Analyse statique de programmes concurrents avec variables numériques

Vincent Botbol

Sorbonne Université
École Doctorale Informatique, Télécommunications et Électronique

Analyse statique de programmes concurrents avec variables numériques

Vincent Botbol
sous la direction d'Emmanuel Chailloux
et de Tristan Le Gall

THÈSE
pour obtenir le titre de
Docteur en Sciences mention Informatique

Encadrement

Pr. Emmanuel Chailloux	Sorbonne Université
Dr. Tristan Le Gall	CEA LIST

Rapporteurs

Pr. Ahmed Bouajjani	Université Paris Diderot
Dr. HDR Laure Gonnord	Université Claude Bernard Lyon 1

Examinateurs

Pr. Gaétan Hains	Huawei Technologies
Pr. Antoine Miné	Sorbonne Université

Remerciements

Je tiens en premier lieu à remercier mes deux encadrants Emmanuel et Tristan qui, au cours de cette thèse, ont su me guider dans mes recherches, me soutenir lors des périodes difficiles et avec qui j'ai pris un grand plaisir à travailler.

Je tiens à remercier très chaleureusement Laure Gonnord et Ahmed Bouajjani d'avoir accepté de rapporter cette thèse et pour leurs remarques pertinentes et bienveillantes qui ont permis d'améliorer la qualité de ce manuscrit. Je remercie également Antoine Miné et Gaëtan Hains d'avoir accepté de faire partie de mon jury de thèse.

Je tiens à remercier de l'équipe APR qui m'ont accueilli dans un cadre chaleureux et dynamique et avec qui j'avais déjà eu le plaisir, pour la plupart, d'avoir eu comme enseignant au cours de mes études. Merci donc à Michèle, Frédéric, Romain, Binh, Antoine, Pascal, Maryse, Tong, Annick, Philippe (les deux), Christian et Jean.

Un grand merci à mes (ex-)compagnons thésards, aux post-doc' et stagiaires du LIP6 qui m'ont qui m'ont accompagné tout au long de cette grande aventure qui n'aurait probablement pas abouti sans eux. Merci, dans un ordre chronologique, à Étienne, Benjamin, Mathias, Grégoire, Philippe, Aurélien, Jérémie, Matthieu D., Dahmun, Rémy, Pierre, Pierrick, Ghiles, Steven, Ali, Alice, Marwan, Grégory, Marie, Matthieu J., Christelle, Thibault, Yi-Ting, Boubacar, Abdelraouf, Antoine, Clément, Medhi Raphaël, Martin, Adrien, Marc et à tous ceux que j'ai involontairement pu omettre.

Également, un grand merci à ma deuxième « famille » du LSL avec qui j'ai pu constaté (et constate encore) la construction du plateau de Saclay. Merci à Jean-Christophe, Lionel, Steven, Hugo, David, Robin, Benjamin, Boris, Julien, Zaynah et à tous les autres ! Merci également à Florent et Patricia pour m'avoir accorder leur confiance en me donnant la possibilité d'effectuer cette thèse.

Je souhaite également remercier ceux avec qui j'ai eu le plaisir de partager mes années de licence et de master. Merci de m'avoir supporté en tant que binôme et d'avoir partagé avec moi ces moments d'insouciance forts en mots croisés. Merci Mathieu C., Yohan, Alex, Oscar, Romain, Margaux, Marguerite, Tahar, Roven et à tous ceux que j'ai oublié.

Merci ensuite à mes collègues de la chaîne de blocs avec qui j'ai partagé ces derniers mois forts en émotion ! Merci Marco, Pietro, Bruno, Quyen, Oana, Raphaël, Vincent, Pierre (les deux), Nadine, Nicolas, Jean-Gabriel, Christian, Jérôme, Arthur et Diego.

Enfin, merci à ma famille qui m'a accompagné jusqu'ici. Merci à mes parents pour m'avoir encourager à poursuivre mes études. Merci à mes sœurs qui ont toujours été là pour moi. Merci à mes $2,5$ adorables nièces (ça sera une fille, Aurélie). Et à tous les autres membres de ma famille proche ou lointaine.

Enfin, je remercie tous ceux qui m'ont aidé de près ou de loin, qui m'ont accompagné d'une manière ou d'une autre et qui ont rendu cette thèse plus agréable.

Table des matières

Chapitre 1

Introduction

1.1 Évolution du matériel

Dans un ordinateur, l'ensemble des calculs, et donc du « travail » de la machine se fait dans des unités de calcul appellées processeurs. Cela peut être le processeur central (Central Processing Unit – CPU), qui est généraliste, ou les processeurs spécialisés comme par exemple ceux que l'on trouve dans des cartes graphiques (Graphical Processing Unit – GPU). La puissance des processeurs se mesure généralement par le nombre d'opérations qu'ils sont capables d'effectuer à la seconde. Ce résultat dépend bien entendu de la fréquence de l'unité de calcul, qui, pour un processeur moderne, est actuellement de l'ordre de quelques GHz ce qui correspond en moyenne à plusieurs centaines de millions d'opérations effectuées chaque seconde. Jusqu'au début des années 2000, il a été possible d'augmenter régulièrement cette cadence grâce aux avancées des techniques de miniaturisation. Ainsi, il a été observé que environ tous les deux ans, la fréquence d'un processeur doublait. Cette tendance est communément surnommée « loi de Moore ». Cependant, après les années 2000, les constructeurs de processeurs ont atteint des limites physiques (environ $15nm$) ne permettant plus d'assurer ce gain de performance. Pour pallier ce problème, les constructeurs ont fait le choix d'introduire plusieurs unités de calcul différentes au sein des processeurs. On parle alors de processeurs *multi-cœurs*. Chacune de ces unités est capable d'effectuer une tâche de manière indépendante des autres ; cette technique se nomme parallélisme. De fait, cette parallélisation permet d'augmenter le nombre global d'opérations en additionnant les possibilités de chaque unité de calcul. Par exemple, pour obtenir des prédictions météorologiques précises, il est nécessaire de prendre en compte un grand nombre de paramètres (vitesse des vents, pression, température, etc.). La résolution de telles équations en n'utilisant qu'une seule unité de calcul terminerait bien après la date de la prédiction recherchée. Ce type de calcul est, en général, effectué dans des « fermes de calcul » concentrant plusieurs dizaines de milliers de nœuds de calcul et pouvant atteindre des puissances de calcul de 10^{16} opérations à la seconde ; pour donner un ordre de grandeur, cela correspondrait à la combinaison d'un million d'ordinateurs grand public. La parallélisation des architectures ne s'est pas limitée aux domaines spécialisés mais concerne bel et bien tout le monde. Ainsi, aujourd'hui, même les téléphones portables disposent de processeurs à plusieurs cœurs capables de travailler sur différents calculs indépendants et de combiner leur puissance pour effectuer une tâche plus importante.

Plus généralement, en informatique, on appelle *concurrence* la théorie s'intéressant à considérer un ensemble de tâches (ou *processus*) pouvant chacune s'exécuter dans un ordre indéterminé. Au niveau matériel, il n'est pas souvent possible de déterminer dans quel ordre et à quel moment les unités de

calcul vont être actives. La concurrence s'intéresse alors à formaliser les différents comportements que cela entraîne. Ce domaine de recherche a été actif dès les années 60 et dispose de nombreux résultats théoriques. En pratique, ces résultats sont, par exemple, appliqués sous la forme de langages de programmation ou traits de langage permettant de décrire aisément des calculs parallèles, de bibliothèques logicielles implantant des algorithmes parallèles hautement optimisés ou encore d'outils de modélisation facilitant le développement de programmes parallèles...

Malgré les avantages de la programmation parallèle, il existe une industrie qui reste même aujourd'hui assez réticente à l'idée d'utiliser le parallélisme : celle qui produit des *logiciels embarqués critiques*. On appelle logiciels critiques, certains programmes destinés à être employé dans l'industrie de l'avionique, du ferroviaire ou encore du nucléaire. Ils sont critiques au sens où la moindre défaillance peut avoir pour conséquence la mort des usagers ou encore causer une perte économique importante. Pour s'assurer de l'absence d'erreurs, ces logiciels sont soumis au niveau le plus élevé de certification qui leur permettent d'atteindre des garanties fortes quant à la sûreté de leur utilisation. La réticence des autorités de certification à promouvoir l'utilisation d'architectures parallèles provient du fait que, comme nous le verrons par la suite, la concurrence complexifie fortement le comportement des programmes. Cependant, avec le temps, les besoins évoluent : l'analyse d'images en temps réel pour les véhicules autonomes, la complexification des réseaux de communication entre satellites, la multiplication des capteurs dans une centrale nucléaire, etc demandent de plus en plus de puissance de calcul. Il devient donc très difficile pour les industriels produisant des logiciels critiques de se passer de parallélisme.

