Compilation de réseaux de Petri : modèles haut niveau et symétries de processus

Lukasz Fronc

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ D'ÉVRY-VAL D'ESSONNE

Spécialité

Informatique

École doctorale Sciences et Ingénierie (ED511)

Présentée par

Łukasz FRONC

Pour obtenir le grade de **DOCTEUR**

Compilation de réseaux de Petri

modèles haut niveau et symétries de processus

Soutenue le 28 novembre 2013

Devant le jury composé de :

Directeurs :	Mme. Hanna Klaudel
	M. Franck Pommereau
Rapporteurs :	M. Jean-François Pradat-Peyre
	M. Olivier H. Roux
Examinateurs :	Mme. Laure Petrucci
	M. Denis Poitrenaud

Sommaire

1 Introduction

La *science informatique* est en plein essort depuis ces 50 dernières années. Cette thèse concerne une de ses branches majeures, la *vérification* [37]. L'objectif de la vérification est d'aider au développement de systèmes informatiques ou automatisables de manière *fiable*. Cela correspond à garantir diverses propriétés sur ces systèmes comme des propriétés de sûreté ou de sécurité. Il peut par exemple s'agir de vérifier qu'un train automatisé n'ouvrira pas ses portes en plein trajet, ou bien que nos identifiants bancaires ne peuvent pas nous être volés lorsqu'on accède à notre compte via une interface web. Plus généralement, toutes les applications ou systèmes critiques devraient être vérifiés, en amont de leur réalisation pratique. Malheureusement cela n'est pas toujours possible car les systèmes qu'on souhaite réaliser sont toujours *plus grands* et toujours *plus complexes* ce qui rend la tâche difficile.

Pour mettre en oeuvre la vérification, on réalise tout d'abord un *modèle* du système. C'est-à-dire une représentation mathématique sur laquelle on raisonne et vérifie des propriétés qui nous intéressent. Il faut garder à l'esprit qu'un modèle est une *abstraction* d'un système, il capture ses aspects importants mais ne lui sera jamais complètement fidèle. Un tel modèle est réalisé avec un *formalisme de modélisation*, par exemple des réseaux de Petri ou des automates. Le choix de ce formalisme est important puisqu'il influe souvent sur les techniques de vérification utilisées par la suite. L'*expressivité* de ce formalisme a de plus un fort impact sur la qualité du modèle, au niveau de la mise en oeuvre, mais aussi sur la possibilité de le vérifier par la suite. L'utilisation d'un formalisme expressif peut grandement aider à la modélisation mais peut être bloquante pour l'étape de vérification en elle-même.

Par la suite, les propriétés qu'on souhaite vérifier sur ces modèles sont exprimées, par exemple, en logiques temporelles, ou plus généralement modales. Un ensemble de telles propriétés correspond à une *spécification* que doit satisfaire le système. Ici aussi, ces propriétés sont sous une représentation mathématique et donc sujettes à des erreurs de spécification de l'ingénieur ou du chercheur les réalisant, tout comme le modèle.

Une fois le modèle et sa spécification réalisés, on procède à une analyse pour savoir si le modèle satisfait la spécification. Cela peut être fait avec différentes techniques comme des *assistants de preuve* ou du *model checking*. Des approches hybrides comme la *méthode B* [12], aident à réaliser le modèle et la spécification conjointement, puis à appliquer dessus des techniques de preuves

et de model checking.

Cette phase d'analyse permet de garantir les propriétés sur le modèle, cependant cela ne garantit pas une propriété sur le système, puisqu'on l'a vérifiée sur une abstraction de celui-ci. Si le modèle n'était pas assez précis alors la propriété peut ne pas être vraie sur le système, de même si la propriété a été mal spécifiée. La réalisation d'un modèle et de sa spécification est donc une étape non triviale qui peut demander un considérable degré d'expertise sur le système et sur les méthodes mises en oeuvre.

Ces différentes approches de vérification ne sont pas parfaites et chacune possède ses limites et son lot de difficultés. Avec un assistant de preuve la spécification est définie sous forme de théorèmes qui doivent être prouvés individuellement et sont non triviaux en général. Ces preuves demandent une *grande expertise* de l'assistant de preuve et du modèle. De plus elles sont réalisées *manuellement* dans la majorité des cas. En pratique il existe des méthode automatiques pour réaliser des preuves mais elles sont limitées, par exemple des équations et inéquations dans l'arithmétique de Presburger. L'avantage cependant est qu'on réalise une *certification* du modèle, c'est-à-dire que nous avons une *preuve mathématique* qu'il vérifie sa spécification.

Avec le model checking, nous sommes essentiellement limités par les capacités de calcul. Il s'agit d'une méthode *complètement automatique* qui consiste à considérer toutes les exécutions possibles du modèle, et donc les exécutions simplifiées du système, puis à observer qu'elles ne violent pas la spécification. L'avantage principal est la non nécessité d'expertise dans les techniques de vérification, un ingénieur peut aisément utiliser ce type de techniques. De plus lorsque certaines techniques de model checking sont utilisées, si une propriété n'est pas satisfaite, alors on peut avoir accès à un *contre-exemple* qui correspond à un scénario qui mène à l'erreur. Cela est très intéressant d'un point de vue d'ingénierie puisqu'on peut aisément corriger l'erreur. Cependant, les modèles ont tendance à être grands et considérer toutes leurs exécutions peut être impossible. En général, on a un nombre *exponentiellement plus grand* d'exécutions par rapport à la taille du modèle. Dans ce cas on parle du problème d'*explosion combinatoire* qui est un des freins de ces approches exhaustives. Pour pallier ce problème des *techniques de réduction* sont nécessaires.

Plus généralement toutes les méthodes de vérification sont limitées par :

- la *décidabilité* qui concerne directement la faisabilité de l'approche, des propriétés non décidables telles que la terminaison ne peuvent pas être garanties dans le cadre général ;
- les *complexités en temps et en espace* qui sont des freins à l'efficacité, les systèmes sont en général de grande taille leur calcul doit donc être efficace en temps mais aussi en mémoire ;
- l'*expressivité* des formalismes de modélisation qui marquent les limites entre la facilité de mise en oeuvre et l'efficacité ou la faisabilité, un langage très expressif est difficilement efficace en temps de calculs et réciproquement un langage bas niveau rend la modélisation difficile ;
- les techniques de *model checking* utilisées qui influent sur la qualité des résultats obtenus mais aussi limitent les degrés de libertés qu'on a sur les leviers ci-dessus, l'utilisation de model checking symbolique limite l'expressivité du modèle mais le model checking explicite est limité par la taille des modèles ;

– la *certification* qui est en lien direct avec la confiance qu'on peut avoir en l'outil de vérification et son résultat.

Cette thèse attaque les questions d'un *compromis* entre les différents aspects abordés ci-dessus.

1. Pour aborder la décidabilité on utilise une approche optimiste :

   a) On considère des modèles qui embarquent un langage *Turing complet*, c'est-à-dire qu'on est *semi-décidable*. Cela implique que c'est à l'utilisateur, ou au programmeur, de faire attention à son modèle. Il s'agit de la même démarche que lors du débogage d'un programme informatique, par exemple, la présence d'une boucle infinie sera détectée à l'exécution puis corrigée.

   b) On considère des *domaines infinis* ce qui rend aussi notre approche *semi-décidable*. Ici aussi c'est à l'utilisateur de détecter les erreurs, par exemple, un compteur dans le modèle qui serait incrémenté à l'infini.

   Cette approche optimiste permet de faciliter la modélisation : on ne limite pas l'utilisateur qui dans le cadre de notre approche utilise un sous-ensemble efficace du langage mis à sa disposition, tout en permettant d'utiliser des constructions plus complexes du langage, ou du modèle, au besoin. Ainsi un utilisateur n'est que rarement au contact des problèmes de semi-décidabilité du langage utilisé.

2. Un de nos objectifs dans ce manuscrit est de concilier l'expressivité et l'efficacité en temps. Pour cela on utilise des *réseaux de Petri de haut niveau* annotés par un langage de programmation standard et une approche de compilation qui est une transformation du réseau en programme informatique permettant de calculer tous ses états plus rapidement. Les réseaux de Petri sont un formalisme qui a l'avantage d'avoir une représentation graphique intuitive et facile à appréhender et donc à utiliser. Un langage standard permet une compilation efficace des annotation du réseau grâce à un compilateur classique. Cependant, ce choix de privilégier l'expressivité des modèles nous oblige à l'utilisation de méthodes de model checking *explicites* : les états et les changements d'états du modèle sont considérés individuellement. En effet, les méthodes de model checking symbolique, qui considèrent des opérations sur des ensembles d'états, peuvent le faire essentiellement car le langage d'annotations utilisé est limité et exhibe des propriétés qui le permettent. Les modèles qu'on utilise sont abordés avec un état de l'art dans le chapitre 2 , la compilation est présentée dans le chapitre 3.

3. La problématique de certification et de la confiance dans l'outil est aussi traitée. Dans les chapitres 4 et 5, on s'intéresse à montrer comment on peut prouver le model checker, c'est-à-dire garantir qu'on engendre des programmes corrects. Il faut remarquer qu'il ne s'agit pas d'une certification totale, on prouve que les algorithmes générés sont corrects, pas que le compilateur engendre effectivement ces algorithmes. Ce serait réalisable mais hors du cadre de cette thèse.

4. La complexité en espace est abordée dans le cadre de réductions par symétries. Les réductions par symétries ont été abondamment étudiées,

on s'intéresse donc à une extension particulière où de multiples processus exécutant le réseau sont mis en oeuvre avec la possibilité de créer de nouveaux processus à l'exécution. Dans ce cadre on propose une méthode nouvelle de réduction d'espaces d'états qui élimine des exécutions redondantes (équivalentes). Cela est traité dans le chapitre 6.

5. Finalement dans le chapitre 7, on présente l'outil *Neco* qui a été réalisé dans le cadre de cette thèse et implémente les approches ci-dessus. Avec cette implémentation on présente des études de cas pour chacune des approches développées. On conclue le manuscrit en donnant quelques perspectives. Une annexe est aussi présente et contient des résultats auxiliaires.

2 Notions préliminaires et état de l'art

Les *réseaux de Petri* sont des modèles mathématiques introduits en 1962 par Carl Adam Petri lors de sa thèse de doctorat [141]. Ces modèles permettent de représenter des systèmes automatisables (informatiques, industriels, ...), puis par diverses techniques (*model checking*, invariants, ...) des propriétés sur ces systèmes peuvent être vérifiées [37]. Les domaines d'application sont nombreux et couvrent un large panorama, on retrouve les réseaux de Petri par exemple dans la vérification de circuits intégrés [143], de protocoles de sécurité [13], de problèmes à caractère social [41], et bien d'autres.

Un réseau de Petri est représenté comme un graphe avec deux types de noeuds : les *places* et les *transitions*. Les places contiennent les données appelées *jetons*, et les transitions indiquent les calculs à réaliser avec les jetons. Le tirage d'une transition (une étape de calcul) correspond à la consommation et production de jetons des places. Les places sont connectées aux transitions avec des arcs entrants (*inputs*) indiquant les jetons qui doivent être consommés, et des arcs sortants (*outputs*) indiquant les jetons qui doivent être produits dans la place. L'état du réseau est représenté par le contenu des places. Un exemple de réseau de Petri est donné en figure 2.1, ce réseau modélise une porte et on y retrouve ses deux état possibles : ouverte ou fermée. Le tirage de la transition "`ferme porte`" permet de passer de l'état fermé à l'état ouvert en consommant le jeton de la place "`ouverte`" et en produisant un jeton dans la place "`fermée`".

On appelle les réseaux de Petri décrits ci-dessus *réseaux bas niveau* ou *réseaux place-transition*. Ce modèle a été étendu de nombreuses manières pour aider à la vérification mais aussi modéliser des systèmes de manière plus fidèle, ou plus simple et naturelle en gagnant de l'expressivité. Il s'agit souvent d'extensions pour des domaines spécifiques d'application. En effet, un modèle reste une abstraction de la réalité et son expressivité doit être limitée (sous la barre de la décidabilité, ou dans des complexités raisonnables) si on souhaite pouvoir l'analyser. Parmi les extensions largement répandues, on retrouve par exemple :

1. Les *réseaux colorés* [102, 104, 103, 106] où chaque jeton a une couleur représentant la nature de la donnée stockée. Cette couleur a un rôle similaire au type d'une variable dans un programme informatique. Un réseau bas niveau peut être vu comme un réseau coloré avec un seul type de jetons : le jeton noir •.

2. Les *réseaux hiérarchiques* où les éléments du réseau sont eux mêmes des réseaux de Petri [103, 95]. Selon la nature de l'extension il peut s'agir, par exemple, de noeuds (places ou transitions) qui sont substitués par des réseaux de Petri [95] ou encore de jetons qui sont des réseaux de Petri [14]. La motivation derrière ce type d'extension est l'observation que les systèmes de grande taille ont besoin d'être manipulés à plusieurs niveaux d'abstraction, la hiérarchisation est un mécanisme qui le permet [95]. Le second type de hiérarchisation mentionné ci-dessus, peut aussi être détourné comme dans [14] où le jeton qui est un réseau de Petri est modifié de manière dynamique.
3. Les réseaux étendus par des contraintes de temps qui sont des réseaux de Petri paramétrés par des horloges, ce qui permet de modéliser des contraintes temporelles en divers points de l'exécution [146]. On peut citer deux tendances par exemple les *Time Petri nets* [128] et *les Timed Petri nets* [147]. Dans les premiers, deux temps sont associés à chaque transition et délimitent le laps de temps où la transition est tirable, hors de cet intervalle la transition ne peut être tirée. Dans les seconds chaque transition est associée à une durée qui est le temps nécessaire pour effectuer le tir. De cette manière le tirage de transitions soumis à des horloges permet de garantir des propriétés sur les *temps de réponse* [128] ou l'évaluation de *performances* [147, 152].
4. Les *réseaux stochastiques* où des probabilités dans le temps ont été introduites pour paramétrer le tirage de transitions. Si une seule transition est tirable, elle le sera nécessairement, mais si plusieurs le sont, surtout en cas de conflit, alors la probabilité permet de choisir laquelle tirer d'abord. Ainsi le tirage d'une transition est soumis à une loi de probabilité et donc un état du réseau n'est atteignable que sous une certaine probabilité [130].

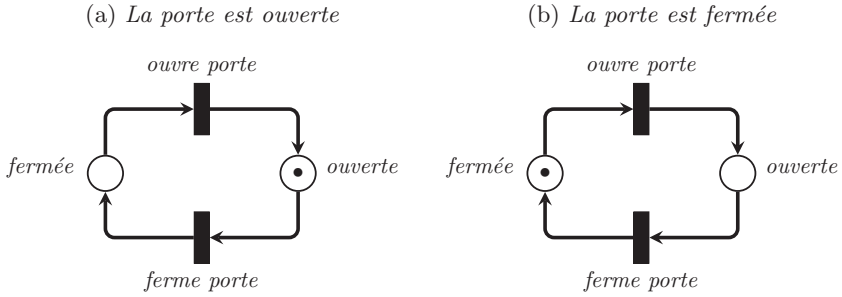


Figure 2.1 – *Une porte modélisée avec un réseau de Petri.*

Dans ce manuscrit, nous nous intéresserons à des réseaux de *haut niveau*, qui peuvent être vus comme des réseaux colorés. Ils impliquent donc un *domaine de couleurs*. On restera détaché d'un domaine de couleur en particulier pour rester dans le cadre le plus général possible, sauf mention contraire. Les *domaines de couleur* des réseaux haut niveau fournissent des valeurs pour les données, des variables, une syntaxe pour les expressions, des règles de typage, etc. En général un domaine de couleur élaboré est utilisé pour faciliter la modélisation ; par exemple, un langage fonctionnel peut être considéré [103, 148], ou le fragment

fonctionnel (expressions) d'un langage impératif [58, 144]. Ici, on ne s'attache pas spécifiquement à un domaine de couleurs, un domaine de couleurs concret peut être considéré comme implémentation d'un *domaine de couleurs abstrait*.

2.1 Domaine de couleurs abstrait

Le domaine de couleurs abstrait utilisé dans ce manuscrit est composé de :
- un ensemble de *valeurs* de données $\mathbb{D}$ contenant les valeurs booléennes *true* et *false* ;
- un ensemble de *variables* $\mathbb{V}$ disjoint de $\mathbb{D}$;
- un ensemble d'*expressions* $\mathbb{E}$ tel que $\mathbb{D} \cup \mathbb{V} \subseteq \mathbb{E}$, *i.e.*, les données et variables sont donc considérées comme expressions.

On note par $vars(e)$ l'ensemble des variables d'une expression $e \in \mathbb{E}$. On ne fait *aucune supposition* sur le *typage* ou la *syntaxe* des expressions. Cependant, on suppose que chaque expression bien formée peut être évaluée de manière *déterministe* comme une valeur de $\mathbb{D}$, et qu'une expression mal formée est évaluée à la valeur $\perp \notin \mathbb{E}$ (valeur non définie). Pour évaluer les expressions nous devons assigner des valeurs aux variables, on définit par $\beta(v)$ la valeur d'une variable v par l'application de la *valuation* $\beta : \mathbb{V} \nrightarrow \mathbb{D}$ (fonction partielle). On étend naturellement cette notation aux expressions en étendant l'ensemble image des valuations à la valeur $\perp \notin \mathbb{D}$. On note par $\beta(e)$ l'évaluation de l'expression $e \in \mathbb{E}$ par $\beta : vars(e) \to \mathbb{D} \cup \{\perp\}$. On se place dans un cadre optimiste en supposant que toute expression mal formée s'évalue à $\perp$, cela vaut pour les opérations non définies telles que $\frac{x}{0}$ ou syntaxiquement incorrectes.

Un *multiensemble* sur un ensemble X est une application $m : X \to \mathbb{N}$ qui à chaque élément de X associe le nombre d'*occurrences* de cet élément dans le multiensemble. On note les multiensembles comme des ensembles avec répétition d'éléments, par exemple le multiensemble sur $\mathbb{N}$ défini par $\{23 \mapsto 3, 42 \mapsto 2\}$ est noté $\{23, 42, 23, 23, 42\}$ (l'ordre des éléments n'étant pas important). Chaque *ensemble* est un multiensemble particulier où chaque nombre d'occurrences non nul est égal à 1. On utilise donc la notation *ensemble vide* $\emptyset$ pour noter un multiensemble vide.

Étant donné deux multiensembles m et m' sur l'ensemble X, on définit les opérations suivantes :
- la *somme* $+$ définie par $\forall x \in X, (m + m')(x) = m(x) + m'(x)$;
- la *différence* $-$ définie par $\forall x \in X, (m - m')(x) = max(m(x) - m'(x), 0)$.

2.2 Réseaux de haut niveau

Un réseau de Petri est composé de deux parties : une partie *statique* et une partie *dynamique*. La partie statique définit la structure du réseau de Petri : où les données sont stockées et comment les données interagissent entre elles. La partie dynamique définit quel est l'état initial du système. En effet, un même réseau de Petri n'aura pas le même comportement selon son état initial, il est donc important de séparer les deux concepts. De plus, on verra plus tard dans ce manuscrit que la partie statique étant fixée une fois pour toutes pour

le réseau de Petri, on essayera de s'en détacher le plus possible en encodant toutes les informations nécessaires directement dans les algorithmes ; c'est le coeur du travail réalisé par l'étape de compilation.

Definition 2.1. *Un réseau de Petri haut niveau est un tuple (S, T, ℓ) où S est un ensemble fini de places, T (disjoint de S) un ensemble fini de transitions, et ℓ est une fonction d'étiquetage sur $S \cup T \cup (S \times T) \cup (T \times S)$ telle que :*
- *pour tout $s \in S$, $\ell(s) \subseteq \mathbb{D}$ est le type de s,* i.e.*, l'ensemble des valeurs que s peut contenir ;*
- *pour tout $t \in T$, $\ell(t) \in \mathbb{E}$ est la garde de t,* i.e.*, une condition pour son exécution ;*
- *pour tout $(x, y) \in (S \times T) \cup (T \times S)$, $\ell(x, y)$ est un multiensemble sur $\mathbb{E}$ et défini l'arc de x vers y.* ◊

Lorsqu'on manipule un réseau de Petri (S, T, ℓ), il est utile d'identifier les nœuds (places ou transitions) adjacents les uns aux autres par rapport aux arcs. Pour chaque nœud $x \in S \cup T$ on définit son *preset* par ${}^\bullet x \stackrel{\text{df}}{=} \{y \in S \cup T \mid \ell(y, x) \neq \emptyset\}$ et son *poset* par $x^\bullet \stackrel{\text{df}}{=} \{y \in S \cup T \mid \ell(x, y) \neq \emptyset\}$.

Dans ce manuscrit on suppose que tout arc en entrée d'une transition est étiqueté par un multiensemble de variables. On suppose aussi que chaque expression sur un arc sortant d'une transition n'a pas de variables libres. Ce sont des restrictions importantes dans deux sens : ça limite ce qu'on peut faire (pas de transitions non déterministes par exemple), ça facilite grandement les algorithmes (pas d'unification ou résolution d'expressions).

Supposition 2.1. *Soit (S, T, ℓ) un réseau de Petri. Pour tout $t \in T$:*
- *pour tout $s \in {}^\bullet t$, on a $\ell(s, t)$ un multiensemble sur $\mathbb{V}$;*
- *pour tout $s \in t^\bullet$, pour tout $e \in \ell(t, s)$ on a $vars(e) \subseteq \bigcup_{s' \in {}^\bullet t} \ell(t, s')$.* ◊

L'état d'un réseau de Petri est caractérisé par son *marquage*, c'est-à-dire une assignation de jetons aux places.

Definition 2.2. *Un marquage d'un réseau de Petri $N \stackrel{\text{df}}{=} (S, T, \ell)$ est une application qui associe à chaque place $s \in S$ un multiensemble dans $\ell(s)$. L'ensemble de tous les marquages de N est $\mathbb{M}_N$, ou simplement $\mathbb{M}$ en l'absence d'ambiguïté.* ◊

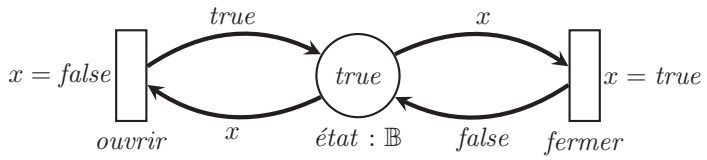
Le passage d'un état du réseau de Petri à un autre est réalisé par une opération de *tir* de transition. Cette opération est locale à une transition et n'est réalisée que sous certaines conditions. Si ces conditions sont satisfaites alors un nouveau marquage du réseau peut être produit en consommant et produisant des jetons.

Definition 2.3. *À partir d'un marquage M d'un réseau de Petri (S, T, ℓ), une transition $t \in T$ peut être tirée en utilisant une valuation β et produire un nouveau marquage M', ce qu'on note $M[t, \beta\rangle M'$, si et seulement si :*
- *il y a assez de jetons dans les places en entrée de la transition : pour tout $s \in S$, $M(s) \geq \beta(\ell(s, t))$;*
- *la garde est satisfaite : $\beta(\ell(t)) = true$*
- *le type des places est respecté : pour tout $s \in S$, $\beta(\ell(t, s))$ est un multiensemble sur $\ell(s)$;*

– *M' est obtenu à partir de M en consommant et en produisant les jetons selon les arcs : pour tout $s \in S$, $M'(s) = M(s) - \beta(\ell(s,t)) + \beta(\ell(t,s))$.*

Une valuation qui respecte les conditions ci-dessus est appelée mode *de la transition t dans le marquage M.* ◊

Un réseau de Petri peut être représenté sous forme de graphe avec les places dessinées sous forme de noeuds circulaires et les transitions sous forme de noeuds rectangulaires. Les arcs entre les noeuds sont étiquetés par les valeur de la fonction d'étiquetage leur correspondantes. Les jetons d'un marquage, quant à eux, sont dessinés dans les places. Un exemple est donné en figure 2.2.



$$(S,T,\ell) \stackrel{\text{df}}{=} \begin{cases} S \stackrel{\text{df}}{=} \{état\}, & T \stackrel{\text{df}}{=} \{ouvrir, fermer\}, \\ \ell(état) \stackrel{\text{df}}{=} \mathbb{B}, & \\ \ell(ouvrir) \stackrel{\text{df}}{=} \{x = false\}, & \ell(fermer) \stackrel{\text{df}}{=} \{x = true\}, \\ \ell(état, ouvrir) \stackrel{\text{df}}{=} \{x\}, & \ell(état, fermer) \stackrel{\text{df}}{=} \{x\}, \\ \ell(ouvrir, état) \stackrel{\text{df}}{=} \{true\}, & \ell(fermer, état) \stackrel{\text{df}}{=} \{false\}, \end{cases}$$

$$M_0 \stackrel{\text{df}}{=} \{état \mapsto \{true\}\}$$

FIGURE 2.2 – *Exemple de réseau de Petri de haut niveau (S,T,ℓ) avec son marquage M_0, sous sa forme graphique et textuelle.*

L'espace d'états d'un réseau de Petri est l'ensemble des états accessibles à partir du marquage initial. Cet ensemble est le plus petit ensemble de marquages contenant le marquage initial et clos par l'opération de tir de transition. Il est utilisé pour tester de propriétés d'accessibilité, par exemple lors de la vérification d'un protocole de sécurité on pourrait chercher un état dans lequel un espion connaît des données confidentielles.

DEFINITION 2.4. *Soit $N \stackrel{\text{df}}{=} (S,T,\ell)$ un réseau de Petri et M son marquage initial. L'ensemble des états de N accessibles à partir de M est le plus petit ensemble $states(N,M)$ tel que :*

– *$M \in states(N,M)$;*
– *s'il existe une transition $t \in T$, une valuation β et un marquage $M \in states(N,M)$ tels que $M[t,\beta\rangle M'$ alors $M' \in states(N,M)$.* ◊

On citera souvent le terme de place *n-bornée*, cela veut dire que la place en question contient au plus n jetons dans tout les états de l'espace d'états. Par exemple, une place 1-bornée ne peut contenir qu'un seul jeton ou aucun.

Souvent, il est aussi intéressant de considérer le graphe d'accessibilité d'un réseau de Petri. Les noeuds de ce graphe sont les marquages accessibles et les arcs sont étiquetés par les transitions qui permettent de passer d'un état à l'autre. Dans certains cas, il est aussi pertinent d'étiqueter les arcs avec les

modes des transition en plus. Le graphe d'accessibilité du réseau de la figure 2.2 est donné en figure 2.3, son espace d'états est l'ensemble de noeuds de ce graphe.

Definition 2.5. *Soit (S, T, ℓ) un réseau de Petri et M son marquage initial. Le graphe d'accessibilité $G = (V, A)$, où $V \subset \mathbb{M}$ et $A \subset \mathbb{M} \times T \times \mathbb{M}$, est le plus petit graphe tel que :*

- *$M \in V$*
- *$M' \in V$ et $(M, t, M') \in A$ s'il existe $M \in V$, une transition $t \in T$ et β un de ses modes tels que $M[t, \beta\rangle M'$.* ◊

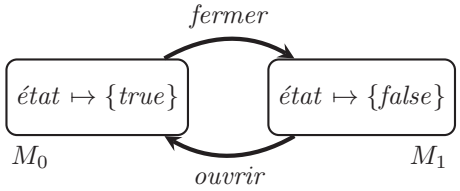


Figure 2.3 – *Graphe d'accessibilité du Réseau de Petri de la figure 2.2.*

2.3 État de l'art sur l'exploration des espaces d'états

Lors de la vérification, l'espace d'états calculé a une taille exponentielle par rapport à la taille du modèle, son calcul peut alors être long et consommer beaucoup de mémoire. Il est donc important d'optimiser ces deux aspects. Ainsi, les notions de base étant définies, nous allons dresser un état de l'art des techniques d'accélération.

Pour calculer les espaces d'états des réseaux de Petri, il y a essentiellement deux approches : l'approche explicite qui stocke et calcule les états individuellement et l'approche symbolique qui stocke et calcule les états par lots. Pour cela l'approche symbolique stocke les états dans des structures de données avancées tels que les diagrammes de décision [29, 32, 33] ainsi que leurs dérivés [45, 79, 30].

L'approche symbolique permet de calculer des espaces d'états de taille bien plus grande que l'approche explicite, cependant elle est aussi limitée. L'expressivité des modèles acceptées est directement liée aux structures de données utilisées (*BDD*, *OBDD*, *DDD*, *etc.*) et leurs extensions sont non triviales. Souvent l'utilisation d'un langage de couleurs riche introduit de l'inefficacité à la détection d'équivalences dans la structure de données et rend difficile le partage qui est la force des diagrammes de décision. C'est pourquoi en général les couleurs acceptées restent proches des réseaux algébriques [158, 50, 112, 113]. Lors de la vérification de formules de logique l'extraction de contre-exemples est difficile. On sait vérifier qu'une formule de logique est satisfaite ou non par un model checker symbolique cependant il est beaucoup plus difficile d'obtenir un contre-exemple, c'est-à-dire un scénario qui rend la formule fausse, il y a cependant des travaux qui adressent ce problème [83, 38, 134, 124]. Un enjeu majeur est de fournir un contre-exemple minimal et intuitif permettant à un ingénieur de le rejouer pour comprendre et corriger les erreurs.

L'approche explicite a aussi des défauts, elle est naturellement plus lente que l'approche symbolique parce qu'elle traite tous les états individuellement de plus elle est beaucoup plus sujette au problème d'explosion d'espaces d'états. La récupération de contre-exemples lors de vérification de formules logiques est quant à elle triviale en général. Enfin, les contre-exemples sont intuitifs pour les ingénieurs non spécialistes en model checking et permettent de facilement déboguer des modèles ou programmes en général. De plus, un langage de couleurs riche est moins problématique qu'en model checking symbolique.

Dans ce manuscrit on s'intéresse au model checking explicite de par le souhait d'utiliser un langage de couleur très expressif. L'utilisation de langages riches permet de modéliser des systèmes plus aisément mais aussi de réduire la taille des espaces d'états [103]. En effet chaque état aura tendance à occuper plus de mémoire mais le nombre des états accessibles sera moindre. L'utilisation de couleurs riches interdit aussi l'utilisation en général toute transformation vers les réseaux bas niveau en raison d'une explosion exponentielle du nombre de places et transitions [24]. Il faut donc les traiter directement tels quels avec des algorithmes et des techniques appropriés.

2.3.1 Compilation de réseaux de Petri

La compilation de réseaux de Petri est utilisée dans le cadre du model checking explicite pour accélérer le tirage de transitions. Elle est utilisée dans les outils efficaces tels que Helena [57], Spin [87] ou encore Maria [132].

Cette approche peut être appliqué aussi bien aux réseaux de Petri bas niveau qu'aux réseaux de haut niveau. Dans [42] où les auteurs s'attaquent à l'implémentation de réseaux de Petri bas niveau, l'approche proposée n'est pas de la compilation cependant les idées restent très proches. Il s'agit de proposer une implémentation de réseaux bas niveau presque 1-bornées, les transitions sont supposées 1-tirables dans tous les cas, *i.e.*, exactement un seul jeton en entrée est nécessaire pour tirer une transition. Une implémentation de réseaux haut niveau est aussi considérée mais non utilisable en pratique puisqu'elle correspond en une expansion vers un réseau de Petri de bas niveau.

L'étape de compilation consiste à produire un moteur d'exploration du réseau de Petri qui lui est spécifique. C'est-à-dire que le programme qu'on utilise sur le modèle du réseau de Petri n'est pas le model checker mais un compilateur qui produit un model checker en sortie. Cette approche est souvent mal comprise car confondue avec des réductions statiques qui transforment le modèle en un modèle plus simple suffisant pour vérifier le but recherché.

L'utilisation d'une étape de compilation est importante pour des modèles colorés à cause des annotations riches apparaissant sur les arcs. Sans compilation, un évaluateur d'expressions est utilisé lors du tir de transitions, cet évaluateur est écrit dans un langage efficace mais le calcul de l'expression est lent puisque interprété. La compilation a pour but d'encoder ces expressions de manière native pour permettre un tirage beaucoup plus rapide. Le gain de performances par compilation d'annotations a d'ores et déjà été montré dans le cadre du model checker Maria [126, 132] et aussi appliquée dans le model checker Helena [57, 58]. Dans Helena d'autres optimisations sont utilisés lors de l'exploration pour accélérer le tirage des transitions [56].

Dans ce manuscrit (chap. 3) on développe une approche par compilation en s'attaquant en plus à la structure du réseau. C'est-à-dire qu'en plus de compiler

les annotations d'arc on compile la fonction de transition pour éliminer toute requête à une représentation quelconque du réseau de Petri, seule la structure de marquage reste utilisée.

2.3.2 Réductions d'ordres partiels

Les réductions d'ordre partiels ont pour but de réduire le non déterminisme du tir de transitions. Pour cela, certaines transitions vont être privilégiées et tirées avant toutes les autres, ou, selon le point de vue, le tir de certaines transitions sera retardé. On peut ainsi parler, par exemple, de transitions prioritaires qui doivent être tirées avant toutes les autres [161]. On pourrait critiquer le fait qu'il s'agit d'une réduction réalisée sur la structure du modèle qui nécessite une bonne connaissance du système modélisé et non pas une approche générale. Elle peut donc cacher des comportements non désirés à cause d'une erreur de choix dans les priorités sur le modèle. Cependant, dans certaines classes de modèles, comme par exemple les protocoles de sécurité le choix des priorités des transitions peut être fait automatiquement [65].

Une autre famille de réductions d'ordre partiels [70, 72, 162, 138, 139] similaires aux *stubborn sets* [153, 154, 156], est basée sur l'observation que des transitions concurrentes sont indépendantes de l'ordre dans lequel elles sont tirées, tous les entrelacements ne sont pas toujours pertinents, des exécutions peuvent donc être omises. Pour la détection de *deadlocks* (état avec aucune transition tirable) par exemple, on peut retarder les transitions le plus longtemps possible sans entraver la correction du résultat de l'analyse.

Initialement, le calcul des stubborn sets d'un réseau de Petri de haut niveau nécessite un dépliage en réseau de bas niveau [154], de plus le calcul le calcul d'un *bon* stubborn set peut être aussi coûteux que le dépliage du réseau complet [117]. Une approche sans dépliage a été donc proposé mais pour une sous classe de réseaux de Petri colorés (*Process-Partitioned CP-Nets*) qui produisent un *bon* stubborn set sans énumération explicite de modes [117] (*i.e.*, sans dépliage explicite ou implicite). Un second algorithme symbolique de calcul de stubborn sets (*i.e.*, sans énumération des modes) a été proposé [62] dans le cadre du model checker Helena [57] pour une autre sous classe de réseaux de Petri colorés. Cependant, à cause du résultat de complexité du calcul des stubborn sets, une approche efficace ne pourra jamais traiter qu'une sous classe de réseaux. Une autre remarque importante est que certaines de ces techniques de réduction ne sont pas compatibles avec tous les algorithmes d'exploration d'espaces d'états [92].

Un ajout intéressant au techniques de réduction par ordres partiels [139] est de pouvoir les appliquer à la volée, c'est-à-dire lors de la vérification d'une formule logique [80]. L'avantage est de pouvoir ignorer une partie de l'espace d'états non pertinente du point de vue de la propriété vérifiée et ainsi avoir une réduction plus importante. Typiquement les approches à la volée ne sont pas applicables aux méthodes symboliques qui doivent d'abord calculer l'ensemble des états accessibles.

Les approches présentées dans ce manuscrit sont compatibles avec les techniques d'ordres partiels : les ordres statiques sur les transitions sont très faciles à mettre en oeuvre, les techniques dynamiques quant à elles sont moins évidentes à mettre en place. La raison de cette difficulté est notre langage de couleur qui est riche et comme cité ci-dessus des techniques comme les stubborn sets sont

difficiles à appliquer sans déplier le réseau. Il est cependant possible de les appliquer de manière symbolique mais avec des contraintes sur les couleurs utilisées, ce qui sort du cadre de cette thèse.

2.3.3 Occupation mémoire

L'occupation mémoire est un des problèmes majeurs du model checking explicite. À l'inverse du model checking symbolique où les différents états partagent des données grâce aux diagrammes de décision, une approche par table de hachage est généralement utilisée en model checking explicite. Des expériences pour utiliser des diagrammes de décision pour stocker des états lors de l'exploration explicite ont été réalisées [160] ; en particulier l'investigation porte sur les OBBD [29, 111], la combinaison des OBBD avec d'autres approches, notamment *Collapse* [67, 160, 88] et les ordres partiels [73, 91, 139, 155], est aussi traitée.

Les résultats des expériences furent décevants, en effet même si en général on a une très bonne compression de l'espace d'états le surcoût de l'exploration est très important. La technique *Collapse* [67] et sa révision *Recursive Indexing* [88] sont des techniques de compression des états par partage de mémoire. Collapse part de l'observation que l'explosion de l'espace d'états est rarement engendrée par un seul processus, en général ce sont des processus concurrents qui engendrent l'essentiel de la combinatoire. L'idée est donc de diviser le marquage en processus stockés indépendamment dans des tables de hachage. L'état du réseau sera un vecteur de bits composé de références vers ces états, plus des places partagées. L'apport du *Recursive Indexing* est d'utiliser des vecteurs de bits de tailles variable en spécifiant la taille de chaque champ dynamiquement, la motivation est d'autoriser des tailles de tables hachages de taille arbitraire pour les différents processus et économiser de l'espace si un processus a peu d'états.

Une approche pour stocker l'espace d'états sous forme de graphe (*GETS : Graphe Encoded Tuple Sets*) a été proposée dans [76], elle présentait cependant des résultats similaires à l'expérience avec les OBBD : une compression importante de l'espace d'états mais un surcoût important en temps de calcul. En 1999, une autre tentative de stocker les espaces d'états sous forme de graphe a été réalisée [93]. Cette fois le point de vue était différent : le graphe en question était un automate fini déterministe (DFA) minimal responsable de reconnaître les états présents dans l'espace d'états. Les performances en temps et espace étaient comparables à GETS, mais lorsque la technique était alliée à Collapse, la consommation mémoire pouvait être encore réduite (ce n'était pas le cas en général).

Des tests de ces différentes techniques ont été réalisés dans le cadre du model checker *Spin*. Sur une moyenne de 14 applications [93], on peut observer que si on utilise la technique Collapse la consommation de mémoire est divisée par 3 alors que les temps d'exécution ne sont multipliés que par 1.3 ; lors de l'utilisation de DFA avec Collapse la consommation mémoire était divisée par 5.5 mais les temps d'exécution multipliés par 6.2 ; lors de l'utilisation de DFA seule l'utilisation mémoire était divisée par 7.2 et les temps d'exécution multipliés par 4.9 ; lors d'utilisation de GETS l'utilisation mémoire était réduite de 7.3 et les temps d'exécution multipliés par 5.2. Les OBBD n'ont pas été comparées pendant ces tests parce qu'ils était plus lents d'un ordre de gran-

deur. On peut donc voir que même si des approches de type symbolique pour le stockage d'espaces d'états ont été testées c'est l'approche purement explicite qui s'en sort le mieux (Collapse).

Une autre approche semi-explicite appelée Δ-marquages [61] a été introduite dans le cadre du model checker Helena. Elle consiste à ne stocker qu'une seule partie de l'espace d'états et à reconstruire certains états au besoin, ces autres états sont donc stockés de manière symbolique. Ils sont appelés Δ-marquages, et possèdent une référence vers leur prédécesseur et une instance de la transition tirée (plus précisément une transition et son mode). Ainsi le marquage pourra être reconstruit en tirant la séquence de transitions sur les arcs depuis le premier prédécesseur explicite. Cette technique est compatible avec Collapse et des résultats expérimentaux sont présentées dans [61, 58], on y remarque que les Δ-marquages permettent de diviser jusqu'à 6.5 fois l'occupation mémoire de Collapse tout en gardant des temps d'exécution raisonnables. De plus ces deux techniques sont compatibles avec les réductions par symétries et réductions d'ordres partiels.

D'autres techniques comme celles du model checker MARIA [127, 132] se focalisent sur les représentations de multiensembles et des marquages, pour les représenter de manière compacte en exploitant la taille des domaines des types de jetons. Cependant cette approche ne réussit pas à traiter efficacement les espaces d'états avec des jetons de domaines de couleur riches [57].

Certaines approches peuvent aussi être vues comme des optimisations de mémoire. Le *bit state hashing* utilisé dans le model checker Spin [85, 89], consiste à ne stocker qu'un bit par marquage, plus précisément allouer un segment mémoire d'une grande taille, puis pour chaque marquage M découvert calculer son hachage et marquer le bit de valeur de ce hachage modulo la taille du segment. Il s'agit d'un filtre de Bloom qui supprime les doublons en se basant sur la valeur de hachage. Cette approche n'est pas complète, certains marquages peuvent être omis, typiquement on considère un marquage déjà visité lorsque le bit correspondant à son hachage est marqué. Ce genre d'approximation peut être pertinent dans la perspective de détection d'erreurs mais ne peut pas garantir leur absence. Dans ce cadre, des techniques pour réduire la probabilité de collisions ont aussi été considérées [163, 89].

On peut aussi parler des techniques qui se basent sur l'observation que les espaces d'états peuvent être explorés sans stocker tous les états en mémoire [86, 100]. Le nombre d'états nécessaires en mémoire est celui du plus long chemin acyclique dans l'espace d'états. Une amélioration de cette approche consiste à garder une cache d'états visités en mémoire afin de détecter des cycles entre différentes traces et éviter des calculs redondants, cette stratégie s'appelle *state space caching* [84]. Elle fut par la suite améliorée [71] en exploitant des techniques d'ordres partiels [70, 73] notamment les *sleep sets* qui permettent de ne pas tirer certaines transitions tout en préservant les propriétés d'accessibilité. Des techniques similaires ont été proposées par la suite tels que la méthode *sweep-line* qui utilise un *garbage collector* pour supprimer des états de la mémoire lorsqu'ils ne sont plus nécessaires [36]. Une approche similaire au *sweep-line* a été utilisée pour réaliser du model checking CTL parallèle de protocoles de sécurité [66] en exploitant les propriétés spécifiques à ces modèles, on remarque que certaines états n'ont pas besoin d'être partagés, de plus une fois certaines transitions tirées tous les états précédents peuvent être oubliés et donc supprimés de la mémoire.

Ces différentes techniques explicites de réduction d'occupation mémoire sont compatibles avec les techniques d'exploration et de réduction de ce manuscrit. En effet, on n'a que peu de contraintes sur les représentations d'états et on pourrait aisément les implémenter.

2.3.4 Réductions par symétries

Les réductions par symétries ont été largement étudiées [108]. Ce type de réductions s'applique aussi bien aux réseaux place/transition [150, 151, 109, 108] qu'aux réseaux de haut niveau [97, 96, 105, 107, 108]. Intuitivement il s'agit de détecter des comportement équivalents dans le système qui ne diffèrent que par une permutation de places, de transitions ou encore de données. Le but est de ne stocker qu'un état par classe d'équivalence d'états (*i.e.*, états symétriques), mais évidemment sans construire l'espace d'états complet. C'est réalisé en définissant un groupe sur le système considéré dont l'action sur l'espace d'états sera de produire des symétries [108]. Les réductions par symétries sont activement étudiées car elles peuvent offrir une réduction exponentielle de la taille de l'espace d'états [54]. Tester si deux états sont dans la relation d'équivalence induite par le groupe est généralement coûteux, il s'agit du problème d'orbite qui est prouvé être au moins aussi difficile que le problème d'isomorphisme de graphes [40], cependant on ne sait pas s'il est NP-complet. Pour que l'approche soit utilisable, il est courant de rechercher des représentants canoniques de chaque classe d'équivalence en utilisant des heuristiques [27, 26, 25], même si la réduction produite n'est pas maximale, lorsque l'exécution du test est suffisamment rapide alors on peut gagner à la fois en espace mais aussi en temps d'exécution.

De manière classique les réductions par symétries considèrent des automorphismes internes à l'espace d'états qui préservent la relation de transition [39, 53]. Cependant, l'usage d'automorphismes n'est pas suffisant pour détecter des symétries dans des systèmes avec un aspect de calcul dynamique caractérisé avec création de *nouveaux* objets [99]. Il peut s'agir de nouveaux composantes qui sont infiniment souvent crées le long d'un chemin d'exécution tels que la création de nouveaux processus ou objets. Si un seul groupe de permutations est considéré comme le groupe d'automorphismes du système alors on ne pourra pas détecter les symétries entre des successeurs qui créent de nouveaux composants [99, 98]. Pour traiter ce genre de systèmes il faut considérer une famille de groupes et les choisir de manière dynamique selon le nombre de composants dans l'état du système. Une approche de ce type est donnée dans [99] pour traiter les symétries de tas et de processus dans le cadre de vérification de programmes.

Dans ce manuscrit, on s'attaque à un problème plus général que les symétries de processus en y ajoutant la question de la préservation de relations entre processus. En effet, dans [99] on suppose qu'il n'y a pas de références entre les processus. Une séries de travaux a été réalisée sur ce sujet [114, 115, 116] en fournissant une base théorique pour la détection de symétries de ce type. Cependant aucun algorithme efficace de test d'orbite n'a été proposé. L'algorithme proposé consiste en une transformation de marquage en graphe puis un test d'isomorphisme entre ce graphe et les graphes correspondants à tous les états précédemment visités. Le problème principal qui empêche l'algorithme d'être efficace en pratique est cette comparaison avec tous les états précédemment

visités, ce qui ne peut pas être accéléré par des techniques de tables de hachage utilisés habituellement en model checking. De plus le calcul d'isomorphisme de graphe est un problème difficile. Il a cependant été montré sur des études de cas dans [116] que le temps de calcul d'isomorphisme pour ce type de graphe demeure raisonnable.

Dans le chapitre 6 on propose une *nouvelle méthode* complète de détection de symétries ne souffrant pas de ce problème, de plus on propose aussi une seconde méthode optimiste suffisante dans certains cas.

2.3.5 Autres techniques de réduction

D'autres techniques de réduction existent, telles que les réductions statiques. Ces réductions sont réalisées avant l'exploration de l'espace d'états pendant une étape de pré-traitement. Il s'agit essentiellement de réductions sur les domaines des variables ou des données du modèle, ou de fusions de transitions. On peut par exemple citer les agglomération de transitions qui fusionnent plusieurs transitions en une transitions atomiques [59, 77, 60, 78, 18, 19]. Certaines des réductions statiques peuvent cependant être vues comme des implémentation de réductions d'ordre partiels [87].

Ces techniques peuvent aisément être appliquées à notre approche par compilation, il suffit de les appliquer avant de compiler le réseau. Cependant dans certains cas il faudrait restreindre le langage de couleurs.

2.4 Problématique de certification de model checkers

Une question complémentaire à celle de l'efficacité du model checking est celle de sa correction. En effet, on peut se demander quelle confiance avoir en un model checker ? En pratique, dans le domaine il est rare de prouver la correction d'un model checker, la correction est souvent déterminée par comparaison avec d'autres outils [KLB+12, KLB+13]. Ce qui a d'ailleurs provoqué un problème lors d'évaluation de formules CTL dans [KLB+13] puisqu'on ne pouvait pas déterminer quel outil fournissait les bons résultats lorsque tous les outils étaient en désaccord. L'évaluation a donc porté sur le temps de réponse aux problèmes, la quantité de réponses, et non la correction des résultats fournis.

Cette discussion n'est pas nouvelle. Lorsqu'on fournit un nouvel algorithme de model checking, il est de pratique courante de fournir une preuve de correction de celui-ci, mais aussi d'en montrer l'efficacité pratique. Cependant il est rare de voir une preuve quelconque de l'étape d'implantation dans un outil, de plus les étapes expérimentales sont aussi souvent remises en question [90].

Dans [16] il est argumenté que les outils de vérification n'ont pas besoin d'être vérifiés car même s'ils le sont on n'aura jamais une garantie qu'ils sont totalement corrects, essentiellement à cause de l'intégration de ces outils dans un environnement qui lui n'est pas certifié correct. Cependant, des preuves même partielles ne sont pas à proscrire, en effet elles permettent de nous persuader que nos hypothèses sont correctes et ainsi avoir confiance en nos algorithmes et leur impact. De plus, si la certification de l'outil entrave ses performances, son utilisation ne sera pas pertinente en pratique. Il est donc conseillé de privilégier les tests.

La certification est utile dans le cas de non réfutation de propriétés lors du model checking. En effet, lorsqu'on teste une propriété qui est fausse sur un modèle alors on obtient une trace d'exécution qui mène à sa réfutation. Cette trace peut être rejouée pour vérifier si la formule est effectivement fausse. Lorsqu'une formule est vraie alors le model checker ne fournit pas de trace, il répond juste que la formule est satisfaite. On ne peut pas vérifier si l'outil se trompe ou pas, c'est le type de cas où une certification apporterait une certaine garantie. Dans la même lignée de pensée que [16], ne pas certifier un model checker peut avoir des avantages notamment pour les performances. Pour ne pas les entraver et avoir une garantie dans le cas de formules vraies, [133] propose de produire un certificat de validité de formule lors du model checking. L'avantage d'un certificat produit par l'outil est que l'outil n'a pas besoin d'être lui même certifié. Le certificat peut être traité par un assistant de preuves pour tester sa véracité. Cependant la création d'un tel certificat peut être coûteuse et sa vérification encore plus. Ce type d'approche est souvent utilisé dans les assistants de preuves tels que coq [10] pour accélérer la recherche automatique de preuves. Ces méthodes appelées *réflexives* [20, 75] se basent sur un noyau externe non certifié et un théorème de validité dans le prouveur qui permet d'interpréter le résultat de ce noyau, c'est-à-dire un programme qui va réaliser une phase de recherche de preuve puis renvoyer un résultat (une preuve) qui va être vérifié par le prouveur.

Il ne faut pas oublier que la certification formelle est une tâche difficile qui prend beaucoup de temps. Certifier un model checker efficace peut s'avérer impossible à cause de diverses optimisations, sans compter les optimisations réalisées par le compilateur. Ce qui est plus abordable c'est montrer la correction des techniques qui sont utilisées. Par exemple, dans [35] on prouve la correction (partielle) d'une réduction par ordre partiel en utilisant l'assistant de preuves HOL [11], cette preuve à elle seule prend presque 7 500 lignes de code. Sachant qu'un model checker est un logiciel beaucoup plus grand on peut se demander s'il est abordable de le prouver, d'autant plus que tout changement dans le programme invalide la preuve en général et souvent de manière drastique.

Dans ce manuscrit on s'intéresse à la problématique de certification au niveau de la correction des algorithmes générés. C'est-à-dire qu'on montre comment prouver que les programmes qu'on génère lors de la compilation sont corrects. Pour une certification complète, il faudrait prouver la correction du compilateur, ce qui est réalisable mais sort du cadre de la thèse.

2.5 Autres notions

2.5.1 Formalismes de modélisation structurés

Dans ce manuscrit on citera souvent des *formalismes de modélisation structurés*, par cela on entend essentiellement les formalismes comme les *algèbres de réseaux de Petri* [22] et leur dérivés [21, 23, 31, 48]. Il s'agit d'algèbres de processus [129, 82] à sémantiques dans les réseaux de Petri [157, 74, 131]. Cette famille d'algèbres a essentiellement les opérateurs suivants :

– la séquence $E_1; E_2$: l'exécution de E_1 suivie de E_2 ;

- le choix $E_1 \square E_2$: l'exécution de E_1 ou bien de E_2 ;
- la composition parallèle $E_1 || E_2$: E_1 et E_2 sont exécutés de manière concurrente ;
- l'itération $E_1 \circledast E_2$: E_1 est exécuté un nombre arbitraire de fois (possiblement nul) suivi d'une fois E_2.
- d'autres opérateurs selon la variante utilisée.

Chacun de ces opérateurs opère sur des réseaux de Petri avec comme brique de base une transition, ainsi si $E \stackrel{\text{df}}{=} t$ le réseau correspondant sera t avec une unique place en entrée et une unique place en sortie de type jeton-noir. L'application d'un opérateur sur deux réseaux de Petri produira un réseau plus grand en composant les deux, le tout réalisé par une opération de substitution de transitions ou des réseaux opérateurs [49, 22].

L'avantage de ce type de formalisation du point de vue d'implémentation est que les réseaux ainsi obtenus exhibent des propriétés intéressantes par construction qui peuvent être utilisées pour compiler le réseau de manière plus efficace. Par exemple, ils introduisent un contrôle de flot qui est 1-borné et formé d'ensembles de places exclusives, dans le chapitre 3 on propose une optimisation qui exploite cette propriété. Ces propriétés sont obtenues par existence d'invariants dans le réseau [46, 47, 22]

2.5.2 Propriétés d'accessibilités et logiques temporelles

On citera des propriétés d'accessibilité qui peuvent être vérifiées lors du model checking, il s'agit essentiellement de savoir si un état est accessible ou bien si un état tel qu'*une certaine propriété* est vraie est accessible. On pourra par exemple se demander s'il existe un état tel qu'une transition est tirable ou non, ou si un état est un *deadlock* (aucune transition tirable) ou plus généralement s'il existe un état avec un marquage particulier.

On parlera aussi de logiques temporelles mais on ne traitera jamais celles-ci directement ; ce n'est pas notre objectif. On citera essentiellement la logique LTL [142, 69, 80, 65] qui est utilisé par la bibliothèque Spot [52, 51]. Ces logiques sont appelées modales de par l'ajout de modalités permettant de spécifier des conditions sur le temps. Ainsi, la vérité d'une propriété dépendra des évènements considérés dans d'une exécution d'un système et non à un instant particulier.

2.6 Conclusion

Dans ce chapitre nous avons vu les notions de base sur les réseaux de Petri et en particulier les variantes qui seront utilisées par la suite. Ensuite, on a vu un panorama des techniques utilisées pour le model checking explicite avec quelques liens vers le model checking symbolique. Parmi toutes ces techniques nous allons nous intéresser essentiellement à la compilation et aux réductions par symétries. Cependant, il est intéressant de voir les diverses techniques pour comprendre comment elles peuvent interagir entre elles. Ainsi, les techniques présentées dans ce manuscrit, sont compatibles avec les techniques existantes même si cela n'est pas explicité. Les techniques présentés dans les chapitres 3 et 6, en majorité, sont compatibles avec les différentes techniques d'ordres partiels,

les techniques de réduction de mémoire, et autres réductions par symétries. En particulier les techniques de réduction de mémoire et techniques d'ordres partiels sont compatibles avec les techniques du chapitre 6 car la méthode de détection de symétries présentée n'influe pas sur la représentation des marquages. Pour l'approche présentée dans le chapitre 3 une adaptation des algorithmes doit être réalisée mais les primitives présentées ne subiraient que peu de changements. Ces approches sont aussi compatibles avec la vérification de propriétés d'accessibilité et propriétés temporelles.

3 Compilation de réseaux de Petri

Lors de l'exploration explicite d'espaces d'états deux principaux obstacles se dressent devant nous :

- *la vitesse d'exploration* qui est liée au tirage de transitions, aux comparaisons entre états, et aux structures de données ;
- *le nombre d'états* qui reflète l'occupation mémoire utilisée.

Dans ce chapitre on aborde le premier problème dans le cadre de réseaux de Petri de haut niveau.

Dans l'approche *classique* du calcul explicite des espaces d'états de réseaux de Petri, le modèle est représenté sous la forme d'une structure de données que l'outil d'exploration manipule constamment. Cette structure de données est interrogée afin de réaliser les calculs nécessaires au tir. À chaque tirage de transition, on requête cette structure pour récupérer les places d'entrée et de sortie, les annotations d'arcs, etc., tous ces calculs sont *redondants* puisque réalisés à chaque fois que la transition est tirée. Ces données sont aussi utilisées pour récupérer les jetons du marquage, lui aussi représenté d'une manière générique. La conséquence immédiate de ce type d'exploration est donc l'*encapsulation* des expressions (ou plus généralement des annotations) dans des structures de données. Ainsi, cette approche peut être vue comme une *interprétation* du modèle. Son principal avantage est la réalisation d'un moteur d'exploration qui est *générique*, c'est-à-dire une implémentation unique pour tous les modèles, ce qui est plus facile à réaliser et maintenir. L'inconvénient est le *temps d'exécution*, nous devons requêter les structures de données, évaluer les expressions, interpréter une structure de marquage générique, *etc.*

Dans ce chapitre on propose une approche par *compilation* qui évite les différentes étapes d'interprétation dans l'objectif d'accélérer le calcul des espaces d'états. Les questions relatives à la correction de l'approche sont traités dans le chapitre 4 dans le cadre d'une implémentation spécifique.

Ce chapitre est structuré comme suit : en premier on présente l'approche par compilation ; ensuite, on décrit l'objet produit et sa réalisation ; on présente des optimisations simples qui en découlent ; on donne un exemple ; finalement on présente une optimisation applicable sur des modèles définis avec des formalismes structurés.

Ce chapitre est basé sur [FP11a], la section 3.5 n'a pas été publiée.

3.1

Approche par compilation

La *compilation* de modèles est une étape qui permet d'accélérer le model checking explicite de réseaux de Petri. Son objectif n'est pas de réaliser le calcul de l'espace d'états mais de générer un moteur d'exploration *spécifique* au réseau de Petri. Par la suite, ce moteur est utilisé pour effectivement calculer l'espace d'états. L'aspect spécifique de ce moteur est important puisqu'il s'agit de *spécialiser* les *structures de données* et les *algorithmes* en se basant sur la structure du modèle. Ainsi, on n'a plus un moteur d'exploration général, comme dans le cas classique, mais un moteur d'exploration *optimisé* pour le modèle traité.

L'approche par compilation est déjà utilisée dans les *meilleurs outils*. L'outil *Helena* [58, 136] compile vers le langage `C` pour exécuter rapidement les annotations du réseau et représenter efficacement les états ; une approche similaire est aussi utilisée dans le model checker *Spin* [94]. Il a d'ores et déjà été montré que l'approche par compilation permet de *grandement accélérer* le calcul d'espaces d'états grâce à la production d'un moteur d'exploration performant [126] ; cette analyse a été faite dans le cadre du model checker *Maria* [132], lui aussi utilisant la compilation de modèles pour accélérer l'évaluation des annotations du réseau. Il faut noter que l'étape de compilation introduit un surcoût en amont de l'exploration, cependant si l'espace d'états est suffisamment grand alors ce surcoût est négligeable.

Le moteur d'exploration généré est une *bibliothèque logicielle* qui contient un ensemble de primitives permettant le calcul de l'espace d'états. Cette bibliothèque ne manipule pas le réseau de Petri, contrairement à l'approche classique, mais l'encode dans sa structure. Cela découle de l'observation que le réseau de Petri est non mutable, c'est-à-dire qu'il ne change pas durant tout le calcul de l'espace d'états, et donc qu'on peut distribuer ses données dans le code source du moteur d'exploration. Ainsi, l'utilisation d'un ensemble de primitives qui ne manipulent que la structure de marquage est cohérent.

On observe que lorsqu'on souhaite tirer une transition, on doit récupérer les jetons de ses places d'entrée. Or, cet ensemble de places est connu à l'avance de manière *statique*. Il en est de même pour les places de sortie nécessaires à la production de jetons, on peut donc déterminer quelles places extraire du marquage à l'avance sans interroger le modèle. Les expressions qui annotent le réseau, telles que les gardes de transitions et les arcs sont aussi connues, on peut les exploiter sans devoir requêter le réseau de Petri. Plus important, on peut encoder ces expressions de manière *native* dans le langage cible et ainsi exploiter toute son efficacité. Au final, à aucun moment nous n'avons besoin d'accéder à la structure du réseau de Petri, on n'accède qu'à la structure de marquage et on peut déterminer à l'avance quelles opérations réaliser en nous basant seulement sur la structure du réseau et son marquage initial.

Cette approche peut encore être améliorée. En effet, en exploitant les particularités des modèles : on peut *optimiser* le code généré. Dans la section 3.3, on verra par exemple que les places 1-bornées peuvent être exploitées à la fois pour réduire le coût mémoire mais aussi accélérer le tir. Le point intéressant dans ces particularités est que si on utilise des *techniques adéquates pour la modélisation* [144], alors elles peuvent être obtenues *par construction*. Ainsi,

on évite une analyse potentiellement coûteuse du réseau avant son exploration. Par exemple, la connaissance de bornes de certaines places peut nécessiter le calcul de tout l'espace d'états.

3.2

Structure de la bibliothèque

La traduction efficace d'un réseau de Petri vers une bibliothèque logicielle nécessite des suppositions à propos des annotations du réseau. On suppose tout d'abord qu'on considère des réseaux de Petri dont les arcs en entrée des transition sont étiquetés par une seule variable (*multiensembles singletons* contenant une seule variable). Cette supposition permet de simplifier la formalisation, et une extension aux multiensembles de variables demeure simple : il suffit de s'assurer qu'on ne consomme pas deux fois le même jeton d'une place.

SUPPOSITION 3.1. *Soit (S, T, ℓ) un réseau de Petri. Pour tout $t \in T$ et pour tout $s \in {}^\bullet t$ on a $\ell(s, t) = \{x\}$ avec $x \in \mathbb{V}$.* ◊

Notre seconde supposition concerne l'identité de ces variables, on supposera qu'elles sont toutes distinctes. L'utilisation d'une même variable sur un arc en entrée permet de réaliser de la détection de motifs (*pattern matching*) sur les jetons. En effet si deux arcs en entrée d'une transition sont étiquetés par x alors les 2 jetons correspondants doivent être égaux. Cette supposition n'enlève rien à l'expressivité du modèle, en effet on peut aisément simuler ce comportement en ajoutant un test d'égalité sur les variables dans la garde de la transition. Un exemple est donné en figure 3.1.

SUPPOSITION 3.2. *Soit (S, T, ℓ) un réseau de Petri. Pour tout $t \in T$ et pour tout $s, s' \in {}^\bullet t$, si $s \neq s'$ alors $\ell(s, t) \neq \ell(s', t)$.* ◊

Il faut noter cependant, que l'outil présentée dans le chapitre 7 n'a pas ces restrictions et implémente leurs généralisations.

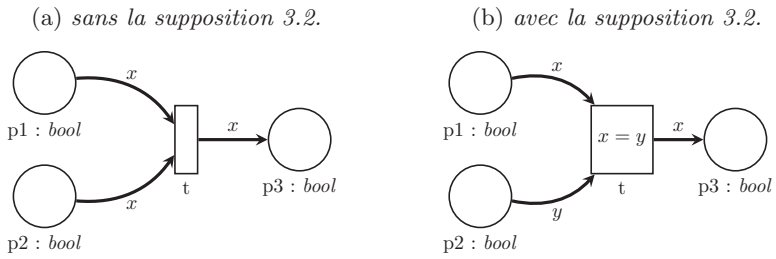


FIGURE 3.1 – *Exemple de pattern-matching de jeton sans et avec la supposition 3.2.*

Parce que le réseau de Petri est compilé vers un langage de programmation cible, on doit aussi supposer que les annotations du réseau sont compatibles avec ce langage. Pour cela, il faut que le langage définisse la notion de type (de manière dynamique ou statique) et la notion de fonction (avec arguments).

Definition 3.1. *Un réseau de Petri (S, T, ℓ) est compatible avec un langage cible si et seulement si :*

- *pour toute place $s \in S$, $\ell(s)$ est un type du langage cible, et implémente un sous-ensemble de $\mathbb{D}$;*
- *pour toute transition $t \in T$, $\ell(t)$ est un appel de fonction booléenne dont les arguments sont des éléments de $vars(t)$;*
- *pour tout $s \in t^\bullet$, $\ell(t, s)$ peut être remplacée par une fonction $f_{t,s}$ du langage dont les arguments sont des éléments de $vars(t)$ et qui renvoie un multiensemble sur $\ell(s)$,* i.e.*, est équivalent à une simple instruction "**renvoyer** $\ell(t, s)$" ;*
- *toutes les fonctions qui annotent le réseau terminent.* ◊

La définition 3.1 généralise les concepts habituels. Par exemple, la garde est une expression booléenne dont les variables libres correspondent à des jetons sur les places d'entrée, elle peut aisément être remplacée par une fonction dont les arguments sont ces variables libres. Il en est de même pour les expressions sur les arcs sortants.

Notre objectif est de calculer l'ensemble des états accessibles à partir de l'état initial du modèle. Pour cela, on compile le réseau de Petri en une bibliothèque logicielle. Ce procédé de compilation vise à supprimer toute interaction avec une structure de données représentant le réseau de Petri pendant les calcul de nouveaux états. Dans notre cas pour atteindre cet objectif, on fournit des primitives de calcul qui ne manipulent qu'une structure de marquage représentant un état du réseau.

La bibliothèque, une fois générée, sera utilisée par un *programme client* pour calculer l'espace d'états. Ce programme client peut être, par exemple, un model checker ou un simulateur. Pour garantir une interopérabilité avec ces outils, elle doit respecter une *API* (*Application Programming Interface*) stricte qui garanti un interfaçage correct. De plus, cette bibliothèque repose sur un ensemble de *primitives préexistantes* (telles que des implémentations d'ensembles et de multiensembles) mais aussi du code extrait directement du modèle, par exemple les gardes de transitions. Cela est présenté sur la figure 3.2.

Cette bibliothèque est formée de deux parties principales, des *structures de données* (une structure de marquage et des structure auxiliaires), et des *fonctions pour l'exploration* d'espaces d'états. La structure de marquage est générée de manière efficace soit en réutilisant des composants génériques, soit en réalisant ces composants directement à la compilation. Pour que la réutilisation soit possible, on a défini un ensemble d'interfaces pour les composants de manière à ce qu'ils s'ajustent facilement. Les fonctions générés sont essentiellement :

- une fonction de *tirage par transition*, elle renvoie le marquage résultant du tir d'une transition à partir d'un marquage donné et d'un mode ;
- une fonction *successeur par transition*, elle renvoie tous les successeurs d'un marquage donné par une transition spécifique ;
- une fonction *successeur globale* qui renvoie l'ensemble des successeurs d'un marquage ;
- une fonction qui renvoie le *marquage initial*.

La fonction de marquage initial est triviale a réaliser et donc non présentée. Cependant on donne un exemple d'implémentation dans un cas particulier en section 3.4.

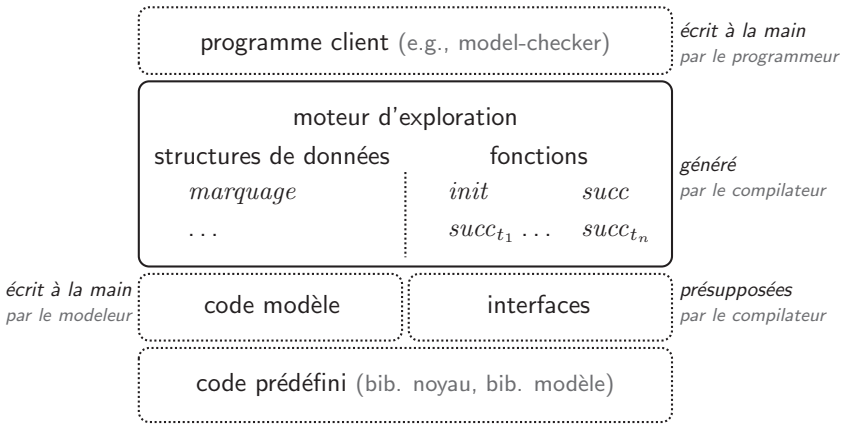


FIGURE 3.2 – *Structure de la bibliothèque.*

3.2.1 Tirage des transitions

Pour chaque transition $t \in T$ du réseau de Petri, on réalise une fonction de tirage de transition $fire_t$ qui lui est spécifique. De cette manière on peut *spécialiser le tirage* et le réaliser plus efficacement.

Soit $t \in T$ une transition telle que ${}^\bullet t = \{s_1, \ldots, s_n\}$ et $t^\bullet = \{s'_1, \ldots, s'_m\}$. Dans ce contexte, la fonction $fire_t$ peut être réalisé comme présenté sur l'algorithme 3.1. Cette fonction crée une copie M' du marquage M, supprime les jetons consommés $(x_{s_1,t}, \ldots, x_{s_n,t})$ et ajoute les jetons produits en évaluant les annotations sur les arcs (de *manière native*), finalement cette fonction renvoie le marquage M'. Le point important de cette fonction est l'absence de parcours des places du réseau de Petri, au lieu de ces requêtes on réalise directement une séquence de modifications sur la structure de marquage. En effet, ceci est plus efficace et le code qui en résulte est plus facile à produire. Le second point important est l'absence de structure de données pour les modes. Pour chaque transition on utilise un ordre fixe sur les variables $x_{s_1,t}, \ldots, x_{s_n,t}$ apparaissant sur les arcs en entrée.

3.2.2 Calcul des successeurs

Le calcul des successeur est réalisé avec deux fonctions. Une fonction successeur réalisée de manière *spécifique* aux transitions et une fonction successeur qui renvoie l'ensemble des successeurs par toutes les transitions.

