a, c\} \text{ et } z' = \max_{\leq}\{b, d\}$$

Plus grande borne inférieure Elle est symbolisée par l'opération ensembliste duale de l'union : l'intersection $\cap_{\mathbb{I}}$.

$$\forall I,\ \emptyset \cap I \triangleq \emptyset$$

$$a..b \cap c..d \triangleq z..z' \quad \text{où } z = \max_{\leq}\{a, c\} \text{ et } z' = \min_{\leq}\{b, d\}$$

Plus grand élément C'est $\infty..\infty$. Il représente l'ensemble de tous les entiers.

Plus petit élément C'est naturellement l'intervalle vide $\emptyset$.

Le domaine $(\mathbb{I}, \subseteq, \cup, \cap, \infty..\infty, \emptyset)$ ainsi obtenu est un treillis complet. Il est en relation de Galois avec l'ensemble des parties des entiers.

En effet, soient $\alpha_{\mathbb{I}} \in \mathcal{P}(\mathbb{Z}) \to \mathbb{I}$ une fonction d'abstraction et $\gamma_{\mathbb{I}} \in \mathbb{I} \to \mathcal{P}(\mathbb{Z})$ une fonction de concrétisation définies comme suit :

$$\begin{array}{rcll} \alpha_{\mathbb{I}}(Z) & \triangleq & \emptyset_{\mathbb{I}} & \text{si } Z = \emptyset \\ & & \infty..\infty & \text{si } Z = \mathbb{Z} \\ & & \infty..\max(Z) & \text{sinon et si } \forall z \in Z,\ \exists z' \in Z,\ z' < z \\ & & \min(Z)..\infty & \text{sinon et si } \forall z \in Z,\ \exists z' \in Z,\ z < z' \\ & & \min(Z)..\max(Z) & \text{sinon} \end{array}$$

$$\gamma_{\mathbb{I}}(\emptyset_{\mathbb{I}}) \triangleq \emptyset$$

$$\gamma_{\mathbb{I}}(z_1..z_2) \triangleq \{z \in \mathbb{Z} \mid z_1 \leq_{\infty} z \leq_{\infty} z_2\}$$

Alors,

$$(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha_{\mathbb{I}}}{\overset{\gamma_{\mathbb{I}}}{\leftrightarrows}} (\mathbb{I}, \subseteq_{\mathbb{I}})$$

On utilise les définitions des opérateurs d'élargissement et de rétrécissement suivantes :

$$a..b \ \nabla\ c..d \ = \ (a \text{ si } a \leq c \text{ et } \infty \text{ sinon})..(b \text{ si } d \leq b \text{ et } \infty \text{ sinon})$$

$$a..b \ \triangle\ c..d \ = \ (c \text{ si } a = \infty \text{ et } \mathit{min}\{a, c\} \text{ sinon})..(d \text{ si } b = \infty \text{ et } \mathit{max}\{b, d\} \text{ sinon})$$

Les opérations arithmétiques sur les intervalles peuvent être définies comme suit, où les opérations sur les entiers sont étendues sur $-\infty$ et $+\infty$ qui y sont absorbants et respectent les conventions sur les signes. Sont indéfinies les opérations sur l'intervalle vide et les divisions

par un intervalle qui contient 0. Étant donné un symbole d'opération op, on note $op^\sharp$ sa fonction abstraite associée au lieu de $\phi^\sharp_{op}$.

$$
\begin{array}{lcl}
I_1\ op^\sharp\ I_2 & = & \emptyset \quad \text{si } I_1 \text{ ou } I_2 = \emptyset \\
a..b +^\sharp c..d & = & (a+c)..(b+d) \\
a..b -^\sharp c..d & = & (a-c)..(b-d) \\
a..b \times^\sharp c..d & = & \min(Z)..\max(Z) \quad \text{où } Z = \{ac, ad, bc, bd\} \\
1..1\ /^\sharp\ a..b & = & \begin{cases} (0 \text{ si } b = \infty \text{ et } 1/b \text{ sinon})..(0 \text{ si } a = \infty \text{ et } 1/a \text{ sinon}) & \text{si } 0 < a \text{ ou } b < 0 \\ \emptyset & \text{sinon} \end{cases} \\
\text{random}^\sharp_{\mathbb{Z}}(a..b, c..d) & = & a..d
\end{array}
$$

$$
a..b <^\sharp c..d \ = \ \begin{cases} \mathbf{true}^\sharp & \text{si } b < c \\ \mathbf{false}^\sharp & \text{si } a \geq d \\ \top_{\mathbb{B}^\sharp} & \text{sinon} \end{cases}
\qquad
a..b \leq^\sharp c..d \ = \ \begin{cases} \mathbf{true}^\sharp & \text{si } b \leq c \\ \mathbf{false}^\sharp & \text{si } a > d \\ \top_{\mathbb{B}^\sharp} & \text{sinon} \end{cases}
$$

$$
a..b ==^\sharp c..d \ = \ \begin{cases} \mathbf{true}^\sharp & \text{si } a = b = c = d \\ \mathbf{false}^\sharp & \text{si } a..b < c..d \text{ ou } a..b > c..d \\ \top_{\mathbb{B}^\sharp} & \text{sinon} \end{cases}
$$

Les opérations arithmétiques arrières sur les intervalles prennent en plus un intervalle dont on souhaite qu'il contienne le résultat de l'opération. Le résultat de ces opérations est une valeur raffinée pour chaque argument de l'opération. On note $I_1\ \overleftarrow{op}^\sharp\ I_2 : I$ la sémantique abstraite arrière de l'opération op.