1.2 *Bug* logiciel

Pour comprendre la difficulté à intégrer les architectures parallèles dans l'embarqué critique, on peut commencer par présenter les difficultés existantes dans un monde non concurrent. Lors du développement d'un programme, il est usuel de constater la présence d'erreurs. Ces erreurs, que l'on surnomme *bugs*, sont la différence entre le comportement désiré d'un programme et son comportement effectif. Leurs provenances peuvent être très diverses : un défaut de conception où une entrée du programme n'a pas été prévue, une faute du développeur lors de l'écriture du code source, une défaillance du matériel, etc. Un des exemples les plus célèbres de bug logiciel est l'explosion de la fusée Ariane 5 (1996). Quelques secondes après son décollage, une erreur due à un oubli de traitement d'une conversion de valeur a engendré un dépassement d'entier imprévu [Le 97]. Cette simple faute entraîna la panne du système de navigation et, finalement, sa destruction causant la perte de 370 millions de dollars. Également, entre 1985 et 1987, un appareil médical, le `Therac-25`, destiné à détruire certaines tumeurs comportait un comportement inattendu [LT93] : la machine projetait au patient des doses de radiation 100× plus importantes que prévu et entraîna la mort de plusieurs personnes et des blessures graves. Cette erreur est issue de l'interaction entre deux unités de calcul où l'une, contrôlant les radiations, venait lire une valeur dans la mémoire trop tôt avant qu'une autre unité n'ait eu le temps de l'écrire et récupérant ainsi une valeur erronée.

Dans certains cas, une simple relecture du code source peut être suffisante pour retrouver l'origine d'une erreur. Cependant, pour des projets de tailles conséquentes (*e.g.* le système de navigation de Ariane 5) inspecter le code source n'est pas une solution raisonnable. Par ailleurs, certaines erreurs ne fournissent pas de détails précis sur leurs origines : une utilisation trop importante de la mémoire ou

encore des accès en dehors de la mémoire allouée du programme vont arrêter sèchement l'exécution du programme avec des messages d'erreur très génériques (*e.g.* `"Stack overflow"`, `"Segmentation Fault"`, etc.). Bien que ce genre d'erreurs soit courantes, leurs provenances peuvent être difficiles à identifier. Un exemple classique d'erreur engendrant un accès illégal à la mémoire et que l'on retrouve souvent dans les programmes de débutants en programmation est le parcours incorrect de cases mémoires qu'on appelle plus communément tableaux. Ces tableaux contiennent n cases qui sont indicées par des entiers. En informatique, les indices des tableaux commencent généralement à 0. Ainsi, pour parcourir la totalité des cases d'un tableau, on doit inspecter celles dont les indices vont de [0 à $(n-1)$]. En revanche, si l'on dépasse cet intervalle, on peut obtenir une erreur à l'exécution. Les débutants, oubliant souvent ce détail, effectuent ce parcours de [1 à n] inclus ou encore de [0 à n] et accèdent alors à une partie non autorisée de la mémoire. La solution pour retrouver l'origine de ce type d'erreur consiste à utiliser des outils appelés *debugger* permettant d'inspecter les points dans le code où le programme « plante » et permettant au développeur d'accéder à plus de détails pour détecter la cause de l'erreur.

Certaines erreurs peuvent être plus difficiles à détecter et où l'utilisation d'un *debugger* n'est pas suffisante. Un exemple peu intuitif est le comportement des nombres à virgules flottantes dans le langage de programmation *C*. Pour la machine, il est très simple de mémoriser et de faire des opérations arithmétiques sur les nombres entiers mais cela est beaucoup plus compliqué voir impossible de représenter les nombres réels (au sens mathématique du terme) de manière exacte et efficace. La solution employée est de les arrondir vers une valeur représentable par la machine. Cependant, des effets indésirables surviennent lorsque l'on effectue plusieurs opérations d'arrondi consécutives. Le programme et ses résultats présentés par la Figure 1.1 illustrent bien ces effets indésirables. De fait, en *C*, le langage de programmation le plus utilisé pour les logiciels critiques, si l'on utilise un algorithme naïf pour calculer $0{,}001 \times 1000$, on obtient $0{,}999991$ au lieu de $1{,}0$. Pire encore, pour $0{,}001 \times 100.000.000$, on obtient $32.768{,}0$ au lieu de $100.000{,}0$. Ce dernier résultat est dû à un problème encore différent : quand un nombre flottant devient très grand, l'arrondi effectué après l'ajout d'une trop petite valeur n'a aucun effet laissant la valeur inchangée. Ce phénomène s'appelle « absorption ».

```
int main(int a, char *s[]){
  int i = 0, n = atoi(s[1]);
  float x = 0.0;
  for (i=0; i < n; i++)
    x += 0.001;
  printf("x = %f\n", x);
  return 0;
}
```

Entrée	Résultat attendu	Résultat obtenu
100	0,1	0,1
1.000	1,0	0,999991
100.000	100,0	99,956696
100.000.000	100.000,0	32.768,0

FIGURE 1.1 – Programme incrémentant une valeur flottante

Dans le monde de la concurrence, de nouvelles catégories d'erreurs viennent s'ajouter à celles existantes. Considérons, par exemple, un carrefour où 4 voitures arrivent exactement au même moment (Figure 1.2) et toutes souhaitent aller tout droit. En suivant la règle française de priorité à droite, aucune n'est en droit d'avancer. Dans la réalité, les automobilistes s'impatienteraient et l'un d'entre-eux prendrait l'initiative d'avancer débloquant les autres véhicules. Si chaque voiture est opérée par une machine, le problème est plus compliqué puisque, par définition, un ordinateur suit exactement toute directive reçue. La règle de priorité à droite les empêcherait tous de progresser et les laisserait bloqués dans cette

position indéfiniment. En programmation concurrente, cette situation se nomme un *interblocage* et survient lorsque que deux ou plusieurs tâches (ou processus) s'attendent mutuellement. Une autre situation peut survenir lorsque une voiture doit céder le passage avant de s'engager sur une route. Si cette route est saturée et qu'aucun autre automobiliste ne la laisse s'engager alors elle ne pourra jamais s'insérer. On parle alors de *famine* car le processus, ici la voiture, ne parvient pas à accéder à la ressource qui lui est nécessaire pour continuer.

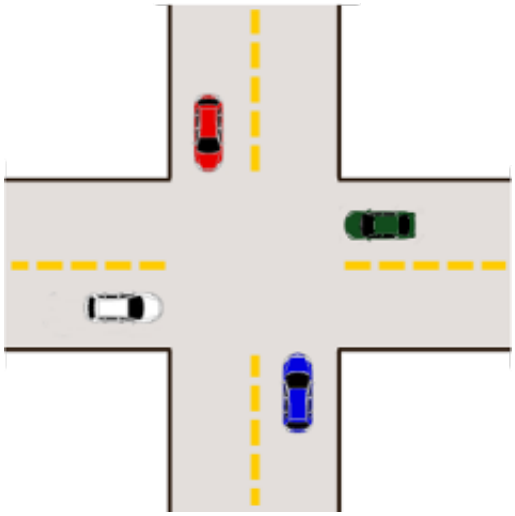


FIGURE 1.2 – Une situation d'inter-blocage

Un autre type de problématique est le partage de ressources entre plusieurs unités de calcul. Dans certains cas, notamment pour des raisons de performance, il est possible que plusieurs unités de calcul partagent une même ressource (la mémoire d'un ordinateur). Cette ressource peut également leur permettre de communiquer entre elles (*e.g.* une unité écrit tandis qu'une autre lit). Un problème peut survenir si l'on réécrit une ressource en cours d'utilisation par d'autres processus ou qu'un processus n'a pas fini de lire. On peut comparer cette situation à un professeur effaçant le tableau alors que ses élèves n'ont pas fini de le recopier : les élèves n'auront peut-être pas suffisamment d'informations leur permettant de faire leurs devoirs.