La fonction spécifique à une transition $t \in T$ énumère les modes de cette transition, et si la garde de celle-ci est satisfaite, calcule le nouvel état grâce à la fonction $fire_t$ correspondante. Finalement le nouvel état est ajouté à un ensemble d'états. Cette fonction est donnée en algorithme 3.2. Le point pertinent est la manière dont sont énumérés les modes, c'est une imbrication de boucles qui est constituée à la compilation pour récupérer les jetons du marquage des places en entrée de la transition. De cette manière on évite de manipuler le réseau de Petri, les opérations sont réalisées directement sur la structure de marquage sans requêter le réseau pour connaître les places nécessaires. De plus,

entrée : $M \in \mathbb{M}$, $x_{s_1,t} \in \ell(s_1)$, ..., $x_{s_n,t} \in \ell(s_n)$
sortie : M' successeur du marquage M par la transition t
`//copier` M
$M' \leftarrow copy(M)$;
`//consommer les jetons`
$M'(s_1) \leftarrow M(s_1) - \{x_{s_1,t}\}$;
...
$M'(s_n) \leftarrow M(s_n) - \{x_{s_n,t}\}$;
`//produire les jetons`
$M'(s'_1) \leftarrow M(s_1) + f_{t,s'_1}(x_{s_1,t}, \ldots, x_{s_n,t})$;
...
$M'(s'_m) \leftarrow M(s_m) + f_{t,s'_m}(x_{s_1,t}, \ldots, x_{s_n,t})$;
renvoyer M' ; `//renvoyer le successeur`

Algorithme 3.1: Tir de transition spécifique à une transition t.

le mode étant défini avec les variables du langage cible, celles-ci vont être compilées efficacement et aucune allocation mémoire n'est nécessaire pour stocker le mode. Ce n'est pas le cas par exemple chez Helena [57, 58] où les modes sont stockés dans une structure spécifique. Remarquons que cette approche directe n'est possible que parce qu'on génère une fonction par transition (Helena fait de même mais n'exploite pas cette possibilité complètement).

La fonction générale de calcul des successeurs prend un marquage en entrée et renvoie l'ensemble de ses marquages successeurs. Pour cela, un ensemble de marquages vide est créé puis chacune des fonctions successeur spécifiques aux transitions est appelée. Ces fonctions spécifiques vont ajouter les états successeurs par chacune des transitions et ainsi l'ensemble d'états contiendra tous les successeurs du marquage. Elle est présentée en algorithme 3.3.

entrée : $M \in \mathbb{M}$, $next \in 2^{\mathbb{M}}$
sortie : $next \in 2^{\mathbb{M}}$
pour x_n **in** $M(s_n)$ **:**
  ...
  pour x_1 **in** $M(s_1)$ **:**
    si $g_t(x_{s_1,t}, \ldots, x_{s_n,t})$ **alors**
      $next \leftarrow next \cup \{fire_t(M, x_{s_1,t}, \ldots, x_{s_n,t})\}$;
    fin si
  fin pour
fin pour
`//`$next$ `a été enrichi de nouveaux successeurs`

Algorithme 3.2: Fonction successeurs spécifique à une transition t.

entrée : $M \in \mathbb{M}$, $x_{s_1,t} \in \ell(s_1)$, ..., $x_{s_n,t} \in \ell(s_n)$
sortie : $next \in 2^{\mathbb{M}}$ ensemble de marquages successeurs de M
$next \leftarrow \emptyset$;
$succ_{t_1}(M, next)$;
$succ_{t_2}(M, next)$;
...
$succ_{t_n}(M, next)$;
renvoyer $next$; `//renvoyer l'ensemble de marquages`

Algorithme 3.3: Fonction successeurs générale.

3.3

Optimisations dirigées par la structure du modèle

Dans cette section, on présente diverses optimisations qui peuvent être réalisées lors de la génération du code source. Un exemple les utilisant est donné dans la section suivante (sec. 3.4).

3.3.1 Typage des jetons et expressions

La première optimisation concerne directement le langage cible et est liée à la traduction réalisée. La majorité des langages de programmation fournissent un système de types pour les valeurs qu'ils manipulent. En général un sous ensemble de ces types appelé types primitifs existe. Un type primitif a l'avantage de pouvoir être manipulé plus efficacement, et souvent de manière native. Dans le cadre d'un langage compilé cela signifie que ces valeurs peuvent être manipulées directement par le processeur de la machine cible.

L'optimisation consiste à utiliser le plus de ces types pour représenter les jetons du réseau de Petri. Pour cela, il faut que les types du formalisme de modélisation d'entrée puissent être aisément traduits. En général c'est le cas pour des types comme les entiers relatifs bornés, les booléens, les chaînes de caractère, *etc.* Les annotations de type des places du réseau permettent de rapidement décider de l'encodage du jeton et de la place. De plus, les modes étant représentés par des variables qui stockent un jeton de la place (ou un pointeur sur le jeton), elles peuvent aussi être typées en utilisant ces informations.

Les fonctions étiquetant les arcs de sortie des transitions, quant à elles, sont plus délicates à gérer. Si on suppose ces fonctions bien typées, alors chacune d'elles renvoie forcément une valeur de type de la place qui est connectée à l'arc correspondant. Si on n'a pas cette garantie, il faut s'assurer que le type de retour de la fonction convient, dans le cas contraire la transition ne devrait pas être tirée. Selon la sémantique du formalisme, l'absence de garantie de type des expression peut être une caractéristique recherchée.

Dans le chapitre 7 on décrit l'implémentation de l'outil *Neco* en mettant l'accent sur la non trivialité de cette optimisation dans notre cas.

3.3.2 Calcul des modes

Chaque fonction $succ_t$ énumère les jetons des places en entrée de sa transition. L'ordre dans lequel se font ces énumérations est arbitraire. En effet, la sémantique usuelle des réseaux de Petri ne définit pas d'ordre sur les arcs en entrée, et donc cela n'a aucune incidence sur la correction de l'approche. Cependant, si on change l'ordre dans lequel ces énumérations sont réalisées, le calcul des successeurs peut être fait de manière beaucoup plus rapide [56]. Par exemple, étant donné un réseau de Petri (S, T, ℓ), une transition $t \in T$ et deux places $s_1, s_2 \in {}^\bullet t$. Si on sait que s_1 est 1-bornée mais non s_2, alors il est plus efficace d'énumérer s_1 avant s_2, parce que si s_1 est vide alors s_2 ne sera pas énumérée. Plus généralement, l'ordre des arcs en entrée de la transition peut être choisi en privilégiant d'abord les places dont la borne est la plus petite, puis les places dont le type a un plus petit domaine. L'optimisation présentée dans [56] est plus générale et repose sur les variables qui sont valuées sur les arcs, ce qui n'est pas applicable à notre cas vu les restrictions sur les expressions étiquetant les arcs et les gardes de transitions.

En général, les bornes des places de réseaux de Petri arbitraires ne sont connues qu'en calculant l'espace d'états complet du modèles ou bien des invariants de places [68, 101]. Cependant, ce genre de propriétés peut être connu en utilisant des techniques de modélisation adéquates : en particulier, les places de contrôle de flot dans les algèbres de réseaux de Petri colorés [144] peuvent être garanties comme 1-bornées.

On peut pousser l'optimisation plus loin en utilisant les places 1-bornées. En effet, par définition ce type de place ne peut avoir qu'un jeton au plus et donc l'utilisation d'une boucle devient inutile. Ainsi une boucle peut être remplacée par un "**si**" pour chaque place 1-bornée, ce qui revient à tester si la place est vide ou non. Pour un maximum d'efficacité ces tests doivent être réalisés avant les boucles pour n'être exécutés qu'une seule fois. Tout cela s'applique aussi pour les places à types singletons tels que les jetons noirs.

3.3.3 Encodage des places

Ce type d'optimisation consiste à choisir l'encodage de chaque place de manière pertinente. C'est une conséquence du typage des places, en effet grâce au type des places on peut choisir l'encodage de celle-ci pour un maximum d'efficacité, c'est un des avantages principaux de la passe de compilation. Cependant en exploitant les propriétés du modèle nous pouvons aller plus loin.

L'implémentation d'une structure de marquage peut être vue comme un vecteur de bits. Ce vecteur est segmenté en places qui peuvent soit être des pointeurs vers des structures de données complexes, soit être des valeurs qui sont interprétées comme un certain marquage de la place.

Si on considère les types finis, par exemple des types énumérés alors on sait quel est le nombre de bits maximal nécessaire pour les représenter ; si n est le nombre de valeurs qu'accepte ce type alors le nombre de bits nécessaires est $k = \lceil log_2(n) \rceil$. On peut alors utiliser un vecteur de k bits pour représenter chaque valeur.

Pour les places 1-bornées de type quelconque, ce procédé peut-être poussé à encore plus loin en réservant l'espace à l'avance dans la structure de marquage. Il faut cependant faire attention à la représentation des places vides.

Si la place 1-bornée est vide, alors il faut réserver un bit dans la structure de marquage qui l'indique. Une autre approche est d'utiliser un pointeur dans la structure de marquage sur l'unique valeur et stocker un pointeur nul si la place est vide. Cette seconde approche est moins efficace que la première à cause de l'espace utilisé, des déréférencements pour accéder aux données et de l'allocation mémoire nécessaire pour l'unique jeton. Les allocations dynamiques de mémoire doivent en général être réduites au minimum pour un gain de performances maximal, il est toujours plus efficace d'allouer un bloc de mémoire en une seule fois plutôt que le segmenter avec de petites allocations. Un exemple de l'optimisation est donné en algorithme 3.4.

```
entrée : M ∈ 𝕄, next ∈ 2^𝕄
sortie : next ∈ 2^𝕄
pour x_{s_1,t} in M(s_1) :
    ...
    pour x_{s_{i-1},t} in M(s_{i-1}) :
        si M(flag_{s_i})          //teste si la place est marquée
        alors
            x_i ← M(s_i);                //récupère l'unique jeton
            pour x_{s_{i+1},t} in M(s_{i+1}) :
                ...
                pour x_{s_n,t} in M(s_n) :
                    si g_t(x_{s_1,t}, ..., x_{s_n,t}) alors
                        next ← next ∪ {fire_t(M, x_{s_1,t}, ..., x_{s_n,t})};
                    fin si
                fin pour
            fin pour
        fin si
    fin pour
fin pour
```

Algorithme 3.4: Énumération d'une place 1-bornée.

Dans un réseau de Petri de haut niveau, on manipule des jetons de divers types. Selon le type de jeton, il existe souvent des implémentations plus efficaces qui peuvent être utilisées. Par exemple, pour représenter un multiensemble de valeurs booléennes on pourrait utiliser l'approche ci-dessus en utilisant un bit par jeton, l'implémentation serait donc un vecteur de bits plus la taille de ce vecteur. Cependant si la place contient un grand nombre de jetons alors elle peut être représentées avec deux compteurs, un compteur pour les jetons *true* et un compteur pour les jetons *false*. Ce deuxième encodage sera plus efficace en temps de calcul puisque l'énumération de la place peut se traduire en un branchement conditionnel comme présenté sur l'algorithme 3.5.

De manière similaire une place de type jeton-noir peut être représentée par un compteur qui indique le nombre de jetons. Ainsi l'énumération de cette place revient à tester si la place est vide ou non. Puisque toutes les instances du type jeton-noir sont équivalentes, la consommation ou la production d'un

```
entrée : M ∈ 𝕄, next ∈ 2^𝕄
sortie : next ∈ 2^𝕄
pour x_{s_1,t} in M(s_1) :
    ...
    pour x_{s_{i-1},t} in M(s_{i-1}) :
        //énumération de la place en deux cas:
        //1. cas 'true'
        si M(s_i)(true) > 0 alors
            x_i ← true;                    //compteur 'true' non nul
            pour x_{s_{i+1},t} in M(s_{i+1}) :
                ...
                pour x_{s_n,t} in M(s_n) :
                    si g_t(x_{s_1,t}, ..., x_{s_n,t}) alors
                        next ← next ∪ {fire_t(M, x_{s_1,t}, ..., x_{s_n,t})};
                    fin si
                fin pour
            fin pour
        fin si
        //2. cas 'false'
        si M(s_i)(false) > 0 alors
            x_i ← false;                   //compteur 'false' non nul
            pour x_{s_{i+1},t} in M(s_{i+1}) :
                ...
                pour x_{s_n,t} in M(s_n) :
                    si g_t(x_{s_1,t}, ..., x_{s_n,t}) alors
                        next ← next ∪ {fire_t(M, x_{s_1,t}, ..., x_{s_n,t})};
                    fin si
                fin pour
            fin pour
        fin si
    fin pour
fin pour
```

Algorithme 3.5: Énumération d'une place de type *bool*.

tel jeton n'est pas influencée par sa valeur mais juste sa présence. Si en plus elle est 1-bornée, alors 1 seul bit est suffisant pour la représenter.

De manière générale une optimisation similaire peut être appliqué à tout type de place pour un maximum d'efficacité.

3.4 Exemple

Dans cette section nous allons compiler le réseau de Petri donné en figure 3.3. Les places sont de trois types : les jetons noirs (•), les booléens (*bool*) et les entiers 32 bits (*int*). On suppose qu'on sait par construction que la place s_3 est 1-bornée. La place s_3 étant 1-bornée, s_4 l'est forcément. Le réseau ne possède qu'une seule transition que nous allons compiler. On s'intéresse aux fonctions $succ_t$, $fire_t$ et $init_t$ correspondantes.

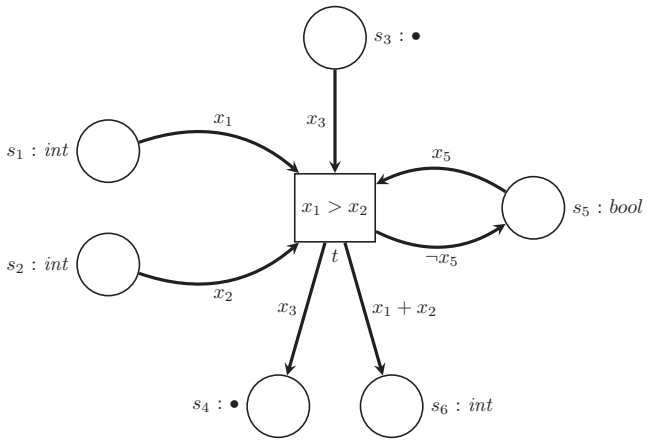


FIGURE 3.3 – *Exemple de réseau de Petri, la place s_3 est supposée 1-bornée.*

La fonction *succ* spécifique à la transition t est donnée en algorithme 3.6. L'ordre des énumérations des places correspond à l'optimisation décrite dans la section précédente : on énumère les place les plus bornées et de plus petit domaine en premier. On commence donc par la place s_3 de type jeton-noir. En utilisant une implémentation adéquate nous pouvons la représenter par un seul bit sachant qu'elle est 1-bornée, on exploite donc cette information pour remplacer l'énumération par un simple test qui vérifie si la place est marquée ou non. En deuxième on énumère la place de type *bool* en utilisant la décomposition en deux compteurs, ainsi on a deux blocs indépendants créés : l'un pour une instance du jeton *true*, l'autre pour l'instance du jeton *false*. Ensuite on énumère les places de type *int*, comme on n'a pas d'informations sur ces places, on les énumère de manière classique. Cependant ces énumérations ne sont réalisées que si les places s_3 et s_5 ne sont pas vides ce qui est détecté très rapidement.

La fonction *fire* spécifique à la transition t est donné en algorithme 3.7. Elle consiste à copier le marquage, consommer les jetons en utilisant les valeurs

```
entrée : M ∈ 𝕄, next ∈ 2^𝕄
sortie : next ∈ 2^𝕄
si M(s_3) alors
    //on n'utilise pas la variable x_3 parce que c'est •
    si M(s_5)(true) > 0 alors
        x_5 ← true
        pour x_1 in M(s_1) :
            pour x_2 in M(s_2) :
                si g_t(x_1, x_2, x_5) alors
                    next ← next ∪ {fire_t(M, x_1, x_2, x_5)};
                fin si
            fin pour
        fin pour
    fin si
    si M(s_5)(false) > 0 alors
        x_5 ← false
        pour x_1 in M(s_1) :
            pour x_2 in M(s_2) :
                si g_t(x_1, x_2, x_5) alors
                    next ← next ∪ {fire_t(M, x_1, x_2, x_5)};
                fin si
            fin pour
        fin pour
    fin si
fin si
```

Algorithme 3.6: Exemple de calcul de successeurs de la transition t de la figure 3.3.

```
entrée : M ∈ 𝕄, x_{s_1,t} ∈ ℓ(s_1), ..., x_{s_n,t} ∈ ℓ(s_n)
sortie : M' successeur du marquage M par la transition t
//copy marking M
M' ← copy(M);
//consommer les jetons
M'(s_3) ← 0;                          //on sait que s_3 est 1-bornée
M'(s_5)(x_5) ← M(s_5)(x_5) − 1;        //on consomme le booléen
M'(s_1) ← M(s_1) − {x_1};             //on consomme l'entier x_1
M'(s_2) ← M(s_2) − {x_2};             //on consomme l'entier x_2
//produire les jetons
M'(s_4) ← 1;        //on sait que s_4 est 1-bornée et x_3 est •
M'(s_5)(¬x_5) ← M(s_5)(¬x_5) + 1;   //réalisée par une fonction
M'(s_6) ← M(s_2) + {x_1 + x_2};       //production de la somme
renvoyer M' ;                          //marquage successeur
```

Algorithme 3.7: Exemple de tirage de la transition t de la figure 3.3.

des arguments, puis utiliser les annotations d'arcs pour produire les nouveaux jetons.

La dernière fonction est celle du marquage initial et est donnée en figure 3.8. Elle consiste simplement à allouer un nouveau marquage et y ajouter les jetons nécessaires.

entrée : vide
sortie : M marquage initial du réseau de Petri
$M \leftarrow cons_{mrk}()$; `//allouer le marquage`
`//ajouter les jetons`
$M(s_1) \leftarrow \{2, 4\}$;
$M(s_2) \leftarrow \{3\}$;
$M(s_3) \leftarrow 1$; `//on sait que` s_3 `est 1-bornée de type` $\bullet$
$M(s_5)(true) \leftarrow 1$; `//un jeton` *vrai* `dans` s_5
$M(s_5)(false) \leftarrow 0$; `//aucun jeton` *faux* `dans` s_5
renvoyer M;

ALGORITHME 3.8: Exemple de fonction de marquage initial pour le réseau de la figure 3.3.

3.5

Réduction du contrôle de flot

Lorsqu'on utilise des formalismes de modélisation structurés dérivés des *algèbres de réseaux de Petri* [22], on compose différents processus. L'application de la composition séquentielle, du choix, ou de l'itération sur deux processus séquentiels produisent un processus séquentiel.

Les processus séquentiels des algèbres de réseaux de Petri ont la bonne propriété de produire un contrôle de flot 1-borné de places de type jeton noir : c'est un ensemble de places de type jeton noir 1-bornées et exclusives. Plus formellement cela signifie que ces places forment un invariant de valeur 1 et donc exactement une seule place est marquée. La composition parallèle de deux processus séquentiels produit un réseau formé de deux processus concurrents, dont l'ensemble des places de contrôle de flot n'est plus exclusif, cependant on peut toujours aisément identifier deux sous ensembles de places exclusives et qui correspondent au contrôle de flot de chacun des processus. Ainsi par la suite, si on compose des réseaux contenant des processus concurrents avec des processus séquentiels ou d'autres processus concurrents, on peut toujours identifier le contrôle de flot de chacun des processus du réseau résultant.

L'optimisation présentée dans cette section s'applique aux place de contrôle de flot mais peut être généralisée à tout sous-ensemble de place 1-bornées exclusives de type jeton noir.

Soit S_f un ensemble de places 1-bornées exclusives de type jeton noir. Puisque les places sont exclusives, si une transition possède deux places de S_f en entrée alors elle ne sera *jamais* tirable. De même, une transition qui possède deux places de S_f en sortie ne sera pas tirable puisque ça violerai l'hypothèse d'exclusivité. On peut donc les supprimer ce type de transitions

sans modifier l'ensemble des états accessibles. On suppose donc que ce type de transition n'existe pas dans le réseau.

Pour chaque place s de S_f on peut identifier un ensemble de transitions exclusives T_s défini par $T_s \stackrel{\text{df}}{=} \{t \in T \mid s \in {}^\bullet t\}$, on notera aussi $T_f \stackrel{\text{df}}{=} \cup_{s \in S_f} T_s$. On sait que si la place s n'est pas marquée alors les transitions de T_s ne sont pas tirables. Réciproquement, si s est marquée alors quel que soit $s' \in S_f$ telle que $s' \neq s$ on sait que les transitions de $T_{s'}$ ne sont pas tirables car s et s' sont exclusives. On peut donc dire que chaque place de S_f identifie un ensemble de transitions potentiellement tirables. Ces transitions ne sont tirables que si la place correspondante est marquée, il s'agit donc d'une condition nécessaire à leur tir.

Si on a n places dans S_f alors on peut les encoder avec $\lceil log_2(n) \rceil$ bits, il suffit d'identifier la place marquée par une valeur entre 0 et $\lceil log_2(n) \rceil - 1$. L'optimisation peut alors correspondre à fusionner ces places de contrôle de flot en une seule place.

Un exemple est donné en figure 3.4. Les places de contrôle de flot sont $\{s_{f_0}, s_{f_1}, s_{f_2}, s_{f_3}\}$ on les identifie par une valeur de 0 à 3. Elles sont toutes encodées avec la place s_f dans le réseau réduit. Ainsi, dans le réseau réduit à l'état initial on a $M_0(s_f) = 0$ ce qui correspond à la place s_{f_0} marquée. On peut tirer la transition t_1 seulement si s_{f_0} est marqué, ce qui est représenté de manière équivalente dans le réseau réduit avec la garde $x = 0$. On procède de la même manière pour les deux autres transitions. On remarque que les deux autres transitions ont la même garde $x = 1$, c'est cohérent puisqu'elles sont potentiellement tirables que si la place s_{f_1} est marquée. Les places en entrée et sortie des transitions correspondent au contrôle de flot, ainsi si la place s_{f_0} était marquée et qu'on tire la transition t_1, on obtient un état où la place s_{f_1} est marquée, ce qui correspond à la création du jeton 1 pour la place s_f dans le réseau réduit.

(a) *Réseau non réduit.* (b) *Réseau réduit.*

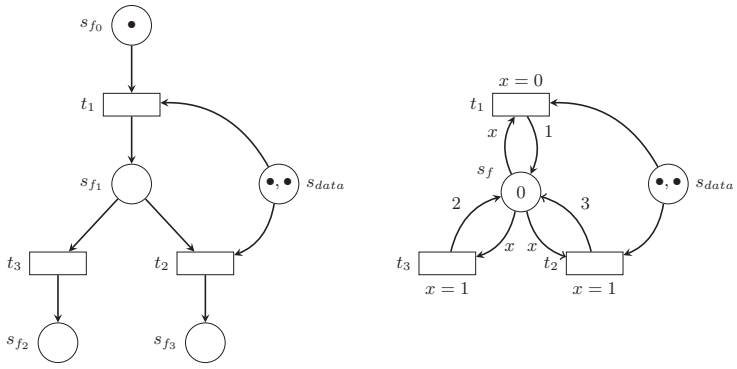


Figure 3.4 – *Exemple fusion de contrôle de flot.*

En regardant le réseau de la figure 3.4 on a l'impression de n'avoir rien gagné à part $n - \lceil log_2(n) \rceil$ bits dans la structure de marquage. De plus, on a l'impression de perdre à cause de l'évaluation de gardes qui a été introduite. Cependant lorsqu'on réfléchit en termes de compilation on peut gagner beaucoup

plus en évitant de manipuler cette place autant que possible.

En effet, selon la valeur du jeton dans la place s_f on peut directement sélectionner les transitions tirables et écarter toutes les autres. L'algorithme 3.9 présente cette étape de sélection. On remarque que les fonctions correspondantes aux autres transitions ne sont pas appelées : on a épuré l'ensemble des transitions potentiellement tirables. Les fonctions successeur et tir doivent être adaptées en conséquence, maintenant on n'effectue plus l'énumération de la place s_f dans les fonctions successeurs et on met à jour le contrôle de flot dans la fonction de tir. On donne l'implémentation de la transition t_2 de la figure 3.4 aux algorithmes 3.10 et 3.11.

Cette optimisation permet donc de rapidement éliminer les transitions non tirables et donc accélérer l'exploration d'espaces d'états.

entrée : $M \in \mathbb{M}$, $next \in 2^{\mathbb{M}}$
sortie : $next \in 2^{\mathbb{M}}$
selon $M(s_f)$ **faire**
  cas *0*
    $succ_{t_1}(M, next)$;
  fin
  cas *1*
    $succ_{t_2}(M, next)$;
    $succ_{t_3}(M, next)$;
  fin
  autres cas
    skip
  fin
fin

Algorithme 3.9: Fonction successeur pour les transitions exclusives de la figure 3.4.

entrée : $M \in \mathbb{M}$, $next \in 2^{\mathbb{M}}$
sortie : $next \in 2^{\mathbb{M}}$
pour y **in** $M(s_{data})$ **:**
  $next \leftarrow next \cup \{fire_t(M, y)\}$;
fin pour
renvoyer *next* ; `//renvoyer l'ensemble des successeurs`

Algorithme 3.10: Fonction successeur spécifique à la transition t_2 de la figure 3.4.

entrée : $M \in \mathbb{M}$, $x_{s_1,t} \in \ell(s_1)$, ..., $x_{s_n,t} \in \ell(s_n)$
sortie : M' successeur du marquage M par la transition t
$M' \leftarrow copy(M)$;
$M'(s_f) \leftarrow \{3\}$; `//place de contrôle suivante`
$M'(s_{data}) \leftarrow M(s_{data}) - 1$; `//place encodée avec un entier`
renvoyer M' ; `//renvoyer le successeur`

Algorithme 3.11: Fonction de tir spécifique à la transition t_2 de la figure 3.4.

3.6

Conclusion

Dans ce chapitre on a présenté une approche de compilation pour les réseaux de Petri de haut niveau. La bibliothèque générée fournit des primitives d'exploration pour les espaces d'états basées sur des algorithmes simples. Ces algorithmes ne manipulent pas la structure du réseau de Petri qui est maintenant encodée dans les calculs, les diverses opérations sont essentiellement réalisées sur la structure de marquage. De cette manière on évite les étapes d'interprétation qui sont nécessaires en général. On a aussi présenté différentes optimisations qui exploitent les particularités des modèles pour réduire l'occupation mémoire et accélérer l'exploration. Ainsi, le coeur de la contribution est la définition de l'approche par compilation dans un cadre général et indépendant du langage cible, mais aussi les optimisations basés sur les propriétés structurelles des modèles.

Dans le chapitre suivant on continue à adresser la compilation mais cette fois du point de vue de l'implémentation dans un langage de bas niveau. Dans ce cadre on réalisera un travail de certification en fournissant des preuves de correction de l'approche. Les résultat expérimentaux quant à eux sont présentés dans le chapitre 7.

4 Compilation vers un langage de bas niveau

Le procédé de compilation d'un réseau de Petri peut cibler différents langages de programmation. Dans ce chapitre on s'intéresse à la compilation vers un langage de bas niveau : *LLVM*. Comme on l'a vu dans le chapitre précédent, la compilation d'un réseau de Petri supprime les structures de données qui l'encodent et les remplace par un ensemble de fonctions pour l'exploration. Par conséquent, il faut s'assurer que cette transformation préserve la sémantique du réseau de Petri originel puisqu'il n'est plus utilisé.

La contribution présentée ici, est d'une part de mener la démarche de compilation au bout sur un langage réel, et d'autre part de prouver la correction de l'approche. Le langage cible choisi est *LLVM* qui fait partie de la suite d'outils *LLVM : Low Level Virtual Machine*. Notre choix pour ce langage résulte des constats suivants :

- ce langage proche de l'assembleur possède des instructions de haut niveau suffisamment expressives pour décrire nos algorithmes de manière raisonnable ;
- ce langage est assez bas niveau pour être équipé d'une sémantique formelle pour prouver la correction de programmes.

Pour arriver à nos fins on choisit un fragment de LLVM suffisant pour réaliser notre bibliothèque d'exploration, ensuite on définit une sémantique opérationnelle de celui-ci pour prouver des propriétés sur les programmes produits par notre compilateur. Au moment de la publication de ce travail [FP11c], il s'agissait de la *première* sémantique formelle pour le langage LLVM. De plus, même si les model checkers sont des outils largement répandus, il y a peu de tentatives pour les prouver au niveau de l'implémentation [159], contrairement au domaine des assistants de preuves [43, 137] pour lesquels il est courant de prouver la correction de l'outil.

Ce chapitre est basé sur [FP11c] et [FP11b].

4.1

LLVM et LLVM-IR

Le *projet LLVM*(*Low Level Virtual Machine*) [120] est une boite à outils *moderne* et *modulaire* pour le développement de compilateurs. Elle est largement utilisée par divers projets commerciaux, open source, ou encore académiques [121, 122].

Le langage *LLVM*, ou plus précisément *LLVM-IR* (*LLVM Itermediate Representation*) [119], fait partie du projet LLVM. Il s'agit d'un langage de représentation intermédiaire qui est bas niveau et multiplateforme. Tout programme écrit dans ce langage peut être exécuté dans une *machine virtuelle* ou *compilé en code natif* pour toutes les plateformes compatibles avec le projet LLVM. Ces programmes vont être optimisés par le compilateur LLVM qui exécute une large variété de passes d'optimisation sur cette représentation. Tout cela nous permet de générer du *code simple* sachant qu'il sera *optimisé* par la suite.

4.1.1 Syntaxe et sémantique intuitive

Dans ce chapitre nous présentons d'abord le langage LLVM-IR de manière formelle afin de pouvoir lui donner une sémantique par la suite. Une descriptions informelle du langage (mais suffisamment précise) peut être trouvée dans la documentation du projet [120].

Un programme LLVM est composé d'un ensemble de *blocs* (*i.e.*, séquences d'instructions) identifiés par des étiquettes. Le passage d'un bloc à un autre est toujours explicite avec : une instruction de branchement (conditionnelle ou non), un appel de sous-programme ou une instruction de retour de sous-programme.

Pour définir la syntaxe de ce langage, on considère les ensembles deux à deux disjoints suivants :

- un ensemble de *pointeurs* $\mathbb{P}$;
- un ensemble de *types LLVM* $\mathbb{T}$, défini inductivement comme le plus petit ensemble contenant l'ensemble de *types primitifs* $\mathbb{T}_0 \stackrel{\text{df}}{=} \{int, bool, \ldots\}$ (entiers, booléens, et autres types qu'on n'utilisera pas ici) et tel que si $t_0, \ldots, t_n \in \mathbb{T}$ alors $struct(t_0, \ldots, t_n) \in \mathbb{T}$, ce qui représente une structure de données avec $n+1$ champs de types $t_0, \ldots, t_n$, de plus pour tout type $t \in \mathbb{T}$ on a aussi un type $ptr(t) \in \mathbb{T}$ qui correspond au type pointeur sur une valeur de type t ;
- un ensemble d'*étiquettes* $\mathbb{L}$ contenant des identifiants arbitraires mais aussi des étiquettes particulières notées $f(\!| a_1, \ldots, a_n |\!)$, où $a_i \in \mathbb{V}$ pour $1 \leq i \leq n$ et qui correspondent aux *entrées de sous-programmes* (avec les arguments formels $a_1, \ldots, a_n$). On défini l'ensemble $\mathbb{L}_\perp \stackrel{\text{df}}{=} \ell \cup \{\perp\}$ où $\perp \notin \mathbb{L}$ dénote l'étiquette *"non définie"*.

Ces définitions ne sont valables que pour ce chapitre et le chapitre suivant, les symboles utilisés ici pourront être réutilisés dans les autres chapitres pour d'autres objets.

Un *programme* LLVM est représenté comme une fonction partielle P de $\mathbb{L}$ vers l'ensemble de blocs, *i.e.*, il associe à chaque étiquette de son domaine une séquence d'instructions (définies par la suite).

Dans notre cas, on s'intéresse à un fragment de LLVM présenté sous forme de grammaire en figure 4.1. Ce fragment est composé de trois classes syntaxiques : les séquences *seq*, les commandes *cmd* (*i.e.*, instructions) et expressions *expr*. Une exception existe cependant pour les séquences. Une séquence est une liste de commandes qui peut se terminer par une expression, et dans ce cas la séquence est considérée elle-même comme expression (ce qui n'est pas mis en évidence sur la figure 4.1). On suppose que nos programmes sont *syntaxiquement corrects* et *bien typés*, ce qui permet de simplifier la syntaxe en omettant les annotations de types dans le code source LLVM ce qui donne cette syntaxe simplifié.

seq	::=	*cmd*	(commande)
	|	*expr*	(expression)
	|	*cmd*; *seq*	(séquence d'instructions)
cmd	::=	*br label*	(branchement non conditionnel)
	|	*br rvalue*, *label*, *label*	(branchement conditionnel)
	|	*pcall label*$(rvalue, \ldots, rvalue)$	(appel de procédure)
	|	*ret*	(retour d'une procédure)
	|	*var* = *expr*	(assignation de variable)
	|	*store rvalue*, *rvalue*	(écriture à un emplacement pointé)
	|	*skip*	(séquence vide)
expr	::=	*add rvalue*, *rvalue*	(addition)
	|	*load rvalue*	(lecture d'une valeur pointée)
	|	*gep rvalue*, 0, *nat*	(obtenir pointeur sur un champ de structure)
	|	*icmp op*, *rvalue*, *rvalue*,	(comparaison d'entiers)
	|	*fcall label*$(rvalue, \ldots, rvalue)$	(appel de fonction)
	|	*alloc type*	(allocation de mémoire)
	|	*ret rvalue*	(renvoyer une valeur d'une fonction)
	|	*rvalue*	(variable ou valeur)
	|	*phi* $(rvalue, label), \ldots, (rvalue, label)$	(lire une variable après un branchement)

FIGURE 4.1 – *Syntaxe de notre fragment de LLVM, avec* $label \in \mathbb{L}$, $rvalue \in \mathbb{D} \cup \mathbb{P} \cup \mathbb{V}$, $var \in \mathbb{V}$, $type \in \mathbb{T}$, $nat \in \mathbb{N}$ *et* $op \in \{<, \leq, =, \neq, \geq, >\}$.

SUPPOSITION 4.1. *Les programmes sont syntaxiquement corrects et bien typés.* ◊

Pour écrire les séquences sur une seule ligne, on introduit l'opérateur de séquencement ";" qui correspond aux fins de lignes en LLVM. On introduit aussi la commande *skip* qui dénote la séquence vide. On peut remarquer que les instructions *pcall* (appel de procédure) et *fcall* (appel de fonction) n'existent pas en LLVM mais sont deux instances de l'instruction *call*. Cette différenciation peut être facilement réalisée en LLVM parce que l'instruction *call* contient le type de retour du sous-programme (fonction ou procédure) que nous avons omis dans notre variante syntaxique. Nous détailleront les instructions à mesure que ce sera utile ; en première approche il suffit de noter que :

- l'instruction *store* (resp. *load*) est l'action d'écrire (resp. lire) des données dans (resp. de) la mémoire en utilisant un pointeur ;
- l'instruction *icmp* compare deux entiers ;
- l'instruction *phi* est utilisée pour accéder à des variables assignés dans d'autres blocs (variable *rvalue*, du bloc *label*), par exemple, *phi* $(i, a), (i, b)$ vaut i si on vient du bloc a et j si on vient du bloc b ;
- l'instruction *gep* correspond à l'arithmétique de pointeurs, on fixe le second argument de celle-ci à 0, ce qui permet d'accéder aux attributs d'une structure en utilisant indice du champ demandé.

L'instruction *phi* est très importante. En effet LLVM est en forme *SSA* (*static single assignment*) qui est une contrainte syntaxique sur les variables apparaissant dans le programme. Chaque variable ne peut être assignée qu'à un seul emplacement dans le code source et la portée d'une variable est limitée au bloc de sa déclaration. Ainsi si on souhaite l'utiliser dans plusieurs blocs, lors de la réalisation d'une boucle par exemple, nous ne pouvons pas le faire directement. Pour cela, il faut utiliser l'instruction *phi* et affecter une nouvelle variable.

4.2 Modèles traités

Les modèles qu'on compile sont des réseaux de Petri supposés compatibles avec LLVM qui est notre langage de couleurs (Définition 3.1). Pour cela, on suppose que chaque réseau de Petri (S, T, ℓ) est défini dans le contexte d'un programme P, et qu'on a :

- pour toute place $s \in S$, $\ell(s)$ dénote un type du langage LLVM dans $\mathbb{T}$, interprété comme un sous-ensemble de $\mathbb{D}$;
- pour toute transition $t \in T$, $\ell(t)$ est un appel à une fonction booléenne dans P dont les paramètres sont dans $vars(t)$;
- pour tout $s \in t^\bullet$, $\ell(t, s)$ est un multiensemble singleton dont l'unique élément est un appel à une fonction dans P avec comme arguments formels des éléments de $vars(t)$ et comme type de retour $\ell(s)$.

On a donc concrétisé les types des places avec des types LLVM et chaque expression est maintenant implémentée par une fonction LLVM appelée à partir de l'annotation correspondante. Comme dans le chapitre 3, pour simplifier la présentation, on restreint aussi les arcs sortants des transitions à être des multiensembles singletons, mais ceci peut être facilement généralisé. De plus, on suppose toujours que les annotations (arcs et gardent) terminent (*cf.* Définition 3.1).

On étend la notion de valuation et mode de transition à LLVM, on les nomme *LLVM-valuations* et *LLVM-modes*. Une LLVM-valuation est une fonction partielle $\beta : \mathbb{V} \nrightarrow \mathbb{D} \cup \mathbb{P}$ qui assigne à chaque variable de son domaine un pointeur ou une valeur, de plus on étend son domaine à $\mathbb{D} \cup \mathbb{V}$ par la fonction identité. On pose $\mathbb{B}$ l'ensemble des LLVM-valuations. Un LLVM-mode est donc une LLVM-valuation qui rend tirable une transition d'un réseau de Petri annoté par le langage LLVM.

Dans la suite de ce chapitre et le chapitre suivant, on appellera par valuations les LLVM-valuations et par modes les LLVM-modes.

4.3 Implémentation de l'exploration d'espaces d'états

Étant donné un marquage initial M_0, l'espace d'états qu'on souhaite calculer est donné par l'ensemble R des marquages accessibles, *i.e.*, le plus petit ensemble R tel que $M_0 \in R$ et, si $M \in R$ et $M[t, \beta\rangle M'$ (pour une transition t et une valuation β) alors $M' \in R$. La correction et la terminaison de l'implémentation présentée dans cette section est discutée dans la section 4.7.

Nos algorithmes sont implémentés sur la base de structures de données prédéfinies, telles que les multiensembles, les marquages, et les ensembles de marquages, qui doivent respecter certaines interfaces. Une interface est un ensemble de procédures ou fonctions qui manipulent une structure de données à travers un pointeur ; il s'agit d'interfaces *à la C*.

Pour chacune des fonctions, ou procédures, de ces interfaces on donne une *spécification formelle de comportement*. Elle se base sur une *interprétation explicite* de la structure de données pointée avant et après l'appel du sous-programme (la fonction ou procédure). La manière de décrire ce genre de spécifications est donnée dans la section 4.5, une fois la sémantique formelle de LLVM définie. L'ensemble des interfaces est donné dans la section 4.6.

Parmi les interfaces qu'on suppose, on a tout d'abord les multiensembles. Un multiensemble stocke des valeurs d'un type $d \in \mathbb{T}$. En particulier, on suppose que l'interface du multiensemble contient :

- une fonction $cons_{mset}(\!|\,|\!)$ qui renvoie un nouveau multiensemble vide ;
- une procédure $add_{mset}(\!|\, p_{mset}, elt \,|\!)$ qui ajoute un élément *elt* dans le multiensemble pointé par le pointeur p_{mset} ;
- une procédure $rem_{mset}(\!|\, p_{mset}, elt \,|\!)$ qui supprime un élément *elt* du multiensemble pointé par le pointeur p_{mset} ;
- une fonction $size_{mset}(\!|\, p_{mset} \,|\!)$ qui renvoie la taille du domaine du multiensemble ;
- une fonction $nth_{mset}(\!|\, p_{mset} \,|\!)$ qui renvoie le n-ième élément du domaine du multiensemble (on considère un ordre arbitraire fixé pour les valeurs dans un multiensemble) ;
- une fonction $copy_{mset}(\!|\, p_{mset}, p'_{mset} \,|\!)$ qui copie le contenu du multiensemble pointé par p_{mset} dans le multiensemble pointé par p'_{mset} ;
- $clear_{mset}(\!|\, p_{mset} \,|\!)$ qui supprime tous les éléments du multiensemble pointé par p_{mset} ;
- une fonction $notempty_{mset}(\!|\, p_{mset} \,|\!)$ qui renvoie *true* si le multiensemble pointé par p_{mset} est vide et *false* sinon.

Chaque place est un conteneur de jetons qui est un multiensemble. L'interface d'une place est donc exactement celle d'un multiensemble mais n'est pas nécessairement réalisée avec la même structure de données que celui-ci. Nous verrons plus tard des implémentations spécifiques guidées par les optimisations (sec. 4.8).

On pourrait aussi ajouter une fonction qui renvoie le nombre d'occurrences d'un élément dans un multiensemble. Cela peut être utile pour lever la restriction sur les arcs d'entrée de transitions, supposés ensembles singletons de variables. Par exemple, pendant l'énumération d'une place pour récupérer deux jetons, si les indices de ces jetons sont égaux alors il faudrait vérifier que le nombre d'occurrences est supérieur à deux.

Pour différencier les réalisations des places de la réalisation d'un multiensemble, on reprend la même interface avec les fonctions et procédures annotées par le nom de la place, c'est-à-dire qu'au lieu de l'annotation *mset* on utilisera l'annotation s, si la place est s. Par exemple, add_s est la fonction d'ajout de jeton dans la place s.

On suppose aussi une interface pour les marquages, elle contient :

- pour chaque place s, une fonction $get_s(\!| p_{mrk} |\!)$ qui renvoie un pointeur sur la place correspondante ;
- une fonction $copy_{mrk}(\!| p_{mrk} |\!)$ qui copie le marquage pointé par p_{mrk}.

Finalement, la dernière interface est celle des ensembles de marquages et contient :

- une fonction $cons_{set}(\!|\,|\!)$ qui construit un nouvel ensemble de marquages vide ;
- une procédure $add(\!| p_{set}, elt |\!)$ qui ajoute le marquage *elt* à l'ensemble pointé par p_{set} ;
- une fonction $contains(\!| p_{set}, elt |\!)$ qui teste si le marquage *elt* appartient à l'ensemble pointé par p_{set}.

La fonction *contains* n'est pas directement utilisée dans la bibliothèque générée cependant, elle est nécessaire si on veut construire les espaces d'états par la suite.

4.4 Réalisation des algorithmes

Dans cette section on présente la réalisation des fonctions nécessaires à l'exploration : *init*, *succ*, $fire_t$ et $succ_t$.

4.4.1 Fonctions init et succ

La fonction *init* renvoie le marquage initial du réseau de Petri compilé. Elle crée un nouveau marquage puis ajoute des jetons afin d'obtenir le marquage initial du réseau. On se limitera à un exemple de l'implémentation de cette fonction car elle est triviale.

On considère une structure de marquage à deux places s_1, s_2 et le marquage initial $M = \{s_1 \mapsto \{1, 2\}, s_2 \mapsto \emptyset\}$. La fonction *init* correspondante est présentée sur la figure 4.2. Une version plus efficace consisterait à calculer la représentation mémoire du marquage initial à la compilation et ne faire qu'une copie d'un vecteur de bits, cependant cette approche sort de la sémantique présentée ici. De plus, cette fonction n'est appelée qu'une seule fois au début de l'exploration, l'optimisation n'aurait donc pas d'impact notable.

La fonction *succ* renvoie l'ensemble de tous les marquages successeurs d'un marquage. Intuitivement elle appelle toutes les fonctions $succ_t$ en agrégeant les marquages découverts dans un même ensemble. L'algorithme général de la fonction *succ* pour un réseau à n transitions $t_1, \ldots, t_n$ est présenté sur la figure 4.3. Le passage de l'algorithme 3.3 au code LLVM est évident.

$$P(init(\!|\,|\!)) \stackrel{\text{df}}{=} \begin{cases} \textit{// marquage vide} \\ x_{mrk} = fcall\ cons_{mrk}() \\ \textit{// ajout des jeton 1 et 2 à } s_1 \\ x_{s_1} = fcall\ get_{s_1}(x_{mrk}) \\ pcall\ add_{s_1}(x_{s_1}, 1) \\ pcall\ add_{s_1}(x_{s_1}, 2) \\ \textit{// renvoi du marquage produit} \\ ret\ x_{mrk} \end{cases}$$

FIGURE 4.2 – *Un exemple de fonction init.*

$$P(succ(\!|\, x_{mrk} \,|\!)) \stackrel{\text{df}}{=} \begin{cases} x_{next} = fcall\ cons_{set}() \\ pcall\ succ_{t_1}(x_{mrk}, x_{next}) \\ pcall\ succ_{t_2}(x_{mrk}, x_{next}) \\ \dots \\ pcall\ succ_{t_n}(x_{mrk}, x_{next}) \\ ret\ x_{next} \end{cases}$$

FIGURE 4.3 – *Calcul de tous les successeurs d'un marquage donc la structure est pointé par* x_{mrk}.

4.4.2 Tirage de transitions

Le tirage de transitions est réalisé en produisant une fonction de tir par transition. Soit $t \in T$ une transition telle que ${}^\bullet t = \{s_1, \dots, s_n\}$ et $t^\bullet = \{s'_1, \dots, s'_m\}$. La fonction $fire_t$ calcule, pour un marquage M et un mode de la transition t, le marquage successeur M' résultant du tir de t. Le schéma général de cette fonction est présenté en figure 4.4 et est une traduction de l'algorithme 3.1. Cette fonction crée une copie M' du marquage M, consomme les jetons des arcs en entrée correspondants aux variables $x_1, \dots, x_n$ et produit les jetons en utilisant les arcs sortants des transitions à l'aide des annotations $f_{t,s'_1}, \dots, f_{t,s'_m}$.

4.4.3 Calcul des Successeurs

La découverte de tous les modes d'une transition t est réalisée à l'aide d'une fonction $succ_t$ spécifique à la transition t. L'algorithme réalise une énumération de toutes les combinaisons de jetons possibles pour les places en entrée puis, si la garde de la transition est satisfaite, appelle une fonction $fire_t$ spécifique à la transition qui va réaliser le tir effectif. Le marquage obtenu par le tir de la transition est ensuite ajouté à un ensemble de marquages. Ceci est réalisé par l'algorithme 3.2 et sa réalisation en LLVM est présentée en figure 4.5.

Remarquons que la garde g_t est réalisée dans le langage cible cela évite une interprétation de l'expression correspondante et permet de l'exécuter de

$$
\begin{array}{l}
P(\mathit{fire}_t(\!| x_{mrk}, x_1, \ldots, x_n |\!)) \stackrel{\text{df}}{=} \\
\left\{\begin{array}{l}
\quad \textit{// copier la structure de marquage} \\
x'_{mrk} = \mathit{fcall}\ \mathit{copy}_{mrk}(x_{mrk}) \\
\quad \textit{// consommer les jetons} \\
\left.\begin{array}{l} x_{s_i} = \mathit{fcall}\ \mathit{get}_{s_i}(x'_{mrk}) \\ \mathit{pcall}\ \mathit{rem}_{s_i}(x_{s_i}, x_i) \end{array}\right\}_{1 \le i \le n} \\
\quad \textit{// produire les jetons} \\
\left.\begin{array}{l} x_{s'_j} = \mathit{fcall}\ \mathit{get}_{s'_j}(x'_{mrk}) \\ o_{s'_j} = \mathit{fcall}\ f_{t,s'_j}(x_1, \ldots, x_n) \\ \mathit{pcall}\ \mathit{add}_{s'_j}(x_{s'_j}, o_{s'_j}) \end{array}\right\}_{1 \le j \le m} \\
\quad \textit{// renvoyer le nouveau marquage} \\
\mathit{ret}\ x'_{mrk}
\end{array}\right.
\end{array}
$$

Figure 4.4 – *Implémentation en LLVM de l'algorithme du tir de transition. Chaque variable x_i est $\ell(s_i, t)$ pour $1 \le i \le n$ et chaque f_{t,s'_j} est la fonction $\ell(t, s'_j)$ pour $1 \le j \le m$. Les accolades à droite indiquent des séquences produites plusieurs fois pour différentes valeurs de i ou j.*

manière native sans recourir à une machine virtuelle ou un interpréteur.

Dans la réalisation LLVM de l'algorithme, x_{mrk} est un pointeur sur la structure de marquage et x_{next} est un pointeur sur l'ensemble des marquages successeurs. Chaque itération valuant x_k est implémentée par un ensemble de blocs annotés par t, k (pour $n \ge k \ge 1$) ; les blocs annotés par $t, 0$ sont les blocs correspondants aux corps de la boucle la plus interne. Notons l'utilisation de l'instruction *phi* pour mettre à jour les indices i_{s_i} (utilisés pour énumérer les jetons des places s_i) : lorsqu'on entre dans le bloc $\mathit{loop}_{t,i}$ pour la première fois, on vient du bloc $\mathit{header}_{t,i}$, on initialise la valeur de i_{s_i} au dernier index du domaine de s_i (mémorisé dans la variable s_{s_i}) ; par la suite, le programme revient dans le bloc $\mathit{loop}_{t,i}$ à partir du bloc $\mathit{footer}_{t,i}$, et donc on assigne i'_{s_i} à i_{s_i} qui correspond à la valeur $i_{s_i} - 1$ (*i.e.*, l'index précédent).

4.5 Sémantique formelle de LLVM

Dans cette section on définit une sémantique formelle du fragment LLVM qu'on utilise. Pour cela on définit un modèle mémoire, puis un ensemble de règles d'inférences décrivant la sémantique opérationnelle, finalement on définit la notion d'interprétation des objets LLVM en termes de réseaux de Petri, ce qui est nécessaire pour les spécifications des interfaces et des algorithmes.

4.5.1 Modèle mémoire

Avant de donner une sémantique formelle à notre fragment LLVM, nous devons nous munir d'un modèle mémoire, cela inclut : des tas pour stocker les données allouées de manière dynamique (accessibles par les pointeurs), des piles pour stocker les variables locales et arguments de sous programmes.

$$
\begin{aligned}
P(succ_t(\!| x_{mrk}, x_{next} |\!)) &\stackrel{\text{df}}{=} \{\ br\ header_{t,n} \\
P(header_{t,k}) &\stackrel{\text{df}}{=} \left\{ \begin{array}{l} x_{s_k} = fcall\ get_{s_k}(x_{mrk}) \\ s_{s_k} = fcall\ size_{s_k}(x_{s_k}) \\ br\ loop_{t,k} \end{array} \right. \\
P(loop_{t,k}) &\stackrel{\text{df}}{=} \left\{ \begin{array}{l} i_{s_k} = phi\ (s_{s_k}, header_{t,k}), (i'_{s_k}, footer_{t,k}) \\ c_{s_k} = icmp\ >,\ i_{s_k},\ 0 \\ br\ c_{s_k},\ body_{t,k},\ footer_{t,k+1} \end{array} \right. \\
P(body_{t,k}) &\stackrel{\text{df}}{=} \left\{ \begin{array}{l} x_k = fcall\ nth_{s_k}(x_{s_k}, i_{s_k}) \\ br\ header_{t,k-1} \end{array} \right. \\
P(footer_{t,k}) &\stackrel{\text{df}}{=} \left\{ \begin{array}{l} i'_{s_k} = add\ i_{s_k},\ -1 \\ br\ loop_{t,k} \end{array} \right. \\
P(header_{t,0}) &\stackrel{\text{df}}{=} \left\{ \begin{array}{l} c_g = fcall\ g_t(x_1, \ldots, x_n) \\ br\ c_g,\ body_{t,0},\ footer_{t,1} \end{array} \right. \\
P(body_{t,0}) &\stackrel{\text{df}}{=} \left\{ \begin{array}{l} x'_{mrk} = fcall\ fire_t(x_{mrk}, x_1, \ldots, x_n) \\ pcall\ add_{set}(x_{next}, x'_{mrk}) \\ br\ footer_{t,1} \end{array} \right. \\
P(footer_{t,n+1}) &\stackrel{\text{df}}{=} \{\ ret
\end{aligned}
$$

FIGURE 4.5 – *Fonction successeur spécifique à une transition t, pour $1 \leq k \leq n$.*

DEFINITION 4.1. *Un tas est une fonction partielle $H : \mathbb{P} \twoheadrightarrow \mathbb{T} \times (\mathbb{D} \cup \mathbb{P} \cup \{\bot\})^\star$ avec domaine fini. Chaque tas assigne à chaque pointeur dans son domaine un type et un tuple de valeurs ou pointeurs. L'ensemble de tous les tas est $\mathbb{H}$.* ◊

DEFINITION 4.2. *Un tas H est bien formé pour un couple (t, p) si la valeur p est consistante avec t dans H, plus précisément :*

- *H est bien formé pour (t, p) tels que $t \in \mathbb{T}_0$ si p est $\bot$ ou une valeur de type t, par exemple si t est int alors p est un entier ou non défini ;*
- *H est bien formé pour $(struct(t_0, \ldots, t_n), d)$ si d est un tuple $(p_0, \ldots, p_n)$ et H est bien formé pour chaque (t_i, p_i) avec $0 \leq i \leq n$;*
- *H est bien formé pour $(ptr(t), p)$ tels que $t \in \mathbb{T}$ si $p = \bot$ ou $H(p) = (t, p')$ et H est bien formé pour (t, p') ;*

Un tas H est bien formé si H est bien formé pour tout couple (t, p) de son domaine image. ◊

Remarquons que de part notre définition inductive de types (section 4.1.1) un tas bien formé n'a pas de pointeurs cycliques, et par conséquent les types récursifs sont interdits.

Definition 4.3. *L'ensemble de tous les pointeurs accessibles à partir d'un pointeur p dans un tas H est dénoté par $p{\downarrow_H}$ et défini pour tout p dans $\mathbb{P}$ par :*

$$\begin{array}{ll} p{\downarrow_H} \stackrel{\text{df}}{=} \{\} & \text{si } p \notin dom(H) \\ p{\downarrow_H} \stackrel{\text{df}}{=} \{p\} & \text{si } H(p) = (t, v) \text{ et } t \in \mathbb{T}_0 \\ p{\downarrow_H} \stackrel{\text{df}}{=} \{p\} \cup p_0{\downarrow_H} & \text{si } H(p) = (ptr(t), p_0) \\ p{\downarrow_H} \stackrel{\text{df}}{=} \{p\} \cup p_0{\downarrow_H} \cup \cdots \cup p_n{\downarrow_H} & \text{si } H(p) = (struct(t_0, \ldots, t_n), (p_0, \ldots, p_n)) \end{array}$$

◊

On peut montrer que si un tas H est bien formé alors $p{\downarrow_H} \subseteq dom(H)$ pour tout $p \in dom(H)$, et plus généralement que $dom(H) = \bigcup_{p \in \mathbb{P}} p{\downarrow_H}$.

Proposition 4.1. *Soit H un tas bien formé. Si $p \in dom(H)$ alors $p{\downarrow_H} \subseteq dom(H)$* □

Pour parcourir, ajouter et mettre à jour des données dans un tas, on définit une famille de fonctions de parcours et d'écriture.

Definition 4.4. *Pour chaque tas H, on a une fonction de parcours $\cdot[\cdot]_H$. Chaque fonction $\cdot[\cdot]_H$ est une fonction partielle $\cdot[\cdot]_H : \mathbb{P} \times \mathbb{N} \nrightarrow \mathbb{P} \cup \mathbb{D}$ définie par :*

$$p[i]_H \stackrel{\text{df}}{=} \begin{cases} p_i \text{ si } H(p) = (struct(t_0, \ldots, t_n), (p_0, \ldots, p_n)) \text{ et } 0 \leq i \leq n \\ \text{non défini sinon} \end{cases}$$

◊

Definition 4.5. *L'écriture (ou écrasement) de données dans le tas est réalisé avec la fonction $\oplus : \mathbb{H} \times \mathbb{H} \to \mathbb{H}$. L'objet calculé est un tas défini pour tout $p \in \mathbb{P}$ par :*

$$(H \oplus H')(p) \stackrel{\text{df}}{=} \begin{cases} H'(p) & \text{si } p \in dom(H') \\ H(p) & \text{si } p \notin dom(H') \wedge p \in dom(H) \\ \text{non défini sinon} \end{cases}$$

◊

Proposition 4.2. *L'opération d'écrasement $\oplus$ est associative.* □

Remarquons que cette opération est clairement non commutative comme la notation $\oplus$ pourrait le suggérer.

Pour comparer les tas, une équivalence structurelle est définie. Cette relation permet de vérifier que deux tas contiennent les mêmes données, accessibles par différents pointeurs mais avec la même organisation. Plus précisément, pour deux tas H,H' et deux pointeurs p,p', on note $(H, p) =_{st} (H', p')$ si $H(p)$ et $H'(p')$ sont structurellement équivalents.

Definition 4.6 (équivalence structurelle). *Soit H et H' deux tas bien formés, v et v' deux valeurs ou pointeurs. On a $(H, v) =_{st} (H', v')$ si $v \notin dom(H)$, $v' \notin dom(H')$ et $v = v'$; ou bien si :*

- *$H(v) = (struct(t_0, \ldots, t_n), (p_0, \ldots, p_n))$;*
- *$H'(v') = (struct(t_0, \ldots, t_n), (p'_0, \ldots, p'_n))$; et*
- *$(H, p_i) =_{st} (H', p'_i)$ pour $0 \leq i \leq n$.* ◊

PROPOSITION 4.3. *$=_{st}$ est une relation d'équivalence.* □

On définit aussi une opération d'*allocation mémoire* sur les tas. Dans notre cas, on se place d'un point de vue fonctionnel qui consiste à renvoyer un nouveau tas à chaque allocation mémoire. On appelle cette fonction *new*, elle prend en argument un tas et un type puis renvoie un nouveau tas contenant la donnée allouée et un pointeur sur celle-ci.

DEFINITION 4.7. *La fonction $new : \mathbb{H} \times \mathbb{T} \to \mathbb{H} \times \mathbb{P}$ est définie par :*

$$\begin{aligned}
new(H, t) &\stackrel{\text{df}}{=} (H \cup \{p \mapsto (t, \bot)\}, p) \\
&\qquad \text{pour } t \in \mathbb{T}_0 \text{ et } p \text{ un pointeur “frais”} \\
new(H, ptr(t)) &\stackrel{\text{df}}{=} (H \cup \{p \mapsto (ptr(t), \bot)\}, p) \\
&\qquad \text{pour } t \in \mathbb{T} \text{ et } p \text{ un pointeur “frais”} \\
new(H, struct(t_0, \ldots, t_n)) &\stackrel{\text{df}}{=} (H \cup \{p \mapsto (struct(t_0, \ldots, t_n), (\bot, \ldots, \bot))\}, p) \\
&\qquad \text{pour } t_0, \ldots, t_n \in \mathbb{T} \text{ et } p \text{ un pointeur “frais”}
\end{aligned}$$

◊

On peut montrer que *new* renvoie toujours un tas bien formé, et que appeler *new* de la même manière avec deux tas équivalents produit deux tas équivalents.

PROPOSITION 4.4. *Soit H un tas bien formé, et t un type. Si on a $(H', p) = new(H, t)$ alors H' est un tas bien formé.* □

PROPOSITION 4.5. *Soient H_1, H_2 deux tas et t un type. Dans ce contexte, on a $new(H_1, t) =_{st} new(H_2, t)$* □

Pour définir les appels aux sous-programmes, notre modèle mémoire a aussi besoin de définir des *piles* qui contiennent des *frames* qui seront empilés et dépilés de manière implicite par des règles d'inférence dans la sémantique.

DEFINITION 4.8. *Un frame est un tuple $F \in \mathbb{F} \stackrel{\text{df}}{=} \mathbb{L}_\bot \times \mathbb{L} \times \mathbb{B}$. Pour chaque frame $(l_{p,F}, l_{c,F}, \beta_F)$ on a :*

- *la première composante $l_{c,F}$ est l'étiquette du bloc dont on vient, elle est non définie au début du programme ;*
- *la seconde composante $l_{c,F}$ est l'étiquette du bloc courant, le bloc qu'on exécute actuellement ;*
- *finalement, la dernière composante β_F est une LLVM-valuation correspondante au contexte courant d'évaluation.*

On élargit la notation fonctionnelle des valuations aux frames pour noter $F(x)$ l'application de x à la valuation β_F, c'est-à-dire $\beta_F(x)$. ◊

Pour alléger les notations on va toujours noter pour un frame F : $l_{p,F}$ sa première composante, $l_{c,F}$ sa deuxième composante et β_F sa troisième composante.

De la même manière que pour les tas nous avons besoin de lire la pile et mettre à jour les frames. On réutilise donc l'opérateur $\oplus$ pour cette opération puisqu'elle est similaire à la mise à jour des tas.

DEFINITION 4.9. *L'écriture (ou écrasement) pour les valuations est réalisée par la fonction $\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$, la même opération mais pour les frames est*

réalisée par la fonction $\oplus : \mathbb{F} \times \mathbb{B} \to \mathbb{F}$. *Ces deux fonctions sont définies par :*

$$(\beta \oplus \beta')(p) \stackrel{\text{df}}{=} \begin{cases} \beta'(p) & si \quad p \in dom(\beta') \\ \beta(p) & si \quad p \notin dom(\beta') \wedge p \in dom(\beta) \\ non\ défini\ sinon \end{cases}$$

$$(l,\ l',\ \beta) \oplus \beta' \quad \stackrel{\text{df}}{=} \quad (l,\ l',\ \beta \oplus \beta')$$

$\Diamond$

L'équivalence structurelle (def. 4.6) peut être étendue aux paires de tas et frames, on note cette relation par $(H, F) =_{st} (H', F')$ pour deux tas H, H' et deux frames F, F'. Intuitivement, on vérifie que toutes les données accessibles depuis les valuations des frames sont structurellement équivalentes. Cela doit aussi être vrai pour les valeurs stockés directement dans les valuations qui ne sont pas des pointeurs, puisque l'équivalence structurelle de tas correspond à l'égalité sur $\mathbb{D}$.

Definition 4.10. *Soient* H, H' *deux tas et soient* F, F' *deux frames. On a* $(H, F) =_{st} (H', F')$ *si et seulement si :*

- *pour tout* $x \in dom(\beta_F)$, *on a* $x \in dom(\beta_{F'})$ *et* $(H, x) =_{st} (H', x)$;
- *pour tout* $x' \in dom(\beta_{F'})$, *on a* $x' \in dom(\beta_F)$ *et* $(H, x') =_{st} (H', x')$. $\Diamond$

Proposition 4.6. *La relation* $=_{st}$ *avec frames est une relation d'équivalence.* $\square$

4.5.2 Règles d'inférence

La sémantique opérationnelle est définie pour un programme P fixé et immuable. Cela veut dire que aucune fonction, ni bloc ne peuvent être créés ou modifiés pendant l'exécution. On dénote le résultat d'un calcul par $\overline{\cdot}$, par exemple $\overline{2+3}$ est 5. Les objets le plus manipulés dans notre système de règles d'inférences sont des configurations qui représentent un état du programme pendant son exécution.

Definition 4.11. *Une configuration est un tuple* (seq, H, F), *qu'on note par* $\left(\ seq\ \right)_{H,F}$, *où seq est une séquence d'instruction,* H *un tas et* F *un frame.* $\Diamond$

Les règles d'inférence pour les expressions sont présentées à la figure 4.6 et sont évaluées comme des valeurs dans le contexte d'un frame. Les règles d'inférence pour les séquences et les commandes sont données sur la figure 4.7, les séquences et les commandes sont évaluées vers d'autres séquences et commandes dans le contexte d'un tas et d'un frame.

On peut remarquer comment les frames sont empilés sur la pile dans les règles *pcall* et *fcall*, un nouveau frame F_0 remplace le frame courant F dans les prémisses de ces règles. Ce nouveau frame est utilisé pour exécuter le corps du sous-programme appelé. La sémantique mélange les réductions à petit- et grand-pas. En effet, la plupart des règles sont définies avec des réductions à petit-pas sauf les règles *pcall* et *fcall* où l'appel du sous-programme est directement lié au résultat du calcul de son corps, ce qui est réalisée par une séquence de réduction dans la prémisse de la règle.

$$\frac{F(x) = p \quad H(p) = (t, v)}{\left(\ load\ x\ \right)_{H,F} \rightsquigarrow \left(\ v\ \right)_H}\ load \qquad \frac{F(x) = p \quad p[i]_H \text{ défini}}{\left(\ gep\ x,\ 0,\ i\ \right)_{H,F} \rightsquigarrow \left(\ p[i]_H\ \right)_H}\ gep_0$$

$$\frac{\overline{F(x_1) + F(x_2)} = v}{\left(\ add\ x_1,\ x_2\ \right)_{H,F} \rightsquigarrow \left(\ v\ \right)_H}\ add \qquad \frac{(H', p) = new(H, t)}{\left(\ alloc\ t\ \right)_{H,F} \rightsquigarrow \left(\ p\ \right)_{H \oplus H'}}\ alloc$$

$$\frac{\overline{F(x_1)\ op\ F(x_2)} = v \quad op \in \{<, \leq, =, \neq, \geq, >\}}{\left(\ icmp\ op,\ x_1,\ x_2\ \right)_{H,F} \rightsquigarrow \left(\ v\ \right)_H}\ icmp$$

$$\frac{1 \leq i \leq n \quad l_i \neq \bot}{\left(\ phi\ (x_1, l_1), \ldots, (x_n, l_n)\ \right)_{H,(l_i,\ l_c,\ \beta)} \rightsquigarrow \left(\ \beta(x_i)\ \right)_H}\ phi$$

$$\frac{\begin{array}{c} f(\!|\, a_1, \ldots, a_n \,|\!) \in dom(P) \\ F_0 = (\bot,\ f(\!|\, a_1, \ldots, a_n \,|\!),\ \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \left(\ P(f(\!|\, a_1, \ldots, a_n \,|\!))\ \right)_{H,F_0} \rightsquigarrow^* \left(\ v\ \right)_{H'} \end{array}}{\left(\ fcall\ f(r_1, \ldots, r_n)\ \right)_{H,F} \rightsquigarrow \left(\ v\ \right)_{H'}}\ fcall$$

$$\frac{}{\left(\ ret\ r\ \right)_{H,F} \rightsquigarrow \left(\ F(r)\ \right)_H}\ ret \qquad \frac{\text{r constante ou variable}}{\left(\ r\ \right)_{H,F} \rightsquigarrow \left(\ F(r)\ \right)_H}\ rvalue$$

FIGURE 4.6 – *Règles pour les expressions.*

$$\frac{\left(\ cmd\ \right)_{H,F} \rightsquigarrow \left(\ seq'\ \right)_{H',F'}}{\left(\ cmd;\ seq\ \right)_{H,F} \rightsquigarrow \left(\ seq';\ seq\ \right)_{H',F'}}\ seq$$

$$\frac{}{\left(\ skip;\ seq\ \right)_{H,F} \rightsquigarrow \left(\ seq\ \right)_{H,F}}\ skip$$

$$\frac{}{\left(\ br\ l\ \right)_{H,(l_p,\ l_c,\ \beta)} \rightsquigarrow \left(\ P(l)\ \right)_{H,(l_c,\ l,\ \beta)}}\ branch_1$$

$$\frac{(\beta(r) = true \wedge l = l_1) \vee (\beta(r) = false \wedge l = l_2)}{\left(\ br\ r,\ l_1,\ l_2\ \right)_{H,(l_p,\ l_c,\ \beta)} \rightsquigarrow \left(\ P(l)\ \right)_{H,(l_c,\ l,\ \beta)}}\ branch_2$$

$$\frac{\begin{array}{c} f(\!|\, a_1, \ldots, a_n \,|\!) \in dom(P) \\ F_0 = (\bot,\ f(\!|\, a_1, \ldots, a_n \,|\!),\ \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \left(\ P(f(\!|\, a_1, \ldots, a_n \,|\!))\ \right)_{H,F_0} \rightsquigarrow^* \left(\ ret\ \right)_{H',F'} \end{array}}{\left(\ pcall\ f(r_1, \ldots, r_n)\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H',F}}\ pcall$$

$$\frac{\left(\ expr\ \right)_{H,F} \rightsquigarrow \left(\ v\ \right)_{H'}}{\left(\ x = expr\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H',F \oplus \{x \mapsto v\}}}\ assign$$

$$\frac{F(r_p) = p \quad H(p) = (t, d) \quad H' = \{p \mapsto (t, F(r_{new}))\}}{\left(\ store\ r_{new},\ r_p\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H \oplus H', F}}\ store$$

FIGURE 4.7 – *Règles pour les séquences et les commandes.*

Exemple d'utilisation de la sémantique

Pour mettre en pratique les règles d'inférence de la sémantique on considère le programme suivant.

$$\begin{array}{ll} P(f(\!|\, x_1 \,|\!)) \stackrel{\text{df}}{=} & P(l) \stackrel{\text{df}}{=} \\ \quad x_2 = icmp\ <,\ x_1,\ 2 & \quad x_3 = 1 \\ \quad ret\ x_2 & \quad x_4 = f(\!|\, x_3 \,|\!) \end{array}$$

On considère l'étiquette l comme point d'entrée du programme. Le programme consiste à affecter une constante à une variable puis appeler la fonction f qui teste si la valeur de cette constante est inférieure à 2, puis renvoie la valeur de ce test.