$$
\begin{array}{lcl}
I_1\ \overleftarrow{op}^\sharp\ I_2 : I & = & (\emptyset, \emptyset) \quad \text{si } I_1 \text{ ou } I_2 \text{ ou } I = \emptyset \\
I_1\ \overleftarrow{+}^\sharp\ I_2 : I & = & (I -^\sharp I_2,\ I -^\sharp I_1) \\
I_1\ \overleftarrow{-}^\sharp\ I_2 : I & = & (I +^\sharp I_2,\ I +^\sharp I_1) \\
I_1\ \overleftarrow{\times}^\sharp\ I_2 : I & = & (I\ /^\sharp\ I_2,\ I\ /^\sharp\ I_1) \\
I_1\ \overleftarrow{/}^\sharp\ I_2 : I & = & (I \times^\sharp I_2,\ I \times^\sharp I_1) \\
\overleftarrow{\text{random}}^\sharp_{\mathbb{Z}}(I_1, I_2) : I & = & (I,\ I) \\
a..b\ \overleftarrow{<}^\sharp\ c..d : \mathbf{true}^\sharp & = & (\infty..(c-1),\ (b+1)..\infty) \\
I_1\ \overleftarrow{<}^\sharp\ I_2 : \mathbf{false}^\sharp & = & I_2\ \overleftarrow{\leq}^\sharp\ I_1 : \mathbf{true}^\sharp \\
a..b\ \overleftarrow{\leq}^\sharp\ c..d : \mathbf{true}^\sharp & = & (\infty..c,\ b..\infty) \\
I_1\ \overleftarrow{\leq}^\sharp\ I_2 : \mathbf{false}^\sharp & = & I_2\ \overleftarrow{<}^\sharp\ I_1 : \mathbf{true}^\sharp \\
I_1\ \overleftarrow{==}^\sharp\ I_2 : \mathbf{true}^\sharp & = & (I_2,\ I_1) \\
a..b\ \overleftarrow{==}^\sharp\ c..d : \mathbf{false}^\sharp & = & \begin{cases} (\emptyset, \emptyset) & \text{si } a = b = c = d \\ (I_1, I_2) & \text{sinon} \end{cases}
\end{array}
$$

A.7 Combinaisons de Treillis

Cette section reprend des résultats de combinaisons de treillis qui peuvent se trouver dans la littérature [NNH99]. En particulier, nous ne montrons pas la correction des constructions que nous définissons.

Union. Soit $(\mathcal{D}_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \top_1, \bot_1)$ et $(\mathcal{D}_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \top_2, \bot_2)$ deux treillis. Le treillis $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ union de $\mathcal{D}_1$ et $\mathcal{D}_2$ est défini comme suit :

$$\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \cup \{\top, \bot\}$$

$$d \sqsubseteq d' \Leftrightarrow \begin{cases} & d = \bot \\ \text{ou} & d' = \top \\ \text{ou} & d \text{ et } d' \in \mathcal{D}_1 \text{ et } d \sqsubseteq_1 d' \\ \text{ou} & d \text{ et } d' \in \mathcal{D}_2 \text{ et } d \sqsubseteq_2 d' \end{cases}$$

$$d \sqcup d' = \begin{cases} d \sqcup_1 d' & \text{si } d \text{ et } d' \in \mathcal{D}_1 \\ d \sqcup_2 d' & \text{si } d \text{ et } d' \in \mathcal{D}_2 \\ \top & \text{sinon} \end{cases}$$

$$d \sqcap d' = \begin{cases} d \sqcap_1 d' & \text{si } d \text{ et } d' \in \mathcal{D}_1 \\ d \sqcap_2 d' & \text{si } d \text{ et } d' \in \mathcal{D}_2 \\ \bot & \text{sinon} \end{cases}$$

Produit cartésien. Soit $(\mathcal{D}_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \top_1, \bot_1)$ et $(\mathcal{D}_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \top_2, \bot_2)$ deux treillis. Le treillis $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ produit de $\mathcal{D}_1$ et $\mathcal{D}_2$ est défini comme suit :

$$\mathcal{D} = \mathcal{D}_1 \times \mathcal{D}_2$$

$$(d_1, d_2) \sqsubseteq (d'_1, d'_2) \Leftrightarrow d_1 \sqsubseteq_1 d'_1 \text{ et } d_2 \sqsubseteq_2 d'_2$$
$$(d_1, d_2) \sqcup (d'_1, d'_2) = (d_1 \sqcup_1 d'_1, d_2 \sqcup_2 d'_2)$$
$$(d_1, d_2) \sqcap (d'_1, d'_2) = (d_1 \sqcap_1 d'_1, d_2 \sqcap_2 d'_2)$$
$$\top = (\top_1, \top_2)$$
$$\bot = (\bot_1, \bot_2)$$

Fonction totale. Soit $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ un treillis et E une fonction totale. Le treillis $(\mathcal{F}, \sqsubseteq_{\mathcal{F}}, \sqcup_{\mathcal{F}}, \sqcap_{\mathcal{F}}, \top_{\mathcal{F}}, \bot_{\mathcal{F}})$ des fonctions totales de $\mathcal{E}$ vers $\mathcal{D}$ est défini comme suit :

$$\mathcal{F} = \mathcal{E} \to \mathcal{D}$$

$$f \sqsubseteq_{\mathcal{F}} f' \Leftrightarrow \forall e,\ f(e) \sqsubseteq f'(e)$$
$$f \sqcup_{\mathcal{F}} f' = \lambda e,\ f(e) \sqcup f'(e)$$
$$f \sqcap_{\mathcal{F}} f' = \lambda e,\ f(e) \sqcap f'(e)$$
$$\top_{\mathcal{F}} = \lambda e,\ \top$$
$$\bot_{\mathcal{F}} = \lambda e,\ \bot$$