Pour finir, la difficulté majeure de la concurrence est l'absence de déterminisme. Lors de l'exécution d'un programme séquentiel (non concurrent), les instructions de la machine sont effectuées les unes à la suite des autres. Chaque exécution sera donc la même pour une entrée donnée. On dit alors que le programme est déterministe. Dans le monde de la concurrence, on ne peut pas supposer que les unités de calcul travaillent toutes à la même vitesse, en particulier d'une exécution à l'autre. Il est alors impossible de prévoir l'ordre dans lequel les opérations seront effectuées ou encore quelle tâche aura la priorité sur une autre, etc. On parle alors d'exécutions non-déterministes. Il faut donc que le développeur considère tous les cas de figures de son programme s'il ne veut pas commettre une erreur comme celles que nous avons évoquées ci-dessus. Cependant, vérifier précisément un grand nombre (potentiellement infini) d'exécutions possibles d'un programme concurrent est bien plus coûteux (voir impossible) que de vérifier une seule exécution déterministe.

Nous avons vu que détecter des erreurs dans des programmes est en général une tâche difficile et coûteuse en temps pour un humain et plus encore pour les programmes concurrents [AdBO09]. De plus, il n'y a aucune garantie sur la correction du bug par le développeur. C'est pourquoi il est souhaitable de déléguer ce rôle à la machine qui est, elle, beaucoup plus rapide et rigoureuse qu'un humain. Le

domaine proposant des solutions potentiellement automatisables à la détection de certaines [1] erreurs se nomme la vérification logicielle. Ainsi, à l'heure actuelle, il existe une grande variété d'outils permettant de détecter la présence d'erreurs [DKW08]. Dans les projets logiciels importants, et en particulier dans le domaine du logiciel embarqué critique, l'utilisation de ces méthodes est essentielle pour valider l'implantation (par exemple la norme RTCA/DO-178 [GP12] dans le domaine de l'avionique).

1.3 Vérification logicielle

Pour permettre de détecter un maximum d'erreurs lors du développement de projets logiciels, une bonne pratique consiste à établir le comportement attendu du programme dans un document technique que l'on nomme spécification. Ce document, rédigé par l'homme, peut aller d'une description informelle et vague (*e.g.* "lorsque je clique sur ce bouton, une fenêtre apparaît") jusqu'à un niveau de détail très élevé où l'on utilise la logique mathématique pour détailler le comportement d'un programme. Cette dernière forme se nomme : spécification formelle. C'est en suivant ce document qu'il est possible de valider l'implantation produite par les développeurs. Dans l'embarqué critique, cette étape se doit d'être très rigoureuse afin d'obtenir les certifications du logiciel et du matériel nécessaires pour la commercialisation. L'étape suivante est alors de respecter cette spécification en vérifiant que le programme se comporte bien de la manière attendue.

À l'heure actuelle, la méthode de vérification logicielle la plus commune est le test logiciel [Bei90, Tre08]. On peut réduire un test de programme à l'exécution de celui-ci avec des entrées spécifiques et un contexte simulant son utilisation réelle. Le résultat de cette exécution doit alors correspondre à la spécification établie supposant ainsi la correction du comportement attendu. Bien que la méthode du test logiciel bénéficie d'efforts importants du monde de la recherche et de l'industrie, il subsiste de fortes limitations. Il est impossible, en général, de tester toutes les entrées d'un programme puisqu'elles sont potentiellement infinies. Même s'il est impossible de générer une infinité de tests, il est en revanche possible de considérer des sous-ensembles *couvrant* certains comportements du programme. On utilise alors des *critères de couverture* destinés à établir une mesure de précision des tests. Par exemple, le critère de couverture *Modified Condition/Decision Coverage* (MC/DC) possède un très haut-niveau de sécurité et est recommandée par les autorités de certifications [GP12]. Pour valider ce critère, il est nécessaire de s'assurer de certains points comme le fait que chaque point d'entrée et de sortie d'un programme soit emprunté ou encore que chaque décision possible dans un programme soit testée. Bien que cette méthode définisse une limite raisonnable du nombre de tests à écrire tout en restant suffisamment sûre, cela représente une grande quantité de travail engendrant donc un coût conséquent. Il existe également d'autres niveaux de tests : les tests structurels qui, comme MC/DC, s'intéressent directement à la structure interne d'un programme, les tests fonctionnels vérifiant la respect de la spécification ou encore les tests d'intégration vérifiant que les parties testées d'un programmes coopèrent correctement entre elles. Le test logiciel est vu comme une forme *dynamique* de vérification. C'est-à-dire qu'il nécessite au programme de s'exécuter afin de constater un résultat. Un des inconvénients à exécuter le programme peut être le temps nécessaire pour évaluer tous les cas de tests. Les tests de certains projets importants peuvent notamment prendre plusieurs jours. À l'opposé, il existe la vérification statique qui va s'intéres-

1. Il est impossible d'établir une méthode universelle et infaillible permettant d'assurer l'absence d'erreurs dans un programme [Ric53]

ser au comportement d'un programme en inspectant le code source et donc sans avoir besoin d'exécuter le programme.

Dans le cadre de cette thèse, nous nous intéressons à un certain type de vérification : la vérification formelle. Celle-ci est, généralement, réalisée statiquement et est destinée à prouver des propriétés des programmes à l'aide d'outils et de raisonnements mathématiques. Pour illustrer, si l'on considère le système de contrôle d'un train, on peut vouloir montrer qu'il est impossible que les portes puissent s'ouvrir lorsque le train roule. On nomme ce type de comportement attendu, une propriété de programme. Cette propriété est exprimée sous forme de formules logiques et est issue de la spécification formelle. Pour pouvoir garantir cette impossibilité, on va utiliser des outils prouvés ou utilisant des méthodes prouvées mathématiquement corrects. Ces outils vont alors prendre en entrée le programme, la propriété à prouver et vont effectuer une *analyse* permettant d'obtenir un jugement sur cette propriété vis-à-vis du programme considéré. Puisque la méthode de vérification sur laquelle se repose l'outil a été prouvée correcte, le résultat obtenu apporte alors une garantie extrêmement forte sur la validité de cette propriété. Plus généralement, les méthodes employées par ces outils se nomment *méthodes formelles* qui reposent sur des concepts généraux de l'informatique théorique.

Il existe une grande variété de méthodes formelles. Historiquement, le typage statique [Pie02] constitue l'une des premières forme de vérification formelle où l'on cherche à garantir que les valeurs d'un programme sont utilisées dans un contexte approprié. Par exemple, on voudrait s'assurer que l'on ne puisse pas additionner un nombre et du texte (*e.g.* 1 + "hello") ce qui, dans la plupart des langages de programmation, entraînerait une erreur à l'exécution. Pour cela, on définit un système de types, pouvant être vu comme une spécification formelle du langage du programmation, et modélisé par un ensemble de règles que le développeur doit respecter lors de l'écriture d'un programme. Le respect de cette sémantique doit être vérifié. Cette étape s'effectue au début de de la compilation du programme (*i.e.* la traduction du code source, textuel, en code machine, binaire) et permet ainsi de vérifier l'absence d'une certaine catégorie d'erreurs.

Le *Model Checking* [CGP99, QS82] est une autre technique de vérification formelle et qui consiste, à partir d'un programme (ou système) donné, à explorer de manière exhaustive tous les états que le programme peut atteindre. Pour illustrer, si l'on considère un distributeur de boissons, on veut pouvoir vérifier toutes les combinaisons d'actions possibles et son résultat : "En introduisant 1€ puis en appuyant sur la touche soda, j'obtiens bien un soda", « En introduisant 1€ puis en appuyant sur la touche café, j'obtiens bien un café », etc. Le défi de ce domaine est d'être capable de traiter cette explosion du nombre d'états, notamment lorsque les programmes disposent de beaucoup de possibilités (*e.g.* un grand nombre de boutons sur le distributeur). Une des limites de cette technique est qu'il est difficile de considérer des programmes dont les états sont potentiellement infinis comme par exemples des programmes effectuant des calculs algébriques. Par exemple, une division $\frac{23}{y}$ où la variable y peut prendre n'importe quelle valeur possible nécessite de tester toutes les valeurs de y. Cependant, le *model checking* reste une technique de vérification formelle très utilisée et obtenant de bons résultats notamment pour vérifier l'absence d'erreurs dans des systèmes restreints comme des protocoles de communications, des systèmes embarqués. La limitation du *model checking* à vérifier de manière efficace les systèmes effectuant des opérations numériques est souvent une restriction trop forte pour certains types de programmes.