Pour notre réduction on considère que le contexte initial est vide, c'est-à-dire que le tas est $\emptyset$ et que le frame est $(\bot, l, \emptyset)$, on fixe l'étiquette courante à l car c'est le bloc courant. On commence avec $P(l) \stackrel{\text{df}}{=} (x_3 = 1; x_4 = f(\!|\, x_3 \,|\!))$ qui est le code du bloc l. L'ensemble de règles étant dirigé par la syntaxe, on n'a d'autres choix que de commencer la réduction en appliquant la règle *seq*.

$$\dfrac{\dfrac{\dfrac{}{(\ 1\ )_{\emptyset,(\bot,l,\emptyset)} \rightsquigarrow (\ 1\ )_{\emptyset}}\ rvalue}{(\ x_3 = 1\ )_{\emptyset,(\bot,l,\emptyset)} \rightsquigarrow (\ skip\ )_{\emptyset,(\bot,l,\{x_3 \mapsto 1\})}}\ assign}{(\ x_3 = 1; x_4 = f(\!|\, x_3 \,|\!)\ )_{\emptyset,(\bot,l,\emptyset)} \rightsquigarrow (\ skip; x_4 = f(\!|\, x_3 \,|\!)\ )_{\emptyset,(\bot,l,\{x_3 \mapsto 1\})}}\ seq$$

Pour réaliser cette réduction on doit aussi appliquer la règle *assign* en prémisse, qui à son tour a besoin de la règle *rvalue* pour traiter la constante. Grâce à la règle *assign* la valuation du frame est enrichie de l'assignation $x_3 \mapsto 1$.

On continue la réduction avec les règles *skip* et *assign*.

$$\dfrac{}{(\ skip; x_4 = f(\!|\, x_3 \,|\!)\ )_{\emptyset,(\bot,l,\{x_3 \mapsto 1\})} \rightsquigarrow (\ x_4 = f(\!|\, x_3 \,|\!)\ )_{\emptyset,(\bot,l,\{x_3 \mapsto 1\})}}\ skip$$

$$\dfrac{\dfrac{(\ P(f(\!|\, x_1 \,|\!))\ )_{\emptyset,(\bot,l,\{x_1 \mapsto 1\})} \rightsquigarrow^* (\ false\ )_{\emptyset}}{(\ f(\!|\, x_3 \,|\!)\ )_{\emptyset,(\bot,l,\{x_3 \mapsto 1\})} \rightsquigarrow (\ false\ )_{\emptyset}}\ fcall}{(\ x_4 = f(\!|\, x_3 \,|\!)\ )_{\emptyset,(\bot,l,\{x_3 \mapsto 1\})} \rightsquigarrow (\ skip\ )_{\emptyset,(\bot,l,\{x_3 \mapsto 1, x_4 \mapsto false\})}}\ assign$$

Lors de la réduction avec la règle *assign* on doit réaliser une réduction avec la règle *pcall* qui correspond à un appel de sous-programme. Cet appel de sous-programme se réduit à *false* dans le contexte d'un tas vide, cela est présenté par les réductions qui suivent.

On commence la réduction du sous programme $P(f(\!|\, x_1 \,|\!))$ dans le contexte du même tas, mais d'un nouveau frame $F_0 = (\bot, f(\!|\, x_1 \,|\!), \{x_1 \mapsto 1\})$. La valeur de x_1 a été obtenue grâce à la valuation $\{x_1 \mapsto 1\}$ contenue dans le frame utilisé pour l'appel de fonction.

Le premier pas de réduction du sous programme est la règle *seq*, de manière générale on utilise toujours cette règle lorsque le bloc courant a plus d'une instruction. Cette réduction a besoin de la règle *assign* qui à son tour a besoin de la règle *icmp*. Cette dernière, récupère la valeur de x_1 à partir du frame et

la compare avec 2 (on rappelle que la valuation d'un frame est l'identité pour les constantes).

$$\dfrac{\dfrac{\dfrac{\overline{F(x_1) < F(2) = false}}{\left(\ icmp\ <,\ x_1,\ 2\ \right)_{\emptyset,F_0} \rightsquigarrow \left(\ false\ \right)_{\emptyset}}\ icmp}{\left(\ x_2 = icmp\ <,\ x_1,\ 2\ \right)_{\emptyset,F_0} \rightsquigarrow \left(\ skip\ \right)_{\emptyset,F_0\oplus\{x_2\mapsto false\}}}\ assign}{\left(\ x_2 = icmp\ <,\ x_1,\ 2; ret\ x_2\ \right)_{\emptyset,F_0} \rightsquigarrow \left(\ skip; ret\ x_2\ \right)_{\emptyset,F_0\oplus\{x_2\mapsto false\}}}\ seq$$

Pour finir la réduction on applique les règles *skip* et *ret* sur le redex du sous-programme.

$$\dfrac{}{\left(\ skip; ret\ x_2\ \right)_{\emptyset,F_0\oplus\{x_2\mapsto false\}} \rightsquigarrow \left(\ ret\ x_2\ \right)_{\emptyset,F_0\oplus\{x_2\mapsto false\}}}\ skip$$

$$\dfrac{}{\left(\ ret\ x_2\ \right)_{\emptyset,F_0} \rightsquigarrow \left(\ false\ \right)_{\emptyset}}\ ret$$

Finalement, on arrive à la valeur *false* dans le contexte d'un tas vide : l'utilisation de la règle *fcall* ci-dessus était légale.

4.5.3 Interprétation des structures de données

Le lien entre le réseau de Petri et son implémentation en LLVM est formalisé avec une famille de fonctions d'*interprétation* pour toutes les structures de données. Cela permet d'une part de faire un lien entre les deux mondes, nécessaire aux preuves de correction, et d'autre part de formaliser les prérequis comportementaux des interfaces présentées dans la section 4.6.

Definition 4.12. *Une interprétation est une fonction partielle qui associe une paire formée d'un tas et d'un pointeur à un objet de l'univers des réseaux de Petri : un marquage, un ensemble de marquages, un multiensemble de jetons, un jeton. Les interprétations sont notées* $[\![H, p]\!]^\star$*, où* $H \in \mathbb{H}$*,* $p \in \mathbb{P} \cup \mathbb{D}$ *et* $\star$ *est une annotation qui décrit l'objet interprété. De plus chaque fonction d'interprétation doit respecter les deux prérequis 4.1 et 4.2.* ◊

Prérequis 4.1. *Soit* $[\![\cdot,\cdot]\!]^\star$ *une fonction d'interprétation,* H *un tas, et* p *un pointeur. Le résultat de l'interprétation* $[\![H, p]\!]^\star$ *dépend seulement des données accessibles à partir de* p*,* i.e.*,* $p{\downarrow_H}$*.* ◊

Prérequis 4.2. *Soit* $[\![\cdot,\cdot]\!]^\star$ *une fonction d'interprétation,* H, H' *deux tas, et* p, p' *deux pointeurs ou valeurs. Si* $(H, p) =_{st} (H', p')$ *alors* $[\![H, p]\!]^\star = [\![H', p']\!]^\star$*.* ◊

Il faut retenir de la définition 4.12 que le résultat d'une fonction d'interprétation dépend du contexte dans lequel elle est utilisée. Par exemple, si on utilise l'annotation $mset(t)$ à la place de $\star$ alors l'interprétation $[\![H, p]\!]^{mset(t)}$ renvoie un multiensemble d'objets de type t, pour un pointeur p et un tas H. Pour cela il faut évidemment que le pointeur p pointe un multiensemble d'objets de type t dans le tas H.

Comme présenté dans la section 4.4, on utilise des structures de données et des fonctions comme blocs de base pour construire nos algorithmes. Ils sont

soit prédéfinis soit produits par le processus de compilation. Chacune de ces fonctions et structures de données est spécifiée (plus précisément, axiomatisée) par une interface formelle. En particulier, cela permet de rester indépendant et modulaire vis-à-vis des composants à la fois sur le plan formel mais aussi par rapport à l'implémentation. Spécifier une interface mène à la définition d'un ensemble de primitives qui respectent des dérivations et des interprétations dans notre sémantique. Par exemple, si H est un tas et F un frame, tels que $F(x_{mset}) = p_{mset}$ est un pointeur sur une structure de multiensemble qui stocke des éléments de type t, alors la procédure add_{mset} est spécifiée par :

$$\left(\ pcall\ add_{mset}(x_{mset}, x)\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H \oplus H', F}$$

$$dom(H) \cap dom(H') \subseteq p_{mset}{\downarrow_H}$$

$$[\![H \oplus H', p_{mset}]\!]^{mset(t)} = [\![H, p_{mset}]\!]^{mset(t)} + \{[\![H, F(x)]\!]^t\}$$

La première condition décrit le résultat de la réduction de l'appel à la procédure, la deuxième condition est une restriction sur les modifications du tas limitant les effets de bord, et la troisième condition interprète le calcul en termes d'objets de l'univers des réseaux de Petri.

Les différentes implémentations des structures de marquage quant à elles doivent respecter les deux prérequis 4.3 et 4.4.

Prérequis 4.3 (cohérence). *Soient H, H' deux tas bien formés, F, F' deux frames, $p_{mrk} \in dom(H)$ un pointeur vers une structure de marquage et p_s un pointeur sur une place accessible à partir de p_{mrk}* (i.e., $p_s \in p_{mrk}{\downarrow_H}$). *Si $[\![H, p_{mrk}]\!]^{mrk}(s) = [\![H, p_s]\!]^s$, $\left(\ seq\ \right)_{H,F} \rightsquigarrow \left(\ seq'\ \right)_{H \oplus H', F'}$ et $p_{mrk} \notin dom(H')$ alors*

$$[\![H \oplus H', p_{mrk}]\!]^{mrk}(s) = [\![H \oplus H', p_s]\!]^s$$

◊

Prérequis 4.4 (séparation). *Soit p_{mrk} un pointeur sur une structure de marquage dans un tas H bien formé. Si p_s et p'_s sont des pointeur sur des places distinctes dans cette structure alors on a $p_s{\downarrow_H} \cap p_{s'}{\downarrow_H} = \emptyset$* ◊

Le prérequis 4.3 assure que chacune des modifications d'une place s à travers un pointeur p_s renvoyé par get_s sera reflétée sur la structure de marquage (et non une copie). Le prérequis 4.4 assure que les places dans une structure de marquage ne partagent pas de mémoire, par conséquent la modification d'une place ne modifiera aucune autre place de cette structure. Ce dernier prérequis permet donc de contrôler les effets de bord sur les modifications de places. L'ensemble des spécifications est donnée dans la section 4.6. Une réalisation complète d'une structure de marquage est donné en annexe A.1.

4.6 Interfaces formelles

Cette section détaille l'ensemble des interfaces utilisées ainsi que leur axiomatisation. On aborde les multiensembles, les places, les marquages et les ensembles de marquages. Dans l'ensemble de la section on pose F un frame et

H un tas. Toutes les spécifications sont données dans des contextes minimaux. Elles pourront être généralisées par la suite grâce au théorème d'extensionnalité présenté dans la section 4.7.

4.6.1 Interface des multiensembles

On suppose qu'on dispose d'une implémentation correcte de multiensembles pour tout type de données t d'interprétation $[\![\cdot,\cdot]\!]^t$. Chaque structure de multiensemble définie pour le type t a pour type $mset(t)$ et la fonction d'interprétation de cette structure sera notée $[\![\cdot,\cdot]\!]^{mset(t)}$. Soit x_{mset} une variable, p_{mset} un pointeur sur une structure de multiensemble de type $mset(t)$ et x une variable ou une valeur telles que $F(x_{mset}) = p_{mset}$ et $F(x)$ soit de type t. Chacune des fonctions de l'interface est spécifié comme suit :

1. La fonction $cons_{mset}$ construit un multiensemble vide :

$$\left(\ fcall\ cons_{mset}()\ \right)_{\emptyset,F} \rightsquigarrow \left(\ p_{mset}\ \right)_H$$

$$[\![H, p_{mset}]\!]^{mset(t)} = \emptyset$$

2. La procédure add_{mset} ajoute un élément au multiensemble :

$$\left(\ pcall\ add_{mset}(x_{mset}, x)\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H\oplus H',F}$$

$$dom(H) \cap dom(H') \subseteq p_{mset}{\downarrow_H}$$

$$[\![H \oplus H', p_{mset}]\!]^{mset(t)} = [\![H, p_{mset}]\!]^{mset(t)} + \{[\![H, F(x)]\!]^t\}$$

   La deuxième équation décrit le comportement de la fonction sur le tas, tous les pointeurs sur les données modifiées sont localisés dans $p_{mset}{\downarrow_H}$, et donc seule la structure de multiensemble a donc été modifiée, le reste du tas demeure inchangé.

3. La procédure rem_{mset} supprime un élément du multiensemble :

$$\left(\ pcall\ rem_{mset}(x_{mset}, x)\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H\oplus H',F}$$

$$dom(H) \cap dom(H') \subseteq p_{mset}{\downarrow_H}$$

$$[\![H \oplus H', p_{mset}]\!]^{mset(t)} = [\![H, p_{mset}]\!]^{mset(t)} - \{[\![H, F(x)]\!]^t\}$$

4. La fonction $size_{mset}$ renvoie la taille du domaine du multiensemble :

$$\left(\ fcall\ size_{mset}(x_{mset})\ \right)_{H,F} \rightsquigarrow \left(\ card(dom([\![H, p_{mset}]\!]^{mset(t)}))\ \right)_H$$

5. La fonction nth_{mset} renvoie le i-ème élément du domaine du multiensemble ou, selon l'implémentation choisie, un pointeur sur cet élément (pour l'ordre arbitraire préalablement fixé) :

$$\left(\ fcall\ nth_{mset}(x_{mset}, i)\ \right)_{H,F} \rightsquigarrow \left(\ p_i\ \right)_H$$

$$[\![H, p_{mset}]\!]^{mset(t)} = \{x_1 \mapsto k_1, \ldots, x_n \mapsto k_n\}$$

$$[\![H, p_i]\!]^t = x_i$$

6. La procédure $copy_{mset}$ copie le contenu d'un premier multiensemble dans un second multiensemble, pour x'_{mset} une variable, p'_{mset} un pointeur sur une structure de multiensemble de type $mset(t)$ et $F(x'_{mset}) = p'_{mset}$:

$$\left(\ pcall\ copy_{mset}(x_{mset}, x'_{mset})\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H \oplus H', F}$$

$$dom(H) \cap dom(H') \subseteq p'_{mset}{\downarrow_H}$$

$$[\![H, p_{mset}]\!]^{mset(t)} = [\![H \oplus H', p_{mset}]\!]^{mset(t)} = [\![H \oplus H', p'_{mset}]\!]^{mset(t)}$$

7. La procédure $clear_{mset}$ prend en argument un pointeur sur une structure de multiensemble et vide ce multiensemble :

$$\left(\ pcall\ clear_{mset}(x_{mset})\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H \oplus H', F}$$

$$dom(H) \cap dom(H') \subseteq p_{mset}{\downarrow_H}$$

$$[\![H \oplus H', p_{mset}]\!]^{mset(t)} = \emptyset$$

8. La fonction $notempty_{mset}$ renvoie *true* si le multiensemble n'est pas vide et *false* sinon :

$$\left(\ fcall\ notempty_{mset}(x_{mset})\ \right)_{H,F} \rightsquigarrow \left(\ \overline{card(dom([\![H, p_{mset}]\!]^{mset(t)})) \neq \emptyset}\ \right)_H$$

4.6.2 Interface des places

Comme précisé dans la section 4.5, une place est un multiensemble de jetons et s'occupe de la façon dont ils sont stockés. On peut donc l'implémenter par un *multiensemble* et garder la même spécification. Pour différencier les places des multiensembles on annote l'interprétation d'une place s par son nom. La fonction d'interprétation de la place est donc $[\![\cdot, \cdot]\!]^s$. On note $[\![\cdot, \cdot]\!]^{t(s)}$ l'interprétation de la structure qui encode un jeton contenu dans cette place, il s'agit de la même fonction que $[\![\cdot, \cdot]\!]^t$ dans la sous-section précédente.

Remarquons que dans certains cas il est beaucoup plus intéressant de ne pas utiliser l'implémentation par défaut de multiensembles mais de fournir une implémentation spécifique, il s'agit d'une optimisation réalisée au moment de la compilation.

4.6.3 Interface des marquages

Chaque structure de marquage est un conteneur de places, son rôle est de les stocker et de permettre d'y accéder. Dans notre approche la structure de marquage délègue le stockage des jetons aux places.

Soit x_{mrk} une variable et p_{mrk} un pointeur sur une instance de la structure de marquage tels que $F(x_{mrk}) = p_{mrk}$ et sa fonction d'interprétation soit $[\![\cdot, \cdot]\!]^{mrk}$. Dans ce contexte, pour un tas H on dira que p_s est un pointeur sur une place de cette structure, s'il existe une place s d'interprétation $[\![\cdot, \cdot]\!]^s$, telle que $[\![H, p_{mrk}]\!]^{mrk}(s) = [\![H, p_s]\!]^s$ et $p_s \in p_{mrk}{\downarrow_H}$.

On dira qu'une structure de marquage est bien définie si elle fournit les éléments suivants et satisfait leur spécification.

1. Elle fournit une famille de fonctions get_s définie pour chaque place s du marquage, telles que

$$\left(\ fcall\ get_s(x_{mrk})\ \right)_{H,F} \rightsquigarrow \left(\ p_s\ \right)_H$$

$$p_s{\downarrow_H} \subset p_{mrk}{\downarrow_H}$$

$$[\![H, p_{mrk}]\!]^{mrk}(s) = [\![H, p_s]\!]^s$$

   Intuitivement, chaque fonction get_s renvoie un pointeur sur la place s de la structure de marquage ;

2. Une fonction $copy_{mrk}$ qui copie un marquage et renvoie le pointeur correspondant, telle que

$$\left(\ fcall\ copy_{mrk}(x_{mrk})\ \right)_{H,F} \rightsquigarrow \left(\ p'_{mrk}\ \right)_{H \oplus H'}$$

$$dom(H') \cap dom(H) = \emptyset$$

$$p'_{mrk} \in dom(H')$$

$$[\![H, p_{mrk}]\!]^{mrk} = [\![H \oplus H', p_{mrk}]\!]^{mrk} = [\![H \oplus H', p'_{mrk}]\!]^{mrk}$$

3. Une fonction $cons_{mrk}$ qui renvoie une structure de marquage vide, telle que

$$\left(\ fcall\ cons_{mrk}()\ \right)_{\emptyset,F} \rightsquigarrow \left(\ p_{mrk}\ \right)_H$$

$$[\![H, p_{mrk}]\!]^{mrk} = \emptyset$$

De plus chaque structure de marquage doit vérifier les deux propriétés mentionnés dans la section 4.5.3 :

- la propriété de *cohérence* qui assure que lorsqu'on fait des modifications sur une place à l'aide d'un pointeur retourné par get_s alors ces changements sont reflétés sur le marquage ;
- la propriété de *séparation* qui assure que lorsqu'on modifie une place du marquage ces modifications n'ont pas d'effet de bord sur les autres places, ce qui est garanti par l'absence de partage de mémoire entre places.

Un marquage simple peut être implémenté avec une structure qui contient des pointeurs sur des places. Le type de cette structure est $struct(t_{s_0}, \ldots, t_{s_n})$ où t_{s_i} est le type de la i-ème place (la place s_i). Cette implémentation ainsi que les preuves de validité sont données dans l'annexe A.1.

4.6.4 Interface des ensembles de marquages

Une structure pour contenir des marquages est nécessaires afin de calculer les espaces d'états. Cette structure a pour type $set(mrk)$ où mrk est le type du marquage. Comme on n'a qu'un seul type de structure de marquage dans un programme l'argument mrk est omis. L'interface que doit respecter la spécification qui suit. Elle doit fournir :

1. Une fonction $cons_{set}$ qui crée un nouvel ensemble vide de marquages et renvoie le pointeur correspondant, telle que

$$\left(\ fcall\ cons_{set}()\ \right)_{\emptyset,F} \rightsquigarrow \left(\ p_{set}\ \right)_H$$

$$[\![H, p_{set}]\!]^{set} = \emptyset$$

2. Une procédure add_{set} telle que si p_{mrk} est un pointeur sur une structure de marquage et p_{set} un pointeur sur un ensemble de marquages alors

$$\left(\ pcall\ add_{set}(set, p_{mrk})\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H\oplus H',F}$$

$$dom(H') \cap dom(H) \subseteq set{\downarrow_H}$$

$$[\![H \oplus H', set]\!]^{set} = [\![H', set]\!]^{set} = [\![H, set]\!]^{set} \cup \left\{\ [\![H, p_{mrk}]\!]^{mrk}\ \right\}$$

Intuitivement, cette fonction ajoute le marquage pointé par p_{mrk} à l'ensemble pointé par p_{set}.

4.7

Résultats théoriques

Dans cette section, on présente les résultats principaux sur la sémantique. Le théorème de correction des fonctions $fire_t$, le théorème de correction des fonctions $succ_t$, ainsi que les théorèmes de correction des fonctions *init* et *succ*. Tous ces résultats sont énoncés et prouvés dans des contextes minimaux, *i.e.*, le tas contient juste les pointeurs requis aux calculs. On peut se le permettre grâce à un résultat important sur notre sémantique : le théorème d'extensionnalité.

4.7.1 Théorème d'extensionnalité

Le théorème d'extensionnalité permet de considérer les différentes séquences de réductions dans des contextes minimaux, puis les généraliser à des contextes plus riches. L'énoncé du théorème est long parce que nous devons adresser les expressions et le commandes simultanément.

Théorème 4.7 (extensionnalité). *Soient seq, seq' des séquence d'instructions, v' une valeur, H, H' des tas et F, F' des frames. Dans ce contexte, si*

$$\left(\ seq\ \right)_{H,F} \rightsquigarrow^* \left(\ v'\ \right)_{H\oplus H'} \quad ou,\ resp.,\quad \left(\ seq\ \right)_{H,F} \rightsquigarrow^* \left(\ seq'\ \right)_{H\oplus H',F'}$$

et si pour tout tas H_0 et toute valuation β_0 telle que

(i) $dom(H) \cap dom(H_0) = \emptyset$

(ii) $dom(\beta_F) \cap dom(\beta_0) = \emptyset$

(iii) toute variable assignée dans seq (ou un de ses redex) n'est pas dans $dom(\beta_0)$

alors on peut réaliser la même réduction dans un environnement élargi par le tas H_0 et le frame β_0, i.e. *:*

$$\left(\ seq\ \right)_{H\oplus H_0,F\oplus\beta_0} \rightsquigarrow^* \left(\ v''\ \right)_{H\oplus H_0\oplus H''}$$

$$ou,\ resp.,\quad \left(\ seq\ \right)_{H\oplus H_0,F\oplus\beta_0} \rightsquigarrow^* \left(\ seq'\ \right)_{H\oplus H_0\oplus H'',F''\oplus\beta_0}$$

avec v'' une valeur, H'' un tas et F'' un frame tels que :

$$dom(H_0) \cap dom(H'') = \emptyset \tag{4.1}$$

$$\forall p \in dom(H), \quad (H', p) =_{st} (H'', p) \tag{4.2}$$

$$(H \oplus H', v') =_{st} (H \oplus H'', v'') \tag{4.3}$$

ou, respectivement, (4.1)*,* (4.2) *et*

$$(F', H') =_{st} (F'', H'') \tag{4.4}$$

$$dom(\beta_0) \cap dom(\beta_{F''}) = \emptyset \tag{4.5}$$

$$dom(\beta_{F'}) = dom(\beta_{F''}) \tag{4.6}$$

□

Intuitivement, l'équation (4.1) garantit qu'on ne modifie pas le tas H_0, c'est-à-dire que l'évaluation de la séquence n'a pas d'effets de bords. L'équation (4.2) garantit que les modifications sur H' et H'' ont été réalisées de la même manière durant les réductions, c'est-à-dire que chaque structure accessible depuis H a subit les mêmes modifications dans les deux tas. Les équations (4.3) et (4.4) garantissent que les réductions produisent le même calcul. L'équation (4.5) garantit que l'on n'écrase pas les variables dans le frame, LLVM étant en forme SSA (*Single Static Assignment*) qui implique que chaque assignation introduit une nouvelle variable. L'équation (4.6) garantit que les variables ajoutées dans les frames sont les mêmes. La preuve de ce théorème est donnée en section 5.1.

La correction de la fonction *init* est donnée par le théorème 4.8, celle de la fonction *succ* par le théorème 4.9.

THÉORÈME 4.8. *La fonction init est correcte et termine,* i.e.*, Si M est le marquage initial, F un frame alors*

$$\left(\ fcall\ init() \ \right)_{\emptyset,F} \rightsquigarrow \left(\ p_{mrk} \ \right)_H \quad et \quad [\![H, p_{mrk}]\!]^{mrk} = M$$

□

THÉORÈME 4.9. *La fonction succ est correcte et termine,* i.e.*, Si M est un marquage et set l'ensemble de tous les marquages successeurs, alors pour un tas H, un frame F, x_{mrk} une variable et p_{mrk} un pointeur sur une structure de marquage tels que $F(x_{mrk}) = p_{mrk}$ et $[\![H, p_{mrk}]\!]^{mrk} = M$ on a :*

$$\left(\ fcall\ succ(x_{mrk}) \ \right)_{H,F} \rightsquigarrow \left(\ p_{set} \ \right)_{H \oplus H'}$$

$$dom(H) \cap dom(H') = \emptyset$$

$$[\![H \oplus H', p_{set}]\!]^{set} = set$$

□

4.7.2 Correction des fonctions d'exploration

La correction de la fonction de tir d'une transition t est donnée par le Théorème 4.10. Ce théorème est prouvé en utilisant deux lemmes montrant respectivement la correction de l'étape de consommation de jetons et la correction de l'étape de production de jetons. Les deux sont prouvés par induction sur le nombre de places dans la structure de marquage. Ces lemmes et les preuves sont données dans la section 5.2. Le corollaire 4.11 donne la terminaison de cette fonction.

Théorème 4.10. *Soient M un marquage, H un tas et p_{mrk} un pointeur sur une structure de marquage telle que $dom(H) = p_{mrk}{\downarrow_H}$ et $[\![H, p_{mrk}]\!]^{mrk} = M$. Soit $\beta \stackrel{\text{df}}{=} \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ un mode de la transition t, ce qui implique que chaque v_i est une valeur ou un pointeur qui encode un jeton d'une place s_i (*i.e.*, $[\![H, v_i]\!]^{\ell(s_i)} \in M(s_i)$). Soit F un frame tel que $\beta_F \stackrel{\text{df}}{=} \beta \oplus \{x_{mrk} \mapsto p_{mrk}\}$ Si*

$$M[t, \beta\rangle M' \quad et \quad \left(\ fcall\ fire_t(x_{mrk}, x_1, \ldots, x_n)\ \right)_{H,F} \rightsquigarrow \left(\ p'_{mrk}\ \right)_{H \oplus H'}$$

alors

$$[\![H \oplus H', p'_{mrk}]\!]^{mrk} = M' \quad et \quad dom(H) \cap dom(H') = \emptyset$$

Corollaire 4.11. *Sous les mêmes hypothèses, l'appel à $fire_t$ termine.* □

Le dernier théorème présenté ici est le théorème 4.12 qui donne la correction de la fonction successeur spécifique aux transitions. La preuve de ce théorème est réalisée par double inclusion en analysant les interfaces formelles des fonctions appelées. Le détail est donné dans la section 5.3. Le corollaire 4.13 donne la terminaison de cette fonction.

Théorème 4.12. *Soient F un frame et p_{mrk} un pointeur sur le tas H tel que $p_{mrk}{\downarrow_H} = dom(H)$, $[\![H, p_{mrk}]\!]^{mrk} = M$, $\beta_F(x_{mrk}) = p_{mrk}$ et $\beta_F(x_{next}) = p_{next}$. Si*

$$\left(\ fcall\ succ_t(x_{mrk}, x_{next})\ \right)_{H,F} \rightsquigarrow^* \left(\ p_{next}\ \right)_{H \oplus H'} \quad et \quad [\![H, p_{next}]\!]^{set} = E$$

où E est un ensemble de marquages, alors

$$dom(H) \cap dom(H') = \emptyset$$

$$[\![H \oplus H', p_{next}]\!]^{set} = E \cup \{M' \mid \exists \beta,\ M[t, \beta\rangle M'\}$$

Corollaire 4.13. *Sous les mêmes hypothèses, l'appel à $succ_t$ termine.* □

4.8

Réalisation des optimisations

Cette section on reprend les optimisations de la section 3.3. Pour chacune d'elles on propose une réalisation en LLVM. Ce travail a été réalisé essentiellement pour mettre en avant l'aspect flexible du cadre LLVM qu'on a défini dans ce chapitre. Dans les trois cas, on s'intéresse aux places 1-bornées qui ont la propriété de contenir au plus un seul jeton dans tous les marquages accessibles depuis le marquage initial. Les preuves de ces optimisations sont données dans la section 5.4.

4.8.1 Optimisation de calcul de modes

Pour un marquage M, et une transition t telle que $s_1, \ldots, s_n$ sont ses places en entrée, c'est-à-dire $\ell(s_i, t) = x_i$, pour $1 \leq i \leq n$. Supposons que la place s_k est 1-bornée, avec $1 \leq k \leq n$, alors on peut remplacer la k-ème boucle de l'algorithme du calcul des marquages successeurs par un simple test. Cela a été présenté en section 3.3.

Les changements au niveau de l'implémentation LLVM de cette optimisation sont minimes, en effet, il suffit de remplacer par les quatre blocs présentés sur la figure 4.8, les blocs correspondants de l'algorithme 4.5. Bien évidemment, puisqu'on change l'algorithme du calcul des marquages successeurs d'une transition, il faut de nouveau prouver le théorème 4.12, ce qui est fait dans la section 5.4.1.

$$
\begin{aligned}
P(header_k) &\stackrel{\text{df}}{=} \begin{cases} x_{p_{q_k}} = fcall\ get_{q_k}(x_{p_M}) \\ br\ loop_k \end{cases} \\
P(loop_k) &\stackrel{\text{df}}{=} \begin{cases} c_{q_k} = fcall\ notempty(x_{p_{q_k}}) \\ br\ c_{q_k},\ body_k,\ footer_{k+1} \end{cases} \\
P(body_k) &\stackrel{\text{df}}{=} \begin{cases} x_k = fcall\ nth_{q_k}(x_{p_{q_k}}, 1) \\ br\ header_{k-1} \end{cases} \\
P(footer_k) &\stackrel{\text{df}}{=} \{\ br\ footer_{k+1}
\end{aligned}
$$

FIGURE 4.8 – *Nouveaux blocs de $succ_t$ spécifiques à une place s_k 1-bornée.*

4.8.2 Optimisation d'ordre des énumérations

Lorsqu'on a réalisé la fonction $succ_t$, on a considéré un ordre arbitraire sur les variables afin de les énumérer. L'ordre des boucles n'a donc pas d'incidence sur la correction de l'algorithme. Cependant, considérer certaines places avant d'autres permet de gagner en performances [56]. L'optimisation consiste à définir un ordre particulier afin d'énumérer en priorité les places 1-bornées, puis celles dont les types comportent peu d'éléments, et enfin les autres. Comme l'ordre était initialement arbitraire, ce choix n'a pas besoin d'être prouvé et les gains seront mis en évidence pendant les tests de performances.

4.8.3 Optimisation d'une structure de données

Ici on s'intéresse à remplacer une structure de données par une autre. La clef de voûte de cette optimisation est le respect des interfaces, car alors, les algorithmes qu'on a prouvés corrects sont préservés. En effet, ils ne font aucune supposition à propos des structures de données autre que le respect des interfaces (et de leur axiomatisation).

La réalisation de cette optimisation n'est visible qu'au niveau LLVM, et consiste à remplacer la structure qui encode une place s 1-bornée de type jeton-noir. L'implémentation de la nouvelle structure de données pour la place

s est présentée sur la figure 4.9 et consiste à stocker un booléen indiquant si la place est marquée ou pas. La fonction d'interprétation de cette structure est :

$$\begin{array}{lll} [\![H, p_s]\!]^s = & \emptyset & \text{si} \quad H(p_s) = (bool, false) \\ [\![H, p_s]\!]^s = & \{\bullet\} & \text{si} \quad H(p_s) = (bool, true) \end{array}$$

La fonction d'interprétation de l'encodage des jetons est définie par :

$$[\![H, v]\!]^{t(s)} = \bullet \quad \text{si } v \in \{true, false\}$$

Il faut remarquer que l'interprétation de jetons donne le même résultat pour les deux valeurs parce qu'elle n'est utilisée que dans le cas d'une place non vide.

Proposition 4.14. *L'implémentation, 1-bornée de type jeton noir, respecte l'interface d'une structure de place.* □

$P(add_s(\!| x_s, x |\!)) \stackrel{\text{df}}{=}$
store true, x_s
ret

$P(rem_s(\!| x_s, x |\!)) \stackrel{\text{df}}{=}$
store false, x_s
ret

$P(copy_s(\!| x_s, x'_s |\!)) \stackrel{\text{df}}{=}$
$x =$ *load* x_s
store x, x'_s
ret

$P(notempty_s(\!| x_s |\!)) \stackrel{\text{df}}{=}$
$x =$ *load* x_s
ret x

$P(cons_s(\!|\, |\!)) \stackrel{\text{df}}{=}$
store false, x_s
ret

$P(size_s(\!| x_s |\!)) \stackrel{\text{df}}{=}$
$x =$ *load* x_s
br x, l_1, l_2

$P(nth_s(\!| x_s, i |\!)) \stackrel{\text{df}}{=}$
ret true

$P(clear_s(\!| x_s |\!)) \stackrel{\text{df}}{=}$
store false, x_s
ret

$P(l_1) \stackrel{\text{df}}{=}$ *ret* 1

$P(l_2) \stackrel{\text{df}}{=}$ *ret* 0

Figure 4.9 – *Implémentation d'une place 1-bornée de type jeton noir.*

4.9 Conclusion

Dans ce chapitre on a vu l'application de l'approche par compilation du chapitre 3. Un fragment du langage LLVM-IR a été utilisé pour la réalisation des algorithmes et des optimisations. Ce fragment a été muni d'une sémantique opérationnelle pour réaliser des preuves de correction et de terminaison.

La sémantique formelle est présentée sous forme d'un ensemble de règles d'inférence qui manipulent des tas et des piles de manière explicite. L'ensemble des règles est à petits pas à l'exception des appels de sous-programmes. Cette sémantique a donné un théorème important, le théorème d'extensionnalité, qui permet d'obtenir des preuves plus simples car réalisés dans des contextes minimaux puis généralisées à tout contexte, comme nous le verrons dans le chapitre 5.

Il faut cependant noter les limites de ce qu'on prouve. On ne prouve pas une implémentation, mais la correction des primitives dans le langage cible. Le calcul est correct si le générateur crée effectivement les algorithmes et structures données ici, ces primitives sont correctes et réalisent effectivement les interfaces.

5 Preuves sur la sémantique

Ce chapitre regroupe les preuves des résultats du chapitre 4, ainsi que des résultats intermédiaires. Il s'agit donc d'un chapitre technique qui peut aisément être passé en première lecture. On donne tout d'abord la preuve du théorème d'extensionnalité, puis la correction des fonctions $fire_t$ et $succ_t$, finalement on donne les preuves des optimisations.

5.1

Preuve du théorème d'extensionnalité

La preuve est réalisée par induction sur les arbres de dérivation. On utilise un ordre lexicographique sur les deux quantités suivantes :

1. La hauteur maximale des arbres de dérivation durant la réduction.
2. Le nombre de pas de calcul nécessaires à la réduction.

5.1.1 Cas de base : nombre de pas nul

On a la réduction

$$\left(\ seq\ \right)_{H,F} \rightsquigarrow^* \left(\ seq'\ \right)_{H\oplus H',F'}$$

on souhaite obtenir

$$\left(\ seq\ \right)_{H\oplus H_0,F\oplus\beta_0} \rightsquigarrow^* \left(\ seq'\ \right)_{H\oplus H_0\oplus H'',F''\oplus\beta_0}$$

parce que le nombre de pas est nul on a $seq = seq'$, $H' = H'' = \emptyset$ et $F = F' = F''$. L'équivalence structurelle et les égalités sur les domaines sont triviales.

5.1.2 Cas inductifs :

Dans les cas inductifs on considère le premier pas durant la séquence de réduction. En effet, si on prouve que la propriété est vraie pour le premier pas alors on peut conclure par l'hypothèse d'induction puisque le nombre de pas a été décrémenté de 1. La hauteur maximale de l'arbre de dérivation restera quant à elle plus petite ou égale à la hauteur actuelle.

La majorité des cas inductifs sont triviaux, une simple application de l'hypothèse d'induction ou l'application de la dérivation suffit pour conclure. Le cas difficile est *assign* à cause de l'introduction d'une nouvelle variable.

Cas *seq* :

Le premier pas est une application de la règle *seq* :

$$\frac{\left(\ cmd\ \right)_{H,F} \rightsquigarrow \left(\ seq'\ \right)_{H\oplus H',F'}}{\left(\ cmd;\ seq\ \right)_{H,F} \rightsquigarrow \left(\ seq';\ seq\ \right)_{H\oplus H',F'}}\ seq$$

On applique l'hypothèse d'inductions sur la prémisse de la règle, on obtient :

$$\left(\ cmd\ \right)_{H\oplus H_0,F\oplus\beta_0} \rightsquigarrow \left(\ seq'\ \right)_{H\oplus H_0\oplus H'',F''\oplus\beta_0}$$

où le tas H'' et le frame F'' sont tels que :

$$dom(H_0) \cap dom(H'') = \emptyset \tag{5.1}$$

$$\forall p \in dom(H), \quad (H', p) =_{st} (H'', p) \tag{5.2}$$

$$dom(\beta_0) \cap dom(\beta_{F''}) = \emptyset \tag{5.3}$$

$$dom(\beta_{F'}) = dom(\beta_{F''}) \tag{5.4}$$

$$(F', H') =_{st} (F'', H'') \tag{5.5}$$

On réinjecte dans la règle pour le résultat souhaité :

$$\left(\ cmd;\ seq\ \right)_{H\oplus H_0,F\oplus\beta_0} \rightsquigarrow \left(\ seq';\ seq\ \right)_{H\oplus H_0\oplus H'',F''\oplus\beta_0}$$

Cas seq_{skip} :

similaire au cas de base.

Cas $branch_1$ et $branch_2$:

$$\frac{}{\left(\ br\ l\ \right)_{H,(l_p,l_c,\beta)} \rightsquigarrow \left(\ P(l)\ \right)_{H\oplus H',(l_c,l,\beta)}}\ branch_1$$

où $H' = \emptyset$. On a

$$\frac{}{\left(\ br\ l\ \right)_{H\oplus H_0,(l_p,l_c,\beta)\oplus\beta_0} \rightsquigarrow \left(\ P(l)\ \right)_{H\oplus H_0\oplus H'',(l_c,l,\beta)\oplus\beta_0}}\ branch_1$$

où $H'' = \emptyset$, car la règle ne dépend pas du tas ni des frames

$$(l_p, l_c, \beta) \oplus \beta_0 = (l_p, l_c, \beta \oplus \beta_0)$$

$$(l_c, l, \beta) \oplus \beta_0 = (l_c, l, \beta \oplus \beta_0)$$

Le cas $branch_2$ est similaire.

Cas *pcall* :

Le premier pas de réduction est *pcall*. On sait que

$$\frac{\begin{array}{c} f(\!|\, a_1, \ldots, a_n \,|\!) \in dom(P) \\ F_0 = (\bot, f(\!|\, a_1, \ldots, a_n \,|\!), \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \left(\ P(f(\!|\, a_1, \ldots, a_n \,|\!))\ \right)_{H,F_0} \rightsquigarrow^* \left(\ ret\ \right)_{H\oplus H',F_1} \end{array}}{\left(\ pcall\ f(r_1, \ldots, r_n)\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H\oplus H',F}}\ pcall$$

où $H' = \emptyset$. On reconstruit la conclusion de la règle en appliquant l'hypothèse d'induction sur la prémisse avec le tas enrichi par H_0 et le frame F_0 laissé intact (arbre de dérivation plus petit).

$$\frac{\begin{array}{c} f(\!|\, a_1, \ldots, a_n \,|\!) \in dom(P) \\ F_0 = (\bot, f(\!|\, a_1, \ldots, a_n \,|\!), \{a_1 \mapsto F(r_1), \ldots, a_n \mapsto F(r_n)\}) \\ \left(\ P(f(\!|\, a_1, \ldots, a_n \,|\!))\ \right)_{H\oplus H_0, F_0\oplus\emptyset} \rightsquigarrow^* \left(\ ret\ \right)_{H\oplus H_0\oplus H'', F_1\oplus\emptyset} \end{array}}{\left(\ pcall\ f(r_1, \ldots, r_n)\ \right)_{\substack{H\oplus H_0,\\ F\oplus\beta_0}} \rightsquigarrow \left(\ skip\ \right)_{\substack{H\oplus H_0\oplus H'',\\ F\oplus\beta_0}}}\ pcall$$

L'utilisation de la même frame F_0 est correcte : $(F\oplus\beta_0)(a_i) = F(a_i)$ $(0 \leq i \leq n)$ car $dom\beta_F \cap dom(\beta_0) = \emptyset$.

Le cas *fcall* est similaire, l'hypothèse d'induction fournit les conclusions recherchées pour les expressions.

Cas *assign* :

Le premier pas de réduction est *assign*, on sait que

$$\frac{\left(\ expr\ \right)_{H,F} \rightsquigarrow \left(\ v'\ \right)_{H\oplus H'}}{\left(\ x = expr\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H\oplus H', F\oplus\{x\ \mapsto\ v'\}}}\ assign$$

On applique l'hypothèse d'induction sur la prémisse (arbre de dérivation plus petit)

$$\left(\ expr\ \right)_{H\oplus H_0, F\oplus\beta_0} \rightsquigarrow \left(\ v''\ \right)_{H\oplus H_0\oplus H''}$$

où H'' est un tas tel que :

$$dom(H_0) \cap dom(H'') = \emptyset \tag{5.6}$$

$$\forall p \in H, \quad (H', p) =_{st} (H'', p) \tag{5.7}$$

$$(H', v') =_{st} (H'', v'') \tag{5.8}$$

On réinjecte dans la règle et on obtient :

$$\left(\ x = expr\ \right)_{H\oplus H_0, F\oplus\beta_0} \rightsquigarrow \left(\ skip\ \right)_{H\oplus H_0\oplus H'', F\oplus\beta_0\oplus\{x\ \mapsto\ v''\}}$$

On montre les conséquences du théorème :

– les conséquences (4.1) et (4.2) sont satisfaites par les conséquence de l'hypothèse d'induction (5.6) et (5.7) ;

- la variable x n'est pas une variable assignée donc elle n'est pas dans $dom(\beta_0)$, par hypothèse du théorème. On a donc β_0 et $\{x \mapsto v''\}$ qui commutent, par conséquent $F \oplus \beta_0 \oplus \{x \mapsto v''\} = F \oplus \{x \mapsto v''\} \oplus \beta_0$ et $dom(F \oplus \{x \mapsto v''\}) \cap dom(\beta_0) = \emptyset$ ce qui nous donne la conséquence (4.5) ;
- la conséquence (4.6) est triviale parce que $dom(\beta') = dom(\beta'') = \{x\}$.
- il reste à montrer que $(F', H \oplus H') =_{st} (F'', H \oplus H'')$ (4.4), *i.e.*. $(F \oplus \{x \mapsto v'\}, H \oplus H') =_{st} (F \oplus \{x \mapsto v''\}, H \oplus H'')$, il est suffisant de monter que $(\{x \mapsto v'\}, H') =_{st} (\{x \mapsto v''\}, H'')$ car (5.7) implique le résultat pour F. Finalement on a bien $(\{x \mapsto v'\}, H') =_{st} (\{x \mapsto v''\}, H'')$ par (5.8).

Cas *store* :

Le premier pas de la réduction est

$$\frac{F(r_p) = p \quad H(p) = (t, d) \quad H' = \{p \mapsto t, F(r_{new})\}}{\left(\ store\ r_{new},\ r_p\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H \oplus H', F'}}\ store$$

avec $F = F'$, on construit la conclusion en prenant un tas H_0 et une valuation β_0 comme dans l'énoncé du théorème :

$$\frac{\begin{array}{c}(F \oplus \beta_0)(r_p) = F(r_p) = p \qquad (H \oplus H_0)(p) = H(p) = (t, d) \\ H'' = \{p \mapsto t, F(r_{new})\}\end{array}}{\left(\ store\ r_{new},\ r_p\ \right)_{H \oplus H_0, F \oplus \beta_0} \rightsquigarrow \left(\ skip\ \right)_{H \oplus H_0 \oplus H'', F'' \oplus \beta_0}}\ store$$

Puisque $H' = H''$ et $F = F' = F''$, les conséquences sont triviales.

Cas *load* :

Le premier pas de calcul est *load*

$$\frac{F(x) = p \quad H(p) = (t, v)}{\left(\ load\ x\ \right)_{H,F} \rightsquigarrow \left(\ v\ \right)_{H \oplus H'}}\ load$$

où $H' = \emptyset$, le résultat est immédiat

$$\frac{(F \oplus \beta_0)(x) = F(x) = p \quad (H \oplus H_0)(p) = H(p) = (t, v)}{\left(\ load\ x\ \right)_{H \oplus H_0, F \oplus \beta_0} \rightsquigarrow \left(\ v\ \right)_{H \oplus H_0 \oplus H''}}\ load$$

avec $H'' = \emptyset$, car $dom(F) \cap dom(\beta_0) = \emptyset$ et $dom(H) \cap dom(H_0) = \emptyset$.

Cas gep_0 :

le premier pas de calcul est gep_0

$$\frac{F(x) = p \quad p[i]_H \text{ défini}}{\left(\ gep\ x,\ 0,\ i\ \right)_{H,F} \rightsquigarrow \left(\ p[i]_H\ \right)_{H \oplus H'}}\ gep_0$$

où $H' = \emptyset$. Puisque $p[i]_H$ est bien défini, on a $p,\ p[i]_H \in H$, on conclue avec

$$\frac{(F \oplus \beta_0)(x) = F(x) = p \quad p[i]_{H \oplus H_0} = p[i]_H \text{ defined}}{\left(\ gep\ x,\ 0,\ i\ \right)_{H \oplus H_0, F \oplus \beta_0} \rightsquigarrow \left(\ p[i]_{H \oplus H_0}\ \right)_{H \oplus H_0 \oplus H''}}\ gep_0$$

où $H'' = \emptyset$, parce que $dom(H) \cap dom(H_0) = \emptyset$.

Cas *icmp* et *add* :

Le premier pas de réduction est *icmp*.

$$\frac{\overline{F(x_1)\ op\ F(x_2)} = v \quad op \in \{=, \neq, <, \leq\}}{\left(\ icmp\ op,\ x_1,\ x_2\ \right)_{H,F} \rightsquigarrow \left(\ v\ \right)_{H \oplus H'}}\ icmp$$

où $H' = \emptyset$, on conclue immédiatement

$$\frac{\overline{(F \oplus \beta_0)(x_1)\ op\ (F \oplus \beta_0)(x_2)} = v \quad op \in \{=, \neq, <, \leq\}}{\left(\ icmp\ op,\ x_1,\ x_2\ \right)_{H \oplus H_0, F \oplus \beta_0} \rightsquigarrow \left(\ v\ \right)_{H \oplus H_0 \oplus H''}}\ icmp$$

où $H'' = \emptyset$ car $(F \oplus \beta_0)(x_1) = F(x_1)$ et $(F \oplus \beta_0)(x_2) = F(x_2)$. Les égalités structurelles et contraintes des domaines sont triviales.
Le cas *add* case est similaire.

Cas *phi* :

Le premier pas de réduction est *phi*

$$\frac{1 \leq i \leq n}{\left(\ phi\ (x_1, l_1), \ldots, (x_n, l_n)\ \right)_{H,(l_i,l_c,\beta)} \rightsquigarrow \left(\ \beta(x_i)\ \right)_{H \oplus H'}}\ phi$$

avec $H' = \emptyset$, on a

$$\frac{1 \leq i \leq n}{\begin{array}{c}\left(\ phi\ (x_1, l_1), \ldots, (x_n, l_n)\ \right)_{\substack{H \oplus H_0, \\ (l_i,l_c,\beta) \oplus \beta_0}} \\ \rightsquigarrow \left(\ (\beta \oplus \beta_0)(x_i)\ \right)_{H \oplus H_0 \oplus H''}\end{array}}\ phi$$

où $H'' = \emptyset$. Par hypothèse du théorème on a $dom(\beta) \cap dom(\beta_0) = \emptyset$, ce qui implique $(\beta \oplus \beta_0)(x_i) = \beta(x_i)$, donc le calcul produit bien le même résultat dans les deux cas (4.3). De plus $H' = H'' = \emptyset$ donc (4.1) et (4.2) sont triviales.

Cas *alloc* :

Le premier pas de calcul est *alloc*

$$\frac{(H', p') = new(H, t)}{\left(\ alloc\ t\ \right)_{H,F} \rightsquigarrow \left(\ p'\ \right)_{H \oplus H'}}\ alloc$$

On construit la conclusion souhaitée

$$\frac{(H'', p'') = new(H \oplus H_0, t)}{\left(\ alloc\ t\ \right)_{H \oplus H_0, F} \rightsquigarrow \left(\ p''\ \right)_{H \oplus H_0 \oplus H''}}\ alloc$$

par définition de *new* (définition 4.7) et la propriété 4.5, on a $(H', p') =_{st} (H'', p'')$, de plus $dom(H'') \cap dom(H \oplus H_0) = \emptyset$, et donc $dom(H'') \cap dom(H_0) = \emptyset$. (4.1), (4.2) et (4.3) sont vraies.

Cas *ret* et *rvalue* :

Triviaux parce que $(\beta \oplus \beta_0)(r) = \beta(r)$.

Tous les cas ont été traités ce qui conclue la preuve. □

5.2

Correction des fonctions $fire_t$

La preuve de correction de la fonction $fire_t$ est réalisée en trois étapes. Les deux premières sont deux lemmes portant respectivement, sur la consommation de jetons dans la fonction, et sur la production de jetons. La troisième étape est le théorème de correction de la fonction $fire_t$ qui consiste essentiellement à utiliser le théorème d'extensionnalité afin d'appliquer les deux lemmes.

Pour aider à la lisibilité de cette section les noms de variables dans les lemmes ont été choisis de façon à correspondre aux variables du théorème.
Dans la suite de cette section on se placera sous les hypothèses suivantes :

(i) M est un marquage du réseau ;

(ii) t est une transition du réseau ;

(iii) $s_1, \ldots, s_n, s'_1, \ldots, s'_m$ sont des places du réseau telles que $s_1, \ldots, s_n \in {}^\bullet t$ et $s'_1, \ldots, s'_m \in t^\bullet$.

5.2.1 Premier lemme

Lemme 5.1. *Soit H un tas et p'_{mrk} un pointeur sur une structure de marquage tels que $dom(H) = p'_{mrk}\!\downarrow_H$ et $[\![H, p'_{mrk}]\!]^{mrk} = M$. Soit F un frame tel que $\beta_F \stackrel{\text{df}}{=} \{x'_{mrk} \mapsto p'_{mrk}, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ où tous les v_i sont des pointeurs ou des valeurs du type des jetons des places s_i et $[\![H, v_i]\!]^{t(s_i)} \in M(s_i)$ (pour $1 \leq i \leq n$). On considère la séquence seq d'instructions qui correspond à la consommation de jetons dans la fonction $fire_t$:*

$$seq \stackrel{\text{df}}{=} \left(\begin{array}{l} x_{s_1} = fcall\ get_{s_1}(x'_{mrk}) \\ pcall\ rem_{s_1}(x_{s_1}, x_1) \\ \ldots \\ x_{s_n} = fcall\ get_{s_n}(x'_{mrk}) \\ pcall\ rem_{s_n}(x_{s_n}, x_n) \end{array} \right)$$

Si on réduit totalement la séquence, i.e.*, si*

$$\left(\ seq \ \right)_{H,F} \quad \rightsquigarrow^* \quad \left(\ skip \ \right)_{H \oplus H', F \oplus \beta}$$

alors

$$\beta \stackrel{\text{df}}{=} \{x_{s_1} \mapsto p_{s_1}, \ldots, x_{s_n} \mapsto p_{s_n}\} \tag{5.9}$$

$$dom(H) \cap dom(H') \subseteq p_{s_1}\!\downarrow_H \cup \cdots \cup p_{s_n}\!\downarrow_H \tag{5.10}$$

$$[\![H \oplus H', p'_{mrk}]\!]^{mrk}(s_i) = M(s_i) - \{[\![H, v_i]\!]^{t(s_i)}\} \ \textit{pour}\ i \in \{1, \ldots, n\} \tag{5.11}$$

◊

Intuitivement, le résultat (5.9) est la valuation produite pendant l'exécution. Le résultat (5.10) permet de localiser les modifications sur le tas pendant la réduction. Le résultat (5.11) assure qu'on obtient le marquage souhaité, c'est-à-dire qu'on supprime bien les jetons du marquage.

Preuve

On montre les trois conséquences simultanément par induction sur n.
Pour le cas de base, $n = 0$: la séquence est vide, *i.e. skip*, les conséquences sont triviales : (5.9) est vrai car $\beta = \emptyset$, (5.10) est vrai car $H' = \emptyset$ et (5.11) est vrai car $i \in \emptyset$.

Pour le cas inductif, supposons le lemme vrai pour $n-1$. On sait que :

$$\left(\begin{array}{l} x_{s_1} = fcall\ get_{s_1}(x'_{mrk}) \\ pcall\ rem_{s_1}(x_{s_1}, x_1) \\ \ldots \\ x_{s_{n-1}} = fcall\ get_{s_{n-1}}(x'_{mrk}) \\ pcall\ rem_{s_{n-1}}(x_{s_{n-1}}, x_{n-1}) \end{array} \right)_{H,F} \rightsquigarrow^* \left(\ skip \ \right)_{\substack{H \oplus H_0, \\ F \oplus \beta_0}}$$

avec comme conséquences :

$$\beta_0 \stackrel{\text{df}}{=} \{x_{s_1} \mapsto p_{s_1}, \ldots, x_{s_{n-1}} \mapsto p_{s_{n-1}}\} \tag{5.12}$$

$$dom(H) \cap dom(H') \subseteq p_{s_1}{\downarrow_H} \cup \cdots \cup p_{s_{n-1}}{\downarrow_H} \tag{5.13}$$

$$\forall i \in \{1, \ldots, n-1\}, [\![H \oplus H_0, p'_{mrk}]\!]^{mrk}(s_i) = M(s_i) - \{[\![H, v_i]\!]^{t(s_i)}\} \tag{5.14}$$

On va montrer que le lemme est vrai pour n, pour cela on va réaliser une séquence de réductions et vérifier que les conséquences sont vraies. Par hypothèse d'induction on peut réaliser la séquence de réductions suivante :

$$\left(\begin{array}{l} x_{s_1} = fcall\ get_{s_1}(x'_{mrk}) \\ pcall\ rem_{s_1}(x_{s_1}, v_1) \\ \ldots \\ x_{s_{n-1}} = fcall\ get_{s_{n-1}}(x'_{mrk}) \\ pcall\ rem_{s_{n-1}}(x_{s_{n-1}}, v_{n-1}) \\ x_{s_n} = fcall\ get_{s_n}(x'_{mrk}) \\ pcall\ rem_{s_n}(x_{s_n}, v_n) \end{array} \right)_{H,F}$$

$$\rightsquigarrow^* \left(\begin{array}{l} x_{s_n} = fcall\ get_{s_n}(x'_{mrk}) \\ pcall\ rem_{s_n}(x_{s_n}, v_n) \end{array} \right)_{\substack{H \oplus H_0, \\ F \oplus \beta_0}}$$

avec les conséquences (5.12) à (5.14) ci-dessus. On remarque que le résultat (5.13) implique qu'on n'a pas modifié la place s_n. On continue à réduire en utilisant les règles de la sémantique et les spécifications des fonctions get_{s_n} et rem_{s_n} :

$$\begin{array}{l} \left(\begin{array}{l} x_{s_n} = fcall\ get_{s_n}(x'_{mrk}) \\ pcall\ rem_{s_n}(x_{s_n}, v_n) \end{array} \right)_{\substack{H \oplus H_0, \\ F \oplus \beta_0}} \\ \rightsquigarrow^* \left(\ pcall\ rem_{s_n}(x_{s_n}, v_n) \ \right)_{\substack{H \oplus H_0, \\ F \oplus \beta_0 \oplus \{x_{s_n} \mapsto p_{s_n}\}}} \\ \rightsquigarrow^* \left(\ skip \ \right)_{\substack{H \oplus H_0 \oplus H_1, \\ F \oplus \beta_0 \oplus \{x_{s_n} \mapsto p_{s_n}\}}} \end{array}$$

On a alors :

$$\begin{aligned}\llbracket H \oplus H_0 \oplus H_1, p_{s_n} \rrbracket^{s_n} &= \llbracket H \oplus H_0, p_{s_n} \rrbracket^{s_n} - \{\llbracket H, v_n \rrbracket^{t(s_n)}\} \\ &= \llbracket H, p_{s_n} \rrbracket^{s_n} - \{\llbracket H, v_n \rrbracket^{t(s_n)}\} \\ &\qquad \text{car on n'a pas modifié la place } s_n \text{ (5.13)} \\ &= M(s_n) - \{\llbracket H, v_n \rrbracket^{t(s_n)}\}\end{aligned} \tag{5.15}$$

$$dom(H \oplus H_0) \cap dom(H_1) \subseteq p_{s_n}{\downarrow_H} \tag{5.16}$$

Et donc

$$\beta = \beta_0 \oplus \{x_{s_n} \mapsto p_{s_n}\} = \{x_1 \mapsto p_{s_1}, \ldots, x_n \mapsto p_{s_n}\}$$

car tous les x_i sont différents (LLVM est en forme SSA). La conséquence (5.9) est donc vérifiée. (5.10) est une conséquence immédiate de (5.13) et (5.16) car

$$dom(H) \cap dom(H_0 \oplus H_1) = dom(H) \cap (dom(H_0) \cup dom(H_1))$$

Pour prouver la conséquence (5.11), il faut montrer qu'on peut étendre le résultat (5.14) au tas $H \oplus H_0 \oplus H_1$. Une fois ce résultat obtenu on pourra le généraliser de 1 à n grâce au résultat (5.15).

Comme les tas H_0 et H_1 sont disjoints (d'après (5.13) et (5.16) car toutes les places sont disjointes) on a

$$\llbracket H \oplus H_0 \oplus H_1, p_{s_n} \rrbracket^{s_n} = \llbracket H \oplus H_1, p_{s_n} \rrbracket^{s_n} = M(s_n) - \{\llbracket H, v_n \rrbracket^{t(s_n)}\}$$

qui préserve bien le résultat (5.14) obtenu par l'hypothèse d'induction

$$\begin{aligned}\forall i \in \{1, \ldots, n-1\},\ \llbracket H \oplus H_0 \oplus H_1, p_{s_i} \rrbracket^{s_i} &= \llbracket H \oplus H_0, p_{s_i} \rrbracket^{s_i} \\ &= M(s_i) - \{\llbracket H, v_i \rrbracket^{t(s_i)}\}\end{aligned}$$

Il en découle naturellement :

$$\forall i \in \{1, \ldots, n\},\ \llbracket H \oplus H_0 \oplus H_1, p_{s_i} \rrbracket^{s_i} = M(s_i) - \{\llbracket H, v_i \rrbracket^{t(s_i)}\}$$

On conclue avec l'associativité de $\oplus$ et la propriété de cohérence de la structure de marquage (prérequis 4.3) :

$$\begin{aligned}&\forall i \in \{1, \ldots, n\}, \\ &\quad \llbracket H \oplus (H_0 \oplus H_1), p_{mrk} \rrbracket^{mrk}(s_i) = \llbracket H, p'_{mrk} \rrbracket^{mrk}(s_i) - \{\llbracket H, v_i \rrbracket^{t(s_i)}\} \\ &\qquad\qquad = M(s_i) - \{\llbracket H, v_i \rrbracket^{t(s_i)}\}\end{aligned}$$

La conséquence (5.11) est donc vérifiée. Les trois conséquences étant vraies, le lemme est vrai pour tout n. □

5.2.2 Second lemme

Lemme 5.2. *Soit H un tas et p'_{mrk} un pointeur sur une structure de marquage tels que $dom(H) = p'_{mrk}{\downarrow_H}$ et $\llbracket H, p'_{mrk} \rrbracket^{mrk} = M$. Soit F un frame tel que $\beta_F \stackrel{\text{df}}{=} \{x'_{mrk} \mapsto p'_{mrk}, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ où tous les v_i sont des pointeurs ou des valeurs du type des jetons des places s_i (pour $1 \leq i \leq n$). On considère*

la séquence seq d'instructions qui correspond à la production de jetons dans la fonction $fire_t$ *:*

$$seq \stackrel{\text{df}}{=} \left(\begin{array}{l} x_{s'_1} = fcall\ get_{s'_1}(x'_{mrk}) \\ o_{s'_1} = fcall\ f_{s'_1}(x_1, \ldots, x_n) \\ pcall\ add_{s'_1}(x_{s'_1}, o_{s'_1}) \\ \ldots \\ x_{s'_m} = fcall\ get_{s'_m}(x'_{mrk}) \\ o_{s'_m} = fcall\ f_{s'_m}(x_1, \ldots, x_n) \\ pcall\ add_{s'_m}(x_{s'_m}, o_{s'_m}) \end{array} \right)$$

Si on exécute totalement la séquence, i.e.*, si*

$$\left(\ seq \ \right)_{H,F} \quad \rightsquigarrow^* \quad \left(\ skip \ \right)_{\substack{H \oplus H' \\ F \oplus \beta}},$$

alors

$$\beta \stackrel{\text{df}}{=} \left\{ \begin{array}{ll} x_{s'_1} \mapsto p_{s'_1}, & o_{s'_1} \mapsto \overline{f_{s'_1}(v_1, \ldots, v_n)}, \\ \ldots & \\ x_{s'_m} \mapsto p_{s'_m}, & o_{s'_m} \mapsto \overline{f_{s'_m}(v_1, \ldots, v_n)} \end{array} \right\} \tag{5.17}$$

$$dom(H) \cap dom(H') \subseteq p_{s'_1}{\downarrow_H} \cup \cdots \cup p_{s'_m}{\downarrow_H} \tag{5.18}$$

$$[\![H \oplus H', p'_{mrk}]\!]^{mrk}(s'_i) = M(s'_i) + \left\{ [\![H \oplus H', \overline{f_{s'_i}(v_1, \ldots, v_n)}]\!]^{t(s'_i)} \right\} \quad \text{pour } 1 \leq i \leq m \tag{5.19}$$

Preuve

La preuve est similaire à celle du lemme 5.1, il faut cependant utiliser les propriétés des fonctions étiquetant les arcs sortants de la transition t. □

5.2.3 Théorème de correction de la fonction $fire_t$

On rappelle l'énoncé du théorème 4.10.

Soit H *un tas et* p_{mrk} *un pointeur sur une structure de marquage tel que* $dom(H) = p_{mrk}{\downarrow_H}$ *et* $[\![H, p_{mrk}]\!]^{mrk} = M$*. Soit* $\beta \stackrel{\text{df}}{=} \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ *un mode LLVM de la transition* t*, ce qui implique que les* v_i *sont des valeurs ou des pointeurs de type des jetons des places* s_i *et* $[\![H, v_i]\!]^{t(s_i)} \in M(s_i)$*. Soit* F *un frame tel que* $\beta_F \stackrel{\text{df}}{=} \beta \oplus \{x_{mrk} \mapsto p_{mrk}\}$*.*
Si

$$M[t, \beta\rangle M'$$

$$\left(\ fcall\ fire_t(x_{mrk}, x_1, \ldots, x_n) \ \right)_{H,F} \rightsquigarrow \left(\ p'_{mrk} \ \right)_{H \oplus H'}$$

alors

$$[\![(H \oplus H')(p'_{mrk})]\!]^{mrk} = M' \tag{5.20}$$

$$dom(H) \cap dom(H') = \emptyset \tag{5.21}$$

Preuve

Il suffira de montrer (5.20), car (5.21) sera une conséquence immédiate de l'application du théorème d'extensionnalité pendant la preuve. Par définition de la règle *fcall*, il nous suffit de montrer que

$$\left(\ P(\mathit{fire}_t(\!|\, x_{mrk}, x_1, \ldots, x_n \,|\!))\ \right)_{H,F_0} \quad \rightsquigarrow \quad \left(\ \mathit{ret}\ x'_{mrk}\ \right)_{H\oplus H'_0, F'_0}$$

avec $F_0 \stackrel{\text{df}}{=} (\bot,\ \mathit{fire}_t(\!|\, x_{mrk}, x_1, \ldots, x_n \,|\!),\ \beta \oplus \{x_{mrk} \mapsto p_{mrk}\})$, H'_0 un tas et F'_0 un frame tels que $[\![H'_0, F'_0(x'_{mrk})]\!]^{mrk} = M'$.

Pour plus de lisibilité on introduit la *notation* $P(\mathit{fire}_t + i)$ qui dénote la séquence d'instructions de la fonction fire_t à partir de la i-ème instruction en comptant à partir de zéro : $P(\mathit{fire}_t + 0) = P(\mathit{fire}_t)$ et $P(\mathit{fire}_t + 1)$ est le corps de la fonction fire_t en omettant la première instruction.

D'après la spécification de copy_{mrk} on peut effectuer la réduction :

$$\left(\begin{array}{l} x'_{mrk} = \mathit{fcall}\ \mathit{copy}_{mrk}(x_{mrk}) \\ P(\mathit{fire}_t + 1) \end{array}\right)_{H,F_0} \rightsquigarrow^* \left(\ P(\mathit{fire}_t + 1)\ \right)_{H\oplus H_0, F_0 \oplus \beta_0}$$

avec $\beta_0 \stackrel{\text{df}}{=} \{x'_{mrk} \ \mapsto\ p'_{mrk}\}$ et d'en déduire :

$$dom(H) \cap dom(H_0) = \emptyset \tag{5.22}$$

$$p'_{mrk}{\downarrow_{H_0}} = dom(H_0) \tag{5.23}$$

$$[\![H, p_{mrk}]\!]^{mrk} = [\![H_0, p'_{mrk}]\!]^{mrk} = [\![H \oplus H_0, p_{mrk}]\!]^{mrk} \tag{5.24}$$

À partir de maintenant on travaille dans un tas et un frame isolés. On utilise le frame $F_1 \stackrel{\text{df}}{=} (F_0 \oplus \beta_0) \setminus \{x_{mrk} \mapsto p_{mrk}\}$ et le tas H_0. Une fois nos réductions réalisées on appliquera le théorème d'extensionnalité pour conclure.

On est exactement dans les conditions du lemme (5.1), on l'applique et obtient la séquence de réductions suivante :

$$\left(\begin{array}{l} x_{s_1} = \mathit{fcall}\ \mathit{get}_{s_1}(x'_{mrk}) \\ \mathit{pcall}\ \mathit{rem}_{s_1}(x_{s_1}, x_1) \\ \cdots \\ x_{s_n} = \mathit{fcall}\ \mathit{get}_{s_n}(x'_{mrk}) \\ \mathit{pcall}\ \mathit{rem}_{s_n}(x_{s_n}, x_n) \\ P(\mathit{fire}_t + 2n + 1) \end{array}\right)_{H_0,F_1} \rightsquigarrow^* \left(\ P(\mathit{fire}_t + 2n + 1)\ \right)_{H_0\oplus H_1, F_2}$$

avec comme conséquences :

$$F_2 = F_1 \oplus \{x_{s_1} \mapsto p_{s_1}, \ldots, x_{s_n} \mapsto p_{s_n}\}$$

$$dom(H_0) \cap dom(H_1) \subseteq p_{s_1}{\downarrow_{H_0}} \cup \cdots \cup p_{s_n}{\downarrow_{H_0}}$$

$$\forall i \in \{1, \ldots, n\},\ [\![H_0 \oplus H_1, p'_{mrk}]\!](s_i) = M(s_i) - \{[\![H_0, v_i]\!]^{t(s_i)}\}$$

On pose $H_2 = H_0 \oplus H_1$ et on continue à réduire de la même manière avec le lemme 5.2 :

$$\left(\begin{array}{l} x_{s'_1} = fcall\ get_{s'_1}(x_{p'}) \\ o_{s'_1} = fcall\ f_{s'_1}(v_1, \ldots, v_n) \\ pcall\ add_{s'_1}(x_{s'_1}, o_{s'_1}) \\ \ldots \\ x_{s'_m} = fcall\ get_{s'_m}(x_{p'}) \\ o_{s'_m} = fcall\ f_{s'_m}(v_1, \ldots, v_n) \\ pcall\ add_{s'_m}(x_{s'_m}, o_{s'_m}) \\ ret\ x_{p'} \end{array}\right)_{H_2,F_2} \rightsquigarrow^* \left(\ ret\ x_{p'}\ \right)_{H_4,F_3}$$

où $H_4 \stackrel{\text{df}}{=} H_2 \oplus H_3$, avec comme conséquences :

$$F_3 \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x_{s'_1} \mapsto \underline{p_{s'_1}}, \ldots, x_{s'_m} \mapsto p_{s'_m}, \\ o_{s'_1} \mapsto \overline{f_{s'_1}(v_1, \ldots, v_n)}, \\ \ldots \\ o_{s'_m} \mapsto \overline{f_{s'_m}(v_1, \ldots, v_n)} \end{array} \right\}$$

$$dom(H_2) \cap dom(H_3) \subseteq p_{s'_1}{\downarrow_{H_2}} \cup \cdots \cup p_{s'_m}{\downarrow_{H_2}} \subseteq p{\downarrow_{H_2}}$$

$\forall i \in \{1, \ldots, m\}$,

$$[\![H_4, p'_{mrk}]\!]^{mrk}(s'_i) = [\![H_2, p'_{mrk}]\!]^{mrk}(s'_i) + \left\{ \left[\!\left[H_4, \overline{f_{s'_i}(v_1, \ldots, v_n)}\right]\!\right]^{t(s'_i)} \right\}$$

Comme $dom(H) \cap dom(H_0) = \emptyset$ (5.22) et $x_{mrk} \notin \beta_{F_1}$ on peut appliquer le théorème d'extensionnalité. On préserve les résultats obtenus grâce aux conséquences du théorème d'extensionnalité et on a $dom(H) \cap dom(H_0 \oplus H_1 \oplus H_2) = \emptyset$ qui est exactement la conséquence (5.21) de ce théorème.

On vérifie maintenant qu'on a bien obtenu le marquage M'. Soit s une place du réseau

1. Si $s \notin \{s_1, \ldots, s_n\}$ et $s \notin \{s'_1, \ldots, s'_n\}$ alors on n'a pas modifié les données, on a bien :

$$[\![H_2 \oplus H_3, p'_{mrk}]\!]^{mrk}(s) = [\![H, p_{mrk}]\!]^{mrk}(s) = M(s) = M'(s)$$

2. Si $s = s_i$, $s_i \in \{s_1, \ldots, s_n\}$ et $s \notin \{s'_1, \ldots, s'_n\}$ alors on a consommé un jeton :

$$\begin{aligned} [\![H_2 \oplus H_3, p'_{mrk}]\!]^{mrk}(s_i) &= [\![H, p_{mrk}]\!]^{mrk}(s_i) - \left\{ [\![H, v_i]\!]^{t(s_i)} \right\} \\ &= M(s_i) - \left\{ [\![H, v_i]\!]^{t(s_i)} \right\} \\ &= M'(s_i) \end{aligned}$$

3. Si $s = s'_j$, $s \notin \{s_1, \ldots, s_n\}$ et $s'_j \in \{s'_1, \ldots, s'_n\}$ alors on a ajouté un jeton :

$$\begin{aligned} [\![H_4, p'_{mrk}]\!]^{mrk}(s'_j) &= [\![H, p_{mrk}]\!]^{mrk}(s'_j) + \left\{ \left[\!\left[H_4, \overline{f_{s'_j}(x_1, \ldots, x_n)}\right]\!\right]^{t(s'_j)} \right\} \\ &= M(s'_j) + \left\{ \left[\!\left[H_4, \overline{f_{s'_j}(x_1, \ldots, x_n)}\right]\!\right]^{t(s'_j)} \right\} \\ &= M'(s'_j) \end{aligned}$$

4. Si $s = s_i = s'_j$, $s_i \in \{s_1, \ldots, s_n\}$ et $s'_j \in \{s'_1, \ldots, s'_n\}$ alors on a consommé un jeton puis ajouté nouveau un jeton :

$$\begin{aligned}
[\![H_4, p'_{mrk}]\!]^{mrk}(s) &= [\![H, p_{mrk}]\!]^{mrk}(s) - \left\{ [\![H, v_i]\!]^{t(s_i)} \right\} \\
&\quad + \left\{ \left[\!\!\left[H_4, \overline{f_{s'_j}(x_1, \ldots, x_n)} \right]\!\!\right]^{t(s'_j)} \right\} \\
&= M(s) - \left\{ [\![H, v_i]\!]^{t(s_i)} \right\} \\
&\quad + \left\{ \left[\!\!\left[H_4, \overline{f_{s'_j}(x_1, \ldots, x_n)} \right]\!\!\right]^{t(s'_j)} \right\} \\
&= M'(s)
\end{aligned}$$

On a bien $[\![H \oplus (H_0 \oplus H_1), p'_{mrk}]\!]^{mrk} = [\![H_2 \oplus H_3, p'_{mrk}]\!]^{mrk} = M'$, c'est-à-dire que la conséquence (5.20) est vérifiée. Ce qui conclue la preuve. □

5.2.4 Terminaison de la fonction $fire_t$

Corollaire 5.3. *Dans les mêmes conditions que le théorème 4.10, l'appel de la fonction $fire_t$ termine.*

Preuve

Conséquence immédiate du lemme précédent car il correspond exactement à l'appel de la fonction. □

5.3

Correction des fonctions $succ_t$

On souhaite que la fonction $succ_t$, où t est une transition, renvoie l'ensemble des marquages successeurs d'un marquage M par le tir de cette transition. C'est-à-dire tous les marquages tels qu'il existe un mode permettant de tirer la transition t à partir du marquage M.

Les énoncés qui suivent sont définis dans le contexte où M est un marquage, t une transition à n places en entrée, p_{mrk} et p_{set} sont deux pointeur, x_{mrk} et x_{set} sont deux variables.

Pour réaliser la preuve de correction on aura besoin d'un invariant qu'on propagera pendant les réductions, on l'appelle la propriété P.

Definition 5.1 (La propriété P). *Dans cette section, on dira qu'un couple (H, F), avec H un tas et F un frame, vérifie la propriété P si les quatre conditions suivantes sont vraies :*

- $p_{mrk} \in dom(H)$ *et* $[\![H, p_{mrk}]\!]^{mrk} = M$;
- $p_{set} \in dom(H)$ *et* $[\![H, p_{set}]\!]^{set}$ *est un ensemble de marquages ;*
- $F(x_{mrk}) = p_{mrk}$;
- $F(x_{set}) = p_{set}$. ◊

La correction de la fonction $succ_t$ est prouvée en se basant sur deux lemmes. Le premier pour de montrer que l'évaluation un bloc $header_{t,k}$, dans un contexte particulier qui nous convient, aboutit forcément au bloc $footer_{t,k+1}$ sans passer par les blocs d'indice plus grand que k, toujours avec un contexte qui nous convient.

5.3.1 Premier lemme

LEMME 5.4. *Quel que soit le nombre n de places en entrée de la transition t et pour tout tas H et tout frame F tels que (H, F) vérifie la propriété P, on a :*

$$\left(\ P(header_{t,n})\ \right)_{H,F} \rightsquigarrow^* \left(\ P(footer_{t,n+1})\ \right)_{H',F'}$$

avec (H', F') qui a vérifie la propriété P et sans être passé par les blocs d'indice supérieur à n, i.e.*,*

$$\left(\ P(header_{t,n})\ \right)_{H,F} \rightsquigarrow^* \left(\ P(block_{t,i})\ \right)_{H'',F''}$$

pour $i > n$ et $block \in \{header, loop, body, footer\}$ n'est pas un préfixe strict de la réduction ci-dessus.

Preuve

La preuve se fait par induction sur n, avec une deuxième induction sur le nombre de jetons qu'il nous reste à traiter dans la n-ème place au niveau du cas inductif.

1. Cas de base : il suffit de réaliser toutes les réductions possibles, on remarque qu'on traite forcément l'instruction "$br\ footer_{t,1}$". Les couples (H, F) vérifient la propriété P, la condition pour les tas découle des spécifications des fonctions appelées car on ne fait pas de modifications explicites sur le tas et la condition pour les frames vient du fait que LLVM est en forme SSA. On peut s'en assurer en réduisant les blocs concernés :

$$P(header_{t,0}) \stackrel{\text{df}}{=} \begin{cases} c_g = fcall\ g_t(x_1, \ldots, x_n) \\ br\ c_g,\ body_{t,0},\ footer_{t,1} \end{cases}$$

$$P(body_{t,0}) \stackrel{\text{df}}{=} \begin{cases} x'_{mrk} = fcall\ fire_t(x_{mrk}, x_1, \ldots, x_n) \\ pcall\ add_{set}(x_{set}, x'_{mrk}) \\ br\ footer_{t,1} \end{cases}$$

2. Cas inductif : on suppose que pour tout tas H et tout frame F tels que (H, F) vérifie la propriété P, on a :

$$\left(\ P(header_{t,n-1})\ \right)_{H,F} \rightsquigarrow^* \left(\ P(footer_{t,n})\ \right)_{H',F'} \tag{5.25}$$

et (H', F') vérifie la propriété P.

Pour prouver ce cas, on va réduire en partant du bloc $header_{t,n}$ avec un couple (H, F) qui vérifie la propriété P. Une fois arrivé à une certaine position dans le bloc $loop_{t,n}$ on fera une deuxième preuve par induction afin de montrer qu'on arrive à la configuration souhaitée.

Soit (H_0, F_0) un couple qui vérifie la propriété P, avec H_0 un tas et $F_0 \stackrel{\text{df}}{=} (l_{p,F_0},\ header_{t,n},\ \beta_{F_0})$ un frame où l_{p,F_0} est une étiquette quelconque. On déroule la définition de $P(header_{t,n})$:

$$P(header_{t,n}) \stackrel{\text{df}}{=} \begin{cases} x_{s_k} = fcall\ get_{s_k}(x_{mrk}) \\ s_{s_k} = fcall\ size_{s_k}(x_{s_k}) \\ br\ loop_{t,k} \end{cases}$$

On commence les réductions :

$$\left(\ P(header_{t,n})\ \right)_{H_0,F_0} \rightsquigarrow^* \left(\ P(loop_{t,n})\ \right)_{H_0,F_1}$$

où $F_1 = (header_{t,n},\ loop_{t,n},\ \beta_F \oplus \{x_{s_n} \mapsto p_{s_n}, s_{s_n} \mapsto card(M(s_n))\})$. La propriété P étant vérifiée, il faut remarquer que p_{mrk} est un pointeur sur une structure de marquage, p_{s_n} est le pointeur sur la place s_n de cette structure, et donc les deux appels de fonction sont légitimes.
De même on déroule la définition de $P(loop_{t,n})$:

$$P(loop_{t,n}) \stackrel{\text{df}}{=} \begin{cases} i_{s_n} = phi\ (s_{s_n}, header_{t,n}), (i'_{s_n}, footer_{t,n}) \\ c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1} \end{cases}$$

et on continue à réduire :

$$\begin{array}{c} \left(\begin{array}{l} i_{s_n} = phi\ (s_{s_n}, header_{t,n}), (i'_{s_n}, footer_{t,n}) \\ c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1} \end{array} \right)_{H_0,F_1} \\ \rightsquigarrow^* \left(\begin{array}{l} c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1} \end{array} \right)_{H_0,F_2} \end{array} \tag{5.26}$$

où $F_2 = F_1 \oplus \{i_{s_n} \mapsto card(M(s_n))\}$. À partir de maintenant on va montrer que cette configuration se réduit à $footer_{t,n+1}$ par induction sur la valeur de i_{s_n} dans la valuation courante. Plus précisément on montre que pour tout tas H et tout frame F, tels que (H, F) vérifie la propriété P et $l_{c,F} \stackrel{\text{df}}{=} loop_{t,n}$, on a :

$$\begin{array}{l} \forall k \in \mathbb{N}, \quad \left(\begin{array}{l} c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1} \end{array} \right)_{\substack{H, \\ F \oplus \{i_{s_n} \mapsto k\}}} \\ \qquad\qquad \rightsquigarrow^* \left(\ P(footer_{t,n+1})\ \right)_{H',F'} \end{array} \tag{5.27}$$

avec (H', F') qui vérifie la propriété P.
Ce résultat nous suffit car la configuration à laquelle on est arrivé (5.26) est un cas particulier de (5.27).

a) cas de base : soit H_0 un tas et F_0 un frame tels que décrits dans l'énoncé. On a $F_0(i_{s_n}) = 0$, il suffit de réduire :

$$\begin{array}{c} \left(\begin{array}{l} c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1} \end{array} \right)_{H_0,F_0} \\ \rightsquigarrow^* \left(\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1}\ \right)_{H_0,F_1} \\ \rightsquigarrow^* \left(\ P(footer_{t,n+1})\ \right)_{H_1,F_2} \end{array}$$

où

$$F_1 \stackrel{\text{df}}{=} F_0 \oplus \{c_{s_n} \mapsto false\}$$

$$F_2 = (loop_{t,n}, footer_{t,n+1}, \beta_{F_1})$$

On arrive bien au bloc $footer_{n+1}$ avec $H_0 = H_1$ et

$$\beta_{F_2} = \beta_{F_1} = F_0 \oplus \{i_{s_n} \mapsto k\} \oplus \{c_{s_n} \mapsto false\}$$

on a (H_1, F_2) qui vérifie la propriété P.

b) cas inductif : soit H_0 un tas et F_0 un frame tels que décrits dans l'énoncé. On a $F_1(i_{s_n}) = k$, on commence les réductions :

$$\begin{array}{c}\left(\begin{array}{l} c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1} \end{array}\right)_{H_0,F_1} \\ \rightsquigarrow^* \left(\begin{array}{l} br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1} \end{array}\right)_{H_0,F_2} \\ \rightsquigarrow^* \left(\begin{array}{l} P(body_{t,n}) \end{array}\right)_{H_0,F_3}\end{array}$$

où

$$F_2 \stackrel{\text{df}}{=} F_1 \oplus \{c_{s_n} \mapsto true\}$$
$$F_3 \stackrel{\text{df}}{=} (loop_{t,n},\ body_{t,n},\ \beta_{F_2})$$

On continue les réductions :

$$P(body_{t,k}) \stackrel{\text{df}}{=} \left\{\begin{array}{l} x_k = fcall\ nth_{s_k}(x_{s_k}, i_{s_k}) \\ br\ header_{t,k-1} \end{array}\right.$$

$$\left(\begin{array}{l} P(body_{t,n}) \end{array}\right)_{H_0,F_3} \rightsquigarrow^* \left(\begin{array}{l} P(header_{t,n-1}) \end{array}\right)_{H_0,F_4}$$

où

$$F_4 \stackrel{\text{df}}{=} (body_{t,n},\ header_{t,n-1},\ \beta_{F_3} \oplus \{x_k \mapsto v\})$$

Par hypothèse d'induction (5.25) on sait que :

$$\left(\begin{array}{l} P(header_{t,n-1}) \end{array}\right)_{H_0,F_4} \rightsquigarrow^* \left(\begin{array}{l} P(footer_{t,n}) \end{array}\right)_{H_1,F_5}$$

et (H_1, F_5) vérifie la propriété P. On continue de réduire :

$$\left(\begin{array}{l} i'_{s_n} = add\ i_{s_n},\ -1 \\ br\ loop_{t,n} \end{array}\right)_{H_1,F_5} \rightsquigarrow^* \left(\begin{array}{l} P(loop_{t,n}) \end{array}\right)_{H_1,F_6}$$

où

$$F_6 \stackrel{\text{df}}{=} (l_{c,F_4},\ loop_{t,n},\ \beta_{F_4} \oplus \{i'_{s_n} \mapsto k-1\})$$

car $F_5(i_{s_n}) = k$. Cette valeur n'a pas pu être modifiée car on n'est pas repassé dans le bloc $loop_{t,n}$ (LLVM est en forme SSA). On arrive donc à

$$P(body_{t,k}) \stackrel{\text{df}}{=} \left\{\begin{array}{l} i_{s_n} = phi\ (s_{s_n}, header_{t,n}), (i'_{s_n}, footer_{t,n}) \\ c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1} \end{array}\right.$$

$$\begin{array}{c}\left(\begin{array}{l} i_{s_n} = phi\ (s_{s_n}, header_{t,n}), (i'_{s_n}, footer_{t,n}) \\ c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1} \end{array}\right)_{H_1,F_6} \\ \rightsquigarrow^* \left(\begin{array}{l} c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1} \end{array}\right)_{H_1,F_7}\end{array}$$

où $F_7 = F_6 \oplus \{i_{s_n} \mapsto k-1\}$. On conclue par l'hypothèse d'induction (5.27) car la valeur de i_{s_n} a été décrémentée de 1 et (H_1, F_7) vérifie la propriété P.

Ceci conclue l'induction et la preuve du lemme. □

5.3.2 Terminaison de la fonction $succ_t$

Avec le lemme précédent on peut d'ores et déjà prouver la terminaison de la fonction $succ_t$.

Corollaire 5.5. *Dans les mêmes conditions que le lemme 5.4, l'appel de la fonction $succ_t$ termine.*

Preuve

C'est une conséquence immédiate du lemme précédent, car l'évaluation du bloc d'étiquette $succ_t(\!| x_{mrk} |\!)$ conduit au bloc $header_n$ et donc par le lemme précédent au bloc $footer_{n+1}$. □

5.3.3 Second lemme

Le second lemme sert à montrer qu'on énumère toutes les combinaisons possibles de jetons des places en entrée de la transition t. En particulier on l'utilise pour montrer qu'on énumère tous les modes de la transition t.

Lemme 5.6. *Soit H un tas et F un frame tels que (H, F) vérifie la propriété P. Soit $\beta = \{x_n \mapsto v_n, \ldots, x_1 \mapsto v_1\}$ une valuation LLVM telle que $[\![H, v_i]\!]^{t(s_i)} \in M(s_i)$ pour $1 \leq i \leq n$. Si*

$$\left(\ P(header_{t,n})\ \right)_{H,F} \rightsquigarrow^* \left(\ P(header_{t,0})\ \right)_{H',F'}$$

alors $\beta \subseteq \beta_{F'}$ et (H', F') vérifie la propriété P. De plus on s'assure de n'être passé par aucun bloc indexé par i tel que $i > n$, i.e.*,*

$$\left(\ P(header_{t,n})\ \right)_{H,F} \rightsquigarrow^* \left(\ P(block_{t,i})\ \right)_{H'',F''}$$

pour $i > n$ et $block \in \{header, loop, body, footer\}$ n'est pas un préfixe strict de la réduction ci-dessus.

Preuve

La preuve est faite par induction sur n. De même que dans le lemme précédent on fera une deuxième induction au niveau du cas inductif sur le nombre de jetons de la place s_n qu'il nous reste à traiter.

1. Cas de base, $n = 0$: trivial

   $$\left(\ P(header_{t,0})\ \right)_{H,F} \rightsquigarrow^* \left(\ P(header_{t,0})\ \right)_{H',F'}$$

   avec $H = H'$, $F = F'$ et $\beta = \emptyset \subseteq \beta_F$.

2. Cas inductif. On suppose que pour tout couple (H, F) qui vérifie la propriété P on a :

   $$\left(\ P(header_{t,n-1})\ \right)_{H,F} \rightsquigarrow^* \left(\ P(header_{t,0}\ \right)_{H',F'} \tag{5.28}$$

   avec $\{x_{n-1} \mapsto v_{n-1}, \ldots, x_1 \mapsto v_1\} \subseteq \beta_{F'}$ et (H', F') qui vérifie la propriété P. De plus on sait qu'on n'est passé par aucun bloc d'indice $i \geq n$.

Soit H_0 un tas et F_0 un frame tels que décrits dans l'énoncé. On commence à réduire le cas inductif :

$$P(header_{t,n}) \stackrel{\text{df}}{=} \begin{cases} x_{s_n} = fcall\ get_{s_n}(x_{mrk}) \\ s_{s_n} = fcall\ size_{s_n}(x_{s_n}) \\ br\ loop_{t,n} \end{cases}$$

$$\left(\ P(header_{t,n})\ \right)_{H_0,F_0} \rightsquigarrow^* \left(\ P(loop_{t,n})\ \right)_{H_0,F_1}$$

où $F_1 \stackrel{\text{df}}{=} (l_{c,F_0},\ loop_{t,n},\ \beta_{F_0} \oplus \{x_{s_n} \mapsto p_{s_n}, s_{s_n} \mapsto card(dom(M(s_n)))\})$ avec $[\![H, p_{s_n}]\!]^{s_n} = [\![H, p_{mrk}]\!]^{mrk}(s_n) = M(s_n)$.
On continue les réductions :

$$P(loop_{t,n}) \stackrel{\text{df}}{=} \begin{cases} i_{s_n} = phi\ (s_{s_n}, header_{t,n}), (i'_{s_n}, footer_{t,n}) \\ c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n-1} \end{cases}$$

$$\left(\ P(loop_{t,n})\ \right)_{H_0,F_1} \rightsquigarrow^* \left(\begin{array}{l} c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n-1} \end{array}\right)_{H_0,F_2}$$

où $F_2 \stackrel{\text{df}}{=} F_1 \oplus \{i_{s_n} \mapsto k\}$ avec $k = card(M(s_n))$.
Ici on commence la deuxième induction, on montre que cette configuration se réduit à la configuration souhaitée par induction sur k. Plus précisément on va prouver que :
Pour tout tas H et tout frame F tels que (H, F) vérifie la propriété P et $F(i_{s_n}) = k$ on a :

$$\left(\begin{array}{l} c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n-1} \end{array}\right)_{H,F} \rightsquigarrow^* \left(\ P(header_{t,0}\ \right)_{H',F'}$$

avec $\{x_n \mapsto v_n, \ldots, x_1 \mapsto v_1\} \subseteq \beta_{F'}$ et (H', F') qui vérifie la propriété P. De plus on s'assurera qu'on ne passe par aucun bloc d'indice i tel que $i > n$.

a) cas de base $k = 0$: trivialement vrai car il est impossible que

$$[\![H, p_{s_n}]\!]^{s_n} = M(s_n)$$

soit vide par hypothèse du lemme.

b) cas inductif : on suppose que pour tout tas H et tout frame F tels que (H, F) vérifie la propriété P et $F(i_{s_n}) = k - 1$, on a :

$$\left(\begin{array}{l} c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n-1} \end{array}\right)_{H,F} \rightsquigarrow \left(\ P(header_{t,0})\ \right)_{H',F'} \tag{5.29}$$

avec $\{x_n \mapsto v_n, \ldots, x_1 \mapsto v_1\} \subseteq \beta_{F'}$ et (H', F') qui vérifie la propriété P. De plus on sait que cette réduction ne passe par aucun bloc d'indice i tel que $i > n$.

On commence le cas inductif en effectuant la séquence de réductions suivante

$$\begin{array}{c}\left(\begin{array}{l} c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1} \end{array}\right)_{H_0,F_2} \\ \rightsquigarrow^* \left(\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n+1}\ \right)_{H_0,F_3} \\ \rightsquigarrow^* \left(\begin{array}{l} x_n = fcall\ nth_{s_n}(x_{s_n}, i_{s_n}) \\ br\ header_{t,n-1} \end{array}\right)_{H_0,F_4} \\ \rightsquigarrow^* \left(\ P(header_{t,n-1})\ \right)_{H_0,F_5}\end{array}$$

avec les frames :

$$F_3 \stackrel{\text{df}}{=} F_2 \oplus \{c_{s_n} \mapsto true\}$$

$$F_4 \stackrel{\text{df}}{=} (l_{c,F_3},\ body_{t,n},\ \beta_{F_3})$$

$$F_5 \stackrel{\text{df}}{=} (loop_{t,n},\ body_{t,n},\ \beta_{F_2} \oplus \{c_{s_n} \mapsto true, x_n \mapsto v_n\})$$

On a deux cas :

i. $(x_n, v_n) \in \beta$: comme (H_0, F_5) vérifie la propriété P on applique l'hypothèse d'induction (5.28).

$$\left(\ P(header_{t,n-1})\ \right)_{H_0,F_5} \rightsquigarrow^* \left(\ P(header_{t,0})\ \right)_{H_1,F_6}$$

avec $\{x_{n-1} \mapsto v_{n-1}, \ldots, x_1 \mapsto v_1\} \subseteq \beta_{F_6}$ et (H_1, F_6) qui vérifie la propriété P. Mais comme $\{x_n \mapsto v_n\} \subseteq \beta_{F_5}$ et qu'on n'est pas repassé par le bloc $loop_{t,n}$ on a aussi $\{x_n \mapsto v_n\} \subseteq \beta_{F_6}$ puisque LLVM respecte la forme SSA. On a donc bien $\{x_n \mapsto v_n, \ldots, x_1 \mapsto v_1\} \subseteq \beta_{F_6}$.

ii. $(x_n, v_n) \notin \beta$: dans ce cas, on applique le lemme 5.4 et on arrive dans le bloc $footer_{t,n}$ avec un tas H_1 et un frame F_6 tels que (H_1, F_6) vérifie la propriété P.

$$\left(\ P(header_{t,n-1})\ \right)_{H_0,F_5} \rightsquigarrow^* \left(\ P(footer_{t,n})\ \right)_{H_1,F_6}$$

De plus on sait que cette réduction ne passe par aucun bloc d'indice i tel que $i \geq n$. On déroule la définition du bloc $footer_{t,n}$:

$$P(footer_{t,n}) \stackrel{\text{df}}{=} \left\{\begin{array}{l} i'_{s_n} = add\ i_{s_n},\ -1 \\ br\ loop_{t,n} \end{array}\right.$$

On effectue les réductions suivantes :

$$\left(\ P(footer_{t,n})\ \right)_{H_1,F_6} \rightsquigarrow^* \left(\ P(loop_{t,n})\ \right)_{H_1,F_7}$$

avec $F_7 \stackrel{\text{df}}{=} (l_{c,F_6},\ loop_{t,n},\ \beta_{F_6} \oplus \{i'_{s_n} \mapsto k-1\})$ car la valeur de i_{s_n} n'a pas pu être modifiée (on n'est pas repassé par le bloc $loop_{t,n}$). On continue les réductions :

$$\begin{array}{c}\left(\begin{array}{l} i_{s_n} = phi\ (s_{s_n}, header_{t,n}), (i'_{s_n}, footer_{t,n}) \\ c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n-1} \end{array}\right)_{H_1,F_7} \\ \rightsquigarrow^* \left(\begin{array}{l} c_{s_n} = icmp\ >,\ i_{s_n},\ 0 \\ br\ c_{s_n},\ body_{t,n},\ footer_{t,n-1} \end{array}\right)_{H_1,F_8}\end{array}$$

avec $F_8 \stackrel{\text{df}}{=} F_7 \oplus \{i_{s_n} \mapsto k-1\}$. On conclue en appliquant l'hypothèse d'induction (5.29) car (H_1, F_8) vérifie la propriété P et la valeur de i_{s_n} a été décrémentée de 1.

Ce qui conclue la preuve. □

5.3.4 Correction de $succ_t$

Pour prouver la correction des fonctions $succ_t$ on rappelle le théorème 4.12 du chapitre 4 correspondant.

Soient F un frame et p_{mrk} un pointeur d'un tas H tels que $p_{mrk} \downarrow_H = dom(H)$, $[\![H, p_{mrk}]\!]^{mrk} = M$ et $\beta_F(x_{mrk}) = p_{mrk}$. Si

$$\left(\ fcall\ succ_t(x_{mrk})\ \right)_{H,F} \rightsquigarrow^* \left(\ p_{set}\ \right)_{H \oplus H'}$$

alors

$$dom(H) \cap dom(H') = \emptyset \tag{5.30}$$

$$[\![H \oplus H', p_{set}]\!]^{set} = \{M' \mid \exists \beta,\ M[t, \beta\rangle M'\} \tag{5.31}$$

Preuve

D'après la règle *fcall* il nous suffit de montrer que

$$\left(\ P(succ_t(\!|\, x_{mrk} \,|\!))\ \right)_{H,F} \rightsquigarrow^* \left(\ p_{set}\ \right)_{H \oplus H'}$$

où $F \stackrel{\text{df}}{=} (\bot, succ_t(\!|\, x_{mrk} \,|\!), \{x_{mrk} \mapsto p_{mrk}\})$ avec les conséquences (5.30) et (5.31). On commence par montrer la conséquence (5.30) de ce théorème. On remarque qu'on ne manipule ni les pointeurs ni les données du tas H explicitement, ils sont manipulés et éventuellement modifiés à travers des appels de fonction. On vérifie que ces fonctions ne peuvent modifier le tas H :

- les fonction get_{s_k}, $size_{s_k}$ et nth_{s_k} pour $1 \leq k \leq n$: ne modifient pas le tas d'après leur spécification ;
- la garde g_t, n'a pas d'effets de bord visibles du point de vue de notre programme, elle ne modifie pas H ;
- la fonction $fire_t$ ne modifie pas H d'après son théorème de correction (théorème 4.10) ;
- les autres fonctions peuvent être exécutés dans un tas séparé de H donc d'après le théorème d'extensionnalité (4.7), elles ne modifient pas le tas H.

La conséquence (5.30) est bien vérifiée.

On montre la conséquence (5.31) en deux étapes. La première étape consiste à montrer que tous les marquages successeurs sont contenus dans l'ensemble retourné par la fonction, c'est-à-dire que :

$$\{M' \mid \exists \beta,\ M[t, \beta\rangle M'\} \subseteq [\![H \oplus H', p_{set}]\!]^{set}$$

La seconde étape consiste à montrer que les seuls marquages qu'on ajoute à $[\![H \oplus H', p_{set}]\!]^{set}$ sont les marquages successeurs ce qui donne l'inclusion dans l'autre sens.

($\subseteq$) Soit $\beta = \{x_n \mapsto v_n, \ldots, x_1 \mapsto v_1\}$ un mode tel que $M[t, \beta\rangle M'$. Il suffit de montrer que le marquage M' est ajouté dans l'ensemble produit ; en effet, la fonction add_{set} est la seule qui manipule le pointeur p_{set}, les autres fonctions pouvant être exécutées en extensionnalité, elles ne peuvent pas modifier la structure pointée par p_{set} (d'après le théorème d'extensionnalité 4.7 et les spécifications des fonctions).

Soit H_0 un tas et p_{mrk} un pointeur de H_0 tels que décrits dans l'énoncé. Soit $F_0 \stackrel{\text{df}}{=} (\bot, succ_t(\!| x_{mrk} |\!), \{x_{mrk} \mapsto p_{mrk}\})$. On commence les réductions :

$$P(succ_t(\!| x_{mrk} |\!)) \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x_{set} = fcall\ alloc_{set}() \\ br\ header_{t,n} \end{array} \right.$$

$$\left(\ P(succ_t(\!| x_{mrk} |\!))\ \right)_{H_0,F_0} \rightsquigarrow^* \left(\ P(header_{t,n})\ \right)_{H_0 \oplus H_1, F_1}$$

avec

$$F_1 \stackrel{\text{df}}{=} (succ_t,\ header_{t,n},\ \beta_F \oplus \{x_{set} \mapsto p_{set}\})$$
$$[\![H_0 \oplus H_1, p_{set}]\!]^{set} = \emptyset$$

On peut appliquer le lemme 5.6 avec la valuation β car $(H_0 \oplus H_1, F_1)$ vérifie la propriété P. Il en résulte :

$$\left(\ P(header_{t,n})\ \right)_{H \oplus H_1, F_1} \rightsquigarrow^* \left(\ P(header_{t,0})\ \right)_{H_2, F_2}$$

avec (H_2, F_2) qui vérifie la propriété P et $\beta \in \beta_{F'}$. Par définition :

$$P(header_{t,0}) \stackrel{\text{df}}{=} \left\{ \begin{array}{l} c_g = fcall\ g_t(x_1, \ldots, x_n) \\ br\ c_g,\ body_{t,0},\ footer_{t,1} \end{array} \right.$$

Comme β est un mode valide on a la réduction suivante :

$$\left(\ P(header_{t,0})\ \right)_{H_2,F_2} \rightsquigarrow^* \left(\ P(body_{t,0})\ \right)_{H_2 \oplus H_3, F_3}$$

avec

$$F_3 = (header_{t,0},\ body_{t,0},\ \beta_{F_2} \oplus \{c_s \mapsto true\})$$

On continue à réduire avec la définition de la règle *assign* et de la spécification de la fonction $fire_t$:

$$P(header_{t,0}) \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x'_{mrk} = fcall\ fire_t(x_{mrk}, x_1, \ldots, x_n) \\ pcall\ add_{set}(x_{set}, x'_{mrk}) \\ br\ footer_{t,1} \end{array} \right.$$

$$\left(\ P(body_{t,0})\ \right)_{H_2 \oplus H_3, F_3} \rightsquigarrow^* \left(\begin{array}{l} pcall\ add_{set}(x_{set}, x'_{mrk}) \\ br\ footer_{t,1} \end{array} \right)_{H_2 \oplus H_3 \oplus H_4, F_4}$$

De plus on obtient :

$$F_4 \stackrel{\text{df}}{=} F_3 \oplus \{x_{mrk} \mapsto p'_{mrk}\}$$
$$[\![H_4, p'_{mrk}]\!]^{mrk} = M'$$

On pose $H_5 \stackrel{\text{df}}{=} H_2 \oplus H_3 \oplus H_4$ et on conclue avec la dernière réduction :

$$\left(\begin{array}{l} pcall\ add_{set}(x_{set}, x'_{mrk}) \\ br\ footer_{t,1} \end{array} \right)_{H_5,F_4} \rightsquigarrow^* \left(\ br\ footer_{t,1}\ \right)_{H_5 \oplus H_6, F_4}$$

avec par la spécification de la fonction add_{set}

$$[\![H_4 \oplus H_5, p_{set}]\!]^{set} = [\![H_4, p_{set}]\!]^{set} \cup \{M'\}$$

On ajoute bien le marquage M' à l'ensemble.

($\supseteq$) On remarque que les seules opérations effectuées sur p_{set} et donc sur $[\![H \oplus H', p_{set}]\!]^{set}$ sont les ajouts de marquages car la fonction $fire_t$ renvoie un marquage successeur par un mode β (voir théorème de correction 4.10). Il en résulte que si un marquage appartient à $[\![H \oplus H', p_{set}]\!]^{set}$ alors il a été ajouté dans le bloc $body_{t,0}$ par l'instruction

$$pcall\ add_{set}(x'_{mrk}, x_1, \ldots, x_n) \tag{5.32}$$

Les blocs sur lesquels on va raisonner sont :

$$P(header_{t,0}) \stackrel{\text{df}}{=} \left\{ \begin{array}{l} c_g = fcall\ g_t(x_1, \ldots, x_n) \\ br\ c_g,\ body_{t,0},\ footer_{t,1} \end{array} \right.$$

$$P(body_{t,0}) \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x'_{mrk} = fcall\ fire_t(x_{mrk}, x_1, \ldots, x_n) \\ pcall\ set\_add(x_{set}, x'_{mrk}) \\ br\ footer_{t,1} \end{array} \right.$$

On va montrer que si on est arrivé à l'instruction (5.32) alors on effectue bien un ajout de marquage successeur. Supposons qu'on est dans l'état suivant :

$$\left(\begin{array}{l} pcall\ add_{set}(x_{pset}, x_{pmrk}); \\ \ldots \end{array} \right)_{\substack{H \oplus H_0, \\ F \oplus \beta_0}} \tag{5.33}$$

Il faut montrer que $[\![H \oplus H_0, (F \oplus \beta_0)(x_{pmrk})]\!]^{mrk}$ est bien un marquage successeur. Si on est arrivé en (5.33), alors on vient de :

$$\left(\begin{array}{l} x_{pmrk} = fcall\ fire_t(x_{pmrk}, x_1, \ldots, x_n) \\ pcall\ add_{set}(x_{pset}, x_{pmrk}); \\ \ldots \end{array} \right)_{\substack{H \oplus H_1, \\ F \oplus \beta_1}} \tag{5.34}$$

il en suit que $\beta_0 = \beta_1 \oplus \{x_{pmrk} \mapsto p_{pmrk}\}$. Par le théorème de correction de la fonction $fire_t$, $[\![H \oplus H_0, p_{pmrk}]\!]^{mrk}$ est un marquage successeur si

$$\{x_1 \mapsto (F \oplus \beta_1)(x_1), \ldots, x_n \mapsto (F \oplus \beta_1)(x_n)\} \tag{5.35}$$

est un mode valide. Il nous reste à montrer que c'est bien un mode valide.

On continue notre raisonnement, si on est en (5.34) alors on vient du bloc $header_{t,0}$, et plus précisément de la configuration :

$$\left(\begin{array}{l} br\ c_g,\ body_{t,0},\ footer_{t,n} \end{array} \right)_{\substack{H \oplus H_1, \\ F \oplus \beta_1}} \tag{5.36}$$

comme on a fait le branchement en $body_{t,0}$, $(F \oplus \beta_1)(c_g) = true$. Si on est en (5.36) alors on vient de :

$$\left(\begin{array}{l} c_g = fcall\ g_t(x_1, \ldots, x_n) \\ br\ c_g,\ body_{t,0},\ footer_{t,n} \end{array} \right)_{\substack{H \oplus H_2, \\ F \oplus \beta_2}} \tag{5.37}$$

et $\beta_1 = \beta_2 \oplus \{c_g \mapsto true\}$. D'après la sémantique de la règle $assign_{expr}$ on a $(F \oplus \beta_1)(c_g) = true$ seulement si

$$\left(\begin{array}{l} fcall\ g_t(x_1, \ldots, x_n) \end{array} \right)_{\substack{H \oplus H_2, \\ F \oplus \beta_2}} \rightsquigarrow \left(\begin{array}{l} true \end{array} \right)_{H \oplus H_1} \tag{5.38}$$

La réduction (5.38) est possible seulement si

$$\{x_1 \mapsto (F \oplus \beta_2)(x_1), \ldots, x_n \mapsto (F \oplus \beta_2)(x_n)\}$$

est un mode valide. Il suffit de remarquer que

$$F \oplus \beta_1 = F \oplus \beta_2 \oplus \{c_g \mapsto \mathit{true}\} \oplus \{x_{pmrk} \mapsto p_{pmrk}\}$$

et donc $\{x_1 \mapsto (F \oplus \beta_1)(x_1), \ldots x_n \mapsto (F \oplus \beta_1)(x_n)\}$ est un mode valide (5.35). On a bien prouvé que si on ajoute un marquage alors c'est bien un marquage successeur car il correspond à un mode valide.

Par induction sur n, on peut ensuite prouver que les jetons proviennent bien des places en entrée de la transition t, ce qui permet de conclure. □

5.4

Correction des optimisations

Dans cette section, on réalise les preuves des optimisations de la section 4.8.

5.4.1 Optimisation du calcul des modes : la fonction $succ_t$ 1-bornée

Afin de prouver la correction de cette optimisation, on reprend les preuves de la section 5.3 et on les adapte au nouvel algorithme. Rappelons respectivement, les blocs initiaux et les blocs optimisés sur la figure 5.1 et la figure 5.2.

On reprend les mêmes hypothèses que dans la section 5.3. Les énoncés qui suivent sont définis dans le contexte où M esqt un marquage, t une transition à n places en entrée, p_{mrk} et p_{set} sont deux pointeur, x_{mrk} et x_{set} sont deux variables. De la même manière on considère la propriété P (définition 5.1, page 74).

$$P(\mathit{header}_{t,k}) \stackrel{\text{df}}{=} \begin{cases} x_{s_k} = \mathit{fcall}\ \mathit{get}_{s_k}(x_{mrk}) \\ s_{s_k} = \mathit{fcall}\ \mathit{size}_{s_k}(x_{s_k}) \\ \mathit{br}\ \mathit{loop}_{t,k} \end{cases}$$

$$P(\mathit{loop}_{t,k}) \stackrel{\text{df}}{=} \begin{cases} i_{s_k} = \mathit{phi}\ (s_{s_k}, \mathit{header}_{t,k}), (i'_{s_k}, \mathit{footer}_{t,k}) \\ c_{s_k} = \mathit{icmp}\ >,\ i_{s_k},\ 0 \\ \mathit{br}\ c_{s_k},\ \mathit{body}_{t,k},\ \mathit{footer}_{t,k+1} \end{cases}$$

$$P(\mathit{body}_{t,k}) \stackrel{\text{df}}{=} \begin{cases} x_k = \mathit{fcall}\ \mathit{nth}_{s_k}(x_{s_k}, i_{s_k}) \\ \mathit{br}\ \mathit{header}_{t,k-1} \end{cases}$$

$$P(\mathit{footer}_{t,k}) \stackrel{\text{df}}{=} \begin{cases} i'_{s_k} = \mathit{add}\ i_{s_k},\ -1 \\ \mathit{br}\ \mathit{loop}_{t,k} \end{cases}$$

Figure 5.1 – *Les blocs initiaux de la fonction $succ_t$ spécifiques à la place s_k.*

$$
\begin{array}{rcl}
P(header_{t,k}) & \stackrel{\text{df}}{=} & \left\{ \begin{array}{l} x_{p_{q_k}} = fcall\ get_{q_k}(x_{p_M}) \\ br\ loop_{t,k} \end{array} \right. \\
P(loop_{t,k}) & \stackrel{\text{df}}{=} & \left\{ \begin{array}{l} c_{q_k} = fcall\ notempty(x_{p_{q_k}}) \\ br\ c_{q_k},\ body_{t,k},\ footer_{t,k+1} \end{array} \right. \\
P(body_{t,k}) & \stackrel{\text{df}}{=} & \left\{ \begin{array}{l} x_k = fcall\ nth_{q_k}(x_{p_{q_k}}, 0) \\ br\ header_{t,k-1} \end{array} \right. \\
P(footer_{t,k}) & \stackrel{\text{df}}{=} & \{\ br\ footer_{t,k+1}
\end{array}
$$

FIGURE 5.2 – *Les blocs optimisés de la fonction $succ_t$ spécifiques à la place s_k.*

LEMME 5.7. *Quel que soit le nombre n de places en entrée de la transition t et pour tout tas H et tout frame F tels que (H, F) vérifie la propriété P, on a :*

$$\left(\ P(header_{t,n})\ \right)_{H,F} \rightsquigarrow^* \left(\ P(footer_{t,n+1})\ \right)_{H',F'}$$

avec (H', F') qui vérifie la propriété P et sans être passé par les blocs d'indice supérieur à n, i.e.*,*

$$\left(\ P(header_{t,n})\ \right)_{H,F} \rightsquigarrow^* \left(\ P(block_{t,i})\ \right)_{H'',F''}$$

pour $i > n$ et $block \in \{header, loop, body, footer\}$ n'est pas un préfixe strict de la réduction ci-dessus.

Preuve On reprend la preuve du lemme 5.4 de la section 5.3. Dans le cas inductif, on a deux sous-cas à traiter :

1. La place n utilise les blocs initiaux : dans ce cas on a la même preuve.
2. La place n utilise les blocs optimisés : on reprends le même schéma de preuve que pour les blocs initiaux en plus simple : après le bloc $footer_k$ on atteint le bloc $footer_{k+1}$.

□

LEMME 5.8. *Soit H un tas et F un frame tels que (H, F) vérifie la propriété P. Soit $\beta = \{x_n \mapsto v_n, \ldots, x_1 \mapsto v_1\}$ une valuation LLVM telle que $[\![H, v_i]\!]^{t(s_i)} \in M(s_i)$ pour $1 \leq i \leq n$. Si*

$$\left(\ P(header_{t,n})\ \right)_{H,F} \rightsquigarrow^* \left(\ P(header_{t,0})\ \right)_{H',F'}$$

alors $\beta \subseteq \beta_{F'}$ et (H', F') vérifie la propriété P. De plus on s'assure de n'être passé par aucun bloc indexé par i tel que $i > n$, i.e.*,*

$$\left(\ P(header_{t,n})\ \right)_{H,F} \rightsquigarrow^* \left(\ P(block_{t,i})\ \right)_{H'',F''}$$

pour $i > n$ et $block \in \{header, loop, body, footer\}$ n'est pas un préfixe strict de la réduction ci-dessus.

Preuve On reprends la preuve du lemme 5.6 de la section 5.3. Dans le cas inductif, il faut traiter les deux cas suivants :

1. La place n utilise les blocs initiaux : dans ce cas on a la même preuve.
2. La place n utilise les blocs optimisés : la place ne peut pas être vide sinon on contredit les hypothèses (on a un mode et donc on a un jeton qui nous convient). Comme la place ne peut contenir que le jeton $\bullet$, la réduction conduit au bloc $header_{n-1}$, on conclue par l'hypothèse d'induction.

□

Théorème 5.9. *Soient* $F \stackrel{\text{df}}{=} (\bot,\ succ_t,\ \{x_{mrk} \mapsto p_{mrk}\})$ *un frame et* p_{mrk} *un pointeur d'un tas* H *tels que* $p_{mrk}{\downarrow_H} = H$ *et* $[\![H, p_{mrk}]\!]^{mrk} = M$. *Si*

$$\left(\ P(succ_t(\!|\, x_{mrk} \,|\!))\ \right)_{H,F} \rightsquigarrow^* \left(\ p_{set}\ \right)_{H \oplus H'}$$

alors

$$dom(H) \cap dom(H') = \emptyset \tag{5.39}$$

$$[\![H \oplus H', p_{set}]\!]^{set} = \{M' \mid \exists \beta,\ M[t, \beta\rangle M'\} \tag{5.40}$$

Preuve Exactement la même preuve que le théorème 4.12 réalisée en section 5.3 mais avec les lemmes 5.7 et 5.8 qu'on vient de prouver. □

5.4.2 Place jeton noir 1-bornée

Rappelons l'implantation de la place 1-bornée jeton noir sur la figure 5.3 et les fonctions d'interprétation utilisées. La fonction d'interprétation de la structure des places :

$$\begin{array}{lll} [\![H, p_s]\!]^s = \emptyset & \text{si} & H(p_s) = (bool, false) \\ [\![H, p_s]\!]^s = \{\bullet\} & \text{si} & H(p_s) = (bool, true) \end{array}$$

La fonction d'interprétation de l'encodage des jetons :

$$[\![H, v]\!]^{t(s)} = \bullet \quad \text{si} \quad v \in \{true, false\}$$

On ne prouve pas la propriété de validité de la fonction d'interprétation car elle est triviale dans ce cas.

Dans la suite de cette section, on se placera sous les hypothèses suivantes : H est un tas, F est un frame, x et x_s sont deux variables, v est un booléen et p_s est un pointeur sur une place s, 1-bornée jeton noir, tels que $[\![H, p_s]\!]^s = \emptyset$, $F(x_s) = p_s$ et $F(x) = v$.

Dans les preuves qui suivent on utilise le fait que la place s est 1-bornée, c'est à dire que certaines opérations ne sont pas définies si la place est vide ou bien pleine.

Fonctions add_s, rem_s et $clear_s$

Pour la fonction add_s on a défini implantation suivante :

$$\begin{array}{l} P(add_s(\!|\, x_s, x \,|\!)) \stackrel{\text{df}}{=} \\ \quad store\ true,\ x_s \\ \quad ret \end{array}$$

Rappelons la spécification à vérifier :

$$\left(\ pcall\ add_s(x_s, x)\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H \oplus H', F} \tag{5.1}$$

$$dom(H) \cap dom(H') \subseteq p_s{\downarrow_H} \tag{5.2}$$

$$[\![H \oplus H', p_s]\!]^s = [\![H, p_s]\!]^s + \{[\![H, v]\!]^{t(s)}\} \tag{5.3}$$

Observation : *La place étant 1-bornée, si on appelle la fonction add_s alors la place est initialement vide.*

On prouve que cette spécification est correcte en la prouvant pour un tas H et un frame F restreints tels que $H = \{p_s \mapsto (bool, false)\}$ et $\beta_F = \{x_s \mapsto p_s, x \mapsto v\}$. La valeur du booléen dans le tas est connue car la place est initialement vide. On ne montrera pas la conséquence (5.2) car ce sera une conséquence immédiate d'une application du théorème d'extensionnalité (théorème 4.7) afin de revenir à l'énoncé initial. On sait que

$$\left(\ pcall\ add_s(x_s, x)\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H \oplus H', F}$$

seulement si

$$\left(\begin{array}{l} store\ true,\ x_s \\ ret \end{array}\right)_{H,F_0} \rightsquigarrow \left(\ ret\ \right)_{H \oplus H', F'}$$

avec $F_0 = (\bot, add_s(\!|\, x_s, x \,|\!), \{x_s \mapsto p_s, x \mapsto v\})$. Or, on a

$$\left(\begin{array}{l} store\ true,\ x_s \\ ret \end{array}\right)_{H_0,F_0} \rightsquigarrow \left(\ ret\ \right)_{\substack{H_0 \oplus H_1, \\ F_0}}$$

avec $H_1 = \{p_s \mapsto (bool, true)\}$. On a bien le résultat (5.1). Le résultat (5.3) est vrai car $[\![H \oplus H_1, p_s]\!]^s = \{\bullet\} = [\![H, p_s]\!]^s + \{[\![H, v]\!]^{t(s)}\}$.

$P(add_s(\!|\, x_s, x \,|\!)) \stackrel{\text{df}}{=}$
store true, x_s
ret

$P(rem_s(\!|\, x_s, x \,|\!)) \stackrel{\text{df}}{=}$
store false, x_s
ret

$P(copy_s(\!|\, x_s, x'_s \,|\!)) \stackrel{\text{df}}{=}$
x = load x_s
store x, x'_s
ret

$P(notempty_s(\!|\, x_s \,|\!)) \stackrel{\text{df}}{=}$
x = load x_s
ret x

$P(cons_s(\!|\,|\!)) \stackrel{\text{df}}{=}$
store false, x_s
ret

$P(size_s(\!|\, x_s \,|\!)) \stackrel{\text{df}}{=}$
x = load x_s
br x, l_1, l_2

$P(nth_s(\!|\, x_s, i \,|\!)) \stackrel{\text{df}}{=}$
ret true

$P(clear_s(\!|\, x_s \,|\!)) \stackrel{\text{df}}{=}$
store false, x_s
ret

$P(l_1) \stackrel{\text{df}}{=}$
ret 1

$P(l_2) \stackrel{\text{df}}{=}$
ret 0

FIGURE 5.3 – *Implantation d'une place 1-bornée de type jeton noir.*

Observation : *La place étant 1-bornée, si on appelle la fonction rem_s alors la place est initialement marquée.*

On utilise la même approche pour la fonction rem_s en supposant cette fois la place marquée, c'est-à-dire $[\![H, p_s]\!]^s = \{\bullet\}$. Quant à la fonction $clear_s$, on obtient le résultat souhaité par définition de la fonction d'interprétation.

Fonction $size_s$

Pour la fonction $size_s$, on a :

$$\begin{array}{c} P(size_s(\!|\, x_s \,|\!)) \stackrel{\text{df}}{=} \\ x = load\ x_s \\ br\ x,\ l_1,\ l_2 \end{array}$$

$$\begin{array}{ll} P(l_1) \stackrel{\text{df}}{=} & P(l_2) \stackrel{\text{df}}{=} \\ ret\ 1 & ret\ 0 \end{array}$$

La spécification souhaitée est :

$$\left(\ fcall\ size_{mset}(p_{mset})\ \right)_{H,F} \rightsquigarrow \left(\ card(dom([\![H, p_{mset}]\!]^{mset(t)}))\ \right)_H$$

On donne l'intuition de la preuve sans la faire en détail, on a deux cas possibles :

1. Si la place est vide alors on aura $H(p_s) = (bool, false)$, le saut conditionnel nous enverra à l'étiquette l_2 et on renverra alors la valeur 0 (domaine vide).
2. Si la place est pleine alors on aura $H(p_s) = (bool, true)$, le saut conditionnel nous enverra à l'étiquette l_1 et on renverra alors la valeur 1 (un seul élément dans le domaine).

Fonction $notempty_s$

On veut que la fonction

$$\begin{array}{c} P(notempty_s(\!|\, x_s \,|\!)) \stackrel{\text{df}}{=} \\ x = load\ x_s \\ ret\ x \end{array}$$

vérifie sa spécification

$$\left(\ fcall\ notempty_s(p_s)\ \right)_{H,S} \rightsquigarrow \left(\ card(dom([\![H, p_s]\!]^{mset(t)})) \neq \emptyset\ \right)_H$$

Par définition de la structure de données, si la place est vide alors $H(p_s) = (bool, false)$ et donc l'instruction $load\ x_s$ se réduira en $false$, on aura bien le résultat attendu car $\emptyset \neq \emptyset$ est faux. En revanche, si la place est marquée alors $H(p_s) = (bool, true)$ et donc l'instruction $load\ x_s$ se réduira en $true$, on aura le bon résultat.

Fonction nth_s

On a l'implantation suivante :

$$P(nth_s(\!| p_s, i |\!)) \stackrel{\text{df}}{=} \\ ret\ true$$

La spécification qu'elle doit vérifier est la suivante :

$$\left(\ fcall\ nth_s(p_s, i)\ \right)_{H,F} \rightsquigarrow \left(\ p_i\ \right)_H$$

$$[\![H, p_s]\!]^{mset(t)} = \{x_1 \mapsto k_1, \ldots, x_n \mapsto k_n\}$$
$$[\![H, p_i]\!]^t = x_i$$

L'implantation vérifie trivialement la spécification car :

- Si la place est vide alors appeler la fonction est illégal et donc le comportement est indéfini (la valeur renvoyée importe peu).
- Si la place est marquée alors on ne peut renvoyer qu'un seul jeton : le jeton noir. Le jeton noir étant encodé par un booléen quelconque la valeur *true* convient.

Fonction $copy_s$

Sous les mêmes hypothèses, soit p'_s un pointeur sur la place s d'une autre instance de la structure de marquage et x'_s une variable telle que $F(x'_s) = p'_s$. Dans ces conditions la fonction $copy_s$ est définie par :

$$P(copy_s(\!| x_s, x'_s |\!)) \stackrel{\text{df}}{=} \\ x = load\ x_s \\ store\ x,\ x'_s \\ ret$$

La spécification demandée par l'interface des places est :

$$\left(\ pcall\ copy_s(x_s, x'_s)\ \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H \oplus H', F}$$

$$dom(H) \cap dom(H') \subseteq x'_s{\downarrow_H} \tag{5.4}$$
$$[\![H, p_s]\!]^s = [\![H \oplus H', p_s]\!]^s = [\![H \oplus H', p'_s]\!]^s$$

De la même manière que pour la fonction add_s on va restreindre le tas H de façon à conclure en utilisant le théorème d'extensionnalité (théorème 4.7). Soit $H_0 \stackrel{\text{df}}{=} \{p_s \mapsto (bool, v), p'_s \mapsto (bool, v')\}$ un tas, et F_0 un frame tel que $\beta_{F_0} = \{x_s \mapsto p_s, x'_s \mapsto p'_s\}$. La conséquence (5.4) sera une conséquence immédiate de l'application du théorème d'extensionnalité. On sait que :

$$\left(\ pcall\ copy_s(x_s, x'_s)\ \right)_{H_0,F_0} \rightsquigarrow \left(\ skip\ \right)_{H_0 \oplus H', F_0}$$

seulement si

$$\left(\begin{array}{l} x = load\ x_s \\ store\ x,\ x'_s \\ ret \end{array}\right)_{H_0,F_1} \rightsquigarrow \left(\ ret\ \right)_{H_0 \oplus H', F'}$$

avec $F_1 = (\bot, copy_s(\!| x_s, x'_s |\!), \beta_{F_0})$. On commence à réduire :

$$\left(\begin{array}{l} x = load\ x_s \\ store\ x,\ x'_s \\ ret \end{array} \right)_{H_0,F_1} \rightsquigarrow \left(\begin{array}{l} store\ x,\ x'_s \\ ret \end{array} \right)_{H_0,F_2}$$

où $F_2 = F_1 \oplus \{x \mapsto v\}$ car $H(x_s) = (bool, v)$. On continue les réductions :

$$\left(\begin{array}{l} store\ x,\ x'_s \\ ret \end{array} \right)_{H_0,F_2} \rightsquigarrow \left(\begin{array}{l} ret \end{array} \right)_{H_0 \oplus H_1,F_2}$$

où $H_1 = \{p'_s \mapsto v\}$.

On a bien $[\![H_0, p_s]\!]^s = [\![H_0 \oplus H_1, p_s]\!]^s = [\![H_0 \oplus H_1, p'_s]\!]^s$ et on conclue par le théorème d'extensionnalité.

Fonction $cons_s$

La fonction $cons_s$ est définie par :

$$\begin{array}{l} P(cons_s(\!|\,|\!)) \stackrel{\text{df}}{=} \\ \quad x = alloc\ bool \\ \quad store\ false,\ x \\ \quad ret\ x \end{array}$$

Et doit respecter la spécification :

$$\left(\ fcall\ cons_s()\ \right)_{\emptyset,F} \rightsquigarrow \left(\ p_s\ \right)_H$$

$$[\![H, p_s]\!]^s = \emptyset$$

pour $F = (l, l', \{\})$. On reprend toujours la même approche, on a

$$\left(\ fcall\ cons_s()\ \right)_{\emptyset,F} \rightsquigarrow \left(\ p_s\ \right)_H$$

seulement si

$$\left(\begin{array}{l} x = alloc\ bool \\ store\ false,\ x \\ ret\ x \end{array} \right)_{\emptyset,F_0} \rightsquigarrow \left(\ p_s\ \right)_H$$

avec $F_0 = (\bot, cons(\!|\,|\!), \{\})$. On commence les réductions :

$$\left(\begin{array}{l} x_s = alloc\ bool \\ store\ false,\ x_s \\ ret\ x_s \end{array} \right)_{\emptyset,F_0} \rightsquigarrow \left(\begin{array}{l} store\ false,\ x_s \\ ret\ x_s \end{array} \right)_{H_0,F_1}$$

où $H_0 = \{p_s \mapsto (bool, \bot)\}$ et $F_1 = F_0 \oplus \{x_s \mapsto p_s\}$. On continue de réduire :

$$\left(\begin{array}{l} store\ false,\ x_s \\ ret\ x_s \end{array} \right)_{H_0,F_1} \rightsquigarrow \left(\ ret\ x_s\ \right)_{H_1,F_1}$$

où $H_1 = \{p_s \mapsto (bool, false)\}$. On finalise avec la dernière réduction :

$$\left(\ ret\ x_s\ \right)_{H_1,F_1} \rightsquigarrow \left(\ p_s\ \right)_{H_1}$$

On a bien $[\![H, p_s]\!]^s = \emptyset$ par définition de la fonction d'interprétation. □

6 Symétries et création dynamique de processus

L'exploration d'espaces d'états explicite stocke tous les états individuellement en mémoire, de par le problème d'explosion combinatoire, stocker tous ces états est souvent impossible. Il est aussi souvent impraticable de les calculer à cause du temps que ça prendrait. Une manière pour pallier ce problème est de mettre en oeuvre des techniques de réduction d'espaces d'états. Pour être intéressantes, ces réductions doivent préserver des propriétés pertinentes de manière à être complètes pour un but recherché (préservation de deadlocks, préservation de tirage de transitions, *etc.*). Dans notre cas on s'intéresse aux réductions par symétries qui préservent les données des places, le tirage de transitions et leurs modes (modulo données symétriques).

Les réductions par symétries pour les réseaux place-transition et les réseaux colorés ont été largement étudiées [108]. Dans ce chapitre on s'intéresse à une version étendue qui met en oeuvre des processus concurrents qui peuvent être crées dynamiquement pendant l'évolution du système étudié. Ce type de réductions n'a été que peu adressé dans la littérature.

On s'intéresse à des systèmes qui manipulent de multiples *processus* (ou *threads*) de manière explicite. Dans les réseaux de bas niveau différents processus sont représentés par des copies d'un sous-réseau dans le modèle, ainsi on simule le comportement de plusieurs instances d'un même processus [110, 118]. Avec les réseaux de haut niveau, on peut représenter l'identité d'un processus par une valeur, ce qui permet d'avoir plusieurs processus qui exécutent la même partie du réseau. Les deux approches sont équivalentes mais la seconde produit un réseau de Petri plus petit en structure, il est plus riche en jetons cependant. La coexistence de plusieurs processus au sein d'un système conduit immédiatement à une explosion combinatoire due aux entrelacement des opérations qu'ils réalisent. Il s'en suit, une augmentation exponentielle de la taille de l'espace d'états du système par rapport à la taille du réseau ; si l'espace d'état reste fini il peut être traité avec les mêmes techniques qu'habituellement. Cependant, pour être traités plus rapidement et avec moins de mémoire des techniques de réduction sont nécessaires.

En plus de la cohabitation de plusieurs processus, on souhaite ajouter une notion de création dynamique à l'exécution. La coexistence de plusieurs processus dans un système est rencontrée souvent, cependant la création dyna-

mique de ceux-ci l'est beaucoup moins [115, 116, 114]. La première raison de cette tendance est l'accentuation de l'explosion combinatoire due à l'ajout de non-déterminisme dans l'ordre de création des processus ainsi que leurs agencements, tout cela augmente la combinatoire dans le système. D'autre part, il est courant de faire mourir et recréer des processus durant l'exécution du système à l'infini, ce qui peut se traduire en la création de nouveaux processus qui n'avaient pas existé avant et qui peuvent être considérés comme nouvelles données du système. Cela peut conduire à des espaces d'états infinis, ce qui rend les techniques de réduction indispensables. Dans ce chapitre on va s'attaquer essentiellement à deux points :

- traiter ce genre de systèmes à comportements potentiellement infinis ;
- réduire leurs espaces d'états.

Des travaux ont déjà été réalisé pour proposer une base théorique pour adresser ce type de systèmes [115, 116, 114], cependant les méthodes de réduction associées n'étaient pas utilisables en pratique à cause de leur complexité.

Ici on revisite complètement la théorie présentée dans ces travaux et on propose une méthode *nouvelle* et *utilisable* en pratique.

Le contenu de ce chapitre n'a pas encore été publié. Cependant, un travail préliminaire a été publié dans [Fro12a]

6.1

Réseaux de Petri avec processus

Les systèmes qu'on manipules ont de multiples processus (ou threads) *anonymes* qui exécutent différentes parties du modèle. On suppose que le comportement du système ne peut pas être influencé par l'identité d'un processus mais seulement par les relations entre processus. Ici on s'intéresse à la relation parent, *i.e.*, étant donné deux processus a et b on peut savoir si a et un parent ou ancêtre de b et vice versa. Dans [115, 116, 114] une relation frère a aussi été considérée et adressée de manière optimiste dans [Fro12a].

6.1.1 Identifiants de processus

Chaque processus dans le système est identifié par un *identifiant de processus* (pid), en général dénoté par π, qui est un tuple d'entiers positifs. L'ensemble des pids est $\mathbb{P}$ et l'ensemble des préfixes d'un pid dénote ses ancêtres. Par exemple, pour le pid $\langle 1,2,3\rangle$, son parent est $\langle 1,2\rangle$ dont le parent est $\langle 1\rangle$. Cette relation de parent (immédiat) est notée avec l'opérateur $\lhd_1$ et sa clôture transitive (relation ancêtre) est notée $\lhd$. On a donc $\langle 1\rangle \lhd_1 \langle 1,2\rangle$ et $\langle 1\rangle \lhd \langle 1,2,3\rangle$, mais on *n'a pas* $\langle 1\rangle \lhd_1 \langle 1,2,3\rangle$. Un pid est construit en utilisant une opération de concaténation sur des fragments de pids, $\langle 1,2,3\rangle = \langle 1\rangle\cdot\langle 2\rangle\cdot\langle 3\rangle = \langle 1,2\rangle\cdot\langle 3\rangle = \langle 1\rangle\cdot\langle 2,3\rangle$. Finalement on introduit le *pid vide* $\langle\rangle$ pour simplifier les notations, cependant il faut garder à l'esprit que ce pid *ne peut pas* représenter un processus dans un modèle.

Definition 6.1 (identifiants de processus). *Les identifiants de processus (pids) sont des éléments de* $\mathbb{P} \stackrel{\text{df}}{=} (\mathbb{N}^+)^\star$*, l'ensemble des tuples sur des nombres naturels positifs.* $\mathbb{P}$ *équipé avec l'opération de concaténation de tuples et son élément identité, le tuple vide* $\langle\rangle$*, est un monoïde.*

Pour simplifier les notations, on omet la notation de tuples et on utilise la concaténation à la place. Ainsi le pid $\langle a_1, a_2 \dots a_n \rangle$ est noté $a_1 \cdot a_2 \cdot \dots \cdot a_n$. $\Diamond$

Comme donné en définition, on omet la notation de tuples. Le pid $\langle 1, 2, 3 \rangle$ est donc noté $1 \cdot 2 \cdot 3$. Aussi, pour différencier les pids singletons des entiers, on les préfixe de l'opérateur de concaténation, par exemple, $\langle 1 \rangle$ est noté $\cdot 1$.

Maintenant, on définit les opérations utilisées pour comparer des identifiants de processus dans un modèle. Le test d'égalité et les opérations qui suivent sont les seules pouvant être utilisées dans les gardes de transitions et sur les arcs, donc, partout dans le réseau, pour comparer de identifiants de processus.

DEFINITION 6.2. *(opérations sur identifiants de processus) Deux pids $\pi, \pi' \in \mathbb{P}$ peuvent être comparés en utilisant l'égalité et les opérations $\lhd_1$ et $\lhd$ définies par :*

- *$\pi \lhd_1 \pi'$ ssi $\exists a \in \mathbb{N}^+$ tel que $\pi \cdot a = \pi'$;*
- *$\pi \lhd \pi'$ ssi $\exists a_1, \dots, a_n \in \mathbb{N}^+$ tels que $\pi \cdot a_1 \cdot \dots \cdot a_n = \pi'$, $n \neq 1$;* $\Diamond$

6.1.2 Restrictions sur les modèles

On impose des restrictions sur les réseaux de Petri adressés dans ce chapitre, ces restrictions sont nécessaires pour réaliser correctement le mécanisme de création dynamique de processus. Ainsi on suppose que $\mathbb{P} \subseteq \mathbb{D}$, *i.e.*, les pids sont des valeurs, et on note $\mathbb{D}_v \stackrel{\text{df}}{=} \mathbb{D} \setminus \mathbb{P}$. On suppose aussi que les éléments de $\mathbb{D}_v$ sont comparables entre eux, c'est-à-dire qu'on a un ordre total sur $\mathbb{D}_v$. Ceci n'est pas une restriction car cet ordre peut être arbitraire.

SUPPOSITION 6.1. *Soit (S, T, ℓ) un réseau de Petri. On suppose que :*

- *pour tout $s \in S$, le type de s, $\ell(s)$ est un produit cartésien $X_1 \times \dots \times X_k$ ($k \geq 1$) avec $X_i = \mathbb{P}$ ou $X_i \subseteq \mathbb{D}_v$;*
- *on a une place $s_\eta \in S$ de type $\ell(s_\eta) = \mathbb{P} \times \mathbb{P}$, appelée place génératrice ;*
- *si M est un marquage et $\langle \pi_1, \pi_2 \rangle, \langle \pi'_1, \pi'_2 \rangle \in M(s_\eta)$ alors $\pi_1 \neq \pi'_1$.* $\Diamond$

Pour implémenter la création dynamique de processus le réseau possède une place génératrice s_η de type $\mathbb{P} \times \mathbb{P}$. Intuitivement, cette place stocke des paires $\langle \pi, \pi \cdot \langle i \rangle \rangle$ qui mémorisent le fait que le processus de pid π a créé $i - 1$ enfants, et que son prochain fils sera $\pi \cdot \langle i \rangle$. On utilise cette place pour créer de nouveaux processus et on n'autorise pas la réutilisation d'identifiants de processus. Lorsque un processus est créé ou termine, cette place devra être mise à jour de manière adéquate. Étant donné un marquage M, on note par pid_M l'ensemble de ses pids. De plus, on note $\mathcal{M}_\pi$ l'ensemble de marquages tels que chaque jeton possède π en première position, c'est-à-dire que chaque jeton est de la forme $\langle \pi, v_2, \dots, v_n \rangle$ avec $n \geq 1$ et $v_2 \dots v_n \in \mathbb{D}_v \cup \mathbb{P}$.

De plus nous devons supposer qu'aucune valeur de pid n'apparait dans une expression, c'est-à-dire que les pids ne peuvent être comparés que de manière littérale. Par exemple, nous pouvons avoir l'expression $\pi \lhd \pi'$ dans une garde où π et π' correspondraient à des jetons en entrée de la transition, mais nous ne pouvons pas avoir $\pi = 1 \cdot 1$. La même restriction doit s'appliquer à toute formule que nous souhaitons vérifier sur le modèle.

SUPPOSITION 6.2. *Dans un réseau de Petri, on suppose que les pids sont comparés de manière littérale et qu'aucune valeur de pid n'apparait dans une expression.*

Une relation d'équivalence sur les marquage de ce type de réseaux a été définie dans [115, 116, 114]. Cette relation identifie les processus équivalents avec une bijection sur les pids qui préserve la relation parent (et une relation frère qu'on omet ici). Deux processus dans cette relation doivent être associés aux mêmes données dans le réseau, intuitivement, deux processus équivalents doivent être dans le même état. Nous simplifions cette relation d'équivalence en introduisant la notion de marquage *pid-cohérent*. Un marquage pid-cohérent garanti la cohérence entre la place génératrice et les pids des autres places.

Definition 6.3 (marquage pid-cohérent). *Un marquage M d'un réseau de Petri est pid-cohérent ssi pour tout $\langle \pi, \pi \cdot \langle k \rangle \rangle \in M(s_\eta)$ et pour tout $\pi \in pid_M \setminus \{\pi \cdot \langle k \rangle\}$ tel que $\pi' \stackrel{\text{df}}{=} \pi \cdot \langle i, a_1, \ldots, a_n \rangle$ $(n \geq 0)$ on a $i < k$.* ◊

Cela veut dire que si on a un jeton $\langle \pi, \pi.\langle k \rangle \rangle$ dans la place génératrice alors il n'y a aucun autre jeton dans le réseau dont le préfixe est π et son suffixé est supérieur ou égal à $\langle k \rangle$ dans l'ordre lexicographique. Par exemple si on a $\langle 1 \cdot 1, 1 \cdot 1 \cdot 1 \rangle$ dans la place génératrice alors les jetons $1 \cdot 1 \cdot 1$, $1 \cdot 1 \cdot 2$, $1 \cdot 1 \cdot 1 \cdot 1$, *etc.* ne peuvent pas apparaître dans le réseau. En revanche, les pid $1 \cdot 2$, $1 \cdot 2 \cdot 1$, *etc.* peuvent apparaître dans le réseau s'ils sont cohérents avec la place génératrice, *i.e.*, le jeton $\langle \cdot 1, 1 \cdot \langle k' \rangle \rangle$ est dans la place génératrice et k' est supérieur à 2 ou bien aucun jeton de la forme $\langle \cdot 1, \pi \rangle$ n'est présent dans la place génératrice.