L'*Interprétation Abstraite* [CC77] est une théorie visant à combler ce manque. L'intérêt cette théorie est qu'elle permet de vérifier des propriétés sur des programmes dont les états sont potentiellement infinis. L'idée de celle-ci est d'*abstraire* certains états potentiellement infinis d'un programme. Par exemple,

si dans un programme, on choisit une valeur aléatoire entière n comprise entre 1 et 1.000.000, on doit vérifier, pour chaque valeur de n, le comportement du programme. Plutôt que de tester chaque entier exhaustivement : $n = \{1, 2, \ldots, 1.000.000\}$, on peut représenter cet ensemble par une valeur « abstraite » en utilisant par exemple un intervalle $n^\sharp = [1; 1.000.000]$ qui est lui composé de deux valeurs simples : une borne minimum et une borne maximum. Si, par la suite du programme, on multiplie par 2 cet entier, ce qui est simple puisqu'il suffit de multiplier par 2 les deux bornes plutôt que de calculer 1.000.000 de nouveaux entiers alors cette nouvelle valeur abstraite sera $n^{\sharp\prime} = [2; 2.000.000]$. Cependant, on constate que, dans cet intervalle résultat, n ne pourra jamais prendre certaines de ces valeurs comme par exemple les entiers impairs. Néanmoins, toutes les valeurs que n peut prendre sont bel et bien comprises dans cet intervalle, on parle alors de « sur-approximation ». Si l'on s'intéresse à montrer la propriété $P = n > 0$ alors cette sur-approximation est suffisante pour être **sûr** que P est vraie : toutes les valeurs possibles sont bien dans l'intervalle. Cependant, si la propriété est « n est pair » alors nous ne pouvons rien dire. Nous pouvons en revanche utiliser des abstractions plus adaptées, et souvent plus coûteuses à manipuler, pour tenter de gagner la précision nécessaire qui nous permettrait de conclure sur cette propriété.

À ce jour, de nombreux travaux se sont intéressés à la vérification formelle de programmes concurrents. La méthode la plus couramment employée reste le *model checking* utilisant différentes approches pour simplifier le problème. En effet, on cherche à réduire le nombre d'états à explorer, souvent infini, pour n'en considérer qu'un nombre fini. Cette réduction s'effectue souvent en effectuant une « abstraction » du comportement des processus. On ne va conserver que leurs interactions concurrentes comme par exemple les communications entre processus en occultant alors les calculs internes. Ainsi, ces méthodes se concentrent principalement sur la détection d'erreurs spécifiques à la concurrence (interblocage, famine, etc.) sans considérer les erreurs plus classiques d'exécutions liées au calcul.

D'autres méthodes basées sur l'interprétation abstraite [Fer05, Min12], permettent de faire l'analyse statique de programmes concurrents. Nous détaillerons ces méthodes dans le chapitre 2 mais donnons en dès maintenant les grandes lignes. Il s'agit d'analyser « processus par processus » la sémantique du programme, en prenant en compte les interactions (écriture dans une mémoire partagée) des différents processus. Cela nécessite donc d'analyser plusieurs fois chaque processus jusqu'à ce que l'analyse globale termine, et l'analyse peut être compliquée par la création dynamique de nouveaux processus. Nous nous proposons donc de développer une nouvelle méthode qui pourra, entre autre, gérer plus facilement la création dynamique de processus.

1.4 Objectifs et contributions

Dans le cadre de cette thèse, nous souhaitons proposer une méthode générale de vérification formelle de programmes concurrents. Celle-ci est entièrement automatique et vise la preuve de *propriétés de sûreté* : c'est-à-dire, des propriétés que l'on exprime sous forme d'une mauvaise configuration dont on va chercher à prouver l'absence dans l'ensemble des états possibles du programme considéré. En particulier, notre méthode est capable de traiter les propriétés numériques notamment grâce à la théorie de l'interprétation abstraite. Nous nous inspirons fortement de cette théorie pour effectuer nos analyses. Dans le cadre séquentiel, l'interprétation abstraite cherche à *sur-approximer* (ou abstraire) le comportement des programmes afin de rendre la méthode d'analyse plus facile à réaliser. Nous cherchons à généraliser ce mécanisme en abstrayant les ensembles d'états de processus et les comportements du programme concurrent.

Pour parvenir à cela, nous nous avons commencé par nous intéresser à une méthode issue du *model checking* nommée *Regular Model Checking* [AJNS04]. Celle-ci modélise les états de systèmes concurrents par des *langages réguliers* reconnus par des *automates* [HMU06]. En utilisant cette représentation, on encode un état de processus par une lettre (*e.g.* a ou b). De fait, on peut représenter un ensemble d'états de processus par une suite d'états encodés par des lettres. Cette suite de lettres donne donc un mot et représente un état *global* possible d'un ensemble de processus (*e.g.* mot_1 = babab, mot_2 = abbba, ...). Finalement, un langage, qui est composé d'un ensemble de mots, représente un ensemble d'états globaux (*e.g.* $\mathcal{L}_1 = \{\texttt{aaa}, \texttt{bbb}\}, \mathcal{L}_2 = \{\texttt{abab}, \texttt{babab}\}, ..$). Avec cette représentation, on peut voir un programme comme une suite de réécriture d'un langage. Ainsi, la sémantique de tels systèmes peut être modélisée par des *transducteurs* [Ber79], un certain type d'automate permettant de reconnaître et de réécrire des langages. En particulier, les transitions des transducteurs acceptent de reconnaître une lettre d'un alphabet fini et de la réécrire (ou non) vers un alphabet fini potentiellement différent.

Nous nous sommes inspirés du *Regular Model Checking* en intégrant à cette approche la méthodologie de l'interprétation abstraite nous permettant de considérer une classe plus générale de programmes, en particulier, les programmes effectuant du calcul numérique. Pour cela, nous utilisons des automates spéciaux : les *automates de treillis* [Le 08] dont l'alphabet de treillis qu'ils reconnaissent peuvent être des ensembles munis d'une relation d'ordre (*i.e.* des treillis). La première contribution que nous présentons dans cette thèse est la définition des *transducteurs de treillis*, une structure permettant de réécrire les langages reconnus par des automates de treillis. Les transitions de ces transducteurs de treillis sont également capables de reconnaître des suites de lettres plutôt qu'une seule. Nous proposons un langage de programmation simple de *processus communiquants* à *mémoire distribuée* (ou répartie), c'est-à-dire, où chaque processus possède son propre espace mémoire et où les processus sont capables de communiquer entre eux à l'aide de *passage de messages*. Ce langage permet également d'effectuer du calcul numérique et possède un mécanisme de création/destruction dynamique de processus. Nous montrons également comment la sémantique d'un tel langage peut se traduire automatiquement vers ces transducteurs de treillis. Finalement, nous proposons une méthode pour calculer une abstraction de l'ensemble des états possibles que l'on nomme *espace d'atteignabilité* nous permettant de valider des propriétés de sûreté numériques. De plus, de par la nature de notre représentation des états (automates), il est possible d'exprimer des propriétés de sûreté également sous la forme d'automate. Il est ainsi possible de vérifier des propriétés concernant un ensemble de processus (*e.g.* « si pour le premier processus : $x < 100$ alors pour tous les autres processus : $y < 100$ »).