Definition 6.4 (équivalence de marquages). *Deux marquages pid-cohérents M et M' sont h-équivalents s'il existe une bijection $h : pid_M \to pid_{M'}$ telle que :*

- *$\forall \pi, \pi' \in pid_M : \pi \prec \pi'$ ssi $h(\pi) \prec h(\pi')$ pour $\prec$ dans $\{\lhd_1, \lhd\}$;*
- *M' est M après remplacement de chaque pid π par $h(\pi)$.*

On note cela par $M \sim_h M'$, ou simplement $M \sim M'$. ◊

Des contraintes structurelles sur les modèles ont été utilisées dans les travaux précédents [115, 116, 114, Fro12a], elles impliquent la pid-cohérence pour les marquages accessibles. On suppose donc que tous les marquages accessibles sont pid-cohérents. Il est inutile de donner cette supposition sous forme de contraintes structurelles puisqu'elles existent déjà et peuvent être réutilisées. De plus, cela permet aussi d'être plus général.

Supposition 6.3. *Tous les marquages accessibles d'un réseau de Petri sont pid-cohérents.* ◊

Une première contribution présentée dans ce chapitre est le théorème 6.1 qui capture une notion forte de similarité entre marquages. Si deux marquages sont *h-équivalents* alors leur successeurs sont aussi équivalents. Un théorème similaire a été donné dans [114] mais était centré seulement sur les marquages accessibles, ici on relâche cette hypothèse d'accessibilité en la remplacement par une hypothèse de pid-cohérence. Cela sera essentiel dans notre cas, puisqu'on va considérer des états potentiellement non accessibles mais qui sont équivalents à des états accessibles du système.

Théorème 6.1. *Soient M et M' deux marquages h-equivalents et pid-cohérents d'un réseau de Petri (S, T, ℓ), soit $t \in T$ une transition et β une valuation telles que $M[t, \beta\rangle \widetilde{M}$. Alors $M'[t, h \circ \beta\rangle \widetilde{M'}$ et $\widetilde{M} \sim \widetilde{M'}$.* □

Comme décrit dans [114], la relation $\sim_h$ se comporte comme une relation de bisimulation forte et peut être vue comme suffisante pour une réduction d'espaces d'états pour la détection de deadlocks. De plus, parce que l'équivalence préserve le tirage de transitions et les modes (modulo valeurs symétriques), c'est aussi suffisant pour travailler avec des propriétés sur les séquences d'états, et donc compatible avec la vérification de propriétés temporelles.

6.2

Pid-trees

Les *Pid-trees* ont été introduits dans [Fro12a] comme représentations pour marquages. Intuitivement, il s'agit d'une structure de données qui représente une hiérarchie de pids qui apparaissent dans un marquage. Ces pids sont associés à des données en utilisant une notion d'appartenance. La racine de l'arbre contient les données qui sont partagées entre tous les processus et les autres noeuds contiennent les données appartenant à un processus spécifique. Le chemin de la racine à un noeud correspond à l'identité du processus (pid) qui est le *propriétaire* des données du noeud.

Cette intuition reste centrale cependant toute la formalisation de pid-trees a été retravaillée depuis. Les pid-trees sont maintenant utilisés comme primitives de normalisation de marquages et non comme leur représentations dans les espaces d'états. La nouvelle formalisation possède de plus fortes implications et est dirigée par la syntaxe. Les arbres sont représentés avec des tuples et notés avec une flèche comme décrit dans la définition qui suit.

Definition 6.5 (pid-trees). *Pour chaque $\pi \in \mathbb{P}$ on défini l'ensemble Ξ_π par induction comme le plus petit ensemble tel que :*

1. *pour tout $M_\pi \in (\mathcal{M}_\pi \cup \{\bot\})$ on a :*

$$\langle M_\pi, \langle\rangle\rangle \in \Xi_\pi \quad \textit{noté} \quad M_\pi$$

2. *pour tout $a_1, \ldots, a_n \in \mathbb{P}$ et $t_1 \in \Xi_{\pi\cdot a_1}, \ldots, t_n \in \Xi_{\pi\cdot a_n}$ tels que $a_1, \ldots, a_n$ sont tous distincts, et pour $M_\pi \in (\mathcal{M}_\pi \cup \{\bot\})$ on a :*

$$\langle M_\pi, \langle\langle a_1, t_1\rangle, \ldots, \langle a_n, t_n\rangle\rangle\rangle \in \Xi_\pi \quad \textit{noté} \quad M_\pi \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n\rangle$$

En utilisant les ensembles définis ci-dessus, l'ensemble des pid-trees Ξ est $\Xi_{\langle\rangle}$. ◊

Dans la définition ci-dessus chaque M_π est un marquage ou $\bot$. Dans le cas où c'est un marquage, chaque jeton qu'il contient est de la forme $\langle \pi, v_2, \ldots, v_m\rangle$ $(m \geq 1)$ où $\pi \in \mathbb{P}$, $v_2, \ldots, v_m \in \mathbb{D}$, *i.e.*, chaque jeton est un tuple avec π en première composante, les autres composantes sont soit des pids soit des valeurs. La seule exception est la racine, puisque $\langle\rangle$ n'est pas un pid valide, $M_{\langle\rangle}$ contient des jetons de la forme $\langle v_1, \ldots, v_m\rangle$ $(m \geq 1)$ où $v_1 \in D_v$ et $v_2, \ldots, v_m \in \mathbb{D}$. $\bot$ est une valeur particulière dénotant un marquage vide qui sert de marqueur pour des pids inactifs, *i.e.*, pids qui n'apparaissent pas dans le marquage. La notation *"flèche"* permet d'écrire les pid-trees de manière moins verbeuse, par exemple le pid-tree

$$\langle \bot, \langle \cdot 1, \langle M_1, \langle\langle 1 \cdot 1, \langle M_{1\cdot1\cdot1}, \langle\langle \cdot 1, \langle \emptyset, \langle\rangle\rangle\rangle\rangle\rangle, \langle \cdot 2, \langle \bot, \langle\rangle\rangle\rangle\rangle\rangle\rangle\rangle$$

où $M_{\cdot1} \in \mathcal{M}_{\cdot1}$ et $M_{1\cdot1\cdot1} \in \mathcal{M}_{1\cdot1\cdot1}$ est noté par

$$\bot \xrightarrow{\cdot1} M_{\cdot1} \xrightarrow{1\cdot1,\ \cdot2} \langle M_{1\cdot1\cdot1} \xrightarrow{\cdot1} \emptyset, \bot\rangle$$

Les pid-trees sont aussi représentés graphiquement, un exemple du pid-tree ci-dessus est donnée en figure 6.1 où les fragments de pids sont utilisé comme annotations d'arcs et les marquages comme annotations de noeuds.

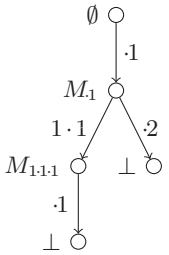


Figure 6.1 – *Pid-tree* $\emptyset \xrightarrow{\cdot1} M_{\cdot1} \xrightarrow{1\cdot1,\ \cdot2} \langle M_{1\cdot1\cdot1} \xrightarrow{\cdot1} \bot, \bot\rangle$ *où* $M_{\cdot1} \in \mathcal{M}_{\cdot1}$ *et* $M_{1\cdot1\cdot1} \in \mathcal{M}_{1\cdot1\cdot1}$

Definition 6.6 (sous-pid-trees). *Soit t un pid-tree, l'ensemble $sub(t)$ des sous-pid-trees de t est défini inductivement par :*

- *pour tout* $\langle M_\pi, \langle\rangle\rangle \in \Xi_\pi$, $sub(\langle M_\pi, \langle\rangle\rangle) = \{\langle M_\pi, \langle\rangle\rangle\}$;
- *pour tout* $t \stackrel{\text{df}}{=} M \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n\rangle \in \Xi$, $sub(t) = \{t\} \cup \bigcup_{0<i<n} sub(t_i)$.

On note $\Xi_{\mathbb{P}} \stackrel{\text{df}}{=} \cup \bigcup_{\pi\in\mathbb{P}} \Xi_\pi$ *l'ensemble de tous les sous-pid-trees.* ◊

Par définition 6.5 on peut voir que chaque sous-arbre peut être identifié par un pid obtenu par la concaténation de tous les fragments de pids sur le chemin de la racine au sous-arbre. Par exemple, le sous-arbre $t \stackrel{\text{df}}{=} M_{1\cdot1\cdot1} \xrightarrow{\cdot1} \emptyset$ dans la figure 6.1 est identifié par le pid $1 \cdot 1 \cdot 1$ qui est la concaténation de $\cdot1$ et $1 \cdot 1$. De plus, on doit avoir $t \in \Xi_{1\cdot1\cdot1}$ et donc tous les jetons dans $M_{1\cdot1\cdot1}$ sont des tuples dont la première composante est $1 \cdot 1 \cdot 1$. Parce qu'on a cette propriété par construction, on va souvent l'exploiter pour désigner un noeud par son pid et vice versa.

On fournit aussi une fonction pour construire un marquage à partir d'un pid-tree, donc chaque pid-tree correspond à un marquage, mais comme on le verra plus tard un marquage peut correspondre à un nombre infini de pid-trees.

Definition 6.7. *Soit $t \in \Xi_\pi$ un sous-pid-tree et $M_\pi \in \mathcal{M}_\pi$ un marquage. La fonction mrk est définie inductivement par :*

$$\begin{aligned}
mrk(\langle \bot, \langle\rangle\rangle) &\stackrel{\text{df}}{=} \emptyset \\
mrk(\langle M_\pi, \langle\rangle\rangle) &\stackrel{\text{df}}{=} M_\pi \\
mrk(\bot \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n\rangle) &\stackrel{\text{df}}{=} mrk(t_1) \cup \cdots \cup mrk(t_n) \\
mrk(M_\pi \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n\rangle) &\stackrel{\text{df}}{=} M_\pi \cup mrk(t_1) \cup \cdots \cup mrk(t_n)
\end{aligned}$$

◊

De même que pour les marquages, si t est un pid-tree, alors on note par pid_t l'ensemble des pids dans l'arbre, *i.e.*, l'ensemble de tous les chemins de la racine à chaque noeud dans l'arbre. En plus, cela veut dire que $pid_{mrk(t)} \subseteq pid_t$ parce que l'arbre peut contenir des noeuds vides ($\emptyset$ ou $\perp$) et donc des pids en plus. Maintenant, grâce à toutes les définitions précédentes on peut définir les pid-trees qui nous intéressent : *les pid-trees bien formés.*

DEFINITION 6.8 (pid-trees clos et bien formés). *Un sous-pid-tree* $t_0 = M_\pi \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n \rangle \in \Xi_\pi$ *est* clos *par rapport à un pid-tree* $t \in \Xi$ *(ou t-clos) ssi :*

1. *Si* $n = 0$ *alors* $M_\pi = \perp \Leftrightarrow \langle \pi', \pi \rangle \in mrk(t)(s_\eta)$.
2. *Quel que soit* $s \in S \setminus \{s_\eta\}$ *pour tout jeton* $\langle v_1, \ldots, v_n \rangle \in M_\pi(s)$, *on a* $v_i \in \mathbb{P}$ *si et seulement s'il y a un sous-pid-tree* $M_{v_i} \xrightarrow{b_1,\ldots,b_m} \langle t'_1, \ldots, t'_m \rangle \in \Xi_{v_i}$ *dans* t *tel que* $M_{v_i} \neq \perp$.
3. $t_1, \ldots, t_n$ *sont t-clos.*

On dit qu'un pid-tree $t \in \Xi$ *est* bien formé *ssi il est t-clos (clos par lui même).* ◊

Dans la définition 6.8 on exprime les conditions qui doivent être satisfaites pour chaque sous-arbre. Cependant, chacune de ces conditions a besoin d'être considérée dans le contexte d'un arbre complet. C'est pour quoi on utilise le terme de sous-pid-tree t-clos où t est un pid-tree. Le premier point, assure que les feuilles de l'arbre ne peuvent être annotés par $\perp$ que si ce sont des pids à créer dans le futur (*i.e.*, ils sont référencés par la place génératrice). Le second point, garantit que si un pid π apparaît dans le marquage d'un noeud, alors il y a un noeud identifié par π dans l'arbre et le marquage correspondant n'est pas $\perp$, et réciproquement si un noeud a un marquage qui n'est pas $\perp$ alors il correspond à un pid dans le marquage. Cela veut dire que si un pid apparaît dans $mrk(t)$ alors il y a un sous-arbre t_0 de t dans Ξ_π tel que $t_0 \stackrel{\text{df}}{=} M_\pi \xrightarrow{b_1,\ldots,b_m} \langle t'_1, \ldots, t'_m \rangle$ ($m \geq 0$) tel que $M_\pi \neq \perp$ mais aussi que si un noeud interne est $\perp$ alors il n'est pas référencé dans le marquage. Finalement, le dernier point, garantit que c'est aussi vrai pour les sous-arbres.

Si on se base sur cette définition alors le plus petit pid-tree bien formé est $\emptyset \stackrel{\text{df}}{=} \langle \emptyset, \langle\rangle \rangle$. Il faut noter que $\perp \stackrel{\text{df}}{=} \langle \perp, \langle\rangle \rangle$ n'est pas bien formé parce qu'une feuille ne peut être $\perp$ que si elle dénote un pid à créer. Cependant, du point de vue du marquage les noeuds $\perp$ et $\emptyset$ représentent la même information mais ont des rôles complémentaires dans les pid-trees : les noeuds $\emptyset$ correspondent aux pids qui apparaissent dans le marquage mais ne possèdent pas de jetons, tandis que $\perp$ correspond aux pids qui sont référencés dans la place génératrice et dénotent les pids à être crées mais aussi les pids qui n'apparaissent pas dans le marquage mais apparaissent dans l'arbre. Par exemple, les pid-tree en figure 6.2 représentent tous le même marquage, mais seulement les pid-trees 6.2a et 6.2b sont bien formés. En effet, dans le pid-tree en figure 6.2c, le marquage au pid $1{\cdot}1$ ne peut pas être vide car ce pid n'est pas référencé, il doit forcément être $\perp$. Le pid-tree en figure 6.2d quand a lui ne possède pas de feuille $\perp$ correspondant au jeton $\langle 1, 1 \cdot 2 \rangle$ de la place génératrice.

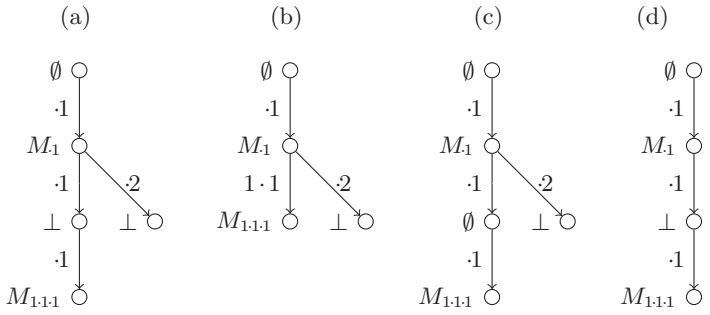


Figure 6.2 – *Exemple de pid-trees dont le marquage est* $\{s_\eta \mapsto \{\langle \cdot 1, 1 \cdot 2 \rangle\}, s_1 \mapsto \{\langle 1 \cdot 1 \cdot 1, d \rangle, \langle \cdot 1, d' \rangle\}\}$ *(pour* $d \in \mathbb{D}_v$*).*

6.2.1 Normalisation de pid-trees

Une amélioration notable dans cette nouvelle formalisation est l'introduction de la normalisation de pid-trees qui permet de remplacer l'équivalence de pid-trees définie dans [Fro12a] par un simple test d'égalité. Pour cela, on défini d'abord une bijection sur les pids qui préserve les relations parent et ancêtre, puis on l'applique récursivement sur l'arbre en transformant les pids sur les arcs et ceux des marquages annotant les noeuds.

Definition 6.9. *Soit* $t \stackrel{\text{df}}{=} M \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n \rangle \in \Xi$ *un pid-tree. Sa bijection de normalisation* $h^t_N : \mathbb{P} \to \mathbb{P}$ *est définie par :*

$$h^t_N \stackrel{\text{df}}{=} \left\{ \begin{array}{l} a_1 \mapsto \langle 1 \rangle, \\ \ldots, \\ a_n \mapsto \langle n \rangle \end{array} \right\} \cup h^{t_1}_{N,\langle 1 \rangle} \cup \ldots \cup h^{t_n}_{N,\langle n \rangle}$$

avec pour chaque $t_i = M' \xrightarrow{b_1,\ldots,b_n} \langle t'_1, \ldots, t'_m \rangle \in \Xi_\pi$

$$h^{t_i}_{N,\pi'} \stackrel{\text{df}}{=} \left\{ \begin{array}{l} \pi \cdot b_1 \mapsto \pi' \cdot b'_1, \\ \ldots, \\ \pi \cdot b_m \mapsto \pi' \cdot b'_m \end{array} \right\} \cup h^{t'_1}_{N,\pi' \cdot b'_1} \cup \ldots \cup h^{t'_m}_{N,\pi' \cdot b'_m}$$

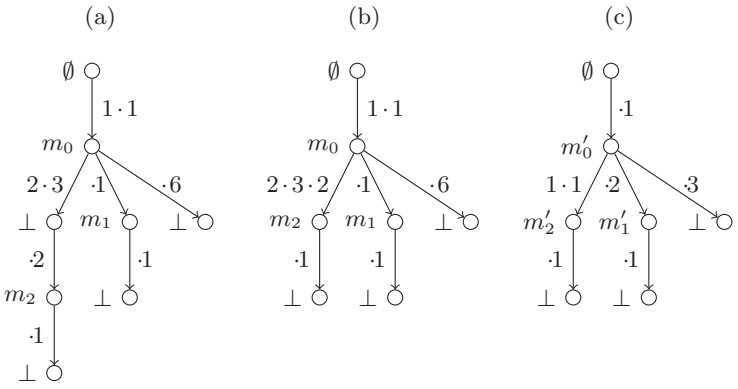
avec $b'_i \stackrel{\text{df}}{=} \left\{ \begin{array}{ll} \langle i \rangle & \text{si } b_i = \langle k \rangle \quad (k \in \mathbb{N}) \\ \langle i, 1 \rangle & \text{sinon.} \end{array} \right.$ ◊

Definition 6.10. *Soit* $t \in \Xi$ *un pid-tree. La forme normalisée de* t *est* $\mathcal{N}(t) \stackrel{\text{df}}{=} h^t_N(t)$*, où* $h^t_N(t)$ *est l'application de la bijection de normalisation* h^t_N *à chaque marquage et fragment de pid dans le pid-tree.* ◊

En pratique, l'application de la bijection correspond à un renommage des pids apparaissant dans l'arbre. Chaque a_i associé avec un fils est remplacé par i si le pid a une longueur de 1, ce qui est suffisant pour préserver la relation

parent, et par $\langle i, 1\rangle$ si la taille du pid est supérieure à 2, ce qui est suffisant pour préserver la relation ancêtre sans perturber la relation parent. Donc, chaque sous-pid-tree $M \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n\rangle$ est transformé en $M \xrightarrow{b_1,\ldots,b_n} \langle t'_1, \ldots, t'_n\rangle$ où chaque b_i est $\langle i\rangle$ si $a_i = \langle k\rangle$ $(k \in \mathbb{N})$ et $\langle i, 1\rangle$ sinon.

Lorsqu'on manipule la racine on peut remplacer chaque a_i par $\langle i\rangle$ parce qu'il n'y a pas de noeuds au dessus et donc aucun ancêtre valide. Un exemple de normalisation est donné en figure 6.3, le pid-tree 6.3c est la forme normalisée du pid-tree 6.3b.



$$m_0 \stackrel{\text{df}}{=} \{\ s_\eta \mapsto \langle 1\cdot1,\ 1\cdot1\cdot6\rangle\ \}$$

$$m_1 \stackrel{\text{df}}{=} \left\{ \begin{array}{lcl} s_1 & \mapsto & \langle 1\cdot1\cdot1,\ 1\cdot1\rangle, \\ s_\eta & \mapsto & \langle\cdot1\cdot1\cdot1,\ 1\cdot1\cdot1\cdot1\rangle \end{array} \right\}$$

$$m_2 \stackrel{\text{df}}{=} \left\{ \begin{array}{lcl} s_1 & \mapsto & \langle 1\cdot1\cdot2\cdot3\cdot2,\ d\rangle, \\ s_\eta & \mapsto & \langle 1\cdot1\cdot2\cdot3\cdot2,\ 1\cdot1\cdot2\cdot3\cdot2\cdot1\rangle \end{array} \right\}$$

$$m'_0 \stackrel{\text{df}}{=} \{\ s_\eta \mapsto \langle\cdot1,\ 1\cdot3\rangle\ \}$$

$$m'_1 \stackrel{\text{df}}{=} \left\{ \begin{array}{lcl} s_1 & \mapsto & \langle 1\cdot2,\ \cdot1\rangle, \\ s_\eta & \mapsto & \langle 1\cdot2,\ 1\cdot2\cdot1\rangle \end{array} \right\}$$

$$m'_2 \stackrel{\text{df}}{=} \left\{ \begin{array}{lcl} s_1 & \mapsto & \langle 1\cdot1\cdot1,\ d\rangle, \\ s_\eta & \mapsto & \langle 1\cdot1\cdot1,\ 1\cdot1\cdot1\cdot1\rangle \end{array} \right\}$$

FIGURE 6.3 – *Deux représentations en pid-tree 6.3a et 6.3b, d'un marquage* $M \stackrel{\text{df}}{=} m_0 \cup m_1 \cup m_2$ *avec* $d \in \mathbb{D}$*, et un pid-tree normalisé 6.3c à partir du pid-tree 6.3b. Le pid-tree 6.3c est une représentation du marquage* $M' \stackrel{\text{df}}{=} m'_0 \cup m'_1 \cup m'_2$*.*

6.3

Pid-trees et symétries

Dans cette section, on définit les représentations de marquages sous forme de pid-trees qui sont essentielles dans notre boite à outils théorique afin de détecter les symétries. D'abord on caractérise ces représentations, puis on propose deux étapes de raffinement.

DEFINITION 6.11. *Soient* (S, T, ℓ) *un réseau de Petri et* M *de ses marquages pid-cohérents.* $\mathcal{R}(M)$ *est l'ensemble des pid-trees tel que* $t \in \mathcal{R}(M)$ *ssi :*

1. *t est bien formé ;*
2. $mrk(t) = M$ *;*
3. *pour tout* $\langle \pi, \pi\cdot\langle k\rangle\rangle \in M(s_\eta)$ *il y a un sous-pid-tree* t_π *de* t *tel que*

$$t_\pi \stackrel{\text{df}}{=} M_\pi \xrightarrow{a_1,\ldots,a_n,\langle k\rangle} \langle a_1, \ldots, a_n, \bot\rangle$$

i.e., *les pids à créer sont les fils les plus à droite.* ◊

La bijection de normalisation d'un pid-tree appliquée à un pid-tree qui est la représentation d'un marquage a la bonne propriété de produire un pid-tree équivalent, et donc est adéquate pour les tests d'équivalence de marquages. Chaque marquage accessible peut être transformé en pid-tree puis normalisé, ce qui donne un marquage équivalent. Il faut cependant remarquer qu'un marquage normalisé peut être non accessible mais restera pid-cohérent ce qui est suffisant si on considère le théorème 6.1.

Proposition 6.2. *Soit M un marquage pid-cohérent d'un réseau de Petri. Si $t \in \mathcal{R}(M)$ est une représentation de M, alors*

1. *$mrk(\mathcal{N}(t))$ est pid-cohérent ;*
2. *M et $mrk(\mathcal{N}(t))$ sont h_N^t équivalents,* i.e., *$M \sim mrk(\mathcal{N}(t))$.* □

6.3.1 Pid-trees réduits.

Les pid-trees réduits (*stripped pid-trees*) sont des arbres où tous les noeuds correspondants au pids inactifs ont été supprimés, c'est-à-dire tous les noeuds internes étiquetés par $\perp$ dans le cas de pid-trees bien formés. Ces arbres sont caractérisés par la définition suivante.

Definition 6.12. *Soit M un marquage pid-cohérent d'un réseau de Petri et $\mathcal{R}(M)$ l'ensemble de ses représentations sous forme de pid-tree. Alors l'ensemble des représentations réduites de M est :*

$$\mathcal{R}_s(M) = \{\ t \in \mathcal{R}(M) \mid pid_t = pid_M\ \}$$

◊

L'avantage principal des pid-trees réduits par rapport aux pid-trees est que l'ensemble de ces représentations est fini et peut être énuméré à partir de chacun de ses éléments. Cela veut dire que si on fournit une fonction de réduction arbitraire pour un marquage alors l'ensemble de ses représentations réduites peut être énuméré.

Definition 6.13. *Soit t et t' deux sous-pid-trees. Alors t' est une permutation de t ssi :*

$$t = M \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n \rangle \quad et \quad t' = M \xrightarrow{a'_1,\ldots a'_n} \langle t'_1, \ldots, t'_n \rangle$$

avec $\langle a'_1, \ldots, a'_n \rangle$ une permutation de $\langle a_1, \ldots, a_n \rangle$ telle que pour chaque $i, j \in \{1, \ldots, n\}$ si $a_i = a'_j$ alors t'_j est une permutation de t_i. ◊

Proposition 6.3. *Soit M un marquage pid-cohérent d'un réseau de Petri. Si $t, t' \in \mathcal{R}_s(M)$ alors t' est une permutation de t.* □

Corollaire 6.4. *Soit M un marquage pid-cohérent d'un réseau de Petri. Alors $\mathcal{R}_s(M)$ est fini et peut être énuméré à partir de chacun de ses éléments.* □

Chacune des représentations réduites d'un marquage est une permutation d'une représentation réduite unique. Cet ensemble étant fini et énumérable à partir d'un de ses éléments, on peut donc énumérer toutes représentations réduites. Cependant, on ne peut pas utiliser des permutations arbitraires pour cela, essentiellement parce que nous devons conserver la pid-cohérence. Heureusement, notre définition de représentation sous forme de pid-tree oblige les futurs pids à être les fils les plus à droite et être étiquetés par $\perp$. Donc, il suffit que les permutations ne modifient pas la position d'un fils le plus à droite d'un sous arbre si celui-ci est étiqueté par $\perp$. Puisqu'on applique la normalisation sur nos arbres, les fils les plus à droite vont avoir les pids les plus grands et donc on garantit que nous n'allons pas recréer de pids qui existent déjà.

Maintenant, on peut voir que cet ensemble fini de représentations peut être utilisé pour détecter des symétries grâce au théorème 6.5

THÉORÈME 6.5. *Soit M et M' deux marquages pid-cohérents d'un réseau de Petri. Alors :*

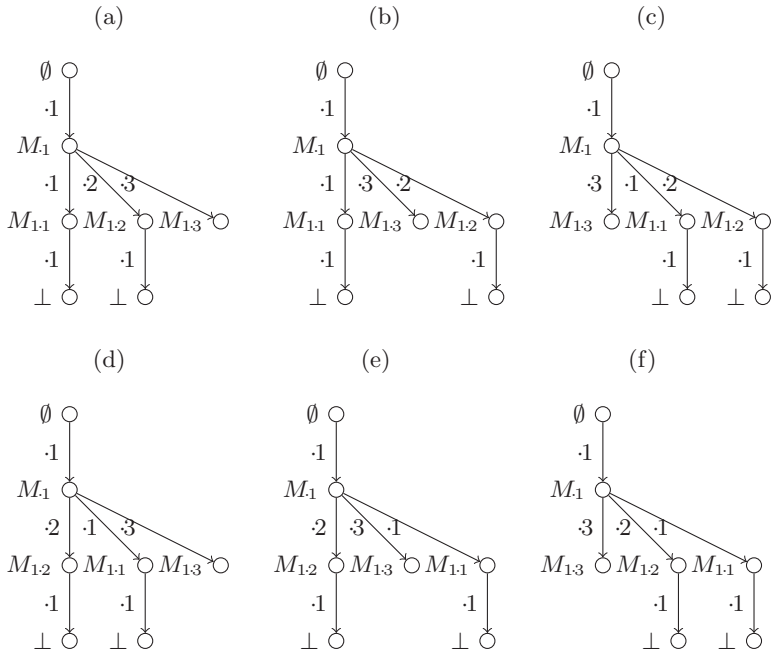
$$M \sim M' \Leftrightarrow \forall t \in \mathcal{R}_s(M), \exists t' \in \mathcal{R}_s(M'), \mathcal{N}(t) = \mathcal{N}(t') \qquad \square$$

Il est important de remarquer que maintenant on a une méthode complète pour détecter les symétries qui est calculable. Pour chaque nouvel état on construit une représentation réduite de celui-ci, puis on énumère toutes ses permutations. Pour chaque permutation de ce pid-tree, on construit un marquage normalisé et on teste si ce marquage est dans l'espace d'états. Si tous les tests échouent le marquage correspond à un nouvel état et on l'ajoute à l'espace d'états. À ce stade, on voit qu'on peut réaliser un grand nombre de permutations. Par exemple, en figure 6.4 on a les 6 permutations d'un pid-tree réduit, or on remarque que certains arbres sont clairement non isomorphes entre eux et donc la normalisation produira forcément des arbres différents. L'étape suivante de notre approche est de raffiner le résultat obtenu en utilisant une heuristique pour réduire le nombre de permutations en ne considérant que des arbres potentiellement isomorphes. Intuitivement, le pid-tree 6.4a est potentiellement isomorphe au pid-tree 6.4d, le pid-tree 6.4b au pid-tree 6.4e et le pid-tree 6.4c au pid-tree 6.4f.

6.3.2 Réduire le nombre de permutations

Réaliser toutes les permutations de pid-trees réduits n'est pas toujours nécessaire. Certaines permutations produisent des pid-trees qui vont clairement produire des marquages qui ne sont pas équivalents. Intuitivement, deux pid-trees produisent des marquages équivalents seulement si les arbres ont la même structure, et contiennent des marquages avec les mêmes données. Ils ne peuvent différer seulement par les pids mais par rien d'autre. Donc si on a deux pid-trees qui produisent des marquages équivalents alors l'application de la même permutation sur les deux produira des pid-trees dont les marquages sont équivalents. L'ordre des enfants n'est pas très important pour la correction puisqu'on peut toujours les permuter à l'exception des pids à créer. L'idée de l'approche présentée dans cette sous-section est de forcer les arbres à respecter un certain ordre de sous arbres afin de réduire le nombre de permutations.

Pour réduire le nombre de permutations on introduit un ordre appelé *pid-free* sur les pid-trees qui oblige à utiliser des arbres ayant la même structure.



$$
\begin{array}{lcl}
M_1 & \stackrel{\text{df}}{=} & \{\ s_1 \ \mapsto\ \langle \cdot 1,\ 1\cdot 3\rangle\ \} \\
M_{1\cdot 1} & \stackrel{\text{df}}{=} & \{\ s_\eta \ \mapsto\ \langle 1\cdot 1,\ 1\cdot 1\cdot 1\rangle\ \} \\
M_{1\cdot 2} & \stackrel{\text{df}}{=} & \{\ s_\eta \ \mapsto\ \langle 1\cdot 2,\ 1\cdot 2\cdot 1\rangle\ \} \\
M_{1\cdot 3} & \stackrel{\text{df}}{=} & \emptyset
\end{array}
$$

Figure 6.4 – *Les permutations d'un pid-tree réduit.*

Definition 6.14. *Soit M un marquage d'un réseau de Petri. La fonction pidfree est définie par :*

$$pidfree(M) \stackrel{\text{df}}{=} \{(s, pidfree(v)) \mid (s, v) \in M\}$$

$$pidfree(\langle x_1, \ldots, x_n\rangle) \stackrel{\text{df}}{=} \langle v_1, \ldots, v_n\rangle$$

où $v_i \stackrel{\text{df}}{=} x_i$ si $x_i \in \mathbb{D}$ et $v_i \stackrel{\text{df}}{=} \langle\rangle$ sinon. ◊

Intuitivement, la fonction *pidfree* efface les pids des marquages leur permettant d'être totalement comparables (à cette étape on considère tous pids égaux entre eux). Maintenant, on étend la définition aux pid-trees. Dans les définitions qui suivent les sous-pid-trees t et t' sont définis pour $n, m \in \mathbb{N}$ par :

$$t \stackrel{\text{df}}{=} M_\pi \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n\rangle \quad \text{et} \quad t' \stackrel{\text{df}}{=} M_{\pi'} \xrightarrow{a'_1,\ldots,a'_n} \langle t'_1, \ldots, t_m\rangle$$

Definition 6.15. *Soit $t, t' \in \Xi_\pi$ deux sous-pid-trees. Les pid-trees t et t' sont* pid-free *équivalents, noté $t \equiv t'$, ssi*

$$pidfree(M_\pi) = pidfree(M_{\pi'}),\ n = m \text{ et } t_i \equiv t'_i \text{ pour } 1 \le i \le n$$

◊

DEFINITION 6.16. *Soit $t, t' \in \Xi_\pi$ deux sous-pid-trees. On a $t \preccurlyeq t'$ ssi*
- *$pidfree(M_\pi) < pidfree(M_{\pi'})$ ou*
- *$pidfree(M_\pi) = pidfree(M_{\pi'})$ et*
  - *$n < m$ ou*
  - *$n = m$ et $\exists j \in \mathbb{N}, j \leq n, \forall i \in \mathbb{N},\ i < j,\ t_i \equiv t'_i \wedge t_j \preccurlyeq t'_j$.* $\Diamond$

Dans la définition ci-dessus, l'ordre $<$ est un ordre total sur les marquages de notre réseau de Petri. On définit d'abord un ordre total sur les places, S étant fini cette étape est triviale. Ensuite on utilise un ordre total sur les multiensembles pour comparer les places, on peut le faire parce qu'on a supposé que les éléments de $\mathbb{D}_v$ sont totalement ordonnés.

Le dernier point de la définition 6.16 est une définition récursive usuelle d'un ordre, si les marquages sont égaux alors soit $n < m$ (t a moins de fils que t') ou $n = m$ et il y a une paire $(t_j, t'_j) \in \{(t_1, t'_1), \ldots, (t_n, t'_n)\}$ telle que pour toute les paires $(t_i, t'_i) \in \{(t_1, t'_1), \ldots, (t_{j-1}, t'_{j-1})\}$ on a t_i équivalent à t'_i et $t_j \preccurlyeq t'_j$.

DEFINITION 6.17. *Un sous-pid-tree $t \in \Xi_\pi$ est pid-free ordonné ssi ses sous-arbres sont pid-free ordonnés,* i.e.*, pour $i, j \in \{1, \ldots, n\}$, $t_i \preccurlyeq t_j$ si $i < j$.* $\Diamond$

Puisqu'on sait comparer les pid-trees, on peut définir un ensemble de pid-trees réduits et ordonnés. Ces arbres sont des pid-trees réduits et ordonnés en utilisant l'ordre pid-free. De plus, on peut caractériser l'ensemble de ces représentations en utilisant la relation $\equiv$ sur les pid-trees réduits, puis les énumérer d'une manière similaire à précédemment.

DEFINITION 6.18. *Soit M un marquage pid-cohérent d'un réseau de Petri. L'ensemble de pid-tree réduits et pid-free ordonnés $\mathcal{R}_{so}(M)$ est défini par :*

$$\mathcal{R}_{so}(M) = \{\ t \in \mathcal{R}_s(M) \mid t \text{ est pid-free ordonné } \}$$ $\Diamond$

PROPOSITION 6.6. *Soit $t \in \mathcal{R}_{so}(M)$ un pid-tree, alors pour tout $t' \in \mathcal{R}_s(M)$ on a :*

$$t \equiv t' \Leftrightarrow t' \in \mathcal{R}_{so}(M)$$ $\square$

COROLLAIRE 6.7. *Soit M un marquage pid-cohérent d'un réseau de Petri. Alors l'ensemble $\mathcal{R}_{so}(M)$ est fini et peut être énuméré à partir de chacun de ses éléments.* $\square$

Comme décrit dans le corollaire 6.7, le sous-ensemble ordonné des pid-trees réduits peut être énuméré à partir d'un élément quelconque de ce même ensemble. Pour réaliser cette énumération, il faut considérer les permutations de pid-trees mais ne permuter que les sous-arbres équivalents (par rapport à la relation $\equiv$). Cependant, dans le pire des cas on devra réaliser toutes les permutations. Finalement le Théorème 6.8 présente que cet sous-ensemble pour être utilisé pour détecter les symétries.

THÉORÈME 6.8. *Soit M et M' deux marquages pid-cohérents d'un réseau de Petri. Alors :*

$$M \sim M' \Leftrightarrow \forall t \in \mathcal{R}_{so}(M), \exists t' \in \mathcal{R}_{so}(M'), \mathcal{N}(t) = \mathcal{N}(t')$$

$\square$

Maintenant, si on reprend les pid-trees de la figure 6.4, et si on suppose $s_\eta < s_1$ (ordre arbitraire) alors les seules permutations de pid-trees autorisés sont les pid-trees 6.4a et 6.4b. En effet, les autres pid-trees ne respectent pas notre ordre. Dans cet exemple nous avons éliminé 4 des 6 permutations possibles, mais bien sûr un pire cas est toujours possible.

6.4

Algorithme de construction de pid-trees

6.4.1 Définitions

Dans cette section on fournit une construction complète de représentation de marquages en termes de pid-trees réduits. Pour cela on suppose qu'on est dans le contexte d'un réseau de Petri (S, T, ℓ). Premièrement, on définit deux fonctions d'*extension* qui étendent les branches dans un pid-tree, puis on définit une fonction de *mise à jour* qui modifie les marquages dans les noeuds.

Definition 6.19. *La fonction d'*extension gauche $\mathcal{E}_l : \Xi_{\mathbb{P}} \times \mathbb{P} \to \Xi_{\mathbb{P}}$ *et la fonction d'*extension droite $\mathcal{E}_r : \Xi_{\mathbb{P}} \times \mathbb{P} \to \Xi_{\mathbb{P}}$ *sont utilisées pour ajouter des branches dans les pid-trees, et sont définies par*

$$\mathcal{E}_l(M \xrightarrow{a_1,\ldots,a_n} \langle t_1,\ldots,t_n\rangle, \pi) =$$
$$\begin{cases} M' \xrightarrow{a_1,\ldots,a_n} \langle t_1,\ldots,t_n\rangle \text{ où } M' = \begin{cases} M & \text{si } M \neq \bot \\ \emptyset & \text{sinon} \end{cases} & \text{si } \pi = \langle\rangle \\ M \xrightarrow{a_1,\ldots,a_n} \langle t_1,\ldots,t_{i-1},\mathcal{E}_l(t_i,\pi'),t_{i+1},\ldots,t_n\rangle & \text{si } \pi = a_i\cdot\pi' \\ M \xrightarrow{\langle k\rangle,a_1,\ldots,a_n} \langle \mathcal{E}_l(\bot,\pi'),t_1,\ldots,t_n\rangle \text{ avec } \pi = \langle k\rangle\cdot\pi' & \text{sinon} \end{cases}$$

$$\mathcal{E}_r(M \xrightarrow{a_1,\ldots,a_n} \langle t_1,\ldots,t_n\rangle, \pi) =$$
$$\begin{cases} M \xrightarrow{a_1,\ldots,a_n} \langle t_1,\ldots,t_n\rangle & \text{si } \pi = \langle\rangle \\ M \xrightarrow{a_1,\ldots,a_n} \langle t_1,\ldots,t_{i-1},\mathcal{E}_r(t_i,\pi'),t_{i+1},\ldots,t_n\rangle & \text{si } \pi = a_i\cdot\pi' \\ M \xrightarrow{a_1,\ldots,a_n,\langle k\rangle} \langle t_1,\ldots,t_n,\bot\rangle & \text{si } \pi = \langle k\rangle \notin \{a_1,\ldots,a_n\} \\ M \xrightarrow{\langle k\rangle,a_1,\ldots,a_n} \langle \mathcal{E}_r(\bot,\pi'),t_1,\ldots,t_n\rangle \text{ avec } \pi = \langle k\rangle\cdot\pi' & \text{sinon} \end{cases}$$

◊

On utilise deux fonctions d'extension parce que la place génératrice doit être manipulée de manière particulière, les futurs pids doivent être les fils les plus à droite d'un noeud d'où le nom d'extension droite. L'extension gauche ajoute toujours un fils à gauche d'un noeud, tandis que l'extension droite ajout les noeuds internes à gauche et les feuilles à droite (avec le marqueur $\bot$).

Definition 6.20. *La fonction de* mise à jour $\mathcal{U} : \Xi_{\mathbb{P}} \times S \times \ell(S) \times \mathbb{P} \to \Xi_{\mathbb{P}}$ *modifie les marquages dans les noeuds, et est définie par :*

$$\mathcal{U}(M \xrightarrow{a_1,\ldots,a_n} \langle t_1,\ldots,t_n\rangle, s, v, \pi) =$$
$$\begin{cases} M' \xrightarrow{a_1,\ldots,a_n} \langle t_1,\ldots,t_n\rangle & \text{si } \pi = \langle\rangle \\ M \xrightarrow{a_1,\ldots,a_n} \langle t_1,\ldots,t_{i-1},\mathcal{U}(t_i,s,v,\pi'),t_{i+1},\ldots,t_n\rangle & \text{si } \pi = a_i\cdot\pi' \end{cases}$$

$$\text{où } \forall s' \in \mathbb{P}, M'(s') = \begin{cases} M(s)\cup\{v\} & \text{si } s = s' \\ M(s') & \text{sinon} \end{cases}$$ ◊

Maintenant, avec la fonction de mise à jour on peut modifier les marquages dans les noeuds en ajoutant des jetons, mais le noeud doit exister. Ces trois fonctions définies, on construit la représentation du marquage en utilisant l'algorithme 6.1.

Entrée : marquage M
Sortie : pid-tree t correspondant
$t \leftarrow \emptyset$;
`//pour toutes les places sauf la place génératrice`
pour $s \in S \setminus \{s_\eta\}$ **:**
  pour $\langle v_0, \ldots, v_n \rangle \in M(s)$ **:**
    pour $v_i \in v_0, \ldots, v_n$ **:**
      si $v_i \in \mathbb{P}$ **alors**
        $t \leftarrow \mathcal{E}_l(t, v_i)$; `//extension gauche`
      fin si
    fin pour
  fin pour
fin pour
`//pour la place génératrice`
pour $\langle v_0, v_1 \rangle \in M(s_\eta)$ **:**
  $t \leftarrow \mathcal{E}_l(t, v_0)$; `//extension gauche`
  $t \leftarrow \mathcal{E}_r(t, v_1)$; `//extension droite`
fin pour
`//insérer les jetons`
pour $s \in S \setminus \{s_\eta\}$ **:**
  pour $\langle v_0, \ldots, v_n \rangle \in M(s)$ **:**
    si $v_0 \in \mathbb{P}$ **alors**
      $t \leftarrow \mathcal{U}(t, s, \langle v_0, \ldots, v_n \rangle, v_0)$; `//mettre à jour le noeud` v_0
    sinon
      $t \leftarrow \mathcal{U}(t, s, \langle v_0, \ldots, v_n \rangle, \langle\rangle)$; `//ajouter à la racine`
    fin si
  fin pour
fin pour
renvoyer t ;

Algorithme 6.1: Algorithme de construction de la représentation en pid-tree.

Proposition 6.9. *L'algorithme 6.1 construit une représentation sous forme de pid-tree d'un marquage pid-cohérent.* □

Une représentation d'un marquage en pid-tree n'étant pas suffisante pour détecter les symétries, on a besoin de réduire cette représentation.

Definition 6.21. *La fonction de* réduction $\mathcal{S} : \Xi_{\mathbb{P}} \to \Xi_{\mathbb{P}}$ *est définie par*

$$\mathcal{S}(M \xrightarrow{a_1,\ldots,a_i,\ldots,a_n} \langle t_1,\ldots,t_{i-1},t_i,t_{i+1},\ldots,t_n\rangle) =$$
$$\begin{cases} \mathcal{S}(M \xrightarrow{a_1,\ldots,a_{i-1},a_i\cdot b_1,\ldots,a_i\cdot b_m,a_{i+1},\ldots,a_n} \langle t_1,\ldots,t_{i-1},t'_1,\ldots,t'_m,t_{i+1},\ldots,t_n\rangle)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad si\ t_i = (\perp \xrightarrow{b_1,\ldots,b_m} \langle t'_1,\ldots,t'_m\rangle) \\ M \xrightarrow{a_1,\ldots,a_n} \langle \mathcal{S}(t_1),\ldots,\mathcal{S}(t_n)\rangle \qquad\qquad\qquad\qquad\qquad\qquad sinon \end{cases}$$
$\Diamond$

La fonction de réduction ci-dessus peut être vue comme une réécriture de graphe réalisant la transformation présentée en figure 6.5. Elle supprime tous les noeuds internes étiquetés par $\perp$, donc les seuls noeuds restants étiquetés par $\perp$ sont les feuilles qui correspondent à des futurs pids. Il faut remarquer qu'on force la racine à être $\emptyset$ ce qui est nécessaire pour satisfaire la définition 6.8.

Proposition 6.10. *La fonction de réduction $\mathcal{S}$ transforme une représentation d'un marquage en une représentation réduite du même marquage.* □

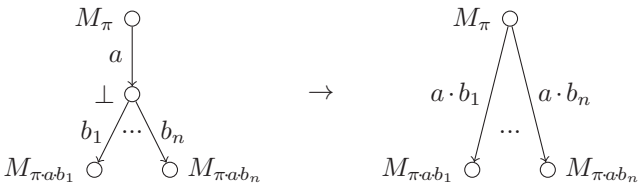


Figure 6.5 – *Règle de réécriture de réduction.*

La dernière étape est de normaliser l'arbre. On construit la bijection de normalisation récursivement, puis on utilise cette fonction pour reétiqueter les arcs et modifier les marquages.

Definition 6.22. *La bijection de normalisation est construite avec la fonction* $\mathcal{N}_{map} : \Xi_{\mathbb{P}} \times \mathbb{P} \times \mathbb{P} \to (\mathbb{P} \to \mathbb{P})$ *qui est définie par*

$$\mathcal{N}_{map}(M \xrightarrow{a_1,\ldots,a_n} \langle t_1,\ldots,t_n\rangle, \pi, \pi') \stackrel{\text{df}}{=} \{\pi \mapsto \pi'\} \cup \bigcup_{1 \le i \le n} \mathcal{N}_{map}(t_i, \pi\cdot a_i, \pi'\cdot b_i)$$

où $b_i = \langle i\rangle$ *si* $a_i = \langle k\rangle$ *(*$k \in \mathbb{N}$*) et* $b_i \stackrel{\text{df}}{=} \langle i, 1\rangle$ *sinon. La fonction de normalisation* $\mathcal{N} : \Xi_{\mathbb{P}} \times (\mathbb{P} \to \mathbb{P}) \to \Xi_{\mathbb{P}}$ *est définie par*

$$\mathcal{N}(M \xrightarrow{a_1,\ldots,a_n} \langle t_1,\ldots,t_n\rangle, f) \stackrel{\text{df}}{=} f(M) \xrightarrow{b_1,\ldots,b_n} \langle \mathcal{N}(t_1,f),\ldots,\mathcal{N}(t_n,f)\rangle$$

où $b_i = \langle i\rangle$ *si* $a_i = \langle k\rangle$ *(*$k \in \mathbb{N}$*) et* $b_i \stackrel{\text{df}}{=} \langle i, 1\rangle$ *sinon, et* $f(M)$ *représente l'application de* f *sur chaque pid dans* M*.* $\Diamond$

Proposition 6.11. *L'application de la fonction de normalisation $\mathcal{N}$ sur un pid-tree produit un pid-tree normalisé.* □

De par la définition de représentations de marquages en pid-trees, on sait que les pids à créer sont les fils les plus à droite marqués par $\perp$. Donc, pour calculer les permutations des pid-trees, on calcule les permutations de tous les fils sauf ceux marqués par $\perp$.

Dans cette section on a montré comment construire un élément de $\mathcal{R}_s(M)$ étant donné le marquage M. Pour obtenir un élément de $\mathcal{R}_{so}(M)$, on doit aussi ordonner l'arbre ce qui revient à comparer les marquages récursivement en ignorant les pids.

6.4.2 Exemple

Soit le marquage $M = \{s_1 \mapsto \{1{\cdot}2{\cdot}3,\ a\}, s_\eta \mapsto \{\langle{\cdot}1,\ 1{\cdot}4\rangle, \langle 1{\cdot}2{\cdot}3,\ 1{\cdot}2{\cdot}3{\cdot}1\rangle\}\}$ d'un réseau de Petri (S, T, ℓ) tel que $s_1 \in S$. D'abord, on construit une représentation de M en pid-tree. On utilise les fonctions d'extension sur toutes les places sauf la place génératrice.

$$\begin{aligned}
\mathcal{E}_l(\emptyset, 1\cdot2\cdot3) &= \emptyset \xrightarrow{\cdot1} \mathcal{E}_l(\bot, 2\cdot3) \\
&= \emptyset \xrightarrow{\cdot1} \bot \xrightarrow{\cdot2} \mathcal{E}_l(\bot, 3) \\
&= \emptyset \xrightarrow{\cdot1} \bot \xrightarrow{\cdot2} \bot \xrightarrow{\cdot3} \mathcal{E}_l(\bot, \langle\rangle) \\
&= \emptyset \xrightarrow{\cdot1} \bot \xrightarrow{\cdot2} \bot \xrightarrow{\cdot3} \emptyset \stackrel{\text{df}}{=} t_1
\end{aligned}$$

Maintenant, on traite les jetons de la place génératrice en utilisant l'extension à gauche pour la première composante, et l'extension à droite pour la seconde (les futur pids).

$$\begin{aligned}
\mathcal{E}_l(t_1, {\cdot}1) &= \emptyset \xrightarrow{\cdot1} \mathcal{E}_l(\bot \xrightarrow{\cdot2} \bot \xrightarrow{\cdot3} \emptyset, \langle\rangle) \\
&= \emptyset \xrightarrow{\cdot1} \emptyset \xrightarrow{\cdot2} \bot \xrightarrow{\cdot3} \emptyset \stackrel{\text{df}}{=} t_2
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}_r(t_2, 1\cdot4) &= \emptyset \xrightarrow{\cdot1} \mathcal{E}_r(\bot \xrightarrow{\cdot2} \bot \xrightarrow{\cdot3} \emptyset, {\cdot}4) \\
&= \emptyset \xrightarrow{\cdot1} \bot \xrightarrow{\cdot2,\ \cdot4} \langle \bot \xrightarrow{\cdot3} \emptyset,\ \bot\rangle \stackrel{\text{df}}{=} t_3
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}_l(t_3, 1\cdot2\cdot3) &= \emptyset \xrightarrow{\cdot1} \mathcal{E}_l(\bot \xrightarrow{\cdot2,\ \cdot4} \langle \bot \xrightarrow{\cdot3} \emptyset,\ \bot\rangle, 2\cdot3) \\
&= \emptyset \xrightarrow{\cdot1} \emptyset \xrightarrow{\cdot2,\ \cdot4} \langle \mathcal{E}_l(\bot \xrightarrow{\cdot3} \emptyset, {\cdot}3),\ \bot\rangle \\
&= \emptyset \xrightarrow{\cdot1} \emptyset \xrightarrow{\cdot2,\ \cdot4} \langle \bot \xrightarrow{\cdot3} \mathcal{E}_l(\emptyset, \langle\rangle),\ \bot\rangle \\
&= \emptyset \xrightarrow{\cdot1} \emptyset \xrightarrow{\cdot2,\ \cdot4} \langle \bot \xrightarrow{\cdot3} \emptyset,\ \bot\rangle \stackrel{\text{df}}{=} t_4
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}_r(t_4, 1\cdot2\cdot3\cdot1) &= \emptyset \xrightarrow{\cdot1} \mathcal{E}_r(\bot \xrightarrow{\cdot2,\ \cdot4} \langle \bot \xrightarrow{\cdot3} \emptyset,\ \bot\rangle, 2\cdot3\cdot1) \\
&= \emptyset \xrightarrow{\cdot1} \emptyset \xrightarrow{\cdot2,\ \cdot4} \langle \mathcal{E}_r(\bot \xrightarrow{\cdot3} \emptyset, 3\cdot1),\ \bot\rangle \\
&= \emptyset \xrightarrow{\cdot1} \emptyset \xrightarrow{\cdot2,\ \cdot4} \langle \bot \xrightarrow{\cdot3} \mathcal{E}_r(\emptyset, {\cdot}1),\ \bot\rangle \\
&= \emptyset \xrightarrow{\cdot1} \emptyset \xrightarrow{\cdot2,\ \cdot4} \langle \bot \xrightarrow{\cdot3} \emptyset \xrightarrow{\cdot1} \bot,\ \bot\rangle \stackrel{\text{df}}{=} t_5
\end{aligned}$$

La structure de l'arbre est complète, maintenant on ajoute les jetons. On ajoute le jeton $\langle 1\cdot2\cdot3,\ a\rangle$ de la place s_1.

$$\begin{aligned}
&\mathcal{U}(t_5, s_1, \langle 1\cdot2\cdot3,\ a\rangle, 1\cdot2\cdot3) \\
&= \emptyset \xrightarrow{\cdot1} \mathcal{U}(\emptyset \xrightarrow{\cdot2,\ \cdot4} \langle \bot \xrightarrow{\cdot3} \emptyset \xrightarrow{\cdot1} \bot,\ \bot\rangle, s_1, \langle 1\cdot2\cdot3,\ a\rangle, 2\cdot3) \\
&= \emptyset \xrightarrow{\cdot1} \emptyset \xrightarrow{\cdot2,\ \cdot4} \langle \mathcal{U}(\bot \xrightarrow{\cdot3} \emptyset \xrightarrow{\cdot1} \bot, s_1, \langle 1\cdot2\cdot3,\ a\rangle, {\cdot}3),\ \bot\rangle \\
&= \emptyset \xrightarrow{\cdot1} \emptyset \xrightarrow{\cdot2,\ \cdot4} \langle \bot \xrightarrow{\cdot3} \mathcal{U}(\emptyset \xrightarrow{\cdot1} \bot, s_1, \langle 1\cdot2\cdot3,\ a\rangle, \langle\rangle),\ \bot\rangle \\
&= \emptyset \xrightarrow{\cdot1} \emptyset \xrightarrow{\cdot2,\ \cdot4} \langle \bot \xrightarrow{\cdot3} \{s_1 \mapsto \{\langle 1\cdot2\cdot3,\ a\rangle\}\} \xrightarrow{\cdot1} \bot,\ \bot\rangle \stackrel{\text{df}}{=} t_6
\end{aligned}$$

On note $M_{1{\cdot}2{\cdot}3} \stackrel{\text{df}}{=} \{s_1 \mapsto \{\langle 1\cdot 2\cdot 3,\ a\rangle\}\}$ et on ajoute le jeton $\langle \cdot 1,\ 1\cdot 4\rangle$ de la place s_η.

$$\begin{aligned}
&\mathcal{U}(t_6, s_\eta, \langle \cdot 1,\ 1\cdot 4\rangle, \cdot 1)\\
&\quad = \emptyset \xrightarrow{\cdot 1} \mathcal{U}(\emptyset \xrightarrow{\cdot 2,\ \cdot 4} \langle \bot \xrightarrow{\cdot 3} M_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot,\ \bot\rangle, s_\eta, \langle \cdot 1,\ 1\cdot 4\rangle, \langle\rangle)\\
&\quad = \emptyset \xrightarrow{\cdot 1} \{s_\eta \mapsto \{\langle \cdot 1,\ 1\cdot 4\rangle\}\} \xrightarrow{\cdot 2,\ \cdot 4} \langle \bot \xrightarrow{\cdot 3} M_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot,\ \bot\rangle \stackrel{\text{df}}{=} t_7
\end{aligned}$$

On note v le jeton $\langle 1\cdot 2\cdot 3,\ 1\cdot 2\cdot 3\cdot 4\rangle$ et M_1 le marquage $\{s_\eta \mapsto \{\langle \cdot 1,\ 1\cdot 4\rangle\}\}$. On ajoute le jeton v de la place s_η.

$$\begin{aligned}
&\mathcal{U}(t_7, s_\eta, v, 1\cdot 2\cdot 3)\\
&\quad = \emptyset \xrightarrow{\cdot 1} \mathcal{U}(M_1 \xrightarrow{\cdot 2,\ \cdot 4} \langle \bot \xrightarrow{\cdot 3} M_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot,\ \bot\rangle, s_\eta, v, 2\cdot 3)\\
&\quad = \emptyset \xrightarrow{\cdot 1} M_1 \xrightarrow{\cdot 2,\ \cdot 4} \langle \mathcal{U}(\bot \xrightarrow{\cdot 3} M_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot, s_\eta, v, \cdot 3),\ \bot\rangle\\
&\quad = \emptyset \xrightarrow{\cdot 1} M_1 \xrightarrow{\cdot 2,\ \cdot 4} \langle \bot \xrightarrow{\cdot 3} \mathcal{U}(M_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot, s_\eta, v, \langle\rangle),\ \bot\rangle\\
&\quad = \emptyset \xrightarrow{\cdot 1} M_1 \xrightarrow{\cdot 2,\ \cdot 4} \langle \bot \xrightarrow{\cdot 3} (M_{1{\cdot}2{\cdot}3} \cup \{s_\eta \mapsto v\}) \xrightarrow{\cdot 1} \bot,\ \bot\rangle \stackrel{\text{df}}{=} t_8
\end{aligned}$$

Maintenant on réduit l'arbre. On note $M'_{1{\cdot}2{\cdot}3}$ le marquage $M_{1{\cdot}2{\cdot}3} \cup \{s_\eta \mapsto v\} = \{s_1 \mapsto \{\langle 1\cdot 2\cdot 3,\ a\rangle\}, s_\eta \mapsto \langle 1\cdot 2\cdot 3,\ 1\cdot 2\cdot 3\cdot 4\rangle\}$ et on continue.

$$\begin{aligned}
\mathcal{S}(t_7) &= \emptyset \xrightarrow{\cdot 1} \mathcal{S}(M_1 \xrightarrow{\cdot 2,\ \cdot 4} \langle \bot \xrightarrow{\cdot 3} M'_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot,\ \bot\rangle)\\
&= \emptyset \xrightarrow{\cdot 1} \mathcal{S}(M_1 \xrightarrow{\cdot 2\cdot 3,\ \cdot 4} \langle M'_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot,\ \bot\rangle)\\
&= \emptyset \xrightarrow{\cdot 1} M_1 \xrightarrow{\cdot 2\cdot 3,\ \cdot 4} \langle \mathcal{S}(M'_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot)\ \mathcal{S}(\bot)\rangle\\
&= \emptyset \xrightarrow{\cdot 1} M_1 \xrightarrow{\cdot 2\cdot 3,\ \cdot 4} \langle M'_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \mathcal{S}(\bot)\ \bot\rangle\\
&= \emptyset \xrightarrow{\cdot 1} M_1 \xrightarrow{\cdot 2\cdot 3,\ \cdot 4} \langle M'_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot,\ \bot\rangle \stackrel{\text{df}}{=} t_9
\end{aligned}$$

Une représentation réduite de M est

$$\emptyset \xrightarrow{\cdot 1} M_1 \xrightarrow{\cdot 2\cdot 3,\ \cdot 4} \langle M'_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot,\ \bot\rangle$$

La dernière étape est de normaliser l'arbre. Pour cela, on calcule la bijection de normalisation en utilisant la fonction $\mathcal{N}_{map}$ puis on normalise l'arbre avec la fonction $\mathcal{N}$.

$$\begin{aligned}
&\mathcal{N}_{map}(t_9, \langle\rangle, \langle\rangle) = \{\langle\rangle \mapsto \langle\rangle\} \cup \mathcal{N}_{map}(M_1 \xrightarrow{\cdot 2\cdot 3,\ \cdot 4} \langle M'_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot,\ \bot\rangle, \cdot 1, \cdot 1)\\
&\quad = \{\langle\rangle \mapsto \langle\rangle, \cdot 1 \mapsto \cdot 1\} \cup \mathcal{N}_{map}(M'_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot, 1\cdot 2\cdot 3, 1\cdot 1\cdot 1) \cup \mathcal{N}_{map}(\bot, 1\cdot 4, 1\cdot 2)\\
&\quad = \{\langle\rangle \mapsto \langle\rangle, \cdot 1 \mapsto \cdot 1, 1\cdot 2\cdot 3 \mapsto 1\cdot 1\cdot 1, 1\cdot 4 \mapsto 1\cdot 2\} \cup \mathcal{N}_{map}(\bot, 1\cdot 2\cdot 3\cdot 1, 1\cdot 1\cdot 1\cdot 1)\\
&\quad = \{\langle\rangle \mapsto \langle\rangle, \cdot 1 \mapsto \cdot 1, 1\cdot 2\cdot 3 \mapsto 1\cdot 1\cdot 1, 1\cdot 4 \mapsto 1\cdot 2, 1\cdot 2\cdot 3\cdot 1 \mapsto 1\cdot 1\cdot 1\cdot 1\} \stackrel{\text{df}}{=} f
\end{aligned}$$

Il faut remarquer qu'on associe $1\cdot 2\cdot 3$ à $1\cdot 1\cdot 1$ parce que la taille de $2\cdot 3$ est supérieure à 1, et donc on utilise $1\cdot 1$ (cas $\langle i, 1\rangle$ dans la définition). En utilisant le fonction de normalisation f on peut normaliser l'arbre.

$$\begin{aligned}
\mathcal{N}(t_9, f) &= \emptyset \xrightarrow{\cdot 1} \mathcal{N}(M_1 \xrightarrow{\cdot 2\cdot 3,\ \cdot 4} \langle M'_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot,\ \bot\rangle, f)\\
&= \emptyset \xrightarrow{\cdot 1} M'_1 \xrightarrow{\cdot 1\cdot 1,\ \cdot 2} \langle \mathcal{N}(M'_{1{\cdot}2{\cdot}3} \xrightarrow{\cdot 1} \bot, f),\ \mathcal{N}(\bot, f)\rangle\\
&= \emptyset \xrightarrow{\cdot 1} M_1 \xrightarrow{\cdot 1\cdot 1,\ \cdot 2} \langle M_{1{\cdot}1{\cdot}1} \xrightarrow{\cdot 1} \mathcal{N}(\bot, f),\ \bot\rangle\\
&= \emptyset \xrightarrow{\cdot 1} M_1 \xrightarrow{\cdot 1\cdot 1,\ \cdot 2} \langle M_{1{\cdot}1{\cdot}1} \xrightarrow{\cdot 1} \bot,\ \bot\rangle
\end{aligned}$$

où $M_1' = f(M_1) = \{s_\eta \mapsto \{\langle \cdot 1,\ 1 \cdot 2\rangle\}\}$ et $M_{1\cdot1\cdot1} = f(M_{1\cdot2\cdot3}) = \{s_1 \mapsto \{\langle 1 \cdot 1 \cdot 1,\ a\rangle\}, s_\eta \mapsto \langle 1 \cdot 1 \cdot 1,\ 1 \cdot 1 \cdot 1 \cdot 1\rangle\}$ Finalement, la représentation normalisée et réduite de M est

$$\emptyset \xrightarrow{\cdot 1} M_1 \xrightarrow{\cdot 1 \cdot 1,\ \cdot 2} \langle M_{1\cdot1\cdot1} \xrightarrow{\cdot 1} \bot,\ \bot\rangle$$

Ce pid-tree est le même que celui donné en figure 6.1.

6.5

Réduction, normalisation, et comportements infinis

Dans cette section on montre pourquoi la réduction et la normalisation de pid-trees sont essentielles pour capturer les comportements infinis. Considérons le réseau de Petri donné en figure 6.6a, son comportement est similaire à de la récursion terminale. Il y a une place s contenant un pid, une place génératrice avec des données adéquates, et une transition t. Chaque fois que la transition t est tirée, le processus π crée un fils et meurt.

Le marquage initial est $\{s \mapsto \{\cdot 1\}, s_\eta \stackrel{\text{df}}{=} \{\langle \cdot 1, 1 \cdot 1\rangle\}\}$. À partir de l'état initial, l'espace d'état construit en utilisant un algorithme d'exploration classique est donné en figure 6.6b et est infini.

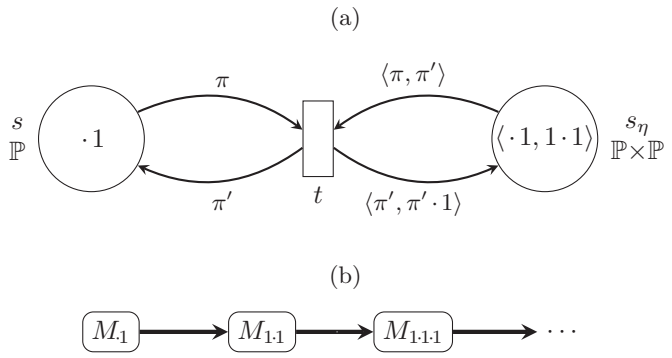


FIGURE 6.6 – *Exemple de réseau de Petri marqué (a), et son espace d'états infini (b) où* $M_\pi \stackrel{\text{df}}{=} \{s \mapsto \{\pi\}, s_\eta \mapsto \{\langle \pi, \pi \cdot 1\rangle\}\}$.

Si on construit les représentations en pid-tree de chacun des marquages dans l'espace d'états, on observe que ces arbres deviennent de plus en plus grands parce que de nouveaux pids sont créés. Lorsqu'on réduit les arbres on observe que les arbres ont toujours le même nombre de noeuds mais les marquages et annotations sur les arcs grossissent toujours. Pour résoudre ce problème on normalise l'arbre et on remplace les pids ce qui mène à une unique représentation en pid-tree pour chaque état de l'espace d'états. Parce qu'on a une unique représentation en pid-tree normalisé, on a aussi une unique classe d'équivalence d'états et donc un espace d'états fini et réduit à un unique état. Les pid-trees et l'espace d'états réduit sont donnés en figures 6.7 et 6.8.

Un autre exemple intéressant à considérer est le même réseau de Petri où l'état initial est $\{s \mapsto \{1 \cdot 1\}, s_\eta \mapsto \{\langle \cdot 1 \mapsto 1 \cdot 2\rangle, \langle 1 \cdot 1 \mapsto 1 \cdot 1 \cdot 1\rangle\}\}$. Cette fois on aura deux classes d'équivalence, une où deux processus sont dans la relation parent et une autre où les deux processus sont dans la relation ancêtre.

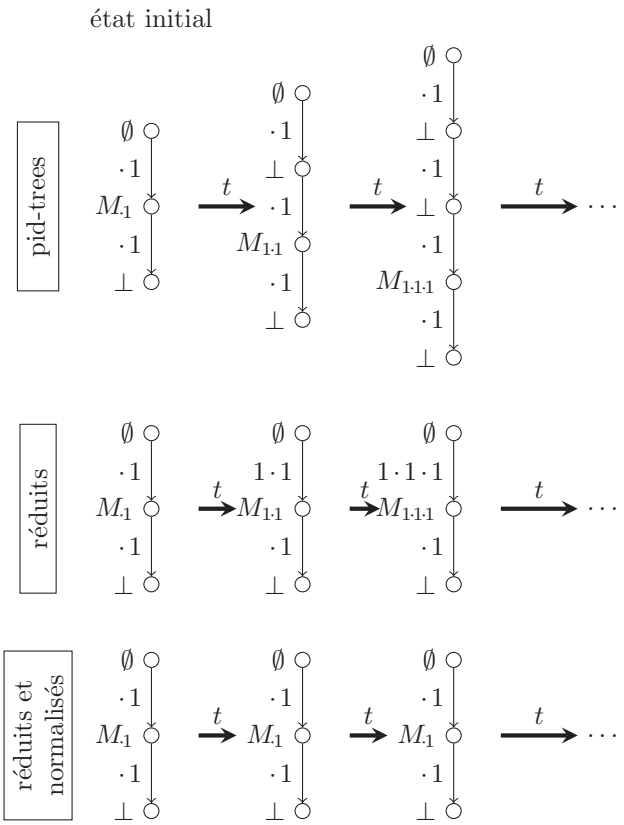


Figure 6.7 – *Les pid-trees correspondants à l'espace d'états du réseau de Petri en figure 6.6a où* $M_\pi \stackrel{\text{df}}{=} \{s \mapsto \{\pi\}, s_\eta \mapsto \{\langle \pi, \pi \cdot 1 \rangle\}\}$.

Figure 6.8 – *Espace d'états réduit du réseau de Petri en figure 6.6a.*

6.6

Algorithmes pour détecter les symétries

6.6.1 Algorithme général

L'algorithme général de détection de symétries dans les réseaux de Petri de haut niveau est basé sur le théorème 6.8. L'idée est de ne stocker que les marquages correspondant à des pid-trees normalisés. C'est correct d'après la proposition 6.2 et le théorème 6.1. Pour cela, pour chaque nouveau marquage on construit un pid-tree, on le réduit, on l'ordonne et on énumère tous les pid-trees équivalents. Pour chacun de ces pid-trees on construit un marquage normalisé et on teste si ce marquage est dans l'espace d'états. Si une permutation satisfait ce test, l'état peut être ignoré, *i.e.*, on vient juste trouver un état qui est équivalent

à un état déjà découvert. Sinon, on ajoute un marquage normalisé dans l'espace d'états. Parce qu'on ne garde qu'un état par classe d'équivalence, et parce qu'on a la possibilité de raffiner l'ensemble potentiel d'états symétriques, on peut proposer un algorithme qui énumère un nombre de pid-trees bien moindre. L'algorithme 6.2 décrit cette approche.

Entrée : marquage $\boldsymbol{M}$, ensemble de marquages ***States***
Sortie : ensemble de marquages contenant un représentant pour $\boldsymbol{M}$

$\boldsymbol{t} \leftarrow$ pid-tree à partir de $\boldsymbol{M}$;
$\boldsymbol{t} \leftarrow$ réduire le pid-tree $\boldsymbol{t}$;
$\boldsymbol{t} \leftarrow$ ordonner le pid-tree $\boldsymbol{t}$; `//avec l'ordre pid-free`

pour toute *permutation pid-free ordonnée* $\boldsymbol{tp}$ *de* $\boldsymbol{t}$ **:**
  $\boldsymbol{tp} \leftarrow N(\boldsymbol{tp})$; `//normaliser l'arbre, changer les pids`
  $\boldsymbol{M'} \leftarrow mrk(\boldsymbol{tp})$; `//construire le marquage normalisé`
  si $\boldsymbol{M'} \in \boldsymbol{States}$ **alors**
    renvoyer ***States***; `//état trouvé`
  fin si
fin pour

`//nouvel état, on met à jour l'ensemble d'états`
`//on ajoute une représentation normalisée`
$\boldsymbol{t} \leftarrow N(\boldsymbol{t})$; `//valeurs minimales pour pids`
$\boldsymbol{M'} \leftarrow mrk(\boldsymbol{t})$; `//marquage équivalent à` $\boldsymbol{M}$
renvoyer $\boldsymbol{States} \cup \{ \boldsymbol{M'} \}$;

Algorithme 6.2: Algorithme général de détection de symétries.

Proposition 6.12. *L'algorithme 6.2 est correct et termine.* □

Cet algorithme a deux avantages par rapport à ceux proposés dans [114, 115, 116]. En effet, le problème principal avec les algorithmes précédents était que chaque état devait être transformé en graphe puis comparé avec tous les autres états de l'espace d'états par une opération d'isomorphisme de graphes. Cela a deux défauts : on doit réaliser des isomorphismes de graphes et on doit le réaliser avec tous les états de l'espace d'états.

Maintenant la complexité de l'approche a été transposée à l'énumération des pid-tree réduits et ordonnées, c'est-à-dire qu'on ne dépend plus de la taille de l'espace d'états et de la taille du marquage, comme c'était le cas pour les isomorphismes, mais juste de la taille du marquage. Plus précisément, la complexité de l'approche dépend du nombre de pids dans le marquage (ce qui était évidemment aussi le cas pour le calcul d'isomorphisme de graphe). Il faut mettre en avant le fait que maintenant le test d'appartenance $M' \in States$ peut être réalisé en temps constant (en général), en utilisant des tables de hachage, tandis que dans [114, 115, 116] il fallait tester tous les états de l'espace d'états en temps linéaire et réaliser un calcul d'isomorphisme de graphe pour chaque état. De plus, l'ordonnancement des pid-trees permet de filtrer les permutations nécessaires des autres, ce qui réduit en général le nombre de permutations, mais ne permet pas bien sûr d'éviter un pire cas.

6.6.2 Algorithme optimiste

Une approche optimiste qui évite l'énumération des permutations est possible, cependant elle ne fournit pas une réduction maximale de l'espace d'états, *i.e.*, elle peut ne pas détecter toutes les symétries. Une telle approche, pour être utilisable en pratique doit fournir un espace d'états fini. C'est le cas ici, en effet, même si on ne détecte pas toutes les symétries on a la garantie que l'espace d'états que l'on construit est fini si le nombre de classes d'équivalence d'états est fini, *i.e.*, s'il est possible d'obtenir une représentation finie de l'espace d'états en utilisant un représentant par classe d'équivalence. Cette approche est donnée en algorithme 6.3 et consiste à construire l'espace d'états en agrégeant les formes normalisés des marquages découverts (une fois réduits et ordonnés). Ajouter les formes normalisées au lieu des marquages rencontrés est correct parce qu'ils sont équivalents. Dans le pire des cas l'algorithme ajoute un marquage par élément de $\mathcal{R}_{so}(M)$ à l'espace d'états pour chaque marquage M découvert, mais $\mathcal{R}_{so}(M)$ étant fini l'algorithme termine.

Entrée : marquage $\boldsymbol{M}$, ensemble de marquages ***States***
Sortie : ensemble de marquages contenant un représentant pour $\boldsymbol{M}$

$\boldsymbol{t} \leftarrow$ pid-tree à partir de $\boldsymbol{M}$;
$\boldsymbol{t} \leftarrow$ réduire le pid-tree $\boldsymbol{t}$;
$\boldsymbol{t} \leftarrow$ ordonner le pid-tree $\boldsymbol{t}$; `//avec l'ordre pid-free`
$\boldsymbol{t} \leftarrow N(\boldsymbol{t})$; `//valeurs minimales pour pids`
$\boldsymbol{M'} \leftarrow mrk(\boldsymbol{t})$; `//un marquage équivalent à M`
renvoyer $\boldsymbol{States} \cup \{\boldsymbol{M'}\}$;

Algorithme 6.3: Algorithme optimiste de détection de symétries.

Proposition 6.13. *L'algorithme 6.3 est correct et termine.* □

Une observation intéressante sur l'algorithme optimiste est qu'il est aussi complet pour une sous classe de modèles. En effet si les jetons ne possèdent des pids qu'en première position alors toutes les symétries sont capturées.

Théorème 6.14. *Soit (S, T, ℓ) un réseau de Petri tel que pour toute place $s \in S$, $\ell(s) = \mathbb{P} \times X_1 \times \cdots \times X_n$ et $X_i \subseteq \mathbb{D}_v$ $(n \geq 0)$,* i.e., *les pids n'apparaissent qu'en première position. Alors, l'algorithme optimiste est complet,* i.e., *capture toutes les symétries.* □

6.7 Récupération de contre-exemples

La vérification de propriétés logiques avec du model checking explicite fournit des contre-exemples lorsque la formule testée n'est pas satisfaite. Il s'agit d'une trace d'exécution qui retrace tout le scénario qui a mené à l'erreur. Ce contre-exemple est très utile pour déboguer le modèle puisqu'on sait exactement comment l'erreur est survenue.

Dès lors qu'on fait des réductions avec équivalences où l'état stocké n'est pas atteignable cette étape de vérification s'avère plus compliquée. Le contre-exemple qu'on construit lors de l'exploration de l'espace d'états est une suite d'états équivalents aux états atteignable. Une telle suite d'états devient très vite difficile à observer dans le modèle car la transformation vers un état de l'espace d'états non réduit peut s'avérer non triviale. On peut alors se poser la question si on peut vraiment arriver à cette situation dans le modèle, ou si c'est une erreur dans le model checker lui même. Il est donc important de reconstruire une trace d'exécution existante dans l'espace d'états non réduit. Évidemment il faut que cette reconstruction n'ait pas besoin de tout l'espace d'états, sinon la méthode de réduction ne serait pas pertinente.

Dans cette section on propose une méthode pour reconstruire une trace d'exécution concrète (trace dans l'espace d'états complet du système) à partir d'une trace d'exécution abstraite (trace dans l'espace d'états réduit).

Remarquons qu'une telle discussion n'avait pas lieu d'être dans [114, 115, 116] puisque l'espace d'états réduit était composé d'états accessibles, ce qui n'est pas notre cas à cause de la fonction de normalisation (on ne stocke que les marquages normalisés).

Une méthode naïve en utilisant le théorème 6.1 peut devenir rapidement coûteuse. Elle consisterait à calculer des marquages de proche en proche de manière à aboutir à une trace concrète. Pour cela, il faudrait donc à chaque fois inverser la bijection de normalisation utilisée et l'appliquer sur le mode utilisé pour calculer le successeur dans la trace. Lors du calcul du contre-exemple concret, il y a beaucoup de bijections intermédiaires qui n'ont pas pu être calculés auparavant, il faut donc les calculer puis les inverser. On peut donc voir que l'opération est quadratique en nombre d'états de la trace abstraite avec comme opérations de base : le calcul de bijection de normalisation de proche en proche, l'inversion de bijection et le tir des successeurs. Cependant il y a deux problèmes : les marquages intermédiaires ne sont pas forcément normalisés, il faut donc calculer les formes normales de tous les marquages intermédiaires et les utiliser pour le calcul de bijections, et les états intermédiaires ne sont pas accessibles pour autant, donc dans un pire cas le calcul avec cet algorithme peut être plus coûteux que le calcul de l'espace d'états complet. Un exemple est donné en figure 6.9, on remarque le grand nombre d'inversions réalisées sur les états intermédiaires. Sachant que ces états doivent être calculés et sont potentiellement inaccessibles l'ensemble du calcul de la trace peut être plus coûteux que l'exploration de l'espace d'états complet.

Une meilleure approche est basée sur l'observation suivante : les marquages de la trace concrète et les marquages de la trace dans l'espace d'états réduit sont équivalents. On peut donc directement utiliser les pid-trees pour calculer la bijection de normalisation qui permet de passer de la trace concrète à la trace abstraite. L'algorithme de calcul de la trace complète est illustré sur la figure 6.10 et est basé sur la démarche suivante :

1. À partir du marquage initial on tire une à une les transitions de la trace abstraite.
2. Pour chaque état successeur concret calculé, on teste s'il est équivalent au successeur de la trace abstraite.
3. S'il est équivalent alors on a notre successeur concret, maintenant on continue les tirs à partir de cet état.

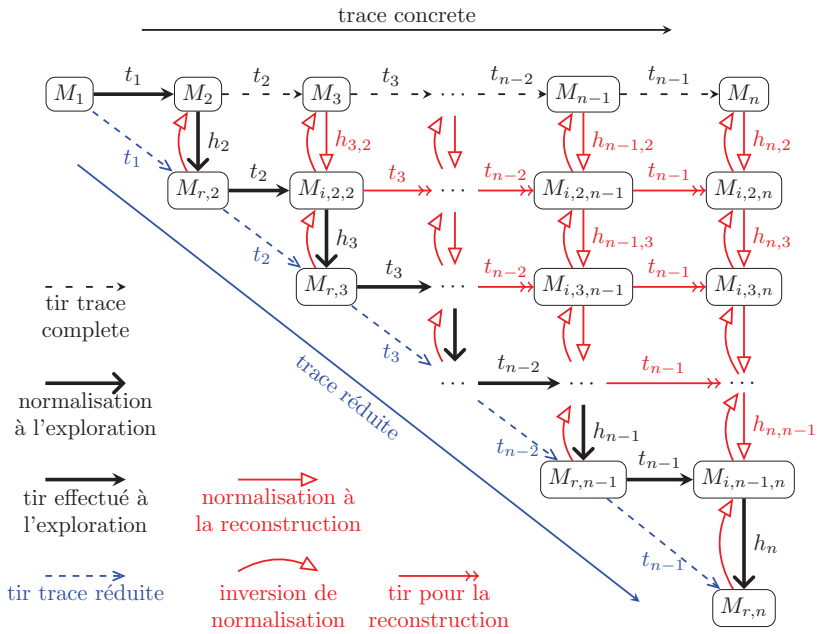


Figure 6.9 – *Exemple de reconstruction de proche en proche.*

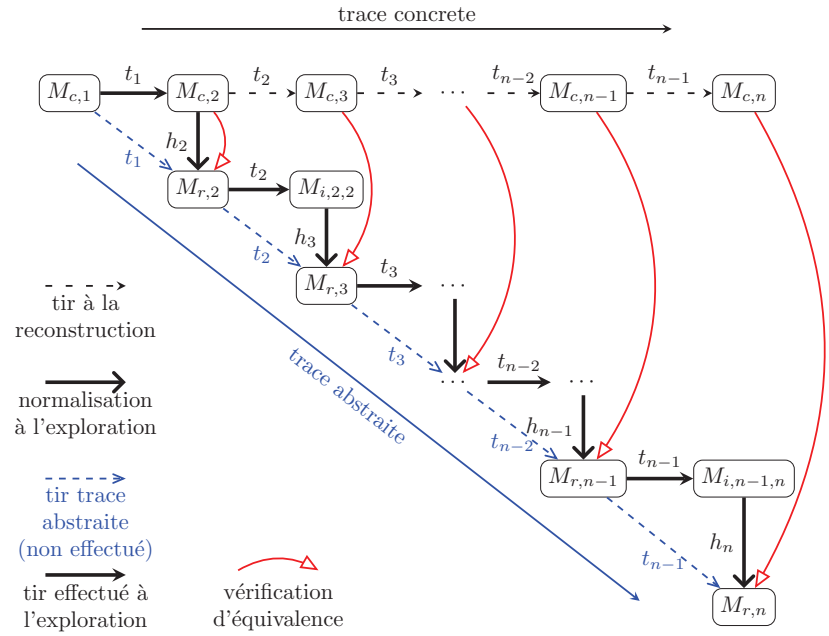


Figure 6.10 – *Exemple de reconstruction.*

De cette manière le calcul de la trace concrète devient linéaire (si le nombre d'états successeurs est borné par une constante) avec comme opération de base

le tir des successeurs et le test d'équivalence avec un état abstrait qui correspond au test réalisé dans l'algorithme général 6.2.

6.8

Preuves des théorèmes

Cette section regroupe les preuves des théorèmes présentés dans ce chapitre, les preuves des résultats auxiliaires peuvent être trouvés dans l'annexe B. Toutes les preuves sont faites dans le contexte d'un réseau de Petri $N = (S, T, \ell)$.

6.8.1 Preuve du théorème 6.1

Soient M et M' deux marquages h-equivalents et pid-cohérents d'un réseau de Petri (S, T, ℓ), et soient $t \in T$ une transition et β une valuation telles que $M[t, \beta\rangle\widetilde{M}$. Alors $M'[t, h \circ \beta\rangle\widetilde{M'}$ et $\widetilde{M} \sim \widetilde{M'}$.

Cette preuve est similaire à la preuve du théorème 3 de [116], la connaissance de ces travaux est donc nécessaire à sa compréhension. Cependant, parce qu'on a changé la définition de la place générateur, introduit la pid-cohérence et adapté la relation d'équivalence, nous avons besoin de garantir que ceux ci sont bien définis.

Pour chaque marquage M on définit l'état correspondant $q_M \stackrel{\text{df}}{=} (\sigma_M, \eta_M)$ donné par :

$$\begin{array}{rcl} \sigma_M & \stackrel{\text{df}}{=} & \{\ (s, M(s)) \mid s \in S\ \} \\ \eta_M & \stackrel{\text{df}}{=} & \{\ \pi \mapsto k \mid \langle \pi, \pi \cdot \langle k \rangle \rangle \in M(s_\eta)\ \} \end{array}$$

Soit M un marquage, alors la *thread configuration* correspondante générée est $ctc_M \stackrel{\text{df}}{=} (G, H)$, où $H = \eta_M$ et G est un ensemble de tous les pics qui apparaissent au marquage M à l'exception des pids à créer, *i.e.*, pour chaque jeton $\langle \pi, \pi' \rangle \in M(s_\eta)$ seulement π est dans G. De cette manière la configuration résultante est exactement la même que dans [116].

Definition 6.23. *Un état q_M génère une* thread-consistent configuration *(ou ct-configuration), si (G, H), comme défini ci-dessus, satisfait :*

- *$dom(H) \subseteq G$,* i.e.*, chaque pid actif est présent dans l'état ; et*
- *pour tout $\pi \in dom(H)$ et $\pi' \in G$, si $\pi \cdot \langle k \rangle$ est un préfixe de π' alors $k \leq \eta_q(\pi)$,* i.e.*, les pids présents dans q ne peuvent être recréés.* ◊

L'étape suivante est de montrer qu'un marquage pid-cohérent produit une ct-configuration.

Proposition 6.15. *Un marquage M est pid-cohérent si et seulement si ctc_M est une ct-configuration.*

Preuve. Soit ctc_M une configuration (H, G) défini comme ci-dessus.

(⇒) On suppose que M est pid-cohérent. On vérifie que ctc_M est une ct-configuration.

- d'abord on doit prouver que $H \subseteq G$: c'est trivial parce que H est construit avec la place générateur donc chaque pid dans H est dans G.

– maintenant, montrons que pour tout $\pi \in dom(H)$ et $\pi' \in G$, si $\pi.k$ est un préfixe de π' alors $k \leq \eta_M(\pi)$, *i.e.*, pids dans M ne peuvent pas être recréés encore : ceci est immédiat par la définition de la pid-cohérence.

($\Leftarrow$) On suppose que ctc_M est une ct-configuration. On vérifie que M est pid-cohérent. C'est immédiat par définition des ct-configurations et de pid-cohérence.

□

Avant d'aller plus loin, on doit vérifier que la définition de l'équivalence de marquages en définition 6.4 est équivalente à l'équivalence d'états dans [116].

Proposition 6.16. *Soient M, M' deux marquages pid-cohérents, et q, q' leurs états correspondants. Alors $M \sim M'$ si et seulement si $q \sim q'$.*

Preuve.

($\Rightarrow$) On montre que $M \sim M' \Rightarrow q \sim q'$.

Soit h la bijection qui satisfait l'équivalence de marquages, alors h peut être utilisé comme candidat pour l'équivalence d'états car $dom(h) = pid_q \cup nextpid_q$ (pour tout $\pi \in dom(\eta_q)$, $nextpid_q(\pi)$ est maintenant dans la place générateur), même argument pour $dom(h^{-1})$ en utilisant q'. Maintenant on montre que h satisfait l'équivalence de [116].

1. $h(dom(\eta_q)) = dom(\eta_{q'})$, parce que M' est M après remplacement de chaque pid π_0 dans M par $h(\pi_0)$ et h est une bijection de pid_M vers $pid_{M'}$.
2. $\forall \pi \in dom(\eta_q), h(next_q(\pi)) = next_{q'}(h(\pi))$.
   Soit $\pi \in dom(\eta_q)$ un pid, alors par définition de η_q, $\langle \pi, next_q(\pi) \rangle \in M(s_\eta)$. De plus, parce que h préserve la relation $\lhd$ et M' est M après remplacement de chaque pid π_0 dans M par $h(\pi_0)$, on sait que $\langle h(\pi), h(next_q(\pi)) \rangle \in M'(s_\eta)$ et donc par définition de $\eta_{q'}$, $h(next_q(\pi)) = next_{q'}(h(\pi))$.
3. $\forall \pi, \pi' \in pid_q : \pi \prec \pi'$ ssi $h(\pi) \prec h(\pi')$ avec $\prec \in \{\lhd_1, \lhd\}$: immédiat parce que h préserve les relations $\lhd_1$ et $\lhd$.
4. $\sigma_{q'}$ est σ_q après remplacement de chaque pid π par $h(\pi)$, immédiat parce que M' est M après remplacement de chaque pid π dans M par $h(\pi)$

Toutes les conditions sont satisfaites.

($\Leftarrow$) $q \sim q' \Rightarrow M \sim M'$

Similaire à ci-dessus, avec h la bijection qui satisfait l'équivalence d'états.

□

Il est prouvé dans [116] que le tir de transition préserve les ct-configurations, *i.e.*, si ctc_M est une ct-configuration et $M[\beta, t\rangle M'$, alors $ctc_{M'}$ est une ct-configuration. Ceci est donné dans la preuve du théorème 2 de [116] qui décrit que les états accessibles correspondent à des ct-configurations.

Donc, de par la proposition 6.15 chaque marquage accessible à partir d'un état pid-cohérent correspond à une ct-configuration. À partir de maintenant la preuve du théorème 6.1 est fait de la même manière que la preuve du théorème 3 dans [116]. □

6.8.2 Preuve du théorème 6.5

Soit M et M' deux marquages pid-cohérents d'un réseau de Petri. Alors :

$$M \sim M' \Leftrightarrow \forall t \in \mathcal{R}_s(M), \exists t' \in \mathcal{R}_s(M'), \mathcal{N}(t) = \mathcal{N}(t')$$

Preuve.

($\Rightarrow$) Supposons que $M \sim_h M'$, et soit $t \in \mathcal{R}_s(M)$ une représentation réduite du marquage M. Par définition de h, h préserve les relations $\lhd_1$ et $\lhd$. Notons t' l'arbre $h(t)$. t' est un pid-tree avec la même structure que t, on a donc $\mathcal{N}(t) = \mathcal{N}(t')$. On doit juste montrer que t' est une représentation sous forme de pid-tree de M' (définition 6.11 page 99) :

1. t' est bien formé (définition 6.8 page 97) : pour chaque sous-pid-tree $t_0 = M_\pi \xrightarrow{a_1,\ldots,a_n} \langle t_1, \ldots, t_n \rangle \in \Xi_\pi$ de t il y a un sous-pid-tree $t'_0 = M_{h(\pi)} \xrightarrow{a'_1,\ldots,a'_n} \langle t'_1, \ldots, t'_n \rangle \in \Xi_{h(\pi)}$ de t' tel que :
   - si $n = 0$ et $M_\pi = \bot$ alors π est un futur-pid dans M, donc $M_{h(\pi)} = \bot$ et parce que h préserve la relation parent, $h(\pi)$ est aussi un futur-pid dans M'.
   - le second point est trivial parce que h préserve la relation parent.
   - parce qu'on utilise des pid-trees réduits il n'y a pas de noeud $\bot$ dans t à l'exception de futur-pids, ceci est aussi vrai pour t'.
2. $mrk(t') = M'$ par définition de h.
3. parce que h préserve la relation parent, il assigne les futur-pids de t à de futur-pids de t' donc les futur pids dans t sont aussi les fils les plus à droite (noeuds $\bot$).

($\Leftarrow$) On suppose que $\exists t \in \mathcal{R}_s(M)$ et $\exists t' \in \mathcal{R}_s(M')$ tels que $\mathcal{N}(t) = \mathcal{N}(t')$. On pose $A \stackrel{\text{df}}{=} mrk(\mathcal{N}(t))$. De par la proposition 6.2, il y a deux bijections h_1, h_2 telles que $mrk(t) = M \sim_{h_1} A$ et $A \sim_{h_2} M' = mrk(t')$. On conclue que $M \sim_{h_1 \circ h_2} M'$.

□

6.8.3 Preuve du théorème 6.8

Soit M et M' deux marquages pid-cohérents d'un réseau de Petri. Alors :

$$M \sim M' \Leftrightarrow \forall t \in \mathcal{R}_{so}(M), \exists t' \in \mathcal{R}_{so}(M'), \mathcal{N}(t) = \mathcal{N}(t')$$

Preuve. Étant donné deux pid-trees réduits qui satisfont le théorème 6.5, on peut les ordonner ce qui donne les égalités recherchées. □

6.8.4 Preuve du théorème 6.14

Soit (S, T, ℓ) un réseau de Petri tel que pour toute place $s \in S$, $\ell(s) = \mathbb{P} \times X_1 \times \cdots \times X_n$ et $X_i \subseteq \mathbb{D}_v$ ($n \geq 0$), i.e.*, les pids n'apparaissent qu'en première position. Alors, l'algorithme optimiste est complet,* i.e.*, capture toutes les symétries.*

Preuve. (schéma) Il faut montrer que toutes les représentation de $\mathcal{R}_{so}$ d'un marquage, tel que dans l'énoncé, ont la même forme normalisée. On sait déjà que tous les pid-trees ont la même structure parce qu'on utilise l'ordre pid-free. Les formes normales ne peuvent différer que par les pid or puisqu'on a les pids qu'en première position la normalisation ne peut produire que des pids égaux. □

6.9 Conclusion

Dans ce chapitre on a vu une manière de détecter des symétries dans les systèmes à création dynamique de processus. Cette approche est décidable et réalisable en pratique. Des résultats expérimentaux sont présentés dans le chapitre 7.

Cette approche est basée sur une structure de données arborescente permettant de facilement détecter les symétries grâce à des formes canoniques. Deux algorithmes ont été proposés un algorithme général qui vise à détecter toutes les symétries, et un algorithme optimiste moins coûteux. Ces deux algorithmes ont un avantage important par rapport aux approches existantes : ils peuvent être utilisés simultanément avec des techniques de hachage pour réaliser des tests rapides de l'existence d'un état dans l'espace déjà exploré.

De plus l'approche présentée dans ce chapitre est économe en mémoire, on stocke des marquages et non des représentations de marquages. C'est un aspect important parce que les model checkers, et les bibliothèques produites par la compilation comme celles présentées dans le chapitre 3, sont optimisés pour stocker des marquages et non des structures exotiques. La réduction n'a donc pas d'impact sur cette représentation.

On a aussi montré que lors de la vérification de propriétés logiques, il est aisé de reconstruire une trace concrète d'exécution. Le prix de cette reconstruction est en général linéaire en la taille de la trace abstraite et donc cette reconstruction est facilement applicable.

7 Neco : implémentation et études de cas

Le framework Neco a débuté en tant que compilateur de réseaux de Petri haut niveau, dans notre cas cela signifie un outil qui prend en entrée un réseaux de Petri et produit une bibliothèque logicielle permettant de calculer son espace d'états. Maintenant, Neco est devenu une boite à outils permettant non seulement de calculer des espaces d'états mais aussi de réaliser du model checking LTL explicite. Ce framework comprend au total : un compilateur pour les réseaux de Petri, un outil de construction d'espaces d'états, un second compilateur pour les formules de logique LTL et un model checker LTL.

Les modèles acceptés par Neco sont une variante très générale des réseaux de Petri haut niveau basée sur le langage Python, *i.e.*, les valeurs, les expressions etc., annotant le réseau sont exprimés en Python. De plus, diverses extensions sont présentes telles que les arcs inhibiteurs (*inhibitor-arcs*), les arcs de lecture (*read-arcs*) et les arcs flush (*whole-place-arcs*). On peut assimiler ces modèles à des réseaux de Petri colorés [103] mais annotés avec le langage Python au lieu d'un dialecte de ML comme c'est traditionnellement le cas. Neco peut être vu comme une implantation des méthodes du chapitre 3 sur cette variante de réseaux de Petri, mais supporte aussi les réductions par symétries présentées dans le chapitre 6.

En particulier, Neco peut exploiter diverses propriétés des réseaux de Petri pour optimiser la représentation du marquage et les algorithmes de tir de transitions, pour améliorer les temps d'exécution mais aussi l'occupation mémoire.

Finalement Neco est capable de typer la majorité du code Python embarqué dans le réseau de Petri grâce aux informations de typage sur les places. Ceci permet de générer du code `C++` efficace grâce au langage Cython sans pénalité sur l'interprétation du code Python en général.

L'ensemble des optimisations crée une accélération notable du tirage des transitions et par conséquent de l'exploration d'espaces d'états [FP11a] et donc du model checking explicite. Ceci a aussi été confirmé lors de la participation de Neco au *Model Checking Contest* (événement satellite de la conférence Petri Nets 2012 et 2013), ce qui a montré que Neco est capable d'être compétitif avec les outils concurrents [KLB+12, KLB+13]

Dans ce chapitre on introduit les concepts principaux de l'architecture de Neco, on présente son utilisation, ainsi que l'évaluation de ses performances sur

des études de cas.

Neco est un logiciel libre distribué sous la licence GNU LGPL et peut être obtenu à l'adresse `http://code.google.com/p/neco-net-compiler` où une documentation est aussi disponible. La documentation inclue un tutoriel, une référence de l'API des bibliothèques générés par Neco et des exemples concrets.

Ce chapitre est basé sur [FP13] et [FDL13]. Les résultats de performances présentés ont été réalisés spécialement pour ce manuscrit.

7.1 Architecture et utilisation de Neco

Neco est une collection de deux compilateurs, un outil d'exploration et un model checker :

- `neco-compile` est le compilateur principal qui produit un moteur d'exploration d'un réseau de Petri sous forme de bibliothèque logicielle ;
- `neco-explore` est un outil d'exploration minimaliste qui calcule l'espace d'états d'un réseau de Petri grâce au moteur d'exploration produit par `neco-compile` ;
- `neco-check` est un compilateur de formules LTL qui produit une bibliothèque nécessaire pour les manipuler ;
- `neco-spot` est un model checker LTL qui utilise un moteur d'exploration construit par `neco-compile` et un adaptateur LTL créé par `neco-check`, mais aussi la bibliothèque *Spot* pour les algorithmes de model checking LTL [52].

En tant que compilateur, Neco propose deux *backends* :

- le backend *Python* qui produit un module Python ;
- le backend *Cython* qui permet de générer une variante de code Python annotée par des informations de types, ce code peut par la suite être compilé en `C++` efficace. Cela donne un module utilisable depuis Python mais aussi depuis `C` ou `C++`.

Un backend LLVM a aussi existé mais n'est plus maintenu, la raison principale de l'abandon est que les performances étaient comparables aux résultats obtenus avec le backend Cython, mais le temps de développement beaucoup plus important. Il devenait alors difficile de développer et tester rapidement des idées d'optimisations. Dans la phase d'expérimentation, le backend Python est le plus utile, lorsqu'on obtient des résultats intéressants, ils peuvent être portés dans le backend Cython. À l'avenir, lorsque Neco sera stabilisé, le backend LLVM pourra être réintroduit pour permettre la preuve des outils comme on l'a vu au chapitre 4.

Cette implémentation utilisait la bibliothèque LLVM-py [125], un binding de la bibliothèque LLVM en Python qui permet de générer du code source LLVM de manière programmatique sans manipuler directement d'arbres de syntaxe abstraite. Des résultats expérimentaux avec une comparaison au model checker Helena [135, 55] sont donnés dans [FP11a].

Un backend *Java* existe aussi mais ne fait pas encore partie de la branche principale de Neco. Il a été réalisé dans le cadre d'un stage de Master, ce qui a permis de mettre en avant l'extensibilité de notre outil et l'indépendance du langage cible. Cette implémentation a pour but de permettre le calcul parallèle

multi-coeur ce qui n'est pas possible en Python à cause d'un verrou global sur l'interpréteur. Les algorithmes de calcul des successeurs restent très similaires, mais utilisent des structures de données partagées.

Chaque outils composant Neco est dédiée à une tâche spécifique, gardant à l'esprit la philosophie unix : faire une seule tâche par outil et la faire aussi bien que possible [63]. Le workflow détaillé d'utilisation de Neco est donné en figure 7.1. Dans cette section on suppose qu'on ne considère que le backend Cython qui est le plus efficace. Cependant la majorité des explications s'appliquent aussi au backend Python. On présente tout d'abord comment est construit le moteur d'exploration et comment l'utiliser pour calculer les espaces d'états. Par la suite, on présente comment réaliser du model checking LTL avec Neco, qui est actuellement une partie non supportée par le backend Python. Cependant le backend Python possède aussi des caractéristiques qui ne sont pas encore disponibles en Cython, comme les réductions par symétries présentés dans le chapitre 6 [Fro12a].

7.1.1 Moteur d'exploration et calcul d'espaces d'états

La première étape lorsqu'on utilise Neco est de créer une bibliothèque `net.so` avec l'outil `neco-compile`. Cette bibliothèque fournit des primitives d'exploration : une structure de marquage, des fonctions successeurs spécifiques aux transitions, et une fonction successeur globale qui appelle les fonctions successeurs spécifiques (chap. 3). Un schéma d'un telle bibliothèque est rappelé en figure 7.2. Ce moteur d'exploration peut être utilisé par un programme client (*e.g.*, un model checker ou un simulateur) pour l'aider à accomplir sa tâche. La bibliothèque générée embarque directement le code du modèle (*i.e.*, les annotations du réseau de Petri en Python) mais se base aussi sur des structures de données préexistantes (en particulier, les ensembles et multiensembles) formant les *bibliothèques noyau*. L'accès à ces bibliothèques se fait par des interfaces normalisées, intuitivement similaires à celles discutées dans les chapitres 3 et 4. Le code source fourni par l'utilisateur annotant le modèle quant à lui a très peu de contraintes et peut utiliser d'autres bibliothèques si nécessaire.

Ce module est construit en utilisant la commande `neco-compile`. Pour cela, Neco prend en entrée un modèle qui peut être :

- soit fourni programmatiquement en tant que module Python avec la bibliothèque *SNAKES* [145] ;
- soit spécifié en utilisant le formalisme *ABCD* [144] qui est une algèbre de réseaux de Petri ;
- soit spécifié en utilisant le standard *PNML* [81].

Une fois le modèle chargé, certains types peuvent être inférés permettant par la suite de typer le code Python, cette étape est importante parce que le langage Cython peut produire du code `C++` optimisé à partir du code Python annoté [17] (principalement en exploitant ces informations de types). D'autres informations peuvent être extraites du modèle, comme par exemple les bornes de certaines places dans le cas de ABCD.

Cependant, parce qu'on autorise un grand degré d'expressivité, tout le code source Python ne peut pas être typé. Dans ce genre de cas Neco va recourir à l'interpréteur Python qui permet de toujours exécuter le code en sacrifiant de l'efficacité. Intuitivement, si le réseau contient des jetons noirs, des entiers, des booléens, des chaînes de caractères statiques, ou des collections Python

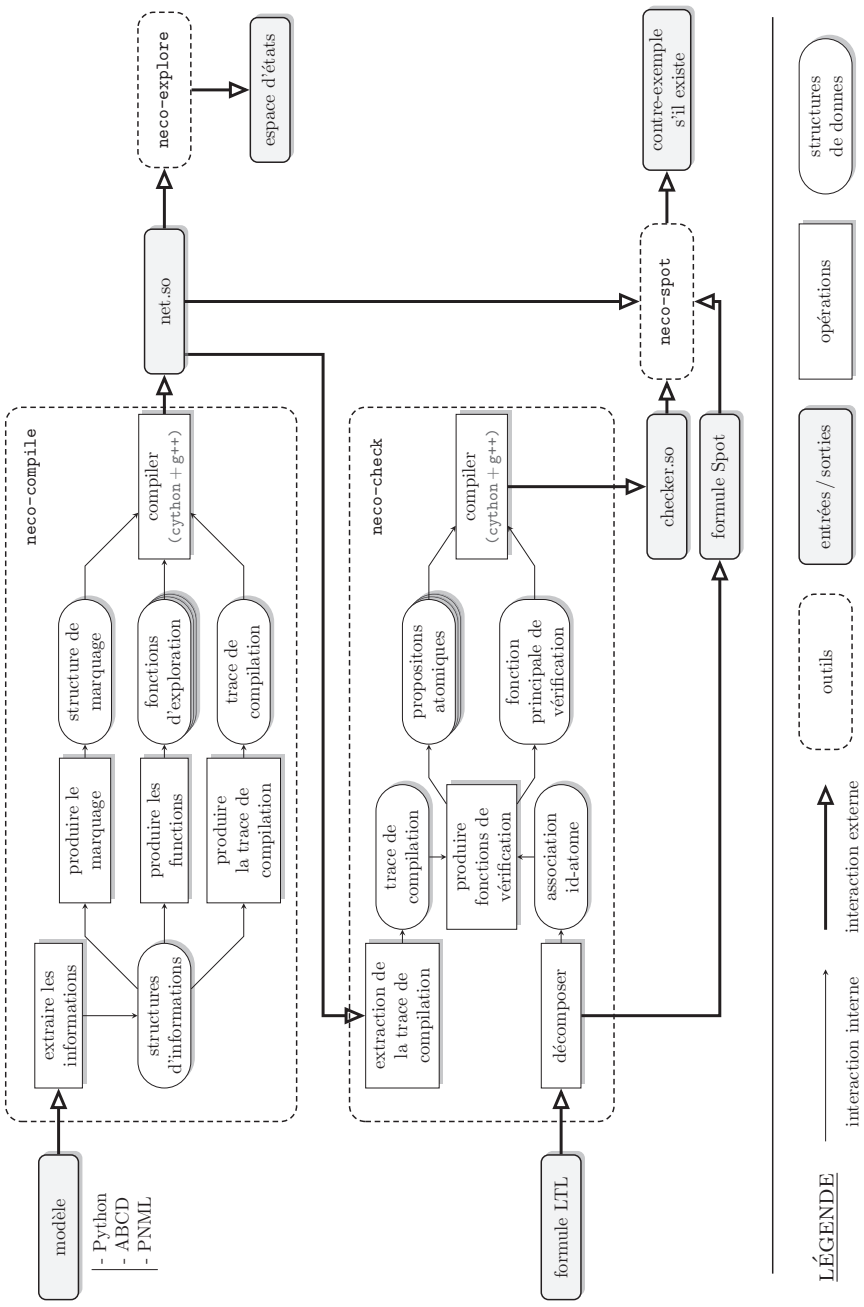


Figure 7.1 – *Flot de compilation est d'exploration des outils Neco (backend Cython).*

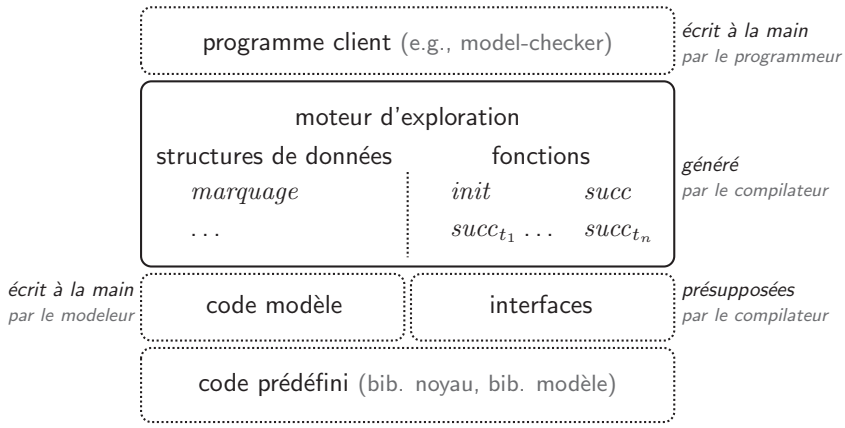


FIGURE 7.2 – *Le moteur d'exploration (boite ligne continue) et son contexte (boites lignes discontinues). [FP11a]*

classiques (tuples, listes, ensembles, dictionnaires) de ces types, alors on peut produire du code `C++` efficace. Mais la possibilité d'utiliser des types plus riches est aussi offerte ce qui permet de jouer sur un compromis entre l'efficacité et l'expressivité, permettant à une partie du réseau d'être efficace et une autre plus lente mais plus expressive. Cette possibilité de compromis est un point fort de Neco.

Cette expressivité a cependant un coût en temps de développement. Neco est réalisé de manière hybride avec plusieurs langages de programmation. Les outils sont réalisés en Python, les compilateurs génèrent du code Python ou Cython mais peuvent réutiliser des blocs préalablement écrits à la main. Ces blocs sont, selon le cas, réalisés soit en Python, soit en Cython, soit en `C++`. Le choix de Python pour le développement des outils résulte du constat que le compilateur n'a pas besoin d'être optimisé puisque la compilation a un coût négligeable par rapport à la construction d'espaces d'états en général.

L'étape suivante est de produire une structure de marquage pour représenter les états du réseau de Petri. Cette structure de données est optimisée grâce aux types précédemment découverts. Cela permet d'utiliser des types natifs ou encore d'utiliser des implémentations spécifiques à chaque place. Par la suite, on peut générer les fonctions d'explorations spécifiques au modèle (principalement une fonction qui renvoie le marquage initial et des fonctions successeurs) qui permettent de calculer les espaces d'états efficacement comme vu au chapitre 3.

Une étape supplémentaire est de produire une *trace de compilation* qui contient des informations à propos de la structure du marquage et du modèle. C'est un ensemble de métadonnées essentielles pour préserver la cohérence avec les autres outils, de plus cela évite à l'utilisateur d'appeler les outils de la chaîne avec tout le temps les mêmes options de qui est propice à l'erreur.

L'étape finale est de compiler le code généré et de produire un module Python natif qui est une bibliothèque logicielle. Dans notre cas cette bibliothèque peut être utilisée depuis le `C++` comme une bibliothèque classique ou depuis Python comme un module. Cette étape est réalisée avec le compilateur Cython et un compilateur `C++`.

Par la suite, l'outil `neco-explore` permet de réaliser l'exploration de l'espace d'états. Cet outil construit l'ensemble des états accessibles ou le graphe d'accessibilité en utilisant un simple algorithme d'exploration qui agrège les états découverts par appel des fonctions successeur. La réalisation de cet outil reste simple.

7.1.2 Model checking LTL

Le model checking LTL est réalisé en utilisant la bibliothèque Spot [52] qui est une bibliothèque générale d'algorithmes de vérification de formules LTL. De part sa généralité, cette bibliothèque n'est pas liée à un formalisme donné et donc ne s'adresse pas de base aux réseaux de Petri ou autre formalisme.

Par conséquent, Spot ne peut pas gérer directement les propositions atomiques apparaissant dans les formules LTL qui sont spécifiques au formalisme utilisé. De plus, puisque nos structures de marquages sont spécifiques aux modèles, on a aussi besoin de produire une manière de vérifier les propositions atomiques pour chaque modèle compilé. C'est le travail du second compilateur `neco-check`.

Cet outil a besoin de deux entrées, une formule LTL dans un format approprié pour Neco [Fro12b], et les métadonnées extraites d'un moteur d'exploration précédemment compilé avec `neco-compile`.

La première étape consiste en une décomposition de la formule : extraire les propositions atomiques et les associer à des identifiants uniques ("association id-atome" sur la figure 7.1). Une formule simplifiée où chaque proposition atomique a été remplacée par ces identifiants est stockée dans un fichier. De cette manière, on peut proposer une interface simple avec un module de vérification. Intuitivement, l'interface publique est une seule fonction `check` qui prend en argument un état du réseau de Petri et l'identifiant d'une proposition atomique, puis renvoie la valeur de vérité de cette proposition atomique dans cet état.

Les propositions atomiques prises en charge sont les comparaisons de multiensembles (tels que les marquages de places ou des multiensembles constants qui peuvent être définis avec du code Python arbitraire), les comparaisons d'expressions entières (opérations arithmétiques, bornes de places, constantes), les tests de tirabilité de transitions et les deadlocks. Ces propositions atomiques peuvent être liées pour former des formules grâce aux opérateurs temporels (A, E, G, F, X, U, W) et les opérateurs logiques ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\oplus$).

L'étape suivante est de générer une fonction check pour chaque proposition atomique en plus de la fonction check générale publique. Pendant cette étape, utiliser la trace de compilation est essentiel parce qu'on a besoin de générer des fonctions qui sont compatibles avec la structure de marquage optimisée, et donc il faut connaître son organisation en mémoire et les types utilisés. On aurait pu fournir des fonctions génériques mais dans ce cas il faudrait transformer la structure de marquage en une structure générique et on perdrait les avantages de la compilation. Finalement le code généré est compilé en utilisant le compilateur Cython et un compilateur `C++`.

Le module *checker.so* finalisé, il peut être utilisé avec le fichier de formule par `neco-spot` qui donnera une contre-exemple s'il en existe un, *i.e.*, si la formule n'est pas satisfaite.

7.2 Interface entre Neco et Spot

Dans cette section on explicite comment l'outil *Neco-Spot* a été réalisé en combinant le moteur d'exploration généré par Neco et la bibliothèque d'algorithmes de model checking Spot [52].

Spot manipule des automates de Büchi généralisés avec transitions d'acceptation (TGBA). Les TGBA permettent une représentation plus compacte des propriétés LTL [51], et peuvent être utilisés de manière efficace pour réaliser les tests de vacuité [44]. Le TGBA est aussi une classe abstraite, `tgba`, en `C++` dans la bibliothèque Spot avec une interface permettant l'exploration à la volée. Les structures de Kripke sont vues comme des sous-classes de `tgba` sans ensembles d'acceptation.

L'approche théorique spécifique aux automates est implémentée dans l'outil `neco-spot` par :

1. Une encapsulation de `net.so` et `checker.so` qui présente le graphe d'accessibilité du modèle comme une sous-classe de la classe `Kripke` de Spot. Cette interface peut être réduite à trois fonctions essentielles :
   - `get_init_state()` qui renvoie l'état initial du modèle ;
   - `succ_iter(s)` qui renvoie un itérateur sur les successeurs d'un état `s` ;
   - `state_condition(s)` qui renvoie la valeur de vérité des propositions atomiques.

   Notons que l'interface permet une exploration à la volée de l'espace d'états, et calcule les résultats de `succ_iter(s)` et `state_condition(s)` à la demande, c'est réalisé simplement en appelant les fonctions adéquates dans `net.so` et `checker.so`.
2. La formule LTL est simplifiée puis convertie en TGBA qui est à son tour simplifié. Toutes ces opérations sont des fonctions fournies par Spot [51].
3. Les deux automates ci-dessus sont alors synchronisés en utilisant la classe `tgba_product` de Spot (une autre sous classe de `tgba`). Ce produit synchronisé est construit en temps constant et les calculs sont différés au besoin (durant l'exploration).
4. Le produit synchronisé est utilisé pour le test de vacuité en utilisant, au choix, un des algorithmes proposés par Spot [52]. Durant ce test de vacuité on va déclencher le calcul à la volée du produit synchronisé, qui va à son tour construire la partie du graphe d'accessibilité qui doit être explorée.
5. Si le produit est vide, un contre exemple est calculé et affiché.

La partie la plus importante du travail d'interfaçage entre Neco et Spot consiste donc surtout à implémenter l'interface pour la classe `Kripke` de Spot ; le reste est juste une suite d'appels à divers algorithmes fournis par Spot.

7.3 Comparaisons et études de cas

Les comparaisons et les études de cas présentés dans cette section servent à évaluer les performances des approches présentées dans les chapitres précédents.

On utilise les backends Python et Cython selon les cas. Tout d'abord on présente les modèles traités, puis on montre les résultats avant de les analyser.

On se concentre sur les performances obtenues par l'approche par compilation avec diverses optimisations introduites, mais aussi les réductions par symétries. On va considérer les modèles présentées dans [FP11a] mais aussi certains modèles participant au Model Checking Contest 2012 [KLB+12] et 2013 [KLB+13]. Des modèles additionnels ont été introduits pour tester les réductions par symétries et les réductions de contrôle de flot.

Les cas présentés ont été choisis de manière à éliminer les temps d'exécutions trop petits : la majorité des exécutions en dessous de 1s a été retirée. De même pour les instances trop grandes : un *time out* de 25 minutes de calcul a été mis en place. Ce *time out* affecte les temps de compilation et les temps d'exploration de manière distincte, ainsi dans le pire des cas un exemple pourra s'exécuter en au plus 50 minutes. Évidemment, si la compilation a dépassé le délai de 25 minutes, l'exploration n'est pas exécutée.

On essaye de se positionner d'un point de vue similaire à celui présenté dans [90]. En effet le but ici n'est pas de prouver que l'outil présenté est meilleur que les autres mais plutôt montrer que les approches employées sont pertinentes et efficaces. Pour une comparaison avec d'autres outils on peut se référer à [FP11a, KLB+12, KLB+13].

Un autre point de comparaison est donné en annexe C. Il s'agit de comparer les performances du backend Python en utilisant la compilation à la volée. Pour cela on utilise *PyPy* [149], une implémentation alternative du langage Python. PyPy est un interpréteur qui compile à la volée le code source Python en utilisant des techniques *just-in-time* pour de meilleures performances. Ce point de comparaison sert essentiellement à vérifier que le travail investi dans le backend Cython est pertinent. Le constat est indéniable, le backend reste beaucoup plus rapide mais PyPy permet d'obtenir de meilleures performances que Python.

7.3.1 Statistiques

L'ensemble de tests est composé de 18 modèles. La majorité des modèles est paramétrable ce qui permet de varier la taille de l'espace d'états associé. L'ensemble de tous les paramétrages choisis donne 97 différents modèles. Chaque cas peut supporter différentes configurations parmi les suivantes :

`SIMPLE`: sans optimisations

`OPT`: optimisations générales (ordonnancements des transitions, utilisation de types spécifiques pour places) ;

`PACK`: optimisations avec *packing* (opérations de bas niveau pour Cython nécessitant un encodage au bit près, il s'agit essentiellement des optimisations du chapitre 3 qui jouent sur l'encodage minimal en bits des places) ;

`FLOW`: optimisation de contrôle de flot ;

`SYM`: réductions par symétries ;

`SYM_NO`: réductions par symétries optimistes.

Certaines optimisations sont dépendantes entre elles, ainsi :

- `PACK` inclue `OPT` ;
- `FLOW` inclue `PACK` et donc `OPT` ;
- `SYM` et `SYM_NO` incluent `PACK` et donc `OPT` ;

Remarquons que la configuration `PACK` ne diffère pas de `OPT` en Python, en effet il s'agit d'opérations bas niveau qui ne sont accessibles qu'en Cython. De plus, puisque les réductions par symétries ne sont disponibles qu'en Python actuellement inclure `PACK` revient à inclure `OPT` pour ces réductions. Les 97 cas supportent les configurations suivantes

- 97 supportent les configurations `SIMPLE`, `OPT`, `PACK` ;
- 20 supportent la configuration `FLOW` ;
- 40 supportent les réductions par symétries `SYM` et `SYM_NO`.

Pour ces configurations, 82 cas supportent l'exécution en Python et 57 cas supportent l'exécution en Cython. Le tout totalisant 531 exécutions. Pour une meilleure fiabilité des résultats chaque cas a été exécuté 5 fois ce qui nous donne un total de 2655 exécutions réalisés en 106 heures et 44 minutes de calcul.

Des statistiques supplémentaires sur les tailles des fichiers produits sont aussi présentés en annexe C. On remarque essentiellement que les optimisations ont pour effet de produire des fichiers plus petits, notamment l'optimisation de réduction de contrôle de flot qui a un effet drastique en Cython.

La *machine utilisée* est un noeud d'une grappe de calculs. Ce noeud possède *8Go* de mémoire, un processeur *quad-core* (virtualisé) cadencé à *2.67 Ghz*. Cependant Neco ne tire pas avantage d'architectures parallèles, *un seul core est donc utilisé*, le reste est laissé aux autres processus et au système d'exploitation. Ce noeud a été isolé pour des *performances homogènes* et est équipé d'un système *GNU/Linux Ubuntu 12.1* en version *32bits*, par conséquent seulement *4Go* de mémoire peuvent être utilisés.

7.3.2 Notations pour création dynamique de processus

Pour décrire nos exemples nous allons alléger les notations. Lorsqu'un réseau de Petri avec création dynamique de processus est présenté graphiquement, on omet la place générateur et les arcs qui y sont connectés. À la place, on utilise la notation $\nu(\pi)$ pour la création d'un fils du processus π et la notation $\chi(\pi)$ pour la terminaison du processus π. Un exemple est donné en figure 7.3. Remarquons que lorsque le processus π crée un fils à la transition t, son successeur dans la place générateur s_η est mis à jour et le jeton $\langle \pi \cdot \langle i \rangle, \pi \cdot \langle i \rangle \cdot \langle 1 \rangle \rangle$ est produit car $\pi \cdot \langle i \rangle \cdot \langle 1 \rangle$ est le premier fils que $\pi \cdot \langle i \rangle$ peut créer.

7.4 Modèles

Les philosophes (*Philosophers*) Le modèle des philosophes est commun dans la littérature depuis longtemps [123] et est souvent utilisé en model checking en tant que modèle académique. On fait varier la taille du modèle en augmentant le nombre de philosophes dans le système. Un exemple de système avec deux philosophes est donné en figure 7.4. On remarquera qu'on assimile l'état de réflexion du philosophe à l'action de ne pas manger. Ce modèle est 1-borné c'est-à-dire que dans chaque état accessible, une place ne peut contenir qu'au plus un seul jeton.

Un modèle de passage à niveau (*Railroad*) Ce modèle généralise le modèle présenté dans [144] à un nombre arbitraire de voies. Il comprend un

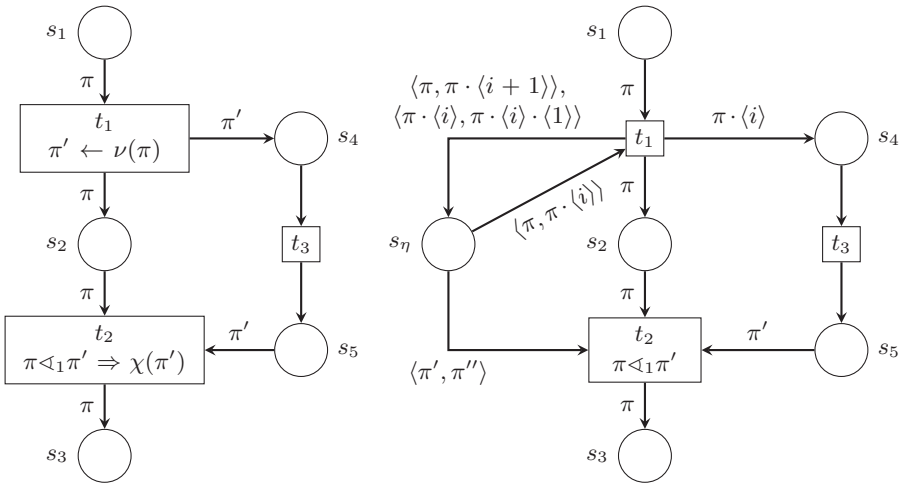


Figure 7.3 – *Exemple de réseau de Petri avec création dynamique de processus : à gauche avec notations simplifiés, à droite avec notations explicites.*

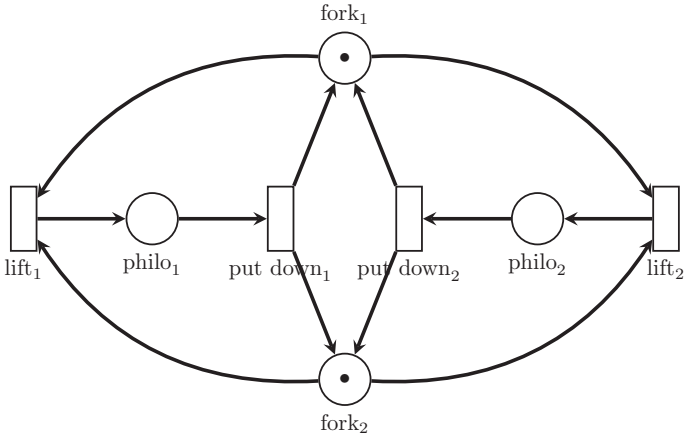


Figure 7.4 – *Un dîner de philosophes avec 2 philosophes.*

passage à niveau, un ensemble de voies équipées de feux de signalisation et un contrôleur pour compter les trains et commander le passage. Pour n voies, ce réseau de Petri comprend :

- $5n + 11$ places de type jeton noir qui sont 1-bornées (places de contrôle de flot ou drapeaux) ;
- 2 places de type entier 1-bornées (compteur de trains et état du passage) ;
- 3 places de type entier (les feux sur les voies) ;
- 1 place n-bornée de type jeton-noir (pour les signaux des voies vers le contrôleur).

Un protocole de sécurité (*Needham Schroeder*) Ce modèle est un cas du protocole cryptographique à clé publique Needham-Schroeder provenant de [64]. Il embarque environ 350 lignes de Python pour implémenter l'algorithme d'apprentissage d'un attaquant Dolev-Yao. Le modèle comprend 17 places :

- 11 places sont de type jeton-noir et sont 1-bornées pour implémenter le contrôle de flot des agents ;
- 6 places sont colorées par le type `object` (*i.e.*, toute donnée) et sont 1-bornées pour stocker les connaissances des agents ;
- 1 place est colorée par le type `object` et n'est pas bornée pour stocker les connaissances de l'attaquant.

Les jetons colorés dans ce modèle sont des tuples d'objets Python arbitraires. Une telle structure est typique pour les modèles de protocoles cryptographiques comme ceux considérés dans [64].

Sur ce type de modèle on voit clairement l'utilité de notre niveau d'expressivité. La réalisation d'un algorithmes d'apprentissage au niveau du modèle est difficile, et beaucoup plus aisée dans un langage de haut niveau tel que Python. Une version de ce protocole a été modélisé pour Helena [28], lors d'une comparaison des temps d'exécution d'une exploration exhaustive entre SNAKES et Helena, on observe des temps équivalents. C'est à cause d'une part de l'exploration des espaces d'états qui est beaucoup plus rapide avec Helena mais qui nécessite des temps de compilation longs, SNAKES quant à lui ne compile pas. D'autre part, Helena prend en charge un nombre limité de structures de données. Ici, les connaissances de l'attaquant et les algorithmes d'apprentissage ont utilisé des listes chaînées et des tableaux, donc des structures séquentielles, ce qui a entraîné de mauvaises complexités. En Python, on a utilisé des dictionnaires et des ensembles, donc des tables de hachage munies d'algorithmes efficaces pour des opérations ensemblistes. La complexité est donc bien moindre et globalement plus optimisé.

Dans le cas de Neco on observe que les temps de compilation sont bons essentiellement parce que les annotations sont des expressions Python qui peuvent être directement embarqués dans le code généré, tandis que Helena a besoin de traduire les annotations du réseau de Petri en code `C`.

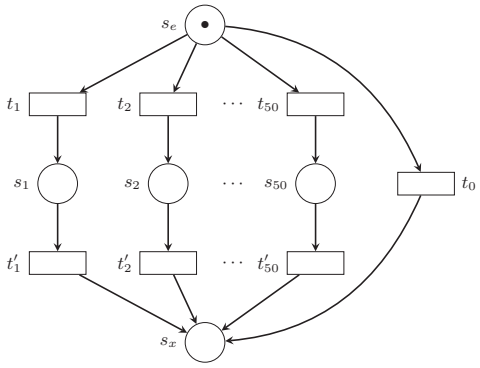
Un premier modèle pour le contrôle de flot (*Linear A*) Pour observer l'impact du contrôle de flot, un modèle spécifique a été réalisé. Il est formé de n processus concurrents instanciés par duplication de sous-réseau. Chacun de ces processus est formé de 51 transitions exécutées en séquence. Le but de ce cas

est de regarder l'impact de la réduction sur des processus possédant un grand nombre de place de contrôle de flot. Un tel processus est donné en figure 7.5b.

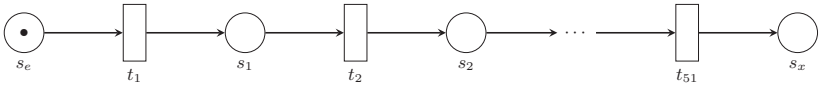
Un second modèle pour le contrôle de flot (*Choice*) Il est important d'essayer de trouver le pire cas d'application pour notre optimisation. Pour cela on a réalisé un modèle similaire au précédent avec n processus concurrents instanciés par duplication de sous-réseau, mais cette fois chaque processus réalise un choix non déterministe parmi les 51 transitions, de cette manière on est obligé de tirer toutes les transitions à partir du même état malgré notre optimisation. Au total un processus contient 101 transitions, un exemple est donné en figure 7.5a.

Un troisième modèle pour le contrôle de flot (*Linear B*) Un second pire cas a été réalisé pour tester l'algorithme en situation avec un plus grand nombre n de processus. Chacun de ces processus possède une séquence de deux transitions qui ne sont tirables qu'une seule fois. Un tel processus est donné en figure 7.5c.

(a) *Un des processus du modèle Choice.*



(b) *Un des processus du modèle Linear A.*



(c) *Un des processus du modèle Linear B.*

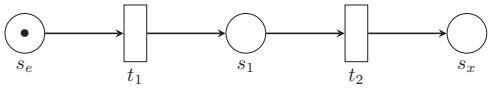


Figure 7.5 – *Réseaux des processus apparaissant dans les modèles* Choice*,* Linear A *et* Linear B*.*

Un système client-serveur (*Client Server*) Ce modèle est un système client-serveur où plusieurs clients communiquent avec un serveur, par exemple

des clients web requêtant une page d'un serveur web. Le réseau de Petri de ce modèle est présenté en figure 7.6 et été coupé en deux parties :

– la partie client où différents processus envoient des requêtes ;
– la partie serveur où un serveur qui gère ces requêtes.

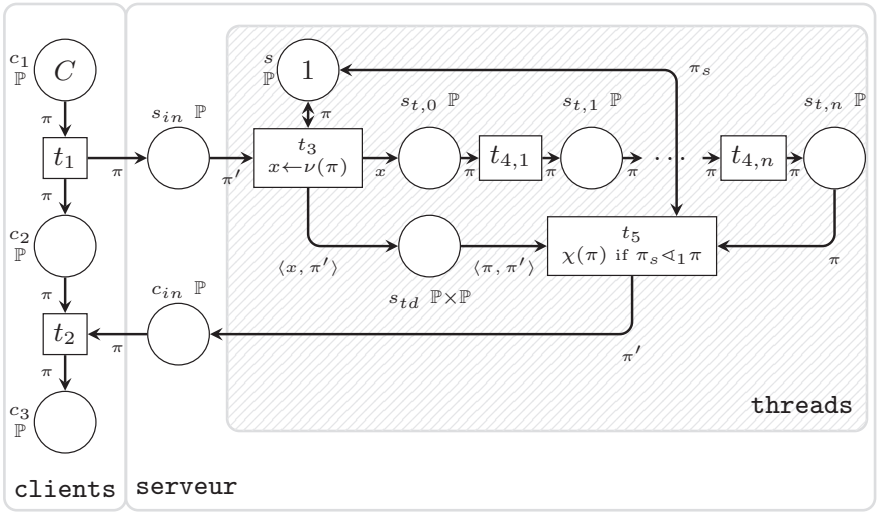


FIGURE 7.6 – *Le modèle client-serveur.*

Dans ce modèle, chaque client est un processus qui envoie une requête au serveur en déposant un jeton dans la place s_{in}. Les requêtes sont présentés comme les pids des clients, de cette manière on peut facilement identifier le client faisant la requête. Une fois la requête envoyée, les clients attendent un jeton dans la place c_{in} correspondant à la réponse du serveur, de plus ce jeton doit être égal au pid du client. Utiliser les identifiants des processus pour marquer l'appartenance de jetons permet de marquer les jetons qui appartiennent à chaque processus. Cela revient à utiliser les pids comme adresse du client.

Le serveur pioche les requêtes dans la place s_{in}, pour chacune d'elles, il crée un thread qui va gérer la requête et place l'identifiant du client qui la soumise dans l'espace local au thread. Cet espace local est modélisé avec la place s_{td} dont les jetons sont des paires d'identifiants de threads et d'identifiants de clients, en utilisant notre politique de modélisation, le thread est propriétaire du jeton. Le reste des opérations est réalisé par le nouveau thread créé. Le traitement de la requête est modélisé par des transitions $t_{4,0}, \ldots, t_{4,n}$ sur la figure 7.6, n est le paramètre qu'on va varier en plus du nombre de clients, plus il est grand plus l'espace d'états est grand. Finalement le thread envoie la réponse au client en utilisant la transition t_5 et termine. De plus, si on a plusieurs serveurs (plusieurs jetons dans la place s), on peut tester si le serveur est le parent du thread en utilisant la relation parent dans la garde de la transition t_5.

Les graphes de contrôle de flot (*CFG*) L'exemple présenté dans cette section montre que l'approche présentée dans le chapitre 6 peut être appliqué

à divers domaines. Par exemple les *graphes de contrôle de flot* (CFGs) [15] qui peuvent être représentés avec des réseaux de Petri en utilisant différents pids pour identifier des threads concurrents. Le but est de différencier les threads et pas seulement la présence de deux threads dans deux blocs de base, sinon on pourrait utiliser un simple réseau place/transition. Les CFGs qu'on utilise ont des blocs de base spéciaux appelées barrières qui sont des primitives de synchronisation. Lorsqu'un thread rencontre une barrière il attend que tous les autres threads atteignent cette barrière avant de continuer. C'est utilisé sur des architectures comme les processeurs graphiques (GPU) où les exécutions qui divergent produisent de mauvaises performances, ainsi le placement de barrières pour synchroniser des threads est meilleur que les laisser diverger.

Le modèle présenté ici a un nombre de threads fixe et ne crée pas de nouveaux threads à l'exécution. Donc les pid-trees résultants ne vont pas grandir en hauteur et avoir une largeur fixe. Le nombre de threads est donc directement lié au nombre de permutations dont on a besoin par état, notre paramètre de mise à échelle sera donc le nombre de threads. De plus le modèle n'incorpore pas de données et donc le filtrage par ordonnancement est réduit à un ordre statique sur les places marqués qui est peu discriminant comme critère.

Le réseau de Petri qu'on considère est une composition séquentielle des sous réseaux présentés en figure 7.7. On crée n instances de ce sous réseau en variant i de 1 à n et joignant les places $s_{i,7}$ et $s_{(i+1),1}$. Il faut mettre en avant que chaque transition $t_{i,8}$ est une barrière, et donc ne peut être franchie que si tous les threads sont dans la place $s_{i,6}$. Le marquage initial du système assigne un multiensemble de n identifiants de processus $\{1{\cdot}1, \ldots, 1{\cdot}n, \}$ à la place $s_{1,1}$.

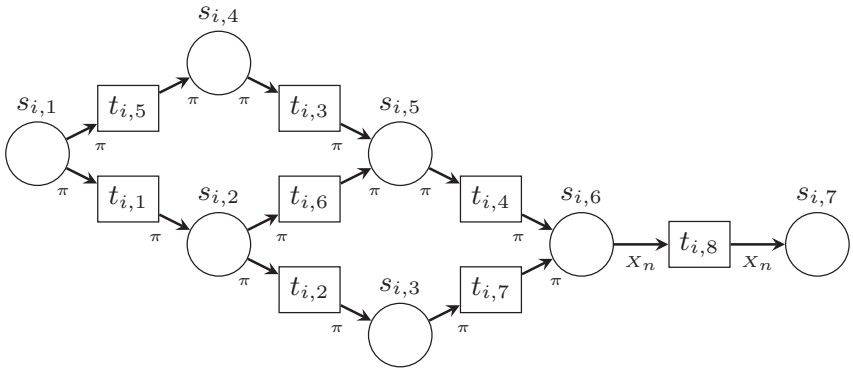


Figure 7.7 – *Sous-réseau utilisé pour le cas de contrôle de flot, où* $X_n \stackrel{\text{df}}{=} \{\pi_1, \ldots, \pi_n\}$.

Les autres modèles Les autres modèles considérés sont tirés de l'évènement Model Checking Contest, satellite des conférences Petri Nets 2012 et 2013. Les modèles sont : CSRepetitions, Dekker, Kanban, Lamport-FastMutEx, Mapk, NeoElection, RessAllocation, RwMutex, SharedMemory, SimpleLoadBal et TokenRing. Leur description peut être trouvée dans [KLB+13]. Il s'agit de réseaux de Petri place-transition. On ne présente pas les résultats pour ces modèles en Python ici, car les résultats avec et sans optimisations sont identiques (la table de résultats correspondante peut cependant être trouvée en annexe C.1). En effet, dans le backend Python, les

optimisations n'ont pas d'effet sur ce type de réseaux car l'implémentation utilisée par défaut est la même que l'implémentation optimisée.

7.4.1 Bilan du model checking contest

Lors du Model Checking Contest 2012, Neco ne réalisait que de l'exploration d'espace d'états. Pour l'édition 2013, il a participé à l'épreuve de vérification de propriétés logiques LTL en plus.

Pour cette épreuve Neco était le seul model checker LTL explicite mais aussi le seul model checker LTL à fournir un résultat dans un temps raisonnable. Les autres outils n'ont pas donné de résultats, ce qui a valu à Neco la première place pour le model checking LTL.

L'épreuve de construction d'espaces d'états a aussi montré que Neco pouvait être compétitif. En effet, le score obtenu était de moitié celui des meilleurs outils qui sont des outils symboliques. Cependant il n'était pas le dernier du concours. Ce qui montre que même si Neco est un outil relativement jeune, il peut être compétitif avec les outils du domaine.

Au final, l'offre d'outils est assez pauvre, malgré une pléthore apparente, et Neco tient une place non négligeable avec un créneau large. Le Model Checking Contest montre la très grande difficulté à comparer les outils. Une des difficulté est le format d'entrée des outils dès qu'on sort du monde des réseaux place/transition. C'est d'ailleurs la raison pour laquelle Neco n'a participé que sur les modèles bas niveau à l'édition 2013 de la compétition. Une seconde difficulté était la vérification de formules logiques, en effet lorsque les outils sont en désaccord, on ne possède aucune implémentation de référence pour déterminer quel outil a raison. C'est directement lié à la problématique de certification des model checkers pour assurer de leur correction. Finalement, une comparaison sérieuse entre les outils est actuellement difficile et correspond à un grand effort en temps et de modélisation.

7.5 Résultats expérimentaux

7.5.1 Résultats pour les optimisations et les réductions de contrôle de flot

Les résultats pour le backend Python La première série de résultats pour les optimisations et réductions de contrôle de flot est présentée en table 7.1, les accélérations ne prennent pas en compte les temps de compilation. Les modèles de cette table sont des réseaux colorés. L'utilisation des optimisations présentées dans le chapitre 3 produit une accélération jusqu'à $\times 12$ en temps d'exploration tout en réduisant les temps de compilation (la bibliothèque produite est plus simple). On observe aussi une faible accélération sur le modèle Needham Schroeder, ceci est essentiellement à dû à l'utilisation de jetons très riches (objets Python) pour représenter l'espion et son algorithme d'apprentissage. Ce code Python ne peut pas être optimisé par notre approche, il s'agit d'une problématique de compilation de Python et non compilation de réseaux de Petri.