La représentation de la sémantique d'un programme par un transducteur souffre de certaines limitations comme, par exemple, l'impossibilité d'effectuer une communication en dehors de son voisinage ; les processus étant ordonnés sous forme de mots et de par la nature des transducteurs, il n'est pas aisé d'exprimer des communications entre processus de manière arbitraire. La deuxième contribution de notre travail est la définition d'une règle de réécriture également capable de reconnaître un langage régulier mais, à la différence du transducteur, les transitions sont désormais capables de reconnaître un ensemble non-borné de processus en employant des expressions régulières. De ce fait, il est possible d'encoder la sémantique d'opérations concurrentes très générales comme les communications point-à-point ou multi-points. Nous sommes alors capables de définir un langage plus expressif que le précédent et munis de primitives de communications généralistes. Nous adaptons la précédente analyse pour l'intégrer à ce nouveau langage et proposons des optimisations permettant d'en améliorer la précision. Enfin, cette méthode est aussi capable de traiter la création dynamique de processus. En effet, pour certaines

applications, il peut être intéressant de « créer » de nouveaux processus au cours de l'exécution du programme dont le but est de traiter une tâche particulière. Cela représente un défi puisque cela complexifie sensiblement l'analyse : à chaque instant du programme de nouveaux états peuvent apparaître augmentant fortement le nombre d'états possibles. Comme pour certaines approches du *regular model checking*, notre méthode traite la création de processus simplement en ajoutant une transition dans l'automate représentant l'ensemble des états atteignables.

Cette méthode offre donc des avantages théoriques substantiels, aussi nous avons voulu montrer qu'elle est applicable à des cas concrets. Pour cela, nous nous sommes intéressés à la bibliothèque de calcul parallèle *Message-Passing Interface* [SOHL+98] (`MPI`). Celle-ci emploie un modèle à mémoire distribuée où un ensemble de nœuds de calcul, chacun possédant un identifiant unique, utilisent un même code source pour effectuer leurs calculs. Pour communiquer, les processus utilisent un mécanisme de passage de messages via l'utilisation de primitives générales de communications synchrones (l'émission et la réception s'effectuent au même moment) ou asynchrones (la réception peut avoir lieu après l'émission). Ces primitives comportent notamment : l'envoi et la réception simple d'un message à un destinataire (*send*/*receive*), l'envoi d'un message d'un processus à tous les autres (*broadcast*) ou encore des opérations de plus haut niveau (réduction, synchronisation, ...). Ce modèle de concurrence en fait donc un excellent candidat pour construire une analyse utilisant notre modèle théorique. La popularité dont jouit cette bibliothèque a accru significativement le nombre de ses primitives (> 200) et dont leur sémantique est parfois difficile à établir de par l'absence de formalisation. Ainsi, nous nous sommes focalisés sur un sous-ensemble en y établissant une sémantique correspondant au comportement des principales implémentations de `MPI`. Ce sous-ensemble est cependant suffisamment expressif pour nous permettre de considérer des programmes intéressants. Ainsi, nous avons établi un prototype d'analyse numérique (ou analyse de valeurs) statique de programmes `MPI` écrits en langage de programmation `C`. Ce prototype prend la forme d'un *greffon* de *Frama-C* [KKP+15], une plate-forme disposant d'outils d'analyse statique de code `C` et utilisée industriellement. L'analyse, entièrement automatisée, calcule une sur-approximation de l'espace d'atteignabilité du programme `MPI/C` permettant de vérifier des propriétés de sûreté numériques et de détecter des erreurs liées à la concurrence.

1.5 Plan du document

Cette thèse vise à proposer une solution à la vérification formelle de programmes concurrent effectuant du calcul numérique. Nous proposons une solution à cette problématique que nous présentons au travers des cinq chapitres principaux suivants :

— **Le chapitre 2** présente l'état de l'art du domaine et des travaux connexes à notre problématique. Nous y introduisons également l'ensemble des notions nécessaires à la bonne compréhension de notre méthodologie et de notre travail.

— **Le chapitre 3** décrit une première approche s'inspirant du *Regular Model Checking* en proposant d'adapter cette méthodologie en utilisant celle de l'interprétation abstraite. Nous introduisons alors un langage simple permettant d'exprimer des calculs, de la création/destruction dynamique de processus et des primitives de communication. À l'aide de la formalisation que nous proposons, nous montrons qu'il est possible de construire une analyse de valeurs de tels programmes.

— **Le chapitre 4** étend la méthodologie précédemment définie en y ajoutant une structure de règle de réécriture permettant d'enrichir sensiblement l'expressivité du modèle précédent. En particu-

lier, cela rend possible la définition d'un nouveau langage disposant, entre autres, de communications point-à-point et multi-point levant ainsi certaines des restrictions induites de la précédente formalisation.

— **Le chapitre 5** présente le standard `MPI` avant d'introduire notre prototype capable de traiter un sous-ensemble de celui-ci. Nous détaillons l'implantation de ce prototype et, en particulier, la phase de traduction automatique du programme `MPI/C` vers notre modèle théorique. Nous y introduisons également des optimisations sur la précision et les performances de l'outil.

— **Le chapitre 6** propose des expérimentations montrant les possibilités et les performances de notre outil. Nous commençons par nous comparer avec l'outil `ISP` [VVD+09] qui analyse la présence d'interblocages dans les programmes `MPI/C` de manière dynamique en montrant qu'il est également possible pour notre outil d'en détecter un certain nombre, bien que cela ne soit pas son utilisation principale. Puis, nous proposons d'appliquer notre outil à des exemples effectuant du calcul numérique pour vérifier des propriétés de sûreté grâce à l'invariant obtenu.

— **Le chapitre 7** conclut en revenant sur l'ensemble des résultats obtenus au cours de cette thèse et propose une discussion sur les perspectives d'améliorations et d'orientations possibles de nos travaux.

Chapitre 2

État de l'art

Avant de présenter les travaux effectués au cours de cette thèse, nous donnons les définitions et rappels destinés à la bonne compréhension de ce document. Nous présentons certains résultats récents de vérification de programmes concurrents.

2.1 Préliminaires

Nous introduisons brièvement quelques notions et résultats théoriques utilisés dans les sections suivantes.

2.1.1 Systèmes de transitions

Un système de transition est un modèle fréquemment utilisé en informatique pour représenter des systèmes à états discrets. Cela signifie que leur évolution n'est pas continue mais « saute » d'une configuration à une autre. On suppose que la modélisation du système comporte des états identifiables et peut évoluer d'un état à un autre à l'aide d'une règle de transition. Ce modèle permet de donner la description formelle d'automates, de programmes informatiques et autres machines. Par exemple, nous pouvons modéliser un feu de circulation disposant de trois états possibles (vert, orange et rouge) et de trois transitions (vert $\xrightarrow{25s}$ orange, orange $\xrightarrow{5s}$ rouge et rouge $\xrightarrow{30s}$ vert). Nous pouvons représenter ce système de transition graphiquement (Figure 2.1). Dans cet exemple, les transitions de ce système sont étiquetées par des durées modélisant le temps nécessaire pour passer d'un état à un autre.

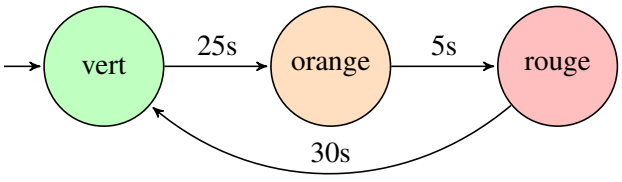


FIGURE 2.1 – Modélisation d'un feu tricolore

Nous pouvons formaliser un tel système par la définition suivante.

Définition 1 (**Système de transition**). *Un système de transition est un n-uplet $\langle C, C_0, \rightarrow \rangle$ où :*

— *C est l'ensemble dénombrable des configurations ;*

— *$C_0 \subseteq C$ est l'ensemble des configuration initiales ;*

— $\rightarrow \subseteq C \times C$ *est la relation de transition ;*

Un système de transition étiqueté est un système de transition $\langle C, \Sigma, C_0, \rightarrow \rangle$ *avec* Σ *un ensemble d'étiquettes et une relation de transition* $\rightarrow \subseteq C \times \Sigma \times C$.

Une *exécution* étiquetée $c_0 \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_{n-1}} c_{n-1} \xrightarrow{\sigma_n} c_n$ part d'une configuration initiale $c_0 \in C_0$ et respecte la relation de transition : $\forall i,\ (c_i, \sigma_{i+1}, c_{i+1}) \in \rightarrow$. Les *traces* d'un système de transition sont les mots composés des étiquettes d'une exécution $m = \sigma_1 \cdot \sigma_2 \dots \sigma_n$. Il est à noter qu'une exécution peut être finie ou infinie.

Cette modélisation est très générale et peut servir à encoder des sémantiques de programmes de haut-niveau comme nous le verrons dans ce document.