Les réductions de contrôle de flot ont un effet drastique sur certains modèles, en effet on peut voir une accélération de ×200 sur le modèle Choice 2, il s'agit pourtant de notre modèle "pire cas" cependant après analyse on se rend compte que la décomposition du contrôle de flot élimine les le test de présence de jeton dans l'unique place en entrée de toutes les transitions, d'où cette accélération. L'optimisation est donc très efficace sur le cas qu'on prévoyait le pire, et qui au final est le meilleur. On remarque que plus le nombre de processus concurrents augmente plus cette accélération est importante par comparaison avec `OPT`. Une accélération est aussi observée pour notre second "pire cas" Choice Bis et notre "meilleur cas" Linear, certes moins impressionnante dans le cas Choice Bis mais non négligeable pour autant. L'approche se comporte donc bien en général, et on n'a pas réussi à cerner la situation lorsque l'optimisation est néfaste, on observe cependant une perte de performances pour le modèle Needham Schroeder. Après analyse par profiling sur ce modèle, on remarque qu'il s'agit d'un problème de fonction hachage, qui se comporte moins bien sur la version avec réduction de contrôle de flot de ce modèle.

Les résultats pour le backend Cython Pour ce backend les résultats pour les modèles haut niveau sont présentés sur la table 7.2. On remarque une accélération avec utilisation d'optimisations (`OPT`) qui est encore plus importante lorsqu'on utilise des encodages bas niveau (`PACK`). On peut voir que pour l'ensemble des modèles l'utilisation d'optimisations de bas niveau est essentielle pour bénéficier des meilleures performances.

Lors de l'utilisation des réductions de contrôle de flot, on observe aussi une accélération, moins marquante que dans le cas du backend Python car les opérations de base sont moins coûteuses mais non négligeables pour autant. Le plus important étant l'absence de décélération lors du calcul d'espaces d'états, ce qui indique que l'optimisation peut toujours être activée sans entraver les performances.

Pour les réseaux de bas niveau, tables 7.3 et 7.4, on remarque un impact nul des optimisations `OPT`. En effet dans ces réseaux toutes les places sont bas niveau, il est donc difficile de définir une stratégie efficace d'encodage et d'exploration sans travailler avec des opérations de bas niveau. Les performances avec la configuration `PACK` sont meilleures en général. Cependant sur certains modèles comme Mapk par exemple, les résultats avec optimisations sont similaires aux performances sans optimisations : c'est essentiellement dû au fait que les optimisations ne s'appliquent pas sur ce modèle.

On peut remarquer que lorsqu'on utilise des langages comme Cython et `C++` les temps de compilation sont plus importants qu'avec Python. Dans certains cas comme par exemple RwMutex w0010w0500 on a un facteur de ×1654 entre en temps de compilation et le temps d'exploration. On a un temps d'exploration très petit et un temps de compilation très long. On peut donc voir que le surcoût de compilation peut être importants si le modèle a un petit espace d'états. La compilation du même modèle en Python (présenté en annexe C.1) prend 106s et son exploration 12s ce qui est plus efficace alors que Python est un langage qui s'exécute beaucoup plus lentement que Cython et `C++`.

Python	Temps comilation (s)			Temps exploration (s)			Accélération			Nombre d'états
	Simple	Opt	Flow	Simple	Opt	Flow	Opt par rapport à Simple	Flow par rapport à Simple	Flow par rapport à Opt	
Choice 2	1.744	1.355	0.245	9.029	0.857	0.045	x10.534	x199.736	x18.961	2,704
Choice 3	3.856	2.849	0.383	mem	99.214	4.158	-	-	x23.861	140,608
Linear A 2	0.885	0.686	0.147	4.098	0.409	0.028	x10.029	x144.051	x14.363	2,704
Linear A 3	1.957	1.425	0.181	mem	48.627	2.424	-	-	x20.058	140,608
Linear A 4	3.480	2.547	0.299	mem	time	199.632	-	-	-	7,311,616
Linear B 10	0.081	0.074	0.044	120.176	9.492	3.728	x12.661	x32.240	x2.546	59,049
Linear B 11	0.092	0.083	0.048	434.056	34.519	12.900	x12.574	x33.649	x2.676	177147
Linear B 12	0.104	0.094	0.053	time	125.313	45.567	-	-	x2.750	531,441
Linear B 13	0.117	0.105	0.058	time	437.960	159.449	-	-	x2.747	1594323
Railroad 5	0.167	0.159	0.121	3.157	0.659	0.521	x4.788	x6.062	x1.266	1,838
Railroad 6	0.164	0.153	0.105	17.315	3.421	2.651	x5.061	x6.531	x1.290	7,502
Railroad 7	0.191	0.176	0.119	91.398	17.406	13.067	x5.251	x6.995	x1.332	30,626
Need.-Sch.	0.049	0.048	0.039	4.187	3.522	4.594	x1.189	x0.911	x0.767	1,234

Table 7.1 – Résultats d'optimisations pour réseaux haut niveau avec Python.

Cython	Temps comilation (s)				Temps exploration (s)				Accélération				Nombre d'états
	simple	opt	pack	flow	simple	opt	pack	flow	opt par rapport à simple	pack par rapport à simple	flow par rapport à simple	flow par rapport à pack	
Choice 2	21.777	21.747	8.176	5.313	0.024	0.023	0.017	0.004	x1.010	x1.363	x5.847	x4.290	2 704
Choice 3	44.302	44.287	14.464	7.698	2.364	2.390	1.737	0.305	x0.989	x1.361	x7.760	x5.702	140 608
Choice 4	74.671	74.694	21.554	10.544	215.869	218.138	144.115	31.719	x0.990	x1.498	x6.806	x4.543	7 311 616
Linear a 2	13.276	13.227	5.210	3.821	0.010	0.010	0.006	0.003	x0.994	x1.669	x4.126	x2.472	2 704
Linear a 3	26.459	26.378	8.654	5.332	0.932	0.919	0.606	0.178	x1.014	x1.538	x5.238	x3.407	140 608
Linear a 4	44.058	44.225	12.688	7.017	87.927	88.351	55.502	17.216	x0.995	x1.584	x5.107	x3.224	7 311 616
Linear b 11	2.884	2.886	2.068	2.071	0.657	0.659	0.519	0.495	x0.998	x1.267	x1.327	x1.047	177 147
Linear b 12	3.078	3.078	2.134	2.133	2.592	2.549	1.962	1.802	x1.017	x1.321	x1.438	x1.089	531 441
Linear b 13	3.298	3.291	2.234	2.269	9.310	9.319	7.293	6.950	x0.999	x1.276	x1.340	x1.049	1 594 323
Linear b 14	3.518	3.511	2.337	2.341	33.087	33.224	25.932	24.314	x0.996	x1.276	x1.361	x1.067	4 782 969
Need.-Sch.	2.686	2.702	2.519	2.501	2.880	2.847	2.820	2.815	x1.012	x1.021	x1.023	x1.002	1 234
Railroad 5	4.038	3.883	2.836	2.834	0.032	0.006	0.005	0.005	x5.071	x6.140	x6.535	x1.064	1 838
Railroad 6	4.429	4.261	3.032	2.996	0.158	0.034	0.029	0.027	x4.617	x5.410	x5.959	x1.101	7 502
Railroad 7	4.862	4.739	3.218	3.138	0.758	0.171	0.138	0.130	x4.446	x5.503	x5.820	x1.057	30 626
Railroad 8	5.362	5.208	3.374	3.333	3.417	0.858	0.675	0.635	x3.983	x5.065	x5.380	x1.062	124 562
Railroad 9	5.859	5.677	3.580	3.490	16.768	4.608	3.650	3.485	x3.639	x4.594	x4.812	x1.047	504 662
Railroad 10	6.452	6.247	3.782	3.645	70.758	22.441	17.564	17.041	x3.153	x4.029	x4.152	x1.031	2 038 166
Railroad 11	6.989	6.838	4.003	3.901	340.479	107.053	83.565	79.848	x3.180	x4.074	x4.264	x1.047	8 211 530

Table 7.2 – Résultats d'optimisations pour réseaux haut niveau avec Cython.

Cython	Temps comilation (s)			Temps exploration (s)			Accélération		Nombre d'états		
	SIMPLE	OPT	PACK	SIMPLE	OPT	PACK	OPT par rapport à SIMPLE	PACK par rapport à SIMPLE	SIMPLE	OPT	PACK
CSRepetitions PT 02	2.674	2.670	2.093	0.004	0.004	0.003	x0.994	x1.113	1,872	1,872	1,872
CSRepetitions PT 03	7.245	7.210	3.918	25.935	25.959	20.057	x0.999	x1.293	3,951,655	3,951,655	3,951,655
Dekker PT 010	9.156	9.131	5.465	0.080	0.080	0.052	x1.004	x1.532	6,144	6,144	6,144
Dekker PT 015	22.337	22.344	11.214	9.869	9.834	6.020	x1.004	x1.639	278,528	278,528	278,528
Dekker PT 020	44.814	44.639	20.066	923.181	924.233	519.840	x0.999	x1.776	11,534,336	11,534,336	11,534,336
Kanban PT 0005	2.077	2.076	2.077	14.169	14.233	14.155	x0.995	x1.001	2,546,432	2,546,432	2,546,432
LamportFastMutEx PT 2	9.309	9.289	4.660	0.001	0.001	0.001	x0.990	x1.102	380	380	380
LamportFastMutEx PT 3	17.749	17.722	7.407	0.110	0.109	0.086	x1.006	x1.280	19,742	19,742	19,742
LamportFastMutEx PT 4	31.608	31.631	11.571	24.455	24.391	17.443	x1.003	x1.402	1,914,784	1,914,784	1,914,784
MAPK PT 008	2.714	2.710	2.709	48.238	48.322	48.179	x0.998	x1.001	6,110,643	6,110,643	6,110,643
NeoElection PT 2	168.566	168.066	38.355	0.012	0.012	0.004	x1.030	x3.240	241	241	241
NeoElection PT 3	time	time	194.130	time	time	81.964	-	-	time	time	974,325
Philosophers 20	3.934	3.924	2.545	0.080	0.080	0.060	x1.006	x1.330	15,127	15,127	15,127
Philosophers 25	5.013	4.990	2.998	1.260	1.248	0.885	x1.010	x1.423	167,761	167,761	167,761
Philosophers 30	6.274	6.212	3.378	22.214	22.148	16.156	x1.003	x1.375	1,860,498	1,860,498	1,860,498
Philosophers 35	7.626	7.603	3.805	342.110	342.749	246.940	x0.998	x1.385	20,633,239	20,633,239	20,633,239
RessAllocation PT R003C010	5.239	5.242	5.235	4.652	4.642	4.721	x1.002	x0.985	823,552	823,552	823,552
RessAllocation PT R005C002	2.131	2.132	2.124	0.000	0.000	0.000	x1.026	x1.019	112	112	112
RessAllocation PT R010C002	3.278	3.284	3.279	0.012	0.012	0.012	x0.999	x1.003	6,144	6,144	6,144
RessAllocation PT R015C002	4.791	4.782	4.768	0.854	0.852	0.857	x1.002	x0.997	278,528	278,528	278,528
RessAllocation PT R020C002	6.672	6.660	6.656	65.597	65.644	65.788	x0.999	x0.997	11,534,336	11,534,336	11,534,336

Table 7.3 – Résultats d'optimisations pour réseaux bas-niveau avec Cython (partie 1).

Cython	Temps comilation (s)			Temps exploration (s)			Accélération		Nombre d'états		
	SIMPLE	OPT	PACK	SIMPLE	OPT	PACK	OPT par rapport à SIMPLE	PACK par rapport à SIMPLE	SIMPLE	OPT	PACK
RwMutex PT r0010w0010	4.774	4.755	3.097	0.005	0.005	0.004	x0.999	x1.392	1,034	1,034	1,034
RwMutex PT r0010w0020	7.578	7.529	4.200	0.006	0.006	0.004	x0.998	x1.570	1,044	1,044	1,044
RwMutex PT r0010w0050	20.240	20.148	8.634	0.011	0.011	0.006	x0.998	x1.701	1,074	1,074	1,074
RwMutex PT r0010w0100	56.137	55.966	18.789	0.023	0.023	0.012	x1.004	x1.945	1,124	1,124	1,124
RwMutex PT r0010w0500	TIME	TIME	218.354	TIME	TIME	0.132	-	-	TIME	TIME	1,524
RwMutex PT r0020w0010	8.408	8.415	4.507	15.150	15.099	9.973	x1.003	x1.519	1,048,586	1,048,586	1,048,586
SharedMemory PT 000005	4.685	4.671	2.992	0.007	0.006	0.005	x1.013	x1.364	1,863	1,863	1,863
SharedMemory PT 000010	28.676	28.377	10.435	31.882	30.829	21.379	x1.034	x1.491	1,830,519	1,830,519	1,830,519
SimpleLoadBal PT 02	3.849	3.830	2.703	0.002	0.002	0.001	x1.012	x1.179	832	832	832
SimpleLoadBal PT 05	13.796	13.746	7.538	0.873	0.862	0.716	x1.013	x1.219	116,176	116,176	116,176
TokenRing PT 005	8.551	8.499	5.276	0.001	0.001	0.001	x1.012	x1.760	166	166	166
TokenRing PT 010	119.497	119.598	45.127	5.700	5.670	4.231	x1.005	x1.347	58,905	58,905	58,905

Table 7.4 – Résultats d'optimisations pour réseaux bas-niveau avec Cython (partie 2).

7.5.2 Résultats pour les réductions par symétries

Pour tester l'approche présentée dans le Chapitre 6, une implémentation a été réalisée dans Neco. Dans nos études de cas on se focalise sur les calculs en utilisant une approche par force brute et les deux réductions précédemment présentées : l'approche complète avec permutations de sous arbres, et l'approche optimiste sans permutations. À notre connaissance ce type de réductions n'est supporté par aucun outil existant, par conséquent aucune comparaison directe n'a été réalisée. Ceci est principalement dû au fait qu'il existe très peu de travaux sur la création dynamique de processus et encore moins d'outils, mais aussi à l'aggravation du problème d'explosion d'états causée par la création de nouveaux processus à l'exécution et donc l'absence de techniques de réduction efficaces. Cependant de nombreuses techniques existantes sont compatibles avec nos réductions et donc n'ont pas à la concurrencer. Par ailleurs, puisque l'implémentation actuelle de la technique a été réalisée en Python une comparaison ne serait pas équitable.

Remarquons que dans le cas des *CFG* on aurait pu utiliser des symétries classiques et se comparer à un outil existant. Il y a des chances que ce serait moins bon car les arbres se justifient que lorsqu'on a de la création dynamique et des relations entre pids, et comme ils ne sont pas gratuits, on devrait perdre par rapport à une autre approche plus simple.

L'implémentation utilisée est basée sur le module `itertools` de Python pour générer les permutations de pid-trees. À chaque fois qu'un itérateur est construit, toutes les permutations sont crées pour être itérées par la suite. Cette approche n'est pas optimale, puisqu'il serait plus efficace d'itérer sur les permutations à la volée et s'arrêter lorsqu'un marquage adéquat est trouvé. Nos expérimentations permettent donc d'évaluer les réductions qu'on peut espérer de notre approche et non ses performances.

Les résultats pour le modèle client-serveur Les résultats pour le modèle client serveur sont présentées en table 7.5. On a exécuté différentes configurations sur ce modèle en faisant varier le nombre de clients c et le nombre n d'opérations par thread. Ainsi, le nom d'une instance Client Server c3n4 signifie que le modèle contient 3 clients et 4 opérations par thread.

On observe une réduction importante des espaces d'états avec un facteur de réduction qui augmente avec la taille des instances, ce qui signifie qu'on a gagné un ordre de grandeur. On voit que pour la plus grande instance calculable sans réductions sur notre machine de tests (Client Server c5n3) on a une réduction de facteur $\times 1133$ et une réduction en temps de calcul d'un facteur $\times 12.69$. La réduction du temps de calcul résulte du calcul d'un espace d'états plus petit, on a moins d'états à traiter. L'évolution des temps de calcul pour le modèle avec 5 clients est montrée en figure 7.8a et l'évolution des tailles des espaces d'états en figure 7.8b.

Les modèles de contrôle de flots (*CFG*) Les résultats pour les graphes de contrôle de flot ont été séparés en deux parties pour différencier les deux paramètres de mise à échelle. En table 7.6 on présente les résultats pour 10 instances du sous réseau en faisant varier le nombre de threads qui exécutent le réseau résultant. La figure 7.9a présente les temps d'exploration pour les

Python	Temps comilation (s)				Temps exploration (s)				Accélération par rapport à		Nombre d'états			Réduction par rapport à OPT	
	SIMPLE	OPT	SYM	SYM NO	SIMPLE	OPT	SYM	SYM NO	SYM	SYM NO	SIMPLE	OPT	SYM	SYM	SYM NO
c3n4	0.037	0.037	0.047	0.047	1.488	1.391	0.928	1.433	x1.498	x0.970	2 494	2 494	165	x15	x6
c3n5	0.040	0.040	0.050	0.050	2.243	2.298	1.311	2.234	x1.752	x1.029	3 664	3 664	220	x17	x6
c4n1	0.029	0.029	0.037	0.037	5.069	4.664	1.067	1.602	x4.373	x2.911	6 336	6 336	126	x50	x21
c4n2	0.031	0.031	0.041	0.041	13.312	13.681	2.556	4.489	x5.352	x3.047	16 216	16 216	210	x77	x22
c4n3	0.035	0.035	0.044	0.044	30.006	31.302	5.147	10.420	x6.081	x3.004	34 304	34 304	330	x104	x22
c4n4	0.037	0.037	0.047	0.047	61.946	56.691	9.007	20.459	x6.294	x2.771	64 056	64 056	495	x129	x23
c4n5	0.040	0.040	0.051	0.051	111.301	101.230	15.097	37.217	x6.705	x2.720	109 504	109 504	715	x153	x23
c5n1	0.029	0.029	0.038	0.038	117.856	116.575	12.819	13.736	x9.094	x8.487	96 384	96 384	252	x382	x67
c5n2	0.033	0.033	0.041	0.041	430.284	382.754	38.475	52.194	x9.948	x7.333	336 184	336 184	462	x728	x72
c5n3	0.035	0.035	0.044	0.045	1 174.190	1 202.540	87.780	138.676	x13.699	x8.672	897 024	897 024	792	x1133	x78
c5n3	0.038	0.038	0.048	0.048	MEM	MEM	170.685	318.248	-	-	MEM	MEM	1 287	-	-
c5n5	0.041	0.041	0.051	0.052	MEM	MEM	301.734	640.295	-	-	MEM	MEM	2 002	-	-

Table 7.5 – Résultats des réductions par symétries pour le modèle *Client-Server* (Python).

(a) *Temps d'exploration.* (b) *Taille des espaces d'états.*

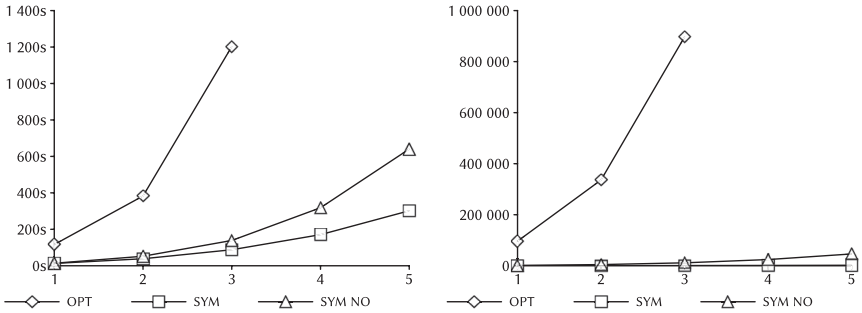


FIGURE 7.8 – *Résultats pour le modèle* CLIENT SERVER C5 *pour un nombre variable de clients (axe horizontal).*

différentes instances testées, on observe que les trois approches ont une croissance exponentielle. Cependant si on regarde le même graphique avec une échelle logarithmique (figure 7.9b) on observe que cette croissance est moindre pour les méthodes avec réductions par symétries : on est passés à un autre ordre de grandeur.

La réduction de la taille des espaces d'états est importante comme on peut le voir sur la figure 7.10a. On réduit les espaces d'états de manière efficace. Il est intéressant d'observer que les deux approches de réduction produisent des espaces d'états de même taille. En effet, ces réseaux de Petri appartiennent à une sous-classe pour laquelle l'approche optimiste est complète. En général, pour tout réseau ou les jetons contiennent au plus un identifiant de processus, l'approche optimiste sera complète. Ici aussi si on observe ces mêmes résultats que précédemment avec une échelle logarithmique (fig. 7.10b) : un gain exponentiel.

La seconde comparaison est réalisée sur le nombre d'instance du sous réseau en fixant le nombre de threads. Les résultats sont présentés dans la table 7.7. Ici, on peut remarquer qu'on obtient toujours la même réduction. Cela est dû à la structure du modèle, en effet on réplique un sous réseau dont le comportement n'est pas influencé par les autres sous réseaux : les threads ne peuvent pas diverger au delà d'une barrière.

7.6 Conclusion

Dans ce chapitre nous avons présenté la boite à outils Neco. Il s'agit d'un ensemble d'outils de compilation de réseaux de Petri et model checking LTL. Ils produisent des bibliothèques dynamiques performantes pouvant être utilisées depuis Python ou de manière native depuis le `C` ou `C++`.

Neco implémente les approches décrites dans ce manuscrit et a été utilisé pour réaliser des études de cas pour les valider. Ces études de cas ont montré l'effet non négligeable de ces approches pour la construction d'espaces d'états.

Python	Temps comilation (s)				Temps exploration (s)				Accélération par rapport à		Nombre d'états			Réduction par rapport	
	SIMPLE	OPT	SYM	SYM NO	SIMPLE	OPT	SYM	SYM NO	SYM	SYM NO	SIMPLE	OPT	SYM	SYM	SYM NO
CFG 10 1	0.411	0.410	0.496	0.465	0.042	0.042	0.125	0.067	x0.334	x0.625	61	61	61	x1	x1
CFG 10 2	0.413	0.412	0.461	0.467	0.473	0.438	0.860	0.542	x0.509	x0.809	361	361	211	x2	x2
CFG 10 3	0.408	0.418	0.465	0.467	3.989	3.986	4.332	2.997	x0.920	x1.330	2 161	2 161	561	x4	x4
CFG 10 4	0.415	0.414	0.464	0.465	31.964	32.251	15.758	11.702	x2.047	x2.756	12 961	12 961	1 261	x10	x10
CFG 10 5	0.411	0.411	0.489	0.467	247.419	245.853	49.642	34.943	x4.953	x7.036	77 761	77 761	2 521	x31	x31
CFG 10 6	0.420	0.412	0.467	0.466	MEM	MEM	134.174	88.992	-	-	MEM	MEM	4 621	-	-
CFG 10 7	0.414	0.419	0.466	0.467	MEM	MEM	396.388	204.150	-	-	MEM	MEM	7 921	-	-
CFG 10 8	0.412	0.413	0.466	0.462	MEM	MEM	TIME	500.157	-	-	MEM	MEM	TIME	12 871	-	-
CFG 10 9	0.415	0.413	0.464	0.465	MEM	MEM	TIME	999.558	-	-	MEM	MEM	TIME	20 021	-	-

Table 7.6 – Résultats des réductions par symétries pour le modèle *CFG* (backend : Python, variable : nombre threads).

Python	Temps comilation (s)				Temps exploration (s)				Accélération par rapport à OPT		Nombre d'états				Réduction par rapport	
	SIMPLE	OPT	SYM	SYM NO	SIMPLE	OPT	SYM	SYM NO	SYM	SYM NO	SIMPLE	OPT	SYM	SYM NO	SYM	SYM NO
CFG 1 5	0.023	0.023	0.029	0.029	4.412	4.405	1.010	0.684	x4.363	x6.438	7 777	7 777	253	253	x31	x31
CFG 2 5	0.044	0.044	0.056	0.055	12.983	13.013	2.948	2.041	x4.414	x6.376	15 553	15 553	505	505	x31	x31
CFG 3 5	0.070	0.075	0.086	0.086	25.899	25.928	5.720	3.950	x4.533	x6.564	23 329	23 329	757	757	x31	x31
CFG 4 5	0.101	0.102	0.123	0.122	43.647	43.491	9.128	6.481	x4.765	x6.711	31 105	31 105	1 009	1 009	x31	x31
CFG 5 5	0.138	0.138	0.166	0.166	65.189	65.491	13.744	9.801	x4.765	x6.682	38 881	38 881	1 261	1 261	x31	x31
CFG 6 5	0.190	0.191	0.223	0.223	91.225	91.076	19.046	13.512	x4.782	x6.740	46 657	46 657	1 513	1 513	x31	x31
CFG 7 5	0.238	0.238	0.273	0.274	122.981	121.993	25.137	18.182	x4.853	x6.710	54 433	54 433	1 765	1 765	x31	x31
CFG 8 5	0.290	0.292	0.332	0.332	155.750	157.270	32.542	22.896	x4.833	x6.869	62 209	62 209	2 017	2 017	x31	x31
CFG 9 5	0.349	0.350	0.396	0.396	197.493	196.618	40.282	28.670	x4.881	x6.858	69 985	69 985	2 269	2 269	x31	x31
CFG 10 5	0.413	0.413	0.466	0.465	239.278	239.967	49.307	35.736	x4.867	x6.715	77 761	77 761	2 521	2 521	x31	x31
CFG 11 5	0.484	0.483	0.571	0.572	287.062	287.295	58.690	42.325	x4.895	x6.788	85 537	85 537	2 773	2 773	x31	x31
CFG 12 5	0.587	0.589	0.650	0.652	340.130	340.189	68.556	49.861	x4.962	x6.823	93 313	93 313	3 025	3 025	x31	x31
CFG 13 5	0.670	0.668	0.733	0.734	392.320	391.388	80.707	57.940	x4.849	x6.755	101 089	101 089	3 277	3 277	x31	x31
CFG 14 5	0.752	0.756	0.828	0.827	450.100	449.457	93.973	66.713	x4.783	x6.737	108 865	108 865	3 529	3 529	x31	x31
CFG 15 5	0.896	0.897	0.971	0.976	513.933	515.591	108.754	78.604	x4.741	x6.559	116 641	116 641	3 781	3 781	x31	x31
CFG 16 5	0.989	0.989	1.075	1.074	581.960	579.245	124.182	89.156	x4.664	x6.497	124 417	124 417	4 033	4 033	x31	x31
CFG 17 5	1.090	1.096	1.178	1.172	659.035	660.999	137.660	99.247	x4.802	x6.660	132 193	132 193	4 285	4 285	x31	x31
CFG 18 5	1.269	1.269	1.368	1.369	736.298	733.120	156.846	112.287	x4.674	x6.529	139 969	139 969	4 537	4 537	x31	x31
CFG 19 5	1.380	1.380	1.485	1.485	816.246	813.213	174.171	126.527	x4.669	x6.427	147 745	147 745	4 789	4 789	x31	x31
CFG 20 5	1.492	1.496	1.722	1.730	908.406	905.724	194.253	141.770	x4.663	x6.389	155 521	155 521	5 041	5 041	x31	x31

Table 7.7 – Résultats des réductions par symétries pour le modèle *CFG* (backend : Python, variable : instances de sous réseau).

(a) *Avec échelle linéaire.* (b) *Avec échelle logarithmique.*

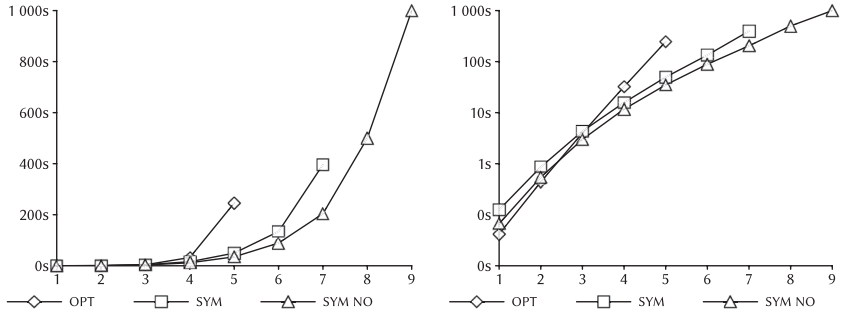


Figure 7.9 – *Temps d'exploration pour le modèle* CFG 10 x *pour un nombre variable de threads (axe horizontal).*

(a) *Avec échelle linéaire.* (b) *Avec échelle logarithmique.*

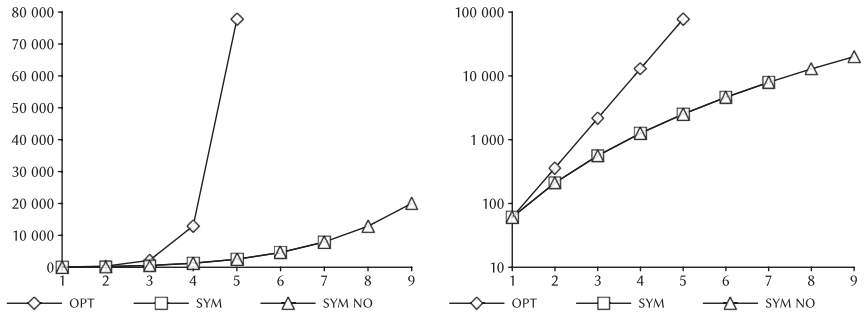


Figure 7.10 – *Taille des espaces d'états pour le modèle* CFG 10 x *pour un nombre variable de threads (axe horizontal).*

La réalisation de cet outil représente un investissement important en temps cependant les résultats sont prometteurs et montrent que Neco est compétitif avec les autres model checkers. De plus les résultats des réductions par symétries offrant des réductions notables en temps et espace.

Neco a d'ores et déjà des utilisateurs, notamment pour la vérification de *protocoles peer-to-peer* pour la réplication de fichiers [34] et une implémentation en réseaux de Petri de *pi-graphes*[1] [140]. Le choix de Neco dans ces deux cas a été rendu plus facile grâce à l'expressivité très riche des langages d'entrée, dans le premier cas la modélisation était réalisée en *ABCD*, et dans le second cas directement en Python grâce à la bibliothèque *SNAKES*.

1. Travail actuellement en cours au laboratoire IBISC.

8 Conclusion

Dans cette thèse nous nous sommes focalisés sur l'approche par compilation du calcul des espaces d'états des réseaux de Petri, et les réductions d'espaces d'états dans les systèmes à création dynamique de processus. Plusieurs contributions sur les plans théoriques et pratiques ont été présentées au fil des chapitres.

Nous avons défini un cadre général pour la compilation de réseaux de Petri. Ce cadre est indépendant du langage cible et est modulaire de par l'utilisation d'interfaces pour divers modules utilisés dans les algorithmes. Nous avons aussi vu une famille d'optimisations dérivées de la structure du réseau de Petri qui accélèrent grandement l'exploration d'espaces d'états et la compilation.

Cette approche de compilation a été appliquée dans le cadre d'un langage de bas niveau, LLVM. Ce langage est suffisamment expressif pour permettre une transcription aisée des algorithmes, mais aussi assez bas niveau pour être équipé d'une sémantique formelle, ce qui nous a permis de réaliser des preuves de correction et terminaison des algorithmes dans cette représentation.

Pour adresser les réductions d'espaces d'états dans les systèmes à création dynamique de processus, un nouveau cadre théorique pour les réductions par symétries a été défini. Cela a mené à la réalisation d'algorithmes effectifs et efficaces. La technique de réduction se base sur une structure arborescente qui est une représentation d'un état du modèle. Une étape de normalisation de ces arbres permet de calculer une représentation canonique d'une classe d'états ce qui permet les réductions. Par la suite, on a aussi vu dans les études de cas que ces réductions peuvent avoir un impact exponentiel sur les tailles des espaces d'états.

Finalement on a présenté la boite à outils Neco, développée pendant la thèse comme application des résultats sur le calcul des espaces d'états et permettant le model checking LTL. On a présenté les 4 outils qui la composent en nous focalisant sur les deux compilateurs, l'un pour les réseaux de Petri, l'autre pour les formules LTL sur un modèle compilé. Les différentes approches présentées dans ce manuscrit y ont été implémentées et des études de cas ont été réalisées. Neco a aussi participé au Model Checking Contest 2012 et 2013, ce qui a montré qu'il est compétitif avec les outils du domaine.

Les perspectives à court terme sont d'abord l'extension des techniques de réduction présentées ici à la relation frère. Il s'agit essentiellement d'adapter la normalisation pour prendre en compte cette relation, puis prouver la correction

de cette extension. Sur le plan pratique il s'agit aussi d'implémenter l'approche actuelle en Cython pour avoir des études de cas qui adressent de plus grands modèles puis implémenter l'extension décrite ci-dessus.

Sur le plan de la compilation il s'agit de continuer à chercher des optimisations en se focalisant sur les formalismes de modélisation structurés. En effet il est difficile de réaliser des optimisations efficaces sur toute la famille des réseaux de Petri. En revanche, dès qu'on ajoute des contraintes sur les modèles on peut trouver des optimisations efficaces comme on l'a vu pour les réductions de contrôle de flot.

Sur le moyen terme, il s'agit essentiellement de stabiliser Neco et d'ajouter des techniques de réduction d'occupation mémoire. Actuellement, le stockage de marquages est réalisé de manière simple et des techniques existantes efficaces pourraient être implémentées. Puis éventuellement, le backend LLVM pourra être réintroduit. Cela permettrait aussi de retravailler le théorème d'extensionnalité pour permettre le partage de mémoire, ce qui n'est pas possible actuellement.

Sur le long terme, il s'agit de s'orienter encore plus vers les réductions par symétries d'espaces d'états. En travaillant sur des formalismes structurées qui aident à la réduction. On a remarqué que l'approche optimiste peut être complète si le modèle a une forme particulière. En analysant les modèles on observe que les permutations des pid-trees ne sont nécessaires que si certains jetons possèdent plus d'un pid, or un jeton avec plus d'un pid peut être vu comme une primitive de communication entre deux processus. Le travail de recherche est donc de trouver un moyen pour les processus de communiquer tout en minimisant le surcoût de cette communication car elle introduit des permutations dans les pid-trees.

Finalement, dans cette thèse nous avons vu qu'un compromis entre la facilité de modélisation et l'efficacité de vérification est possible. Cette thèse montre que nous pouvons être efficaces tout en proposant une grande expressivité, au sens facilité de modélisation, mais aussi décidabilité : les réseaux de Petri qu'on propose sont annotés par un langage Turing complet. En pratique, on reporte donc sur l'utilisateur la responsabilité des problèmes liés à la non-décidabilité de ce langage. Des restrictions syntaxiques et des conseils de mise en oeuvre peuvent aider mais l'utilisateur pourra généralement les contourner s'il n'est pas suffisamment attentif aux risques, ou s'il le souhaite. Ceci n'est pas fondamentalement différent de ce qui arrive lorsqu'on programme, les précautions à prendre et la démarche à suivre en cas d'erreur sont donc essentiellement les mêmes. En contrepartie de cet inconvénient, l'avantage de proposer un langage existant, complet, et répandu est qu'on transmet aussi une grande partie de la décision concernant l'efficacité entre les mains de l'utilisateur : s'il désire passer peu de temps, avoir rapidement des simulations, ou des résultats sur de petits modèles, il peut le faire grâce à un langage de couleurs très haut niveau ; dans Neco, il s'agit de Python. Mais s'il veut progresser ensuite vers l'efficacité, il peut de manière incrémentale améliorer son modèle, spécialiser les types des places et implémenter des structures efficaces en `C`/`C++`/Cython ou tout langage qui s'interface avec Python (donc presque tout langage en pratique). Il en résulte que la démarche de modélisation se rapproche du développement traditionnel en cela que les outils proposés ressemblent à ceux que l'utilisateur connaît. De son point de vue, on lui propose un "vrai" langage de programmation, étendu par des constructions issues des réseaux de Petri qui facilitent l'expression de la

concurrence, des communications et d'autres aspects généralement difficiles à programmer. Ceci est complété par des possibilités de simulation automatique ou interactive et un outil de diagnostique (le model checker) proposant des traces facilement exploitables. Au final, la modélisation devient une activité familière dont il est facile de comprendre les rouages car la technique reste cachée (de même qu'un programmeur n'a pas besoin de comprendre ce que fait vraiment son compilateur). Ainsi facilitée, on peut espérer que la démarche de modélisation, et vérification, pourra être plus largement utilisée de façon plus pertinente.

9 Références

9.1 Références de la thèse

[FDL13] Ł. Fronc and A. Duret-Lutz. Ltl model checking with neco. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology For Verification And Analysis, 11th International Symposium (ATVA'13)*, LNCS. Springer, 2013.

[FP11a] Ł Fronc and F. Pommereau. Optimizing the compilation of Petri nets models. In *Proc. of SUMo'11*, volume 726. CEUR, 2011.

[FP11b] L. Fronc and F. Pommereau. Proving a Petri net model-checker implementation. Technical report, IBISC, University of Évry, `http://goo.gl/WMzhp`, 2011.

[FP11c] Ł. Fronc and F. Pommereau. Towards a certified Petri net model-checker. In *Proc. of the 9th Asian Symposiium on Programming Languages and Systems (APLAS 2011)*, volume 7078 of *Lecture Notes in Computer Science*, pages 322–336, Kenting, Taïwan, Province De Chine, 2011.

[FP13] Ł Fronc and F. Pommereau. Building Petri nets tools around Neco compiler. In *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'13)*, volume 989, Milano, June 2013. CEUR.

[Fro12a] Ł Fronc. Effective marking equivalence checking in systems with dynamic process creation. In *Proceedings of the 14th International Workshop on Verification of Infinite-State Systems (Infinity'12)*, Electronic Proceedings in Theoretical Computer Science, Paris, August 2012.

[Fro12b] Ł Fronc. Neco net compiler wiki. `goo.gl/CXrry`, 2012.

[KLB+12] F. Kordon, A. Linard, D. Buchs, M. Colange, S. Evangelista, Ł. Fronc, L.-M. Hillah, N. Lohmann, E. Paviot-Adet, F. Pommereau,

C. Rohr, Y. Thierry-Mieg, H. Wimmel, and K. Wolf. Raw report on the model checking contest at petri nets 2012. *CoRR*, abs/1209.2382, 2012.

[KLB+13] F. Kordon, A. Linard, M. Beccuti, D. Buchs, Ł. Fronc, L.-M. Hillah, F. Hulin-Hubard, F. Legond-Aubry, N. Lohmann, A. Marechal, E. Paviot-Adet, F. Pommereau, C. Rodríguez, C. Rohr, Y. Thierry-Mieg, H. Wimmel, and K. Wolf. Model Checking Contest @ Petri Nets, Report on the 2013 edition. page 422, September 2013.

9.2

Autres références

[10] The Coq proof assistant. `http://coq.inria.fr/`.

[11] The HOL proof assistant for higher order logic. `http://hol.sourceforge.net/`.

[12] J.-R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[13] Issam Al-Azzoni, Douglas G. Down, and Ridha Khedri. Modeling and verification of cryptographic protocols using coloured petri nets and design/cpn. *Nordic J. of Computing*, 12(3) :201–228, June 2005.

[14] Mourad Amziani, Tarek Melliti, and Samir Tata. Formal Modeling and Evaluation of Service-Based Business Process Elasticity in the Cloud. In *Proc. of the 22nd IEEE International Conference on Collaboration Technologies and Infrastructure (WETICE 2013)*, page (to appear), Hammamet, Tunisie, June 2013.

[15] Michael Bader, Hans-Joachim Bungartz, Dan Grigoras, Miriam Mehl, Ralf-Peter Mundani, and Rodica Potolea, editors. *11th International Symposium on Parallel and Distributed Computing, ISPDC 2012, Munich, Germany, June 25-29, 2012*. IEEE Computer Society, 2012.

[16] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, and Wen mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *In PPoPP*, pages 105–114, 2010.

[17] Bernhard Beckert and Vladimir Klebanov. Must program verification systems and calculi be verified. In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, pages 34–41, 2006.

[18] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython : The best of both worlds. *Computing in Science Engineering*, 13(2) :31 –39, march-april 2011.

[19] Gérard Berthelot. *Transformations et analyse de réseaux de Petri : Application aux protocoles*. PhD thesis, 1983.

[20] Gérard Berthelot et al. Checking properties of nets using transformations. In *Advances in Petri Nets 1985*, pages 19–40. Springer, 1986.

[21] Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular smt proofs for fast reflexive checking inside coq. In *Certified Programs and Proofs*, pages 151–166. Springer, 2011.

[22] Eike Best, Raymond Devillers, and JonG. Hall. The box calculus : A new causal algebra with multi-label communication. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1992*, volume 609 of *Lecture Notes in Computer Science*, pages 21–69. Springer Berlin Heidelberg, 1992.

[23] Eike Best, Raymond Devillers, and Maciej Koutny. *Petri net algebra*. Springer, 2001.

[24] Eike Best, Raymond Devillers, and Maciej Koutny. The box algebra= petri nets+ process expressions. *Information and Computation*, 178(1) :44–100, 2002.

[25] Eike Best, Hans Fleischhack, Wojciech Fraczak, RichardP. Hopkins, Hanna Klaudel, and Elisabeth Pelz. A class of composable high level petri nets. In Giorgio Michelis and Michel Diaz, editors, *Application and Theory of Petri Nets 1995*, volume 935 of *Lecture Notes in Computer Science*, pages 103–120. Springer Berlin Heidelberg, 1995.

[26] Dragan Bošnački, Dennis Dams, and Leszek Holenderski. Symmetric spin. In *SPIN Model Checking and Software Verification*, pages 1–19. Springer, 2000.

[27] Dragan Bošnački, Dennis Dams, and Leszek Holenderski. Symmetric spin. *International Journal on Software Tools for Technology Transfer*, 4(1) :92–106, 2002.

[28] Dragan Bošnački, Leszek Holenderski, and Dennis Dams. A heuristic for symmetry reductions with scalarsets. In *FME 2001 : Formal Methods for Increasing Software Productivity*, pages 518–533. Springer, 2001.

[29] R. Bouroulet, H. Klaudel, and E. Pelz. Modelling and verification of authentication using enhanced net semantics of SPL (Security Protocol Language). In *ACSD'06*. IEEE Computer Society, 2006.

[30] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8) :677–691, 1986.

[31] Didier Buchs and Steve Hostettler. Sigma decision diagrams. In Andrea Corradini, editor, *TERMGRAPH 2009 : Premiliminary proceedings of the 5th International Workshop on Computing with Terms and Graphs*, number TR-09-05, pages 18–32. Università di Pisa, 2009.

[32] Cécile Bui Thanh, Hanna Klaudel, and Franck Pommereau. Box Calculus with Coloured Buffers, 2002. Technical report 2002-16, LACL.

[33] J. R. Burch, E.M. Clarke, K. L. McMillan, D.L. Dill, and L. J. Hwang. Symbolic model checking : 1020 states and beyond. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439, 1990.

[34] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking : 1020 states and beyond. *Information and Computation*, 98(2) :142 – 170, 1992.

[35] S. Chaou and F. Pommereau. Formal Modelling and Analysis of Behaviour Grading within a peer-to-peer Storage System. In *The 2012 Symposium On Theory of Modeling and Simulation*, Florida, 2012. IEEE.

[36] Ching-Tsun Chou and Doron Peled. Formal verification of a partial-order reduction technique for model checking. *Journal of Automated Reasoning*, 23(3) :265–298, 1999.

[37] Søren Christensen, Lars Michael Kristensen, and Thomas Mailund. A sweep-line method for state space exploration. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 450–464. Springer, 2001.

[38] E. Clarke, A. Emerson, and J. Sifakis. Model checking : Algorithmic verification and debugging. *ACM Turing Award*, 2007.

[39] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC '95, pages 427–432, New York, NY, USA, 1995. ACM.

[40] Edmund M Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2) :77–104, 1996.

[41] E.M. Clarke, E.A. Emerson, S. Jha, and A.P. Sistla. Symmetry reductions in model checking. In AlanJ. Hu and MosheY. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 147–158. Springer Berlin Heidelberg, 1998.

[42] Monica Clemente, MariaPia Fanti, AgostinoM. Mangini, and Walter Ukovich. The vehicle relocation problem in car sharing systems : Modeling and simulation in a petri net framework. In José-Manuel Colom and Jörg Desel, editors, *Application and Theory of Petri Nets and Concurrency*, volume 7927 of *Lecture Notes in Computer Science*, pages 250–269. Springer Berlin Heidelberg, 2013.

[43] J.M. Colom, M. Silva, and J.L. Villarroel. On software implementation of petri nets and colored petri nets using high-level concurrent languages. In *Seventh European Workshop on Application and Theory of Petri Nets*, pages 207–241, Oxford, England, 06/1986 1986.

[44] ADT Coq/INRIA. The Coq proof assistant. `http://coq.inria.fr`.

[45] Jean-Michel Couvreur, Alexandre Duret-Lutz, and Denis Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In Patrice Godefroid, editor, *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*, volume 3639 of *Lecture Notes in Computer Science*, pages 143–158. Springer, August 2005.

[46] Jean-Michel Couvreur, Emmanuelle Encrenaz, Emmanuel Paviot-Adet, Denis Poitrenaud, and Pierre-André Wacrenier. Data decision diagrams for Petri net analysis. In *Proceedings of the 23th International Conference on Application and Theory of Petri Nets (ICATPN'02)*, volume 2360 of *Lecture Notes in Computer Science*, pages 101–120, Adelaide, Australia, June 2002. Springer Verlag.

[47] Raymond Devillers. Construction of s-invariants and s-components for refined petri boxes. In Marco Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 242–261. Springer Berlin Heidelberg, 1993.

[48] Raymond Devillers. S-invariant analysis of general recursive petri boxes. *Acta Informatica*, 32(4) :313–345, 1995.

[49] Raymond Devillers, Hanna Klaudel, Maciej Koutny, and Franck Pommereau. Asynchronous box calculus. *Fundamenta Informaticae*, 54(4) :295–344, 2003.

[50] Raymond Devillers, Hanna Klaudel, and Robert-C Riemann. General refinement for high level petri nets. In *Foundations of Software Technology and Theoretical Computer Science*, pages 297–311. Springer, 1997.

[51] Cristian Dimitrovici, Udo Hummert, and Laure Petrucci. Semantics, composition and net properties of algebraic high-level nets. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1991*, volume 524 of *Lecture Notes in Computer Science*, pages 93–117. Springer Berlin Heidelberg, 1991.

[52] Alexandre Duret-Lutz. LTL translation improvements in Spot. In *Proceedings of the 5th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'11)*, Electronic Workshops in Computing, Tunis, Tunisia, September 2011. British Computer Society.

[53] Alexandre Duret-Lutz and Denis Poitrenaud. SPOT : an Extensible Model Checking Library using Transition-based Generalized Büchi Automata. In *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, pages 76–83, Volendam, The Netherlands, October 2004. IEEE Computer Society Press.

[54] E Allen Emerson and A Prasad Sistla. Symmetry and model checking. *Formal methods in system design*, 9(1-2) :105–131, 1996.

[55] E.Allen Emerson and 1]RichardJ. Trefler. From asymmetry to full symmetry : New techniques for symmetry reduction in model checking. In

Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 142–157. Springer Berlin Heidelberg, 1999.

[56] S. Evangelista. *Méthodes et outils de vérification pour les réseaux de Petri de haut niveau*. PhD thesis, CNAM, Paris, France, 2006.

[57] S. Evangelista and J.-F. Pradat-Peyre. An efficient algorithm for the enabling test of colored Petri nets. In *CPN'04*, number 570 in DAIMI report PB. University of Århus, Denmark, 2004.

[58] Sami Evangelista. High level petri nets analysis with helena. In *Applications and Theory of Petri Nets 2005*, pages 455–464. Springer, 2005.

[59] Sami Evangelista. *Méthodes et outils de vérification pour les réseaux de Petri de haut nive*. PhD thesis, CNAM, Paris, France, 2006.

[60] Sami Evangelista, Serge Haddad, and J-F Pradat-Peyre. Syntactical colored petri nets reductions. In *Automated Technology for Verification and Analysis*, pages 202–216. Springer, 2005.

[61] Sami Evangelista, Serge Haddad, and Jean-François Pradat-Peyre. New coloured reductions for software validation. 2004.

[62] Sami Evangelista and Jean-François Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *Model Checking Software*, pages 43–57. Springer, 2005.

[63] Sami Evangelista and Jean-François Pradat-Peyre. On the computation of stubborn sets of colored petri nets. In *Petri Nets and Other Models of Concurrency-ICATPN 2006*, pages 146–165. Springer, 2006.

[64] Mike Gancarz. *The UNIX philosophy*. Digital Press, Newton, MA, USA, 1995.

[65] F. Gava, M. Guedj, and F. Pommereau. A BSP algorithm for the state space construction of security protocols. In *PDMC'10*. IEEE Computer Society, 2010.

[66] Frédéric Gava, Michaël Guedj, and Franck Pommereau. A bsp algorithm for on-the-fly checking ltl formulas on security protocols. In Bader et al. [**?**], pages 11–18.

[67] Frederic Gava, Michaël Guedj, and Franck Pommereau. A bsp algorithm for on-the-fly checking ltl formulas on security protocols. In *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*, pages 11–18. IEEE, 2012.

[68] Jaco Geldenhuys, PJA De Villiers, and John Rushby. Runtime efficient state compaction in spin. In *Theoretical and Practical Aspects of SPIN Model Checking*, pages 12–21. Springer, 1999.

[69] H.J. Genrich and K. Lautenbach. S-invariance in predicate/transition nets. In *European Workshop on Applications and Theory of Petri Nets*, 1982.

[70] Rob Gerth, Doron Peled, Moshe Y Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6. 1 International Symposium on Protocol Specification, Testing and Verification*. IFIP, 1995.

[71] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Computer-Aided Verification*, pages 176–185. Springer, 1991.

[72] Patrice Godefroid, Gerard J Holzmann, and Didier Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3) :227–241, 1995.

[73] Patrice Godefroid, J van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems : an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.

[74] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Computer Aided Verification*, pages 332–342. Springer, 1992.

[75] Roberto Gorrieri and Ugo Montanari. On the implementation of concurrent calculi in net calculi : two case studies, 1995.

[76] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in coq. In *Theorem Proving in Higher Order Logics*, pages 98–113. Springer, 2005.

[77] J-Ch Grégoire. State space compression in spin with getss. In *Proc. Second SPIN Workshop, Rutgers Univ*. Citeseer, 1996.

[78] S Haddad and Jean-François Pradat-Peyre. Efficient reductions for ltl formulae verification. Technical report, Technical report, CEDRIC, CNAM, Paris, 2004.

[79] Serge Haddad. A reduction theory for coloured nets. In *High-level Petri Nets*, pages 399–425. Springer, 1991.

[80] Alexandre Hamez, Yann Thierry-Mieg, and Fabrice Kordon. Hierarchical set decision diagrams and automatic saturation. In *Proceedings of the 29th international conference on Applications and Theory of Petri Nets*, PETRI NETS '08, pages 211–230, Berlin, Heidelberg, 2008. Springer-Verlag.

[81] Moritz Hammer, Alexander Knapp, and Stephan Merz. Truly on-the-fly ltl model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 191–205. Springer, 2005.

[82] L.M. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Trèves. A primer on the Petri Net Markup Language and ISO/IEC 15909-2. In *10th International workshop on Practical Use of Colored Petri Nets and the CPN Tools (CPN'09)*, Oct. 2009.

[83] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, 1978.

[84] Ramin Hojati, RobertK. Brayton, and RobertP. Kurshan. Bdd-based debugging of designs using language containment and fair ctl. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 41–58. Springer Berlin Heidelberg, 1993.

[85] Gerard J Holzmann. Tracing protocols. *AT&T technical journal*, 64(12) :2413–2434, 1985.

[86] Gerard J Holzmann. An improved protocol reachability analysis technique. *Software : Practice and Experience*, 18(2) :137–161, 1988.

[87] Gerard J Holzmann. Algorithms for automated protocol validation. *AT&T technical journal*, 69(1) :32–44, 1990.

[88] Gerard J Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5) :279–295, 1997.

[89] Gerard J Holzmann. State compression in spin : Recursive indexing and compression training runs. In *Proceedings of Third International SPIN Workshop*, 1997.

[90] Gerard J Holzmann. An analysis of bitstate hashing. *Formal methods in system design*, 13(3) :289–307, 1998.

[91] Gerard J Holzmann. On checking model checkers. In *Computer Aided Verification*, pages 61–70. Springer, 1998.

[92] Gerard J Holzmann and Doron Peled. An improvement in formal verification. In *FORTE*, volume 6, pages 197–211, 1994.

[93] Gerard J Holzmann, Doron Peled, and Mihalis Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, volume 32, pages 81–89, 1996.

[94] Gerard J Holzmann and Anuj Puri. A minimized automaton representation of reachable states. *International Journal on Software Tools for Technology Transfer*, 2(3) :270–278, 1999.

[95] G.J. Holzmann and al. Spin, formal verification. `http://spinroot.com`.

[96] P. Huber, K. Jensen, and R.M. Shapiro. Hierarchies in coloured petri nets. In Kurt Jensen and Grzegorz Rozenberg, editors, *High-level Petri Nets*, pages 215–243. Springer Berlin Heidelberg, 1991.

[97] Peter Huber, Arne M. Jensen, Leif O. Jepsen, and Kurt Jensen. Reachability trees for high-level petri nets. *Theoretical Computer Science*, 45(0) :261 – 292, 1986.

[98] Peter Huber, ArneM. Jensen, LeifO. Jepsen, and Kurt Jensen. Towards reachability trees for high-level petri nets. In G. Rozenberg, editor, *Advances in Petri Nets 1984*, volume 188 of *Lecture Notes in Computer Science*, pages 215–233. Springer Berlin Heidelberg, 1985.

[99] Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 254–261. IEEE, 2001.

[100] Radu Iosif. Symmetry reductions for model checking of concurrent dynamic software. *International Journal on Software Tools for Technology Transfer*, 6(4) :302–319, 2004.

[101] Claude Jard and Thierry Jéron. Bounded-memory algorithms for verification on-the-fly. In *Computer Aided Verification*, pages 192–202. Springer, 1992.

[102] K. Jensen. Coloured Petri nets and the invariant-method. *Theoretical Computer Science*, 14(3), 1981.

[103] K. Jensen. *Coloured Petri nets : basic concepts, analysis methods, and practical use*. Number vol. 1 in EATCS monographs on theoretical computer science. Springer-Verlag, 1992.

[104] K. Jensen and L.M. Kristensen. *Coloured Petri Nets : Modelling and Validation of Concurrent Systems*. Springer, 2009, ISBN 978-3-642-00283-0.

[105] Kurt Jensen. *An introduction to the theoretical aspects of coloured petri nets*. Springer, 1994.

[106] Kurt Jensen. Condensed state spaces for symmetrical coloured petri nets. *Formal Methods in System Design*, 9(1-2) :7–40, 1996.

[107] Kurt Jensen, LarsMichael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4) :213–254, 2007.

[108] Jens Baek Jorgensen and Lars Michael Kristensen. Computer aided verification of lamport's fast mutual exclusion algorithm using colored petri nets and occurrence graphs with symmetries. *Parallel and Distributed Systems, IEEE Transactions on*, 10(7) :714–732, 1999.

[109] Tommi Junttila et al. *On the symmetry reduction method for Petri Nets and similar formalisms*. Helsinki University of Technology, 2003.

[110] Tommi A Junttila. New canonical representative marking algorithms for place/transition-nets. In Jordi Cortadella and Wolfgang Reisig, editors, *Application and Theory of Petri Nets 2004*, Lecture Notes in Computer Science, pages 258–277. Springer Berlin Heidelberg, 2004.

[111] Terence Kelly, Yin Wang, Stéphane Lafortune, and Scott Mahlke. Eliminating concurrency bugs with control engineering. *Computer*, 42(12) :52–60, 2009.

[112] Shinji Kimura and Edmund M Clarke. A parallel algorithm for constructing binary decision diagrams. In *Computer Design : VLSI in Computers and Processors, 1990. ICCD'90. Proceedings., 1990 IEEE International Conference on*, pages 220–223. IEEE, 1990.

[113] Ekkart Kindler and Hagen Völzer. *Flexibility in algebraic nets*. Springer, 1998.

[114] Ekkart Kindler and Hagen Völzer. Algebraic nets with flexible arcs. *Theoretical Computer Science*, 262(1) :285–310, 2001.

[115] H. Klaudel, M. Koutny, E. Pelz, and F. Pommereau. Towards efficient verification of systems with dynamic process creation. volume 5160 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2008.

[116] H. Klaudel, M. Koutny, E. Pelz, and F. Pommereau. An approach to state space reduction for systems with dynamic process creation. In *ISCIS'09*, pages 543–548. IEEE Digital Library, 2009.

[117] H. Klaudel, M. Koutny, E. Pelz, and F. Pommereau. State space reduction for dynamic process creation. *Scientific Annals of Computer Science*, 20, 2010.

[118] LarsMichael Kristensen and Antti Valmari. Finding stubborn sets of coloured petri nets without unfolding. In Jörg Desel and Manuel Silva, editors, *Application and Theory of Petri Nets 1998*, volume 1420 of *Lecture Notes in Computer Science*, pages 104–123. Springer Berlin Heidelberg, 1998.

[119] Stéphane Lafortune, Yin Wang, and Spyros Reveliotis. Eliminating concurrency bugs in multithreaded software : An approach based on control of petri nets. In José-Manuel Colom and Jörg Desel, editors, *Application and Theory of Petri Nets and Concurrency*, volume 7927 of *Lecture Notes in Computer Science*, pages 21–28. Springer Berlin Heidelberg, 2013.

[120] C. Lattner. LLVM language reference manual. `http://llvm.org/docs/LangRef.html`.

[121] C. Lattner and al. The LLVM compiler infrastructure. `http://llvm.org`.

[122] C. Lattner and al. LLVM related publications. `http://llvm.org/pubs`.

[123] C. Lattner and al. LLVM users. `http://llvm.org/Users.html`.

[124] D. Lehmann and M. Rabin. On the advantage of free choice : A symmetric and fully distributed solution to the dining philosophers problem (extended abstract). In *Proc. 8th Annual ACM Symposium on Principles of Programming Languages (POPL'81)*, pages 133–138, 1981.

[125] Wanwei Liu, Rui Wang, Xianjin Fu, Ji Wang, Wei Dong, and Xiaoguang Mao. Counterexample-preserving reduction for symbolic model checking. *CoRR*, abs/1301.3299, 2013.

[126] R. Mahadevan. Python bindings for LLVM. `http://www.mdevan.org/llvm-py`.

[127] Marko Mäkelä. Applying compiler techniques to reachability analysis of high-level models. In *Workshop on Concurrency, Specification & Programming*, number 140, pages 129–142, 2000.

[128] Marko Mäkelä. Condensed storage of multi-set sequences. *Practical Use of High-Level Petri Nets, DAIMI report PB-547*, pages 111–125, 2000.

[129] P Merlin. *A Study of the Recoverability of Computer Systems*. PhD thesis, University of California, Irvine, California, 1974.

[130] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[131] M.K. Molloy. *On the Integration of Delay and Throughput Measures in Distributed Processing Models*. Report (University of California, Los Angeles. Computer Science Dept.). Computer Science Department Research Laboratory, 1981.

[132] Ugo Montanari and Daniel Yankelevich. Combining ccs and petri nets via structural axioms. *Fundam. Inf.*, 20(1,2,3) :193–229, April 1994.

[133] Marko Mäkelä. Maria : Modular reachability analyser for algebraic system nets. In Javier Esparza and Charles Lakos, editors, *Application and Theory of Petri Nets 2002*, volume 2360 of *Lecture Notes in Computer Science*, pages 434–444. Springer Berlin Heidelberg, 2002.

[134] Kedar S Namjoshi. Certifying model checkers. In *Computer Aided Verification*, pages 2–13. Springer, 2001.

[135] Gordon Pace, Nicolas Halbwachs, and Pascal Raymond. Counter-example generation in symbolic abstract model-checking. *International Journal on Software Tools for Technology Transfer*, 5(2-3) :158–164, 2004.

[136] C. Pajault and S. Evangelista. Helena : a high level net analyzer. `http://helena.cnam.fr`.

[137] Christophe Pajault and Sami Evangelista. Helena : a high level net analyzer. `http://helena.cnam.fr`.

[138] L. Paulson, T. Nipkow, and M. Wenzel. The Isabelle proof assistant. `http://www.cl.cam.ac.uk/research/hvg/Isabelle`.

[139] Doron Peled. All from one, one for all : on model checking using representatives. In *Computer Aided Verification*, pages 409–423. Springer, 1993.

[140] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1) :39–64, 1996.

[141] Frédéric Peschanski, Hanna Klaudel, and Raymond Devillers. A Petri Net Interpretation of Open Reconfigurable Systems. *Fundamenta Informaticae*, 120 :1–33, 2012.

[142] Carl Adam Petri. *Communication with automata (Kommunikation mit Automaten)*. PhD thesis, Universität Hamburg, 1962.

[143] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, 1977.

[144] I. Poliakov, A. Mokhov, A. Rafiev, D. Sokolov, and A. Yakovlev. Automated verification of asynchronous circuits using circuit petri nets. In *Asynchronous Circuits and Systems, 2008. ASYNC '08. 14th IEEE International Symposium on*, pages 161–170, 2008.

[145] F. Pommereau. *Algebras of coloured Petri nets.* LAMBERT Academic Publishing, October 2010, ISBN 978-3-8433-6113-2.

[146] Franck Pommereau. Quickly prototyping Petri nets tools with SNAKES. *Petri net newsletter*, October 2008.

[147] Louchka Popova-Zeugmann. On time petri nets. *Elektronische Informationsverarbeitung und Kybernetik*, 27(4) :227–244, 1991.

[148] Chander Ramchandani. *Analysis of asynchronous concurrent systems by timed petri nets.* PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering, 1973.

[149] C. Reinke. Haskell-coloured Petri nets. In *IFL'99*, volume 1868 of *LNCS*. Springer, 1999.

[150] A. Rigo and al. PyPy. `http://pypy.org/`.

[151] Karsten Schmidt. How to calculate symmetries of petri nets. *Acta Informatica*, 36(7) :545–590, 2000.

[152] Karsten Schmidt. Integrating low level symmetries into reachability analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 315–330. Springer, 2000.

[153] J. Sifakis. Performance evaluation of systems using nets. In Wilfried Brauer, editor, *Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Science*, pages 307–319. Springer Berlin Heidelberg, 1980.

[154] Antti Valmari. Error detection by reduced reachability graph generation. In *Proceedings of the 9th European Workshop on Application and Theory of Petri Nets*, pages 95–112, 1988.

[155] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, pages 491–515. Springer, 1991.

[156] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4) :297–322, 1992.

[157] Antti Valmari. State of the art report : Stubborn sets. *Petri Net Newsletter*, 46 :6–14, 1994.

[158] Rob van Glabbeek and Frits Vaandrager. Petri net models for algebraic theories of concurrency. In *PARLE Parallel Architectures and Languages Europe*, pages 224–242. Springer, 1987.

[159] Jacques Vautherin. Parallel systems specifications with coloured petri nets and algebraic specifications. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1987*, volume 266 of *Lecture Notes in Computer Science*, pages 293–308. Springer Berlin Heidelberg, 1987.

[160] K.N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *ASIAN'00*, volume 1961 of *LNCS*. Springer, 2000.

[161] Willem Visser and Howard Barringer. Memory efficient state storage in spin. In *Proceedings of the 2nd SPIN Workshop*, volume 21, 1996.

[162] Michael Westergaard and H. M. W. Verbeek. Efficient implementation of prioritized transitions for high-level Petri nets. In Michael Duvigneau, Daniel Moldt, and Kunihiko Hiraishi, editors, *Petri Nets and Software Engineering. International Workshop PNSE'11*, volume 723 of *CEUR Workshop Proceedings*, pages 27–41. CEUR-WS.org, June 2011.

[163] Pierre Wolper and Patrice Godefroid. Partial-order methods for temporal verification. In *CONCUR'93*, pages 233–246. Springer, 1993.

[164] Pierre Wolper and Denis Leroy. Reliable hashing without collision detection. In *Computer Aided Verification*, pages 59–70. Springer, 1993.

A | LLVM

A.1

Implantation des Marquages

Dans cette section on décrit la réalisation d'une structure de marquage. Un marquage simple peut être implémenté avec une structure qui contient des pointeurs sur des places. Le type de cette structure est $struct(t_{s_0}, \ldots, t_{s_n})$ où t_{s_i} est le type de la i-ème place (la place s_i en énumérant les places à partir de 0). La seule chose qu'on suppose est que les structures qui encodent les places ne partagent pas de mémoire.

A.1.1 Interprétation et les propriétés

On définit l'interprétation récursivement sur la structure. Si p_{mrk} est un pointeur sur la structure de marquage dans un tas H bien formé, alors par définition on a

$$H(p_{mrk}) = (struct(t_{s_0}, \ldots, t_{s_n}), (p_{s_0}, \ldots, p_{s_n}))$$

On définit l'interprétation dans ces conditions par :

$$[\![H, p_{mrk}]\!]^{mrk} = \{s_0 \mapsto [\![H, p_{s_0}]\!]^{s_0}, \ldots, s_n \mapsto [\![H, p_{s_n}]\!]^{s_n}\}$$

On vérifie que la fonction d'interprétation est valide (supposition 4.2 page 53).

Soient H, H' deux tas et p, p' deux pointeurs sur des structures de marquages tels que $p \in dom(H)$ et $p' \in dom(H')$. Si $(H, p) =_{st} (H', p')$ alors $[\![H, p]\!]^{mrk} = [\![H', p']\!]^{mrk}$.

Preuve. Par définition de la fonction d'interprétation et de la structure car les fonctions d'interprétation des places vérifient la propriété de validité. □

On vérifie maintenant la propriété de cohérence pour cette structure de données (prérequis 4.3 page 54).

Soient H, H' deux tas, F, F' deux frames, p_{mrk} un pointeur de H sur une structure de marquage, et p_s un pointeur sur une place de cette structure. Si

$p_s \in p_{mrk}\downarrow_H$, $[\![H, p_{mrk}]\!]^{mrk}(s) = [\![H, p_s]\!]^s$, $\left(\ seq\ \right)_{H,F} \rightsquigarrow \left(\ seq'\ \right)_{H\oplus H',F'}$ *et* $p_{mrk} \notin dom(H')$, *alors*

$$[\![H \oplus H', p_{mrk}]\!]^{mrk}(s) = [\![H \oplus H', p_s]\!]^s$$

Preuve. Immédiat par définition de $[\![\cdot,\cdot]\!]^{mrk}$ car

$$(H \oplus H')(p_{mrk}) = H(p_{mrk}) = (struct(t_{s_0}, \ldots, t_{s_n}), (p_{s_0}, \ldots, p_{s_n}))$$

□

La propriété de séparation quant à elle est triviale dans cette structure car les places sont séparées (prérequis 4.4 page 54).

Soit p_{mrk} un pointeur sur une structure de marquage dans un tas H. Si p_s et $p_{s'}$ sont deux pointeurs sur deux places distinctes de cette structure alors on a $p_s\downarrow_H \cap\ p_{s'}\downarrow_H = \emptyset$.

Preuve. Par définition de la structure. □

A.1.2 Les fonctions *get*

Si s_i est la i-ème place du marquage alors on définit la fonction get_{s_i} de la structure de marquage par :

$$P(get_{s_i}(\!|\, x_{mrk}\,|\!)) \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x_{s_i} = gep\ x_{mrk},\ 0,\ i \\ ret\ x_{s_i} \end{array} \right.$$

On vérifie que cette fonction implémente l'interface demandée.

Proposition A.1. *La fonction get_{s_i} vérifie bien sa spécification (cf. 1 page 57),* i.e., *Si*

$$\left(\ fcall\ get_{s_i}(p_{mrk})\ \right)_{H,F} \rightsquigarrow \left(\ p_s\ \right)_H$$

alors $p_s\downarrow_H \subset p_{mrk}\downarrow_H$ et $[\![H, p_{mrk}]\!]^{mrk}(s_i) = [\![H, p_s]\!]^s$.

Preuve. Immédiat par définition de la structure et de $[\![\cdot,\cdot]\!]^{mrk}$. □

A.1.3 La fonction *copy*

On définit la fonction $copy_{mrk}$ de la structure de marquage par :

$$P(copy_{mrk}(\!|\, x_{mrk}\,|\!)) \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x'_{mrk} = alloc\ struct(t_{s_0}, \ldots, t_{s_n}) \\ \left.\begin{array}{l} x_{s_i} = fcall\ get_{s_i}(x_{mrk}) \\ x'_{s_i} = fcall\ get_{s_i}(x'_{mrk}) \\ pcall\ copy_{s_i}(x_{s_i}, x'_{s_i}) \end{array}\right\} \text{pour } 0 \leq i \leq n \\ ret\ x'_{mrk} \end{array} \right.$$

Remarquons que dans le cadre d'une sémantique *copy-on-write*, on ne ferait pas de copie en profondeur mais juste une copie de pointeurs. La copie des

places serait réalisée à la première écriture ce qui permettrait de gagner en performances.

Pour vérifier que cette fonction implémente bien l'interface demandée, on va passer par deux lemmes prouvant respectivement le cœur de la fonction et la spécification du point de vue du corps de la fonction. Le résultat attendu sera une conséquence immédiate du second lemme.

Pour un marquage M, un tas H, un frame F, x_{mrk} une variable et p_{mrk} un pointeur tels que $F(x_{mrk}) = p_{mrk}$ et $[\![H, p_{mrk}]\!]^{mrk} = M$. La spécification demandée de la fonction $copy_{mrk}$ est la suivante :

$$\left(\ fcall\ copy_{mrk}(x_{mrk})\ \right)_{H,F} \rightsquigarrow \left(\ p'_{mrk}\ \right)_{H\oplus H'}$$

$$dom(H') \cap dom(H) = \emptyset$$

$$p'_{mrk}{\downarrow_{H\oplus H'}} \subseteq dom(H')$$

$$[\![H, p_{mrk}]\!]^{mrk} = [\![H \oplus H', p_{mrk}]\!]^{mrk} = [\![H \oplus H', p'_{mrk}]\!]^{mrk}$$

Lemme A.2. *Soit M un marquage, H un tas bien formé, p_{mrk} et p'_{mrk} deux pointeurs de H sur des instances distinctes de la structure de marquage tels que $p_{mrk}{\downarrow_H} \cup p'_{mrk}{\downarrow_H} = dom(H)$ et $[\![H, p_{mrk}]\!]^{mrk} = M$. Soit F un frame, x_{mrk} et x'_{mrk} deux variables tels que $F(x_{mrk}) = p_{mrk}$ et $F(x'_{mrk}) = p'_{mrk}$. Soit seq la séquence d'instructions suivante qui correspond au coeur de la fonction $copy_{mrk}$:*

$$seq \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x_{s_0} = fcall\ get_{s_0}(x_{mrk}) \\ x'_{s_0} = fcall\ get_{s_0}(x'_{mrk}) \\ pcall\ copy_{s_0}(x_{s_0}, x'_{s_0}) \\ \ldots \\ x_{s_n} = fcall\ get_{s_n}(x_{mrk}) \\ x'_{s_n} = fcall\ get_{s_n}(x'_{mrk}) \\ pcall\ copy_{s_n}(x_{s_n}, x'_{s_n}) \end{array} \right.$$

Si on réduit toute la séquence, i.e.*, si*

$$\left(\ seq\ \right)_{H,F} \rightsquigarrow^* \left(\ skip\ \right)_{H\oplus H', F\oplus\beta}$$

alors

$$\beta \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x_{s_0} \mapsto p_{s_0}, \ldots, x_{s_n} \mapsto p_{s_n}, \\ x'_{s_0} \mapsto p'_{s_0}, \ldots, x'_{s_n} \mapsto p'_{s_n} \end{array} \right\} \tag{A.1}$$

$$dom(H) \cap dom(H') \subseteq p'_{s_0}{\downarrow_H} \cup \cdots \cup p'_{s_n}{\downarrow_H} \subseteq p'_{mrk}{\downarrow_H} \tag{A.2}$$

$$[\![H, p_{mrk}]\!]^{mrk}(s_i) = [\![H, p_{s_i}]\!]^{s_i} = [\![H \oplus H', p_{s_i}]\!]^{s_i} = [\![H \oplus H', p'_{s_i}]\!]^{s_i} \quad pour\ 0 \leq i \leq n \tag{A.3}$$

Le résultat (A.1) est le résultat de toutes les affectations exécutées. Le résultat (A.2) assure que seuls les pointeurs de la structure de marquage pointée par p'_{mrk} sont dans le domaine de H', plus précisément les pointeurs accessibles à partir des places $s_0, \ldots, s_n$, et donc c'est la seule partie du tas susceptible d'être modifiée. Cette conséquence aidera à argumenter les conditions sur les domaines de la spécification. Le dernier résultat (A.3) assure qu'on copie bien le contenu des places.

Preuve. On fait la preuve par induction sur n.
Pour le cas de base, n = 0 : On sait que *seq* est de la forme suivante

$$seq \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x_{s_0} = fcall\ get_{s_0}(x_{mrk}) \\ x'_{s_0} = fcall\ get_{s_0}(x'_{mrk}) \\ pcall\ copy_{s_0}(x_{s_0}, x'_{s_0}) \end{array} \right.$$

Soit H un tas, et F un frame tels que décrits dans l'énoncé. On va réduire la séquence et propager les résultats.

$$\left(\ seq\ \right)_{H,F} \rightsquigarrow^* \left(\begin{array}{l} x'_{s_0} = fcall\ get_{s_0}(x'_{mrk}) \\ pcall\ copy_{s_0}(x_{s_0}, x'_{s_0}) \end{array} \right)_{H,F_0}$$

avec $F_0 \stackrel{\text{df}}{=} F \oplus \beta_0$. Par définition de la spécification de la fonction get_{s_0} on a :

$$\beta_0 \stackrel{\text{df}}{=} \{x_{s_0} \mapsto p_{s_0}\}$$

$$p_{s_0}{\downarrow_H} \subset p_{mrk}{\downarrow_H}$$

$$[\![H, p_{s_0}]\!]^{s_0} = [\![H, p_{mrk}]\!]^{mrk}(s_0)$$

On continue à réduire la séquence :

$$\left(\begin{array}{l} x'_{s_0} = fcall\ get_{s_0}(x'_{mrk}) \\ pcall\ copy_{s_0}(x_{s_0}, x'_{s_0}) \end{array} \right)_{H,F_0} \rightsquigarrow^* \left(\ pcall\ copy_{s_0}(x_{s_0}, x'_{s_0})\ \right)_{H,F_1}$$

avec $F_1 \stackrel{\text{df}}{=} F_0 \oplus \beta_1$. De même, par définition de la spécification de la fonction get_{s_0} on a :

$$\beta_1 \stackrel{\text{df}}{=} \{x'_{s_0} \mapsto p'_{s_0}\}$$

$$p'_{s_0}{\downarrow_H} \subset p'_{mrk}{\downarrow_H}$$

$$[\![H, p'_{s_0}]\!]^{s_0} = [\![H, p'_{mrk}]\!]^{mrk}(s_0)$$

On effectue la dernière réduction :

$$\left(\ pcall\ copy_{s_0}(x_{s_0}, x'_{s_0})\ \right)_{H,F_1} \rightsquigarrow^* \left(\ skip\ \right)_{H \oplus H_0, F_1}$$

Et par définition de la procédure $copy_{s_0}$, on a :

$$dom(H) \cap dom(H_0) \subseteq p'_{s_0}{\downarrow_H}$$

$$[\![H, p_{s_0}]\!]^{s_0} = [\![H \oplus H_0, p_{s_0}]\!]^{s_0} = \left[\!\left[H \oplus H_0, p'_{s_0}\right]\!\right]^{s_0}$$

On a bien le résultat souhaité :

$$\beta \stackrel{\text{df}}{=} \beta_0 \oplus \beta_1 = \{x_{s_0} \mapsto p_{s_0}, x'_{s_0} \mapsto p'_{s_0}\}$$

$$dom(H) \cap dom(H_0) \subseteq p'_{s_0}{\downarrow_H} \subseteq p'_{mrk}{\downarrow_H}$$

$$[\![H, p_{mrk}]\!]^{mrk}(s_0) = [\![H, p_{s_0}]\!]^{s_0} = [\![H \oplus H_0, p_{s_0}]\!]^{s_0} = \left[\!\left[H \oplus H_0, p'_{s_0}\right]\!\right]^{s_0}$$

Pour le cas inductif, on suppose le lemme est vrai pour $n-1$, c'est-à-dire que :

$$\left(\begin{array}{l} x_{s_0} = fcall\ get_{s_0}(x_{mrk}) \\ x'_{s_0} = fcall\ get_{s_0}(x'_{mrk}) \\ pcall\ copy_{s_0}(x_{s_0}, x'_{s_0}) \\ \ldots \\ x_{s_{n-1}} = fcall\ get_{s_{n-1}}(x_{mrk}) \\ x'_{s_{n-1}} = fcall\ get_{s_{n-1}}(x'_{mrk}) \\ pcall\ copy_{s_{n-1}}(x_{s_{n-1}}, x'_{s_{n-1}}) \end{array}\right)_{H,F} \rightsquigarrow^* \left(\ skip\ \right)_{\substack{H\oplus H_0,\\ F\oplus\beta_0}}$$

avec comme conséquences :

$$\beta_0 \stackrel{\text{df}}{=} \left\{ \begin{array}{c} x_{s_0} \mapsto p_{s_0}, \ldots, x_{s_{n-1}} \mapsto p_{s_{n-1}}, \\ x'_{s_0} \mapsto p'_{s_0}, \ldots, x'_{s_{n-1}} \mapsto p'_{s_{n-1}} \end{array} \right\} \tag{A.4}$$

$$dom(H) \cap dom(H_0) \subseteq p'_{s_0}{\downarrow_H} \cup \cdots \cup p'_{s_{n-1}}{\downarrow_H} \subseteq p'_{mrk}{\downarrow_H} \tag{A.5}$$

$$[\![H, p_{mrk}]\!]^{mrk}(s_i) = [\![H, p_{s_i}]\!]^{s_i} = [\![H \oplus H_0, p_{s_i}]\!]^{s_i} = [\![H \oplus H_0, p'_{s_i}]\!]^{s_i} \quad \text{pour } 0 \leq i \leq n-1 \tag{A.6}$$

Maintenant on va prouver que le lemme est vrai pour n. On applique l'hypothèse d'induction sur notre but initial, et on obtient :

$$\left(\begin{array}{l} x_{s_0} = fcall\ get_{s_0}(x_{mrk}) \\ x'_{s_0} = fcall\ get_{s_0}(x'_{mrk}) \\ pcall\ copy_{s_0}(x_{s_0}, x'_{s_0}) \\ \ldots \\ x_{s_{n-1}} = fcall\ get_{s_{n-1}}(x_{mrk}) \\ x'_{s_{n-1}} = fcall\ get_{s_{n-1}}(x'_{mrk}) \\ pcall\ copy_{s_{n-1}}(x_{s_{n-1}}, x'_{s_{n-1}}) \\ x_{s_n} = fcall\ get_{s_n}(x_{mrk}) \\ x'_{s_n} = fcall\ get_{s_n}(x'_{mrk}) \\ pcall\ copy_{s_n}(x_{s_n}, x'_{s_n}) \end{array}\right)_{H,F} \rightsquigarrow^* \left(\begin{array}{l} x_{s_n} = fcall\ get_{s_n}(x_{mrk}) \\ x'_{s_n} = fcall\ get_{s_n}(x'_{mrk}) \\ pcall\ copy_{s_n}(x_{s_n}, x'_{s_n}) \end{array}\right)_{\substack{H\oplus H_0,\\ F_0}}$$

avec $F_0 \stackrel{\text{df}}{=} F \oplus \beta_0$ et les conséquences (A.4), (A.5), (A.6) ci-dessus. On continue à réduire de la même manière que dans le cas de base.

$$\left(\begin{array}{l} x_{s_n} = fcall\ get_{s_n}(x_{mrk}) \\ x'_{s_n} = fcall\ get_{s_n}(x'_{mrk}) \\ pcall\ copy_{s_n}(x_{s_n}, x'_{s_n}) \end{array}\right)_{\substack{H\oplus H_0,\\ F\oplus\beta_0}} \rightsquigarrow^* \left(\ pcall\ copy_{s_n}(x_{s_n}, x'_{s_n})\ \right)_{\substack{H\oplus H_0,\\ F_1}}$$

avec $F_1 \stackrel{\text{df}}{=} F_0 \oplus \beta_1$ et

$$\beta_1 \stackrel{\text{df}}{=} \{x_{s_n} \mapsto p_{s_n}, x'_{s_n} \mapsto p'_{s_n}\}$$

$$p_{s_n}{\downarrow_H} \subset p_{mrk}{\downarrow_H}$$

$$p'_{s_n}{\downarrow_H} \subset p'_{mrk}{\downarrow_H}$$

$$[\![H \oplus H_0, p_{s_n}]\!]^{s_n} = [\![H \oplus H_0, p_{mrk}]\!]^{mrk}(s_n)$$

$$[\![H \oplus H_0, p'_{s_n}]\!]^{s_n} = [\![H \oplus H_0, p'_{mrk}]\!]^{mrk}(s_n)$$

Or comme les places sont disjointes, on a

$$dom(H_0) \cap p_{s_n}\!\downarrow_H = dom(H_0) \cap p'_{s_n}\!\downarrow_H = \emptyset$$

Les conclusions (A.5) et (A.6) sont préservées, car on n'a pas modifié le tas. Les x_{s_i} sont tous différents pour $0 \leq i \leq n$ (SSA), donc

$$\beta_0 \oplus \beta_1 = \left\{ \begin{array}{l} x_{s_0} \mapsto p_{s_0}, \ldots, x_{s_n} \mapsto p_{s_n}, \\ x'_{s_0} \mapsto p'_{s_0}, \ldots, x'_{s_n} \mapsto p'_{s_n} \end{array} \right\}$$

De plus, on a :

$$[\![H, p_{s_n}]\!]^{s_n} = [\![H, p_{mrk}]\!]^{mrk}(s_n) = [\![H \oplus H_0, p_{s_n}]\!]^{s_n}$$

$$[\![H, p'_{s_n}]\!]^{s_n} = [\![H, p'_{mrk}]\!]^{mrk}(s_n) = [\![H \oplus H_0, p'_{s_n}]\!]^{s_n}$$

Car la réduction de l'hypothèse d'induction n'a pas pu modifier les places pointées par p_{s_n} et p'_{s_n} (elles n'appartiennent pas au domaine des modifications (A.5)). On finalise les réductions :

$$\left(\ \mathit{pcall}\ \mathit{copy}_{s_n}(x_{s_n}, x'_{s_n})\ \right)_{F_1}^{H \oplus H_0,} \rightsquigarrow^* \left(\ \mathit{skip}\ \right)_{F_1}^{H \oplus H_0 \oplus H_1,}$$

Par la spécification de la procédure copy_{s_n}, on obtient :

$$dom(H \oplus H_0) \cap dom(H_1) \subseteq p'_{s_n}\!\downarrow_H$$

$$[\![H, p_{s_n}]\!]^{s_n} = [\![H \oplus H_0 \oplus H_1, p_{s_n}]\!]^{s_n} = [\![H \oplus H_0 \oplus H_1, p'_{s_n}]\!]^{s_n}$$

Au final on obtient le résultat souhaité car la condition sur les domaines nous assure qu'on n'a pas modifié les places $s_0, \ldots, s_{n-1}$ (donc on préserve les interprétations) et l'opération $\oplus$ est associative.

$$\beta = \beta_0 \oplus \beta_1 = \left\{ \begin{array}{l} x_{s_0} \mapsto p_{s_0}, \ldots, x_{s_n} \mapsto p_{s_n}, \\ x'_{s_0} \mapsto p'_{s_0}, \ldots, x'_{s_n} \mapsto p'_{s_n} \end{array} \right\}$$

$$dom(H) \cap dom(H_0 \oplus H_1) \subseteq p'_{s_0}\!\downarrow_H \cup \cdots \cup p'_{s_n}\!\downarrow_H \subseteq p'_{mrk}\!\downarrow_H$$

$$\begin{array}{rcl} [\![H, p_{mrk}]\!]^{mrk}(s_i) & = & [\![H, p_{s_i}]\!]^{s_i} \\ & = & [\![H \oplus H_0 \oplus H_1, p_{s_i}]\!]^{s_i} \\ & = & [\![H \oplus H_0 \oplus H_1, p'_{s_i}]\!]^{s_i} \\ & & \text{pour } 0 \leq i \leq n \end{array}$$

□

Le lemme qui suit vérifie que le corps de la fonction copy_{mrk} se comporte bien. Une fois qu'on l'aura prouvé, le résultat qu'on cherche sera immédiat.

Lemme A.3. *Soit M un marquage, H un tas bien formé, F un frame, x_{mrk} une variable et p_{mrk} un pointeur de H sur une structure de marquage tels que $p_{mrk}\!\downarrow_H = dom(H)$, $F(x_{mrk}) = p_{mrk}$ et $[\![H, p_{mrk}]\!]^{mrk} = M$. Si on réduit le corps de la fonction copy_{mrk},* i.e., *si*

$$\left(\ P(\mathit{copy}_{mrk}(\!|\, x_{mrk} \,|\!))\ \right)_{H,F} \rightsquigarrow^* \left(\ p'_{mrk}\ \right)_{H \oplus H'}$$

alors

$$dom(H) \cap dom(H') = \emptyset \tag{A.7}$$

$$p'_{mrk}{\downarrow_{H \oplus H'}} \subseteq dom(H') \tag{A.8}$$

$$[\![H, p_{mrk}]\!]^{mrk} = [\![H \oplus H', p_{mrk}]\!]^{mrk} = [\![H \oplus H', p'_{mrk}]\!]^{mrk} \tag{A.9}$$

Le résultat (A.7) nous assure qu'on alloue un nouveau tas. Le résultat (A.8) nous assure que le nouveau marquage pointé par p'_{mrk} ne partage pas de mémoire avec le marquage pointé par p_{mrk}. Et le dernier résultat (A.9) assure qu'on copie bien la structure de marquage.

Preuve. On fait cette preuve en réduisant la séquence $P(copy_{mrk}(\!| x_{mrk} |\!))$. Par définition :

$$P(copy_{mrk}(\!| x_{mrk} |\!)) \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x'_{mrk} = alloc\ struct(t_{s_0}, \ldots, t_{s_n}) \\ \\ \left.\begin{array}{l} x_{s_i} = fcall\ get_{s_i}(x_{mrk}) \\ x'_{s_i} = fcall\ get_{s_i}(x'_{mrk}) \\ pcall\ copy_{s_i}(x_{s_i}, x'_{s_i}) \end{array}\right\} \text{pour } 0 \leq i \leq n \\ \\ ret\ x'_{mrk} \end{array} \right.$$

Soit H un tas et F un frame tels que décrit dans l'énoncé. On commence à réduire :

$$\left(\ P(copy_{mrk}(\!| p_{mrk} |\!))\ \right)_{H,F} \rightsquigarrow^* \left(\begin{array}{l} x_{s_0} = fcall\ get_{s_0}(x_{mrk}) \\ x'_{s_0} = fcall\ get_{s_0}(x'_{mrk}) \\ pcall\ copy_{s_0}(x_{s_0}, x'_{s_0}) \\ \ldots \\ x_{s_n} = fcall\ get_{s_n}(x_{mrk}) \\ x'_{s_n} = fcall\ get_{s_n}(x'_{mrk}) \\ pcall\ copy_{s_n}(x_{s_n}, x'_{s_n}) \\ \\ ret\ x'_{mrk} \end{array} \right)_{H \oplus H_0, F_0}$$

où $F_0 \stackrel{\text{df}}{=} F \oplus \beta_0$ et $\beta_0 \stackrel{\text{df}}{=} \{x'_{mrk} \mapsto p'_{mrk}\}$. Par la définition de l'instruction *alloc* et la fonction *new* on a :

$$dom(H) \cap dom(H_0) = \emptyset \tag{A.10}$$

$$p'_{mrk}{\downarrow_{H \oplus H_0}} = dom(H_0) \tag{A.11}$$

et on sait que H_0 est bien formé. On applique le lemme A.2, les hypothèses nécessaires étant respectées :

$$p_{mrk}{\downarrow_{H \oplus H_0}} \cup p'_{mrk}{\downarrow_{H \oplus H_0}} = dom(H \oplus H_0)$$

$$[\![H \oplus H_0, p_{mrk}]\!]^{mrk} = M$$

$$\beta_{F_0} = \{x_{mrk} \mapsto p_{mrk}, x'_{mrk} \mapsto p'_{mrk}\}$$

Après l'application du lemme, on obtient :

$$\left(\begin{array}{l} x_{s_0} = fcall\ get_{s_0}(x_{mrk}) \\ x'_{s_0} = fcall\ get_{s_0}(x'_{mrk}) \\ pcall\ copy_{s_0}(x_{s_0}, x'_{s_0}) \\ \dots \\ x_{s_n} = fcall\ get_{s_n}(x_{mrk}) \\ x'_{s_n} = fcall\ get_{s_n}(x'_{mrk}) \\ pcall\ copy_{s_n}(x_{s_n}, x'_{s_n}) \\ \\ ret\ x'_{mrk} \end{array}\right)_{H\oplus H_0, F_0} \rightsquigarrow^* \left(\ ret\ x'_{mrk}\ \right)_{H\oplus H_0\oplus H_1, F_1}$$

où $F_1 \stackrel{\text{df}}{=} F_0 \oplus \beta_1$ avec

$$\beta_1 \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x_{s_0} \mapsto p_{s_0}, \dots, x_{s_n} \mapsto p_{s_n}, \\ x'_{s_0} \mapsto p'_{s_0}, \dots, x'_{s_n} \mapsto p'_{s_n} \end{array} \right\} \tag{A.12}$$

$$dom(H \oplus H_0) \cap dom(H_1) \subseteq p'_{s_0}{\downarrow_{H\oplus H_0}} \cup \dots \cup p'_{s_n}{\downarrow_{H\oplus H_0}} \subseteq p'_{mrk}{\downarrow_{H\oplus H_0}} \tag{A.13}$$

$$\begin{aligned} [\![H \oplus H_0, p_{mrk}]\!]^{mrk}(s_i) &= [\![H \oplus H_0, p_{s_i}]\!]^{s_i} \\ &= [\![H \oplus H_0 \oplus H_1, p_{s_i}]\!]^{s_i} \\ &= [\![H \oplus H_0 \oplus H_1, p'_{s_i}]\!]^{s_i} \\ &\qquad \text{pour } 0 \le i \le n \end{aligned} \tag{A.14}$$

On fait la dernière réduction :

$$\left(\ ret\ x'_{mrk}\ \right)_{H\oplus H_0\oplus H_1, F_1} \rightsquigarrow \left(\ p'_{mrk}\ \right)_{H\oplus H_0\oplus H_1, F_1}$$

Le résultat (A.13) implique que

$$dom(H \oplus H_0) \cap dom(H_1) \subseteq dom(H_0)$$

car $p'_{mrk}{\downarrow_{H\oplus H_0}} \subseteq dom(H_0)$. On obtient donc les deux premières conséquences du lemme car l'opération $\oplus$ est associative :

$$dom(H) \cap dom(H_0 \oplus H_1) = dom(H) \cap dom(H_0) = \emptyset$$

$$p'_{mrk}{\downarrow_{H\oplus H_0\oplus H_1}} = p'_{mrk}{\downarrow_{H_0\oplus H_1}} \subseteq dom(H_0 \oplus H_1)$$

La troisième conséquence

$$[\![H, p_{mrk}]\!]^{mrk} = [\![H \oplus H_0 \oplus H_1, p_{mrk}]\!]^{mrk} = [\![H \oplus H_0 \oplus H_1, p'_{mrk}]\!]^{mrk}$$

est vraie par la propriété de cohérence de la structure de marquage en utilisant le résultat (A.14). □

Proposition A.4. *la fonction $copy_{mrk}$ vérifie bien sa spécification,* i.e., *pour un marquage M, un tas H, un frame F, x_{mrk} une variable et p_{mrk} un pointeur tels que $F(x_{mrk}) = p_{mrk}$ et $[\![H, p_{mrk}]\!]^{mrk} = M$, on a :*

$$\left(\ fcall\ copy_{mrk}(x_{mrk})\ \right)_{H,F} \rightsquigarrow \left(\ p'_{mrk}\ \right)_{H\oplus H'}$$

$$dom(H') \cap dom(H) = \emptyset$$

$$p'_{mrk}{\downarrow_{H\oplus H'}} \subseteq dom(H')$$

$$[\![H, p_{mrk}]\!]^{mrk} = [\![H \oplus H', p_{mrk}]\!]^{mrk} = [\![H \oplus H', p'_{mrk}]\!]^{mrk}$$

Preuve. Ceci est une conséquence immédiate du lemme A.3, en utilisant la règle de réduction de l'instruction *fcall*. □

A.1.4 La fonctions *cons*

L'implantation de la fonction $cons_{mrk}$ est définie par

$$P(cons_{mrk}(\!|\,|\!)) \stackrel{\text{df}}{=} \left\{ \begin{array}{l} x_{mrk} = alloc\ struct(t_0, \ldots, t_n) \\ \\ x_{s_1} = fcall\ get_{s_1}(x_{mrk}) \\ pcall\ clear_{s_1}(x_{s_1}) \\ \ldots \\ x_{s_n} = fcall\ get_{s_n}(x_{mrk}) \\ pcall\ clear_{s_n}(x_{s_n}) \end{array} \right.$$

Pour prouver la correction de cette fonction, on se base sur un lemme qui montre que la séquence des fonctions *get* et *clear* "nettoie" bien le marquage. Une fois ce lemme prouvé il nous suffira de montrer qu'on alloue un nouveau marquage d'interprétation quelconque puis on le "nettoie".

Lemme A.5. *La séquence des fonctions get et clear "nettoie" bien la structure de marquage,* i.e.*, si H est un tas, F est un frame, x_{mset} est une variable et p_{mrk} est un pointeur sur une structure de marquage, tels que $dom(H) = p_{mrk}{\downarrow_H}$ et $F(x_{mrk}) = p_{mrk}$, alors*

$$\left(\begin{array}{l} x_{s_0} = fcall\ get_{s_0}(x_{mrk}) \\ pcall\ clear_{s_0}(x_{s_0}) \\ \ldots \\ x_{s_n} = fcall\ get_{s_n}(x_{mrk}) \\ pcall\ clear_{s_n}(x_{s_n}) \end{array} \right)_{H,F} \rightsquigarrow \left(\ skip\ \right)_{H \oplus H', F \oplus \beta}$$

et

$$\beta = \{x_{s_0} \mapsto p_{s_0}, \ldots, x_{s_n} \mapsto p_{s_n}\}$$

$$[\![H \oplus H', p_{mrk}]\!]^{mrk}(s_i) = [\![H \oplus H', p_{s_i}]\!]^{s_i} = \emptyset \quad pour\ 0 \leq i \leq n$$

Preuve. La preuve se fait par induction sur n. Pour le cas de base, on a :

$$\left(\begin{array}{l} x_{s_0} = fcall\ get_{s_0}(x_{mrk}) \\ pcall\ clear_{s_0}(x_{s_0}) \end{array} \right)_{H,F} \rightsquigarrow \left(\ pcall\ clear_{s_0}(x_{s_0})\ \right)_{H,F_0}$$

avec $F_0 = F \oplus \{x_{s_0} \mapsto p_{s_0}\}$ et $[\![H, p_{mrk}]\!]^{mrk}(s_0) = [\![H, p_{s_0}]\!]^{s_0}$. On continue en réduisant grâce à la spécification de la fonction $clear_{s_0}$:

$$\left(\ pcall\ clear_{s_0}(x_{s_0})\ \right)_{H,F_0} \rightsquigarrow \left(\ skip\ \right)_{H \oplus H_0, F_0}$$

avec $[\![H \oplus H_0, p_{s_0}]\!]^{s_0} = \emptyset$. On conclue par la propriété de *cohérence* de la structure de marquage.

Pour le cas inductif, on commence par appliquer l'hypothèse d'induction pour réduire les $n-1$ premières paires d'instructions.

$$\left(\begin{array}{l} x_{s_0} = \mathit{fcall}\ \mathit{get}_{s_0}(x_{mrk}) \\ \mathit{pcall}\ \mathit{clear}_{s_0}(x_{s_0}) \\ \ldots \\ x_{s_{n-1}} = \mathit{fcall}\ \mathit{get}_{s_{n-1}}(x_{mrk}) \\ \mathit{pcall}\ \mathit{clear}_{s_{n-1}}(x_{s_{n-1}}) \\ x_{s_n} = \mathit{fcall}\ \mathit{get}_{s_n}(x_{mrk}) \\ \mathit{pcall}\ \mathit{clear}_{s_n}(x_{s_n}) \end{array}\right)_{H,F} \rightsquigarrow \left(\begin{array}{l} x_{s_n} = \mathit{fcall}\ \mathit{get}_{s_n}(x_{mrk}) \\ \mathit{pcall}\ \mathit{clear}_{s_n}(x_{s_n}) \end{array}\right)_{H\oplus H_0, F_0}$$

avec

$$F_0 = F \oplus \{x_{s_0} \mapsto p_{s_0}, \ldots, x_{s_{n-1}} \mapsto p_{s_{n-1}}\} \tag{A.15}$$

$$[\![H \oplus H_0, p_{mrk}]\!]^{mrk}(s_i) = [\![H \oplus H_0, p_{s_i}]\!]^{s_i} = \emptyset \quad \text{pour } 0 \leq i \leq n-1 \tag{A.16}$$

On réduit la dernière paire d'instructions de la même manière que dans le cas de base :

$$\left(\begin{array}{l} x_{s_n} = \mathit{fcall}\ \mathit{get}_{s_n}(x_{mrk}) \\ \mathit{pcall}\ \mathit{clear}_{s_n}(x_{s_n}) \end{array}\right)_{H\oplus H_0, F_0} \rightsquigarrow \left(\ \mathit{skip}\ \right)_{H\oplus H_0\oplus H_1, F_1}$$

avec $F_1 = F_0 \oplus \{x_{s_n} \mapsto p_{s_n}\}$ et

$$\begin{array}{rcl} [\![H \oplus H_0 \oplus H_1, p_{mrk}]\!]^{mrk}(s_n) & = & [\![H \oplus H_0 \oplus H_1, p_{s_n}]\!]^{s_n} \\ & = & \emptyset \end{array}$$

de plus par la propriété de séparation les places sont disjointes et donc les résultats (A.15) et (A.16) sont préservés pour le tas $H \oplus H_0 \oplus H_1$. On conclue par la propriété de cohérence. □

Proposition A.6. *La fonction $cons_{mrk}$ vérifie bien sa spécification,* i.e.*,*

$$\left(\ \mathit{fcall}\ \mathit{cons}_{mrk}()\ \right)_{\emptyset,F} \rightsquigarrow \left(\ p_{mrk}\ \right)_H$$

$$[\![H, p_{mrk}]\!]^{mrk} = \emptyset$$

Preuve. Conséquence du lemme précédent en utilisant la définition de la règle alloc. □

B | Preuves du chapitre 6

Toutes les preuves sont faites dans le contexte d'un réseau de Petri $N = (S, T, \ell)$ avec les suppositions du chapitre 6.

B.1

Preuves des résultats auxiliaires (section 6.3)

B.1.1 Proposition 6.2

Soit M un marquage pid-cohérent d'un réseau de Petri. Si $t \in R(M)$ est une représentation de M, alors

1. *$mrk(N(t))$ est pid-cohérent ;*
2. *M et $mrk(\mathcal{N}(t))$ sont h_N^t équivalents,* i.e., *$M \sim mrk(\mathcal{N}(t))$.* □

Preuve.

1. Parce que, pour chaque paire $\langle \pi, \pi_\eta \rangle \in M(s_\eta)$ le noeud dans t à la position π_η est le fils le plus à droite du noeud à la position π (définition 6.11), on a $h_N^t(\pi_\eta)$ le plus grand pid qui satisfait $h_N^t(\pi) \lhd_1 h_N^t(\pi_\eta)$ et donc $mrk(N(t))$ est pid-cohérent.
2. h_N^t est une bijection qui assigne aux pids de pid_t les pids dans $pid_{\mathcal{N}(t)}$. Cette bijection préserve les relations $\lhd_1$ et $\lhd$ parce qu'elle ne change que les noms de pids et non leur relations, par conséquent les condition de la définition 6.4 sont satisfaites.

□

B.1.2 Proposition 6.3

Soit M un marquage pid-cohérent d'un réseau de Petri. Si $t, t' \in R_s(M)$ alors t' est une permutation de t. □

Preuve. Par définition $R_s(M) \subseteq R(M)$, on sait que

$$pid_t = pid_{t'} = pid_M \tag{B.1}$$

$$mrk(t) = mrk(t') = M \tag{B.2}$$

si t et t' diffèrent en nombre de branches, ou diffèrent en la longueur de leur branches (longueur minimale car pid-trees réduits) alors (B.1) est violée. S'ils diffèrent en leurs marquages alors (B.2) est violée. Donc les pid-trees t et t' ne peuvent différer que par l'ordonnancement de leurs enfants. □

Le corollaire 6.4 est immédiat.

B.1.3 Proposition 6.6

Soit $t \in R_{so}(M)$ un pid-tree, alors pour tout $t' \in R_s(M)$ on a :

$$t \equiv t' \Leftrightarrow t' \in R_{so}(M)$$

□

Preuve.

($\Rightarrow$) $t \equiv t'$, t' est réduit et pid-free ordonné donc il est dans $R_{so}(M)$.

($\Leftarrow$) $t' \in R_{so}$, donc t' est réduit et pid-free ordonné, t et t' ne peuvent différer qu'en des noeuds équivalents, sinon t' ne serait pas dans $R_{so}(M)$, par conséquent $t \equiv t'$. □

□

B.2

Preuves de l'algorithme de construction (section 6.4)

B.2.1 Proposition 6.9

L'algorithme 6.1 construit une représentation sous forme de pid-tree d'un marquage pid-cohérent.

(schéma). On considère un marquage pid-cohérent M. L'algorithme démarre avec l' pid-tree $\langle \emptyset, \langle \rangle \rangle$.

Si le marquage M est vide alors aucune modification n'est fait, on renvoie ce pid-tree. C'est le plus petit pid-tree bien formé, qui est une représentation de M.

Sinon, s'il y a des jetons dans les places différentes de s_η on les ajoute à gauche, les fils les plus à droite ne sont donc pas modifiés (s'ils existent). Le pid-tree reste bien formé et une représentation de M puisqu'on ajoute les jetons dans l'arbre.

Pour chaque futur-pid, on ajoute un sous arbre à gauche d'un noeud. Parce que le marquage est pid-cohérent, il ne peut y avoir qu'un seul fils de ce type par noeud. De plus le placer en tant que fils le plus à droite force la troisième condition de la définition 6.11 à être vraie. Le jeton correspondant de la place s_η est aussi ajouté à l'arbre, il est donc bien formé et une représentation de M. □

B.2.2 Proposition 6.10

La fonction de réduction $\mathcal{S}$ transforme une représentation d'un marquage en une représentation réduite du même marquage. □

Preuve. Pour respecter la définition des pid-tree bien formés, on ne supprime que les noeuds correspondants aux pids non référencés, par conséquent l'arbre résultant sera bien formé. Le marquage reste le même parce que les noeuds $\perp$ se comportent comme des marquages vides. De plus, la transformation ne peut modifier les noeuds $\perp$ correspondants aux futur-pids car il s'agit de feuilles. L'arbre résultant ne possède plus de noeuds internes étiquetés par $\perp$ et donc pas de pids non référencés, il s'agit d'un arbre réduit. □

B.2.3 Proposition 6.11

L'application de la fonction de normalisation $\mathcal{N}$ sur un pid-tree produit un pid-tree normalisé.

Preuve. Immédiat de part l'implémentation qui est basée directement sur la définition de la normalisation de pid-trees. □

B.3

Preuves des algorithmes (section 6.6)

L'algorithme 6.2 est correct et termine.

(schéma). On vérifie juste que cette fonction renvoie un espace d'états minimal contenant un marquage équivalent à M.

On construit un pid-tree réduit et ordonné du marquage M. On énumère toutes les représentations réduites du marquage M (corollaire 6.7), on normalise chacune de ces représentations et construit le marquage résultant. Si un de ces marquages correspond à un marquage dans l'espace d'états, alors on a déjà un état équivalent (théorème 6.8). Sinon, on ajoute une des représentations à l'espace d'états, ce qui est aussi correct de par le théorème 6.8.

L'espace d'états est minimal car sinon on aurait trouvé un état équivalent dans l'ensemble d'états.

La terminaison est triviale car l'ensemble des représentations réduites et ordonnées des pid-trees est fini. □

L'algorithme 6.3 est correct et termine.

Preuve. Trivial car on ajoute toujours un marquage équivalent à M (proposition 6.2).

De plus l'espace d'états est fini car chaque marquage possède un nombre fini de représentations sous forme de pid-trees réduits, et donc un nombre fini de représentations réduites et ordonnées. □

C Évaluation des performances de Neco

Dans cette annexe on présente les résultats des études de cas du chapitre 7 qui ont été omis. De plus on réalise une autre série d'exécutions avec l'interpréteur *PyPy* [149] pour évaluer l'intérêt du backend Cython. En effet, *PyPy* est une implémentation alternative du langage Python qui réalise de la compilation à la volée pour de meilleures performances en temps d'exécution et en mémoire.

Les Résultats omis du chapitre 7 La table C.1 présente les résultats du backend Python sur les modèles bas niveau.

Les exécutions avec *PyPy* Les tests présentés ici ont été réalisés avec PyPy pour l'exploration d'espaces d'états mais l'étape de compilation a été réalisée avec Python pour des raisons de compatibilité avec la bibliothèque SNAKES.

Les résultats pour les optimisations sur les réseaux bas niveau sont donnés dans la table C.2. Les résultats pour les réseaux haut niveau et réductions de contrôle de flot sont données dans la table C.3. Les résultats pour le modèle client-serveur sont données dans la table C.4. Finalement, les résultats pour les graphes de contrôle de flot sont donnés dans les tables C.5 et C.6.

On observe les mêmes tendances qu'avec Python pour les optimisations, cependant les temps de calculs sont plus rapides avec PyPy mais restent bien supérieurs aux temps atteints avec Cython.

Les fichiers produits par Neco La compilation produit différents fichiers selon le backend utilisé. Le backend Python produit un fichier `net.py` qui est un module python correspondant au moteur d'exploration. Selon le modèle, la taille de ce fichier varie, comme on peut le voir sur les figures C.7, C.8, C.9, C.10, C.11, C.12. On remarque qu'en général on obtient des fichiers de même taille pour les configurations SIMPLE, OPT et OPTPACK, en revanche lorsqu'on utilise l'élimination de contrôle de flot (FLOW) alors le fichier produit est beaucoup plus petit.

Le backend Cython produit plusieurs fichiers. Tout d'abord deux fichiers Cython, `net.pyx` et `net.pxd`, ils sont produits directement par Neco, puis compilés par le compilateur Cython pour produire deux fichiers `net.h` et `net.cpp`

Python	Temps comilation (s)		Temps exploration (s)		Accélération	Nombre d'états	
	SIMPLE	OPT	SIMPLE	OPT	OPT par rapport à SIMPLE	SIMPLE	OPT
CSRepetitions PT 02	0.098	0.099	0.146	0.147	x0.998	1 872	1 872
Dekker PT 010	0.607	0.609	6.651	6.594	x1.009	6 144	6 144
Dekker PT 015	1.667	1.663	921.498	923.739	x0.998	278 528	278 528
LamportFastMutEx PT 2	0.521	0.523	0.047	0.047	x1.005	380	380
LamportFastMutEx PT 3	1.064	1.059	5.035	5.030	x1.001	19 742	19 742
LamportFastMutEx PT 4	2.092	2.089	1,045.760	1,038.740	x1.007	1 914 784	1 914 784
NeoElection PT 2	9.231	9.222	0.174	0.173	x1.001	241	241
Philosophers 20	0.165	0.166	5.460	5.490	x0.995	15 127	15 127
Philosophers 25	0.222	0.224	94.002	93.926	x1.001	167 761	167 761
Philosophers 30	0.291	0.291	TIME	TIME	-	-	-
RessAllocation PT R002C002	0.021	0.021	0.000	0.000	x1.012	8	8
RessAllocation PT R003C002	0.030	0.030	0.000	0.000	x0.981	20	20
RessAllocation PT R003C003	0.047	0.047	0.004	0.004	x1.000	92	92
RessAllocation PT R003C005	0.096	0.097	0.122	0.135	x0.905	1 200	1 200
RessAllocation PT R003C010	0.217	0.218	312.675	313.789	x0.996	823 552	823 552
RessAllocation PT R005C002	0.050	0.050	0.004	0.004	x1.002	112	112
RessAllocation PT R010C002	0.120	0.120	0.644	0.644	x1.000	6 144	6 144
RessAllocation PT R015C002	0.187	0.188	58.786	58.633	x1.003	278 528	278 528
RwMutex PT r0010w0010	0.237	0.239	0.384	0.385	x0.997	1 034	1 034
RwMutex PT r0010w0020	0.437	0.440	0.557	0.556	x1.002	1 044	1 044
RwMutex PT r0010w0050	1.280	1.279	0.943	0.950	x0.993	1 074	1 074
RwMutex PT r0010w0100	3.668	3.691	1.785	1.777	x1.004	1 124	1 124
RwMutex PT r0010w0500	62.125	62.253	12.639	13.080	x0.966	1 524	1 524
RwMutex PT r0020w0010	0.479	0.479	1,218.500	1,223.040	x0.996	1 048 586	1 048 586
SharedMemory PT 000005	0.211	0.212	0.354	0.348	x1.015	1 863	1 863
SharedMemory PT 000010	1.758	1.775	TIME	TIME	-	-	-
SimpleLoadBal PT 02	0.188	0.189	0.076	0.076	x1.000	832	832
SimpleLoadBal PT 05	0.964	0.969	32.034	32.043	x1.000	116 176	116 176

Table C.1 – Résultats des réductions par symétries pour le modèle *CFG* (backend : Python, variable : instances de sous réseau).

PyPy	Temps comilation (s)		Temps exploration (s)		Accélération OPT par rapport à SIMPLE	Nombre d'états	
	SIMPLE	OPT	SIMPLE	OPT		SIMPLE	OPT
CSRepetitions PT 02	0.259	0.259	0.305	0.305	x1.000	1,872	1,872
Dekker PT 010	1.530	1.493	5.246	5.397	x0.972	6,144	6,144
Dekker PT 015	3.042	3.130	time	time	-	-	-
LamportFastMutEx PT 2	1.265	1.238	0.154	0.154	x0.995	380	380
LamportFastMutEx PT 3	2.257	2.322	21.111	21.093	x1.001	19,742	19,742
LamportFastMutEx PT 4	3.176	3.198	time	time	-	-	-
NeoElection PT 2	9.500	9.871	0.426	0.426	x1.000	241	241
Philosophers 20	0.474	0.477	3.341	3.402	x0.982	15,127	15,127
Philosophers 25	0.735	0.722	28.477	33.807	x0.842	167,761	167,761
Philosophers 30	0.742	0.746	421.988	424.585	x0.994	1,860,498	1,860,498
RessAllocation PT R002C002	0.046	0.046	0.000	0.000	x1.003	8	8
RessAllocation PT R003C002	0.081	0.081	0.001	0.001	x1.001	20	20
RessAllocation PT R003C003	0.119	0.120	0.008	0.008	x1.012	92	92
RessAllocation PT R003C005	0.298	0.305	0.282	0.288	x0.980	1,200	1,200
RessAllocation PT R003C010	0.688	0.691	160.307	144.520	x1.109	823,552	823,552
RessAllocation PT R005C002	0.135	0.135	0.008	0.008	x1.000	112	112
RessAllocation PT R010C002	0.359	0.360	1.178	1.094	x1.077	6,144	6,144
RessAllocation PT R015C002	0.616	0.603	32.237	39.479	x0.817	278,528	278,528
RwMutex PT r0010w0010	0.735	0.733	0.490	0.505	x0.969	1,034	1,034
RwMutex PT r0010w0020	1.180	1.216	0.502	0.502	x0.999	1,044	1,044
RwMutex PT r0010w0050	2.582	2.589	2.706	2.851	x0.949	1,074	1,074
RwMutex PT r0010w0100	4.945	4.934	5.077	4.973	x1.021	1,124	1,124
RwMutex PT r0010w0500	?	?	mem	mem	-	mem	mem
RwMutex PT r0020w0010	1.378	1.428	time	time	-	time	time
SharedMemory PT 000005	0.641	0.642	0.647	0.645	x1.003	1,863	1,863
SharedMemory PT 000010	2.828	2.920	time	time	-	-	-
SimpleLoadBal PT 02	0.571	0.574	0.232	0.232	x1.000	832	832
SimpleLoadBal PT 05	2.194	2.270	27.358	27.336	x0.999	116,176	116,176

Table C.2 – Résultats d'optimisations pour réseaux bas niveau avec PyPy.

Temps comilation (s)			Temps exploration (s)			Accélération			Nombre d'états
SIMPLE	OPT	FLOW	SIMPLE	OPT	FLOW	OPT par rapport à SIMPLE	FLOW par rapport à SIMPLE	FLOW par rapport à OPT	
1.741	1.343	0.252	6.026	2.642	0.166	x2.281	x36.292	x15.914	2,704
3.892	2.879	0.386	698.127	354.771	9.658	-	-	x36.734	140,608
0.887	0.695	0.150	2.895	1.358	0.144	x2.131	x20.117	x9.439	2,704
1.959	1.427	0.185	338.790	203.804	5.850	-	-	x34.838	140,608
3.485	2.548	0.302	MEM	TIME	176.226	-	-	-	7,311,616
0.086	0.079	0.049	30.082	3.964	3.724	x7.590	x8.077	x1.064	59,049
0.096	0.089	0.052	108.258	9.899	7.443	x10.936	x14.544	x1.330	177147
0.108	0.099	0.057	561.184	32.849	18.253	-	-	x1.800	531,441
0.120	0.108	0.063	MEM	110.584	56.681	-	-	x1.951	1594323
0.171	0.162	0.124	2.641	1.032	0.900	x2.559	x2.936	x1.147	1,838
0.169	0.158	0.109	13.296	2.906	2.853	x4.575	x4.661	x1.019	7,502
0.196	0.180	0.123	69.134	7.900	7.266	x8.751	x9.515	x1.087	30,626
0.053	0.053	0.044	2.855	2.540	2.984	x1.124	x0.957	x0.851	1,234

Table C.3 – Résultats d'optimisations pour réseaux haut niveau avec PyPy.

PYPY	TEMPS COMILATION (s)				TEMPS EXPLORATION (s)				ACCÉLÉRATION PAR RAPPORT À OPT		NOMBRE D'ÉTATS				RÉDUCTION PAR RAPPORT À OPT	
	SIMPLE	OPT	SYM	SYM NO	SIMPLE	OPT	SYM	SYM NO	SYM	SYM NO	SIMPLE	OPT	SYM	SYM NO	SYM	SYM NO
c3n4	0.117	0.117	0.260	0.261	1.781	1.642	2.064	2.257	x0.796	x0.727	2 494	2 494	165	395	x15	x6
c3n5	0.148	0.148	0.176	0.175	2.359	2.359	2.273	2.322	x1.038	x1.016	3 664	3 664	220	579	x17	x6
c4n1	0.064	0.065	0.159	0.159	3.270	3.210	2.170	2.172	x1.480	x1.478	6 336	6 336	126	308	x50	x21
c4n2	0.126	0.126	0.147	0.147	6.058	6.051	3.331	3.497	x1.817	x1.730	16 216	16 216	210	758	x77	x21
c4n3	0.118	0.119	0.171	0.171	11.757	12.228	4.706	5.518	x2.598	x2.216	34 304	34 304	330	1 539	x104	x22
c4n4	0.129	0.129	0.214	0.215	19.814	20.151	6.658	8.360	x3.026	x2.410	64 056	64 056	495	2 815	x129	x23
c4n5	0.153	0.150	0.177	0.177	33.550	34.027	8.680	12.302	x3.920	x2.766	109 504	109 504	715	4 793	x153	x23
c5n1	0.066	0.066	0.160	0.160	34.299	34.716	7.822	6.021	x4.438	x5.766	96 384	96 384	252	1 447	x382	x67
c5n2	0.128	0.128	0.148	0.148	124.992	129.842	16.517	14.320	x7.861	x9.067	336 184	336 184	462	4 659	x728	x72
c5n3	0.120	0.120	0.171	0.172	348.077	343.174	33.779	30.585	x10.159	x11.220	897 024	897 024	792	11 499	x1133	x78
c5n4	0.132	0.132	0.218	0.267	MEM	MEM	58.486	66.094	-	-	MEM	MEM	1 287	24 337	-	-
c5n5	0.150	0.155	0.179	0.183	MEM	MEM	96.265	119.789	-	-	MEM	MEM	2 002	46 303	-	-

TABLE C.4 – Résultats des réductions par symétries pour le modèle *Client-Server* avec PyPy.

PYPY	Temps comilation (s)				Temps exploration (s)				Accélération par rapport à OPT		Nombre d'états				Réduction par rapport à OPT	
	SIMPLE	OPT	SYM	SYM NO	SIMPLE	OPT	SYM	SYM NO	SYM	SYM NO	SIMPLE	OPT	SYM	SYM NO	SYM	SYM NO
CFG 1 5	0.048	0.049	0.072	0.072	3.989	4.005	1.954	1.222	x2.049	x3.277	7 777	7 777	253	253	x31	x31
CFG 2 5	0.163	0.163	0.281	0.282	4.905	5.074	2.627	1.509	x1.931	x3.363	15 553	15 553	505	505	x31	x31
CFG 3 5	0.271	0.270	0.334	0.334	9.429	9.153	3.626	2.204	x2.524	x4.153	23 329	23 329	757	757	x31	x31
CFG 4 5	0.374	0.374	0.511	0.510	14.604	14.605	5.281	3.228	x2.766	x4.525	31 105	31 105	1 009	1 009	x31	x31
CFG 5 5	0.510	0.513	0.606	0.608	21.081	21.304	6.534	4.446	x3.260	x4.792	38 881	38 881	1 261	1 261	x31	x31
CFG 6 5	0.512	0.511	0.818	0.819	76.557	75.217	12.720	7.087	x5.913	x10.614	46 657	46 657	1 513	1 513	x31	x31
CFG 7 5	0.532	0.529	0.815	0.813	101.095	98.463	18.028	10.069	x5.462	x9.778	54 433	54 433	1 765	1 765	x31	x31
CFG 8 5	0.627	0.637	0.969	0.968	126.502	128.187	21.028	11.734	x6.096	x10.925	62 209	62 209	2 017	2 017	x31	x31
CFG 9 5	0.949	0.931	1.063	1.072	150.240	150.002	24.973	14.439	x6.007	x10.389	69 985	69 985	2 269	2 269	x31	x31
CFG 10 5	0.904	0.908	1.236	1.236	178.036	175.715	28.780	16.913	x6.105	x10.389	77 761	77 761	2 521	2 521	x31	x31
CFG 11 5	0.962	0.966	1.255	1.256	204.817	208.605	33.901	19.838	x6.153	x10.515	85 537	85 537	2 773	2 773	x31	x31
CFG 12 5	1.146	1.150	1.531	1.534	236.652	233.555	38.439	23.205	x6.076	x10.065	93 313	93 313	3 025	3 025	x31	x31
CFG 13 5	1.451	1.481	1.427	1.446	267.601	268.159	43.589	26.842	x6.152	x9.990	101 089	101 089	3 277	3 277	x31	x31
CFG 14 5	1.287	1.327	1.510	1.510	319.087	310.363	49.144	30.395	x6.315	x10.211	108 865	108 865	3 529	3 529	x31	x31
CFG 15 5	1.533	1.575	1.813	1.811	356.076	346.912	55.838	34.288	x6.213	x10.117	116 641	116 641	3 781	3 781	x31	x31
CFG 16 5	1.729	1.736	2.099	2.099	394.226	394.027	62.519	40.059	x6.303	x9.836	124 417	124 417	4 033	4 033	x31	x31
CFG 17 5	1.683	1.738	2.056	2.052	437.426	437.264	68.098	43.739	x6.421	x9.997	132 193	132 193	4 285	4 285	x31	x31
CFG 18 5	1.936	1.973	2.121	2.128	482.188	491.941	76.650	48.363	x6.418	x10.172	139 969	139 969	4 537	4 537	x31	x31
CFG 19 5	1.994	2.034	2.454	2.452	543.358	536.913	84.745	53.091	x6.336	x10.113	147 745	147 745	4 789	4 789	x31	x31
CFG 20 5	2.123	2.118	2.487	2.491	593.968	581.570	91.777	58.627	x6.337	x9.920	155 521	155 521	5 041	5 041	x31	x31

Table C.5 – Résultats des réductions par symétries pour le modèle *CFG* avec PyPy.

PYPY	TEMPS COMILATION (s)				TEMPS EXPLORATION (s)				ACCÉLÉRATION PAR RAPPORT À OPT		NOMBRE D'ÉTATS				RÉDUCTION PAR RAPPORT À	
	SIMPLE	OPT	SYM	SYM NO	SIMPLE	OPT	SYM	SYM NO	SYM	SYM NO	SIMPLE	OPT	SYM	SYM NO	SYM	SYM NO
CFG 10 1	0.905	0.907	1.265	1.234	0.099	0.097	0.188	0.146	x0.517	x0.666	61	61	61	61	x1	x1
CFG 10 2	0.906	0.907	1.236	1.240	0.410	0.411	0.757	0.588	x0.543	x0.700	361	361	211	211	x2	x2
CFG 10 3	0.904	0.906	1.240	1.239	2.919	2.980	2.901	2.220	x1.027	x1.342	2 161	2 161	561	561	x4	x4
CFG 10 4	0.911	0.906	1.235	1.237	23.145	22.974	9.856	6.184	x2.331	x3.715	12 961	12 961	1 261	1 261	x10	x10
CFG 10 5	0.904	0.908	1.236	1.236	178.036	175.715	28.780	16.913	x6.105	x10.389	77 761	77 761	2 521	2 521	x31	x31
CFG 10 6	0.907	0.911	1.238	1.239	1275	1274	79.249	38.734	-	-	466 561	466 561	4 621	4 621	-	-
CFG 10 7	0.909	0.909	1.237	1.239	TIME	TIME	233.327	84.407	-	-	TIME	TIME	7 921	7 921	-	-
CFG 10 8	0.908	0.909	1.233	1.235	MEM	MEM	904.098	170.957	-	-	MEM	MEM	12 871	12 871	-	-
CFG 10 9	0.908	0.909	1.240	1.234	MEM	MEM	TIME	341.238	-	-	MEM	MEM	TIME	20 021	-	-

TABLE C.6 – Résultats des réductions par symétries pour le modèle *CFG* avec PyPy.

qui vont servir à produire le module natif correspondant au moteur d'exploration. Ainsi, on peut voir que ici aussi selon le modèle la taille de ces fichiers varie, ceci est présenté en figures C.13, C.14, C.15 et C.16. On remarque que l'utilisation de la configuration opt pack réduit la taille des fichiers produit, mais l'utilisation des réductions de contrôle de flot a un impact beaucoup plus important.

Python	SIMPLE NET.PY	OPT NET.PY	OPT PACK NET.PY	FLOW NET.PY	DPS NO NET.PY	DPS NET.PY
CFG 10 1	6 287	6 287	6 287	N/A	6 892	6 892
CFG 10 2	6 288	6 288	6 288	N/A	6 893	6 893
CFG 10 3	6 289	6 289	6 289	N/A	6 894	6 894
CFG 10 4	6 290	6 290	6 290	N/A	6 895	6 895
CFG 10 5	6 291	6 291	6 291	N/A	6 896	6 896
CFG 10 6	6 292	6 292	6 292	N/A	6 897	6 897
CFG 10 7	6 293	6 293	6 293	N/A	6 898	6 898
CFG 10 8	6 294	6 294	6 294	N/A	6 899	6 899
CFG 10 9	6 295	6 295	6 295	N/A	6 900	6 900
CFG 11 5	7 422	7 422	7 422	N/A	8 085	8 085
CFG 12 5	8 649	8 649	8 649	N/A	9 370	9 370
CFG 13 5	9 972	9 972	9 972	N/A	10 751	10 751
CFG 14 5	11 391	11 391	11 391	N/A	12 228	12 228
CFG 15 5	12 906	12 906	12 906	N/A	13 801	13 801
CFG 16 5	14 517	14 517	14 517	N/A	15 470	15 470
CFG 17 5	16 224	16 224	16 224	N/A	17 235	17 235
CFG 18 5	18 027	18 027	18 027	N/A	19 096	19 096
CFG 19 5	19 926	19 926	19 926	N/A	21 053	21 053
CFG 20 5	21 921	21 921	21 921	N/A	23 106	23 106

Table C.7 – Nombre de lignes des fichiers produits par le backend Python. (modèles CFG)

Python	SIMPLE net.py	OPT net.py	OPT PACK net.py	FLOW net.py	DPS NO net.py	DPS net.py
CFG 1 5	432	432	432	n/a	515	515
CFG 2 5	699	699	699	n/a	840	840
CFG 3 5	1 062	1 062	1 062	n/a	1 261	1 261
CFG 4 5	1 521	1 521	1 521	n/a	1 778	1 778
CFG 5 5	2 076	2 076	2 076	n/a	2 391	2 391
CFG 6 5	2 727	2 727	2 727	n/a	3 100	3 100
CFG 7 5	3 474	3 474	3 474	n/a	3 905	3 905
CFG 8 5	4 317	4 317	4 317	n/a	4 806	4 806
CFG 9 5	5 256	5 256	5 256	n/a	5 803	5 803
CFG 10 5	6 291	6 291	6 291	n/a	6 896	6 896

Table C.8 – Nombre de lignes des fichiers produits par le backend Python. (modèles CFG)

Python	SIMPLE net.py	OPT net.py	OPT PACK net.py	FLOW net.py	DPS NO net.py	DPS net.py
client server c3n4	552	552	552	n/a	678	678
client server c3n5	590	590	590	n/a	725	725
client server c4n1	452	452	452	n/a	551	551
client server c4n2	484	484	484	n/a	592	592
client server c4n3	518	518	518	n/a	635	635
client server c4n4	554	554	554	n/a	680	680
client server c4n5	592	592	592	n/a	727	727
client server c5n1	454	454	454	n/a	553	553
client server c5n2	486	486	486	n/a	594	594
client server c5n3	520	520	520	n/a	637	637
client server c5n4	556	556	556	n/a	682	682
client server c5n5	594	594	594	n/a	729	729

Table C.9 – Nombre de lignes des fichiers produits par le backend Python. (modèles Client Serveur)

Python	SIMPLE NET.PY	OPT NET.PY	OPT PACK NET.PY	FLOW NET.PY	DPS NO NET.PY	DPS NET.PY
Choice 1	6 775	6 876	6 876	1 222	N/A	N/A
Choice 2	23 805	24 007	24 007	2 397	N/A	N/A
Choice 3	51 339	51 642	51 642	3 774	N/A	N/A
Linear 2	12 505	12 607	12 607	1 497	N/A	N/A
Linear 3	26 589	26 742	26 742	2 274	N/A	N/A
Linear 4	45 977	46 181	46 181	3 153	N/A	N/A
Need.-Sch.	668	677	677	558	N/A	N/A
Railroad 5	1 999	2 101	2 101	1 416	N/A	N/A
Railroad 6	2 344	2 458	2 458	1 606	N/A	N/A
Railroad 7	2 719	2 845	2 845	1 808	N/A	N/A

Table C.10 – Nombre de lignes des fichiers produits par le backend Python. (modèles colorés)

Python	SIMPLE NET.PY	OPT NET.PY	OPT PACK NET.PY	FLOW NET.PY	DPS NO NET.PY	DPS NET.PY
CSRepetitions PT 02	1 416	1 416	1 416	N/A	N/A	N/A
CSRepetitions PT 03	N/A	N/A	N/A	N/A	N/A	N/A
Dekker PT 010	8 978	8 978	8 978	N/A	N/A	N/A
Dekker PT 015	25 118	25 118	25 118	N/A	N/A	N/A
Dekker PT 020	N/A	N/A	N/A	N/A	N/A	N/A
Kanban PT 0005	N/A	N/A	N/A	N/A	N/A	N/A
LamportFastMutEx PT 2	8 741	8 741	8 741	N/A	N/A	N/A
LamportFastMutEx PT 3	18 820	18 820	18 820	N/A	N/A	N/A
LamportFastMutEx PT 4	35 613	35 613	35 613	N/A	N/A	N/A
MAPK PT 008	N/A	N/A	N/A	N/A	N/A	N/A
NeoElection PT 2	165 974	165 974	165 974	N/A	N/A	N/A
NeoElection PT 3	N/A	N/A	N/A	N/A	N/A	N/A
Philosophers 20	2 628	2 628	2 628	N/A	N/A	N/A
Philosophers 25	3 723	3 723	3 723	N/A	N/A	N/A
Philosophers 30	5 018	5 018	5 018	N/A	N/A	N/A
Philosophers 35	N/A	N/A	N/A	N/A	N/A	N/A

Table C.11 – Nombre de lignes des fichiers produits par le backend Python. (modèles bas niveau)

Python	SIMPLE net.py	OPT net.py	OPT PACK net.py	FLOW net.py	DPS NO net.py	DPS net.py
RessAllocation PT R002C002	432	432	432	N/A	N/A	N/A
RessAllocation PT R003C002	542	542	542	N/A	N/A	N/A
RessAllocation PT R003C003	767	767	767	N/A	N/A	N/A
RessAllocation PT R003C005	1 361	1 361	1 361	N/A	N/A	N/A
RessAllocation PT R003C010	3 686	3 686	3 686	N/A	N/A	N/A
RessAllocation PT R005C002	810	810	810	N/A	N/A	N/A
RessAllocation PT R010C002	1 760	1 760	1 760	N/A	N/A	N/A
RessAllocation PT R015C002	3 110	3 110	3 110	N/A	N/A	N/A
RessAllocation PT R020C002	N/A	N/A	N/A	N/A	N/A	N/A
RwMutex PT r0010w0010	3 468	3 468	3 468	N/A	N/A	N/A
RwMutex PT r0010w0020	6 418	6 418	6 418	N/A	N/A	N/A
RwMutex PT r0010w0050	20 068	20 068	20 068	N/A	N/A	N/A
RwMutex PT r0010w0100	58 818	58 818	58 818	N/A	N/A	N/A
RwMutex PT r0010w0500	1 088 818	1 088 818	1 088 818	N/A	N/A	N/A
RwMutex PT r0020w0010	7 138	7 138	7 138	N/A	N/A	N/A
SharedMemory PT 000005	3 531	3 531	3 531	N/A	N/A	N/A
SharedMemory PT 000010	31 556	31 556	31 556	N/A	N/A	N/A
SimpleLoadBal PT 02	2 683	2 683	2 683	N/A	N/A	N/A
SimpleLoadBal PT 05	14 617	14 617	14 617	N/A	N/A	N/A

Table C.12 – Nombre de lignes des fichiers produits par le backend Python. (modèles bas niveau)

Cython	SIMPLE net.pyx	SIMPLE net.pxd	SIMPLE net.h	SIMPLE net.cpp	OPT PACK net.pyx	OPT PACK net.pxd	OPT PACK net.h	OPT PACK net.cpp
Choice 1	7 102	62	215	74 852	2 197	11	164	24 060
Choice 2	24 540	114	368	249 388	5 430	11	265	53 592
Choice 3	52 482	166	521	528 964	10 186	11	366	96 840
Choice 4	90 928	218	674	913 580	15 944	11	467	149 097
Linear 2	12 940	114	268	135 188	2 930	11	165	31 592
Linear 3	27 282	166	371	279 664	5 386	11	216	54 390
Linear 4	46 928	218	474	477 180	8 344	11	267	81 697
Need.-Sch.	716	27	88	12 795	563	17	78	11 025
Railroad 5	2 322	53	130	27 306	1 280	17	94	15 563
Railroad 6	2 712	58	138	31 224	1 430	17	97	16 912
Railroad 7	3 132	63	146	35 440	1 586	17	100	18 315
Railroad 8	3 582	68	154	39 956	1 706	17	103	19 385
Railroad 9	4 062	73	162	44 770	1 871	17	106	20 869
Railroad 10	4 572	78	170	49 886	1 994	17	109	21 966
Railroad 11	5 112	83	178	55 304	2 168	17	112	23 531

Table C.13 – Nombre de lignes des fichiers produits par le backend Cython. (modèles colorés)

CYTHON	OPT				FLOW			
	net.pyx	net.pxd	net.h	net.cpp	net.pyx	net.pxd	net.h	net.cpp
Choice 1	7 102	62	215	74 852	1 296	11	164	16 155
Choice 2	24 540	114	368	249 388	2 626	11	265	28 775
Choice 3	52 482	166	521	528 964	4 158	11	366	43 213
Choice 4	90 928	218	674	913 580	5 892	11	467	59 469
Linear 2	12 940	114	268	135 188	1 526	11	165	19 175
Linear 3	27 282	166	371	279 664	2 358	11	216	27 463
Linear 4	46 928	218	474	477 180	3 292	11	267	36 669
Need.-Sch.	716	27	88	12 795	542	17	78	10 872
Railroad 5	2 390	55	132	27 508	1 229	17	94	15 240
Railroad 6	2 786	60	140	31 460	1 339	17	97	16 239
Railroad 7	3 212	65	148	35 712	1 449	17	100	17 238
Railroad 8	3 668	70	156	40 264	1 601	17	103	18 624
Railroad 9	4 154	75	164	45 116	1 714	17	106	19 650
Railroad 10	4 670	80	172	50 268	1 827	17	109	20 676
Railroad 11	5 216	85	180	55 720	1 991	17	112	22 170

Table C.14 – Nombre de lignes des fichiers produits par le backend Cython. (modèles colorés)

CYTHON	SIMPLE				OPT PACK				OPT			
	NET.PYX	NET.PXD	NET.H	NET.CPP	NET.PYX	NET.PXD	NET.H	NET.CPP	NET.PYX	NET.PXD	NET.H	NET.CPP
CSRepetitions PT 02	1 480	33	113	18 254	760	11	91	10 544	1 480	33	113	18 254
CSRepetitions PT 03	6 745	68	201	70 076	2 295	11	144	23 883	6 745	68	201	70 076
Dekker PT 010	9 837	60	232	93 294	4 333	11	183	37 060	9 837	60	232	93 294
Dekker PT 015	27 047	85	392	253 164	9 952	11	318	79 658	27 047	85	392	253 164
Dekker PT 020	57 507	110	602	540 284	18 531	11	503	145 496	57 507	110	602	540 284
Kanban PT 0005	823	26	94	12 028	823	26	94	12 028	823	26	94	12 028
LamportFastMutEpyx PT 2	9 206	79	227	93 531	2 966	11	159	29 075	9 206	79	227	93 531
LamportFastMutEpyx PT 3	19 611	110	318	195 616	5 343	11	219	49 244	19 611	110	318	195 616
LamportFastMutEpyx PT 4	36 802	145	427	364 939	8 718	11	293	78 015	36 802	145	427	364 939
MAPK PT 008	1 537	32	114	18 828	1 537	32	114	18 828	1 537	32	114	18 828
NeoElection PT 2	168 779	448	857	1 682 496	28 984	11	420	256 994	168 779	448	857	1 682 496
NeoElection PT 3	1 020 619	982	TIME	TIME	150 219	11	1 079	1 327 157	1 020 619	982	TIME	TIME
Philosophers 20	2 767	50	142	31 484	1 087	11	103	13 728	2 767	50	142	31 484
Philosophers 25	3 917	60	162	43 029	1 423	11	113	16 795	3 917	60	162	43 029
Philosophers 30	5 267	70	182	56 574	1 731	11	123	19 601	5 267	70	182	56 574
Philosophers 35	6 817	80	202	72 119	2 059	11	133	22 587	6 817	80	202	72 119
RessAllocation PT R002C002	391	18	76	7 616	391	18	76	7 616	391	18	76	7 616
RessAllocation PT R003C002	521	22	82	8 941	521	22	82	8 941	521	22	82	8 941
RessAllocation PT R003C003	779	28	92	11 526	779	28	92	11 526	779	28	92	11 526
RessAllocation PT R003C005	1 439	40	112	18 136	1 439	40	112	18 136	1 439	40	112	18 136
RessAllocation PT R003C010	3 929	70	162	43 061	3 929	70	162	43 061	3 929	70	162	43 061
RessAllocation PT R005C002	829	30	94	12 071	829	30	94	12 071	829	30	94	12 071
RessAllocation PT R010C002	1 879	50	124	22 696	1 879	50	124	22 696	1 879	50	124	22 696
RessAllocation PT R015C002	3 329	70	154	37 321	3 329	70	154	37 321	3 329	70	154	37 321
RessAllocation PT R020C002	5 179	90	184	55 946	5 179	90	184	55 946	5 179	90	184	55 946

Table C.15 – Nombre de lignes des fichiers produits par le backend Cython. (modèles bas niveau)

CYTHON	SIMPLE				OPT PACK				OPT			
	NET.PYX	NET.PXD	NET.H	NET.CPP	NET.PYX	NET.PXD	NET.H	NET.CPP	NET.PYX	NET.PXD	NET.H	NET.CPP
RwMutepyx PT r0010w0010	3 817	60	152	39 734	1 753	11	103	18 040	3 817	60	152	39 734
RwMutepyx PT r0010w0020	7 057	80	192	69 614	2 909	11	123	26 622	7 057	80	192	69 614
RwMutepyx PT r0010w0050	21 577	140	312	207 254	7 113	11	183	59 010	21 577	140	312	207 254
RwMutepyx PT r0010w0100	61 777	240	512	596 654	15 949	11	283	129 442	61 777	240	512	596 654
RwMutepyx PT r0010w0500	1 103 377	1 040	TIME	TIME	177 149	11	1 083	1 507 542	1 103 377	1 040	TIME	TIME
RwMutepyx PT r0020w0010	7 827	90	202	77 384	3 067	11	123	28 033	7 827	90	202	77 384
SharedMemory PT 000005	3 743	51	158	40 305	1 538	11	118	17 195	3 743	51	158	40 305
SharedMemory PT 000010	32 558	141	403	324 860	7 706	11	273	70 536	32 558	141	403	324 860
SimpleLoadBal PT 02	2 925	42	139	30 962	1 441	11	108	15 462	2 925	42	139	30 962
SimpleLoadBal PT 05	15 822	69	301	149 510	6 234	11	243	51 897	15 822	69	301	149 510
TokenRing PT 005	8 927	46	254	86 928	3 843	11	219	35 000	8 927	46	254	86 928
TokenRing PT 010	155 707	131	1 294	1 513 063	38 212	11	1 174	321 045	155 707	131	1 294	1 513 063

Table C.16 – Nombre de lignes des fichiers produits par le backend Cython. (modèles bas niveau)