Nous introduisons également la notion d'*atteignabilité*. Soit $S \subseteq C$ un ensemble de configurations. Les fonctions Post : $2^C \rightarrow 2^C$ et Pre : $2^C \rightarrow 2^C$ sont définies par les relations de transitions suivantes :

$$\text{Post}(S) = \{c \in C \mid \exists c' \in S, c' \rightarrow c\}$$
$$\text{Pre}(S) = \{c \in C \mid \exists c' \in S, c \rightarrow c'\}$$

Les ensembles de configurations résultants de l'application de ces deux fonctions sont les successeurs et prédécesseurs immédiats des configurations S. L'ensemble de tous les successeurs et de tous les prédécesseurs sont les fermetures transitives et réflexives de $\rightarrow$, $\rightarrow^*$, qui sont définies récursivement par :

1. $\forall c \in C,\ (c, c) \in \rightarrow^*$;
2. $(c_1, c_2) \in \rightarrow^*$ si et seulement si $\exists c_1' \in C, (c_1, c_1') \in \rightarrow \wedge (c_1', c_2) \in \rightarrow^*$.

Définition 2 (Atteignabilité). *Soit c et c′ deux configurations du système de transition* $\langle C, C_0, \rightarrow \rangle$*, s′ est atteignable (resp. co-atteignable) depuis s si et seulement si* $s \rightarrow^* s'$ *(resp.* $s' \rightarrow^* s$*). L'ensemble* $Post^*(S)$ *(resp.* $Pre^*(S)$*) est l'ensemble des configurations atteignables (resp. co-atteignables) depuis S :*

$$Post^*(S) = \{c \in C \mid \exists c' \in S, c' \rightarrow^* c\}$$
$$Pre^*(S) = \{c \in C \mid \exists c' \in S, c \rightarrow^* c'\}$$

L'espace d'atteignabilité d'un système de transition $\langle C, C_0, \rightarrow \rangle$ est défini par $\text{Post}^*(C_0)$.

2.1.2 Ordres partiel et treillis

Définition 3. *Un treillis est une paire* $(S, \sqsubseteq)$ *où :*

— *S est un ensemble (fini ou infini)*

— $\sqsubseteq$ *est une relation d'ordre sur l'ensemble S, i.e. une relation binaire vérifiant les propriétés suivantes :*

  — *réflexivité :* $\forall x \in S, xRx$,

  — *antisymétrie :* $\forall x, y \in S, xRy \wedge yRx \Rightarrow x = y$,

  — *transitivité :* $\forall x, y, z \in S, xRy \wedge yRz \Rightarrow xRz$.

— *Toute paire d'éléments* $\{x, y\}$ *de S admet une borne inférieure et une borne supérieure :*

  — *on note* $x \sqcap y$ *la borne inférieure de x et y, qui est l'unique élément maximal de l'ensemble des minorants (prédécesseurs) de x et y (*$z \in S$ *est un minorant si* $z \sqsubseteq x \wedge z \sqsubseteq y$*).*

— *on note $x \sqcup y$ la borne supérieure de x et y, qui est l'unique élément minimal de l'ensemble des majorants (successeurs) de x et y. ($z \in S$ est un majorant si $z \sqsupseteq x \wedge z \sqsupseteq y$).*

Un treillis est borné si S possède un élément minimal $\bot$ et un élément maximal $\top$ satisfaisant : $\forall x \in S, \bot \sqsubseteq x \sqsubseteq \top$ et est complet si pour tout sous-ensemble $X \subseteq S$, les bornes inférieure et supérieure de X existent.

Par exemple, si l'ensemble des intervalles ordonné par la relation d'inclusion $\subseteq$ est un treillis :

$$\mathcal{I} = \{[a,b] \text{ où } a \in \mathbb{Z} \cup \{-\infty\} \text{ et } b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\} \cup \{\emptyset\}$$

Par la suite, nous ne considérerons que des treillis bornés.

2.1.3 Points-fixes

$x \in S$ est un point-fixe de $f : S \to S$ si $f(x) = x$. Le *plus petit point-fixe* (lfp : *least fix-point*) de f est la plus petite borne inférieure de l'ensemble des points-fixes de f ; de plus, le *plus grand point-fixe* (gfp : *greatest fixpoint*) est la plus petite borne supérieure du même ensemble. x est un *post-point-fixe* si $x \sqsubseteq f(x)$.

Comme expliqué dans la section précédente, l'espace d'atteignabilité $\text{Post}^*(C_0)$ est le plus petit point-fixe de la fonction $X \mapsto \text{Post}(X) \cup C_0$ qui est une fonction définie sur le treillis complet $(2^C, \subseteq)$.

En supposant $(S, \sqsubseteq)$ un treillis complet, il est possible de calculer le point-fixe de la fonction f grâce aux théorèmes suivants :

Définition 4. *Soit $f : S \to S$ une fonction :*

— *f est monotone si :*

$$\forall x, y \in S,\ x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

— *f est continue si :*

$$\forall X \subseteq S,\ f(\sqcup X) = \sqcup\{f(x) \mid x \in X\} \text{ et } f(\sqcap X) = \sqcap\{f(x) \mid x \in X\}$$

Il est à noter que si f est continue alors f est monotone puisque si $x \sqsubseteq y$ alors $y = x \sqcup y$ et $f(y) = f(x) \sqcup f(y) \sqsupseteq f(x)$.

Théorème 1 (Knaster-Tarski). *En supposant f monotone :*

$$lfp(f) = \sqcap\{x \mid x \sqsupseteq f(x)\}$$

Démonstration. On considère $D = \{x \mid x \sqsupseteq f(x)\}$ et $l = \sqcap D$, qui existe puisque $(S, \sqsubseteq)$ est un treillis complet.

Pour tout $x \in D$, on a $x \sqsupseteq l$, alors $x \sqsupseteq f(x) \sqsupseteq f(l)$ puisque f est monotone. Ainsi, $f(l)$ est une borne inférieure de D. Par définition de l, nous avons $f(l) \sqsubseteq l$. Nous appliquons une fois encore f : $f(f(l)) \sqsubseteq f(l)$. $f(l) \in D$ ce qui signifie $l \sqsubseteq f(l) \sqsubseteq l$. Donc, l est un point-fixe de f et, puisque tous les points-fixes de f sont dans D, alors l est le plus petit point-fixe de f. □

Intuitivement, le plus petit point-fixe de f est plus petit que n'importe quel autre point-fixe de f. Le théorème suivant établit une manière de construire un calcul de $\text{lfp}(l)$.

Théorème 2 (Kleene). *En supposant f continue, lfp(l) est la limite de la suite :*

$$u_0 = \bot$$
$$u_{n+1} = u_n \sqcup f(u_n)$$

Démonstration. La limite de cette suite est définie par $x = \bigsqcup_n f^n(\bot)$. Ainsi, x est un point-fixe puisque :

$$\begin{aligned}
f(x) &= f(\bigsqcup_n f^n(\bot)) \\
&= \bigsqcup_n f(f^n(\bot)) && f \text{ est continue} \\
&= \bigsqcup_n f^n + 1(\bot) \\
&= \bot \sqcup \bigsqcup_{n \geq 1} f^{n+1}(\bot) \\
&= \bigsqcup_{n'} f^{n'}(\bot) && n' = n + 1 \\
&= x
\end{aligned}$$

x est donc un point-fixe et nous souhaitons démontrer qu'il est le plus petit point-fixe. Soit y un autre point-fixe de f. Puisque $\bot \sqsubseteq y$, nous avons $f(\bot) \sqsubseteq f(y) = y$. Par induction, nous obtenons : $\forall n,\ f^n(\bot) \sqsubseteq y$ ainsi $x = \bigsqcup_n f^n(\bot) \sqsubseteq y$. □

Le théorème de Kleene donne un algorithme naïf permettant de calculer le plus petit point-fixe d'une fonction f sans aucune garantie de terminaison. En effet, ce calcul peut ne pas terminer lorsque le treillis $(S, \sqsubseteq)$ ne satisfait pas la *condition de chaîne ascendante*, *i.e.* chaque chaîne ascendante $u_0 \sqsubseteq u_1 \ldots \sqsubseteq u_n \sqsubseteq \ldots$ est stationnaire après un nombre fini d'étapes. Une solution pour garantir la terminaison de ce calcul de point-fixe est d'introduire des approximations.

2.2 Interprétation Abstraite

L'interprétation abstraite est une théorie générale introduite par Patrick et Radhia Cousot [CC77] qui consiste en l'approximation de la sémantique de programmes ou de modèles. L'interprétation abstraite a de nombreuses applications dont la vérification *automatique* de programmes. Celle-ci est souvent réductible à une recherche de l'ensemble des états que peut atteindre une machine exécutant un programme. Dans le cas général, cet ensemble d'atteignabilité n'est pas calculable : on peut en effet réduire ce problème à celui de l'arrêt des machines de Turing. L'interprétation abstraite va chercher à construire une sur-approximation de cet ensemble en y ajoutant des états non-atteignables. Cette sur-approximation est, quant à elle, calculaable. L'idée est de vérifier qu'un programme ne peut atteindre d'états mauvais (*e.g.* une porte d'une rame de train pouvant s'ouvrir lorsque le train se trouve entre deux gares) : si ces états mauvais ne sont pas présents dans la sur-approximation alors ils ne peuvent pas être présents dans les états atteignables. A contrario, si un état mauvais est présent dans la sur-approximation alors soit le programme contient une erreur, soit l'abstraction est trop imprécise et doit être raffinée.

2.2.1 L'ensemble d'atteignabilité vu comme un calcul de point-fixe

De fait, la sémantique d'un programme peut être vue comme un système de transition $\langle C, C_0, \rightarrow \rangle$, son ensemble d'atteignabilité est $\text{Post}^*(C_0)$. Nous introduisons alors la fonction :

$$\begin{aligned} F : \ 2^C &\mapsto 2^C \\ X &\mapsto C_0 \cup \text{Post}(X) \end{aligned}$$

Le plus petit point-fixe de cette fonction dans le treillis $\Gamma = 2^C$, noté $\text{lfp}(F)$, est égal à $\text{Post}^*(C_0)$. Le théorème de Kleene nous donne donc un moyen de calculer cet ensemble d'atteignabilité. Cependant, pour rendre ce calcul effectif, il nous faut faire des approximations.

En interprétation abstraite, deux méthodes d'approximation sont employées pour approximer ce plus petit point-fixe :

1. Les approximations statiques : plutôt que de chercher le plus petit point-fixe de F dans son treillis original $(\Gamma, \sqsubseteq)$, nous calculons le plus petit point-fixe de son abstraction $F^\sharp$ dans un *treillis abstrait* $(\Lambda, \sqsubseteq^\sharp)$, lié à Γ à l'aide d'une *correspondance de Galois*. Le treillis abstrait est choisi de telle manière que le calcul d'un point-fixe dans ce treillis est plus simple à calculer que dans l'original.
2. Les approximations dynamiques : quand un treillis abstrait ne satisfait pas la condition de chaîne ascendante, on cherche à *deviner* la limite de la suite $(u_n)_{n \in \mathbb{N}}$ à l'aide d'un *opérateur d'élargissement* qui assure la terminaison du calcul au prix d'une certaine approximation.

Nous allons maintenant détailler ces deux types d'approximation.

2.2.2 Approximations statiques : correspondance de Galois

Nous supposons que le treillis (dit concret) $(\Gamma, \sqsubseteq)$ est un treillis complet et nous considérons un autre treillis, dit abstrait, $(\Lambda, \sqsubseteq^\sharp)$. Une *correspondance de Galois* est donnée par deux fonctions $\alpha : \Gamma \mapsto \Lambda$ et $\gamma : \Lambda \mapsto \Gamma$ qui sont respectivement la fonction d'abstraction et la fonction de concrétisation, définies telles que :

$$\forall s \in \Gamma, \ \forall a \in \Lambda, \ \alpha(s) \sqsubseteq^\sharp a \Leftrightarrow s \sqsubseteq \gamma(a)$$

Cette correspondance de Galois est notée :

$$(\Gamma, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\Lambda, \sqsubseteq^\sharp)$$

Aussi, $F^\sharp$ est définie par $F^\sharp = \alpha \circ F \circ \gamma$, avec la propriété suivante :

$$\text{lfp}(F) \sqsubseteq \gamma(\text{lfp}(F^\sharp))$$

Le calcul de $\text{lfp}(F^\sharp)$ donne alors une sur-approximation de $\text{lfp}(F)$.

Exemple 1. *Pour illustrer, prenons le treillis complet* $(2^{\mathbb{Z}}, \subseteq)$. *Le treillis des intervalles* $(\mathcal{I}, \subseteq)$ *est également un treillis complet. Il existe une correspondance de Galois* $\underset{(2^{\mathbb{Z}}, \subseteq)}{\overset{(\mathcal{I}, \subseteq)}{\leftrightarrows}}$, *où la fonction d'abstraction est* $\alpha(P) = [min(P), max(P)]$ *et la fonction de concrétisation est l'injection canonique* $\mathcal{I} \hookrightarrow 2^{\mathbb{Z}}$. *Les éléments du treillis abstrait peuvent être aisément manipulés mais il existe une infinité de chaînes croissantes (*e.g. $[-n, n], n \in \mathbb{N}$*). Ainsi, le calcul de point-fixe* $\bigcup_{n \in \mathbb{N}} f^n(\bot)$ *peut ne pas terminer sans l'utilisation d'un opérateur d'élargissement que nous présentons dans la sous-section suivante.*

2.2.3 Approximations dynamiques : opérateur d'élargissement

Un opérateur d'élargissement $\triangledown$ extrapole une limite à une chaîne croissante depuis un nombre fini de termes, généralement deux.

Définition 5. *Soit* $(\Lambda, \sqsubseteq)$ *un treillis.* $\triangledown : \Lambda \times \Lambda \to \Lambda$ *est un opérateur d'élargissement si :*

1. $\forall x, y \in \Lambda, x \sqsubseteq x \triangledown y \wedge y \sqsubseteq x \triangledown y$
2. *si* $x_0 \sqsubseteq x_1 \sqsubseteq \ldots x_n \sqsubseteq \ldots$ *est une chaîne croissante alors la chaîne croissante définie par* $y_0 = x_0$ *et* $y_{n+1} = y_n \triangledown x_{n+1}$ *est stationnaire après un nombre fini d'étapes.*

La première condition assure que l'élargissement est une sur-approximation de la plus petite borne supérieure. La seconde condition assure la terminaison du calcul de point-fixe. Ainsi, nous obtenons :

Théorème 3. *[CC77] Soit* $(\Lambda, \sqsubseteq^\sharp)$ *un treillis abstrait complet et* $f : \Lambda \to \Lambda$ *une fonction. La suite définie par :*

$$u_0 = \bot$$
$$u_{n+1} = u_n \triangledown f(u_n)$$

est stationnaire après un nombre fini d'étapes et sa limite est $u_\infty \sqsupseteq^\sharp lfp(f)$.

Exemple 2. *Sur le treillis des intervalles* $(\mathcal{I}, \subseteq)$*, l'opérateur d'élargissement classique est défini par* $[a_1, b_1] \triangledown [a_2, b_2] =$

- $[a_1, b_1] \cup [a_2, b_2]$ *si* $[a_1, b_1] \not\subseteq [a_2, b_2]$ *;*
- $[a, b]$ *sinon, avec* $a = a_1$ *si* $a_1 = a_2$ *et* $a = -\infty$ *si* $a_1 > a_2$*, et* $b = b_1$ *si* $b_1 = b_2$ *et* $b = +\infty$ *si* $b_1 < b_2$.

L'idée de cet opérateur d'élargissement est que s'il détecte que l'intervalle croît, il peut croître indéfiniment.

Cette approximation est souvent peu précise mais peut également être améliorée en définissant un ensemble fini de valeurs intermédiaires $v_1 < \ldots < v_n$ *sur lesquelles les bornes croissantes seront, dans un premier temps, projetées. Nous pouvons alors changer la définition ainsi : si* $b_1 < b_2$ *alors* b *est la plus petite valeur* $v_i > b_2$ *ou* $+\infty$ *si* $b_2 > v_n$*. Cette optimisation se nomme élargissement avec seuils.*

Notons qu'il existe des alternatives à l'utilisation d'un opérateur d'élargissement, comme par exemple l'accélération [Ler03, LS07]. Comme l'élargissement, l'accélération vise à calculer en temps fini la limite d'une suite croissante. Elle ne recourt pas à des approximations, mais calcule, quand cela est possible, l'effet des boucles d'un programme de manière exacte. Notons que l'on peut combiner les deux types de techniques pour calculer des points-fixes [GS13].

2.2.4 Domaines abstraits

Dans le cadre de la vérification de programmes par interprétation abstraite, on nomme domaine abstrait un treillis muni d'opérations abstraites. Celui-ci doit spécifier :

- Un treillis abstrait $(\Lambda, \sqsubseteq^\sharp)$:
  - deux extremums : $\bot$ et $\top$;
  - une borne inférieure binaire : $\sqcap^\sharp$;
  - une borne supérieure binaire : $\sqcup^\sharp$.

— une fonction de concrétisation $\gamma : \Lambda \to \Gamma$ monotone où Γ représente le domaine concret ;
— une fonction d'abstraction $\alpha : \Gamma \to \Lambda$ formant une correspondance de Galois avec γ ;
— des fonctions de transfert (voir plus bas) ;
— un opérateur d'élargissement $\triangledown$ si le treillis possède des chaînes croissantes infinies.

Il existe une large variété de domaines abstraits dans la littérature. Dans une analyse, le choix du domaine abstrait employé dépend du type de l'invariant de programme que l'on souhaite obtenir afin de prouver nos propriétés. Les propriétés les plus communes que l'on cherche à prouver sont des propriétés numériques. Ainsi, il existe une grande diversité de domaines abstraits numériques. La précision des domaines et des invariants résultants est souvent liée au coût des opérations sur ce domaine : un domaine abstrait peu précis comme les intervalles sera très efficace mais peu expressif tandis que des domaines plus complexes comme celui des polyèdres permettra d'encoder des inégalités linéaires établissant des relations entre variables.

Pour donner une intuition graphique de quelques domaines numériques, nous prenons, dans la Figure 2.2, un programme écrit dans un sous-ensemble du langage C et nous allons calculer les invariants possibles à la fin du programme en utilisant différents domaines abstraits. Le domaine concret est représenté par l'ensemble des points présents sur la Figure 2.3a ; la zone rouge représente la zone interdite que nos états de programmes, représentés par un couple (x, y), ne doivent pas prendre. La première abstraction, Figure 2.3b, est calculée à l'aide du domaine des intervalles : nous observons que cette sur-approximation n'est pas suffisamment précise puisque le coin haut gauche de la « boîte » est en intersection avec la zone interdite. Le domaine abstrait des octogones [Min06], Figure 2.3c, permet d'établir des conjonctions de contraintes de la forme $\pm x \pm y \leq c$ où x et y sont des variables du programme et c une constante. De cette manière, il est possible d'établir des *relations* entre variables et d'affiner ainsi la précision de nos états abstraits. Graphiquement, nous constatons que le domaine des octogones est suffisant pour nous permettre de prouver la validité du programme. Le domaine des polyèdres [CH78] est plus coûteux mais aussi plus précis que les octogones et permettent d'obtenir des invariants de la forme $\alpha_1 v_1 + \ldots + \alpha_n v_n \leq c$ où $v_1 \ldots v_n$ sont des variables du programme et $\alpha_1 \ldots \alpha_n, c$ sont des constantes.

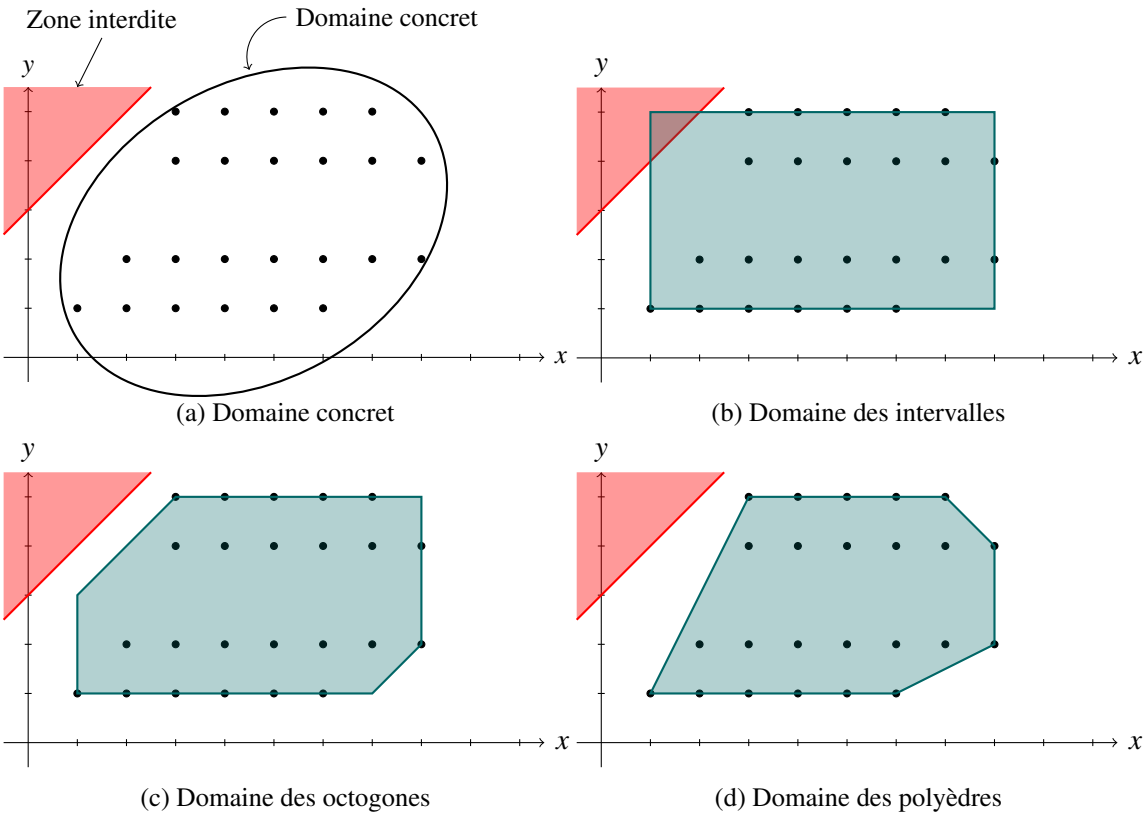
```
1  int x, y;
2  x = 1;
3  y = random(1,5);
4  while (y < 3 && x <= 8){
5    x = x + y;
6    y = 2 * y;
7  }
8  x = x - 1;
9  y = y + 1;
```

FIGURE 2.2 – Programme à vérifier

La sémantique abstraite d'un programme peut être définie via des *fonctions de transfert*. Ces fonctions sur-approximent la sémantique concrète des instructions d'un langage. Pour chaque instruction d'un langage, une fonction de transfert concrète est associée $[\![stmt]\!] : \Gamma \to \Gamma$ ainsi qu'une fonction de transfert abstraite $[\![stmt]\!]^\sharp : \Lambda \to \Lambda$ agissant dans un domaine abstrait.

Définition 6. *Une fonction de transfert pour une instruction* `stmt` *est dite sûre si :*

$$\forall s \in \Lambda,\ [\![stmt]\!](\gamma(s)) \sqsubseteq \gamma([\![stmt]\!]^\sharp(s))$$



(a) Domaine concret

(b) Domaine des intervalles

(c) Domaine des octogones

(d) Domaine des polyèdres

2.2.5 Combinaison de domaines abstraits

Comme présenté dans [CCF13], il est possible de combiner deux domaines abstraits. Une première méthode est d'effectuer un *produit cartésien*. Formellement, soit $(\mathcal{A}, \sqsubseteq_{\mathcal{A}}, \sqcap_{\mathcal{A}}, \sqcup_{\mathcal{A}})$ et $(\mathcal{B}, \sqsubseteq_{\mathcal{B}}, \sqcap_{\mathcal{B}}, \sqcup_{\mathcal{B}})$, deux domaines abstraits, le produit cartésien de ceux-ci donne le domaine abstrait $(C, \sqsubseteq_C, \sqcap_C, \sqcup_C)$ où C est le produit des éléments des treillis et la relation d'ordre $\sqsubseteq_C$ est la conjonction des relations d'ordre : $(a_1, b_1) \sqsubseteq_C (a_2, b_2) \Leftrightarrow a_1 \sqsubseteq_{\mathcal{A}} a_2 \wedge b_1 \sqsubseteq_{\mathcal{B}} b_2$ où a_1, a_2, b_1, b_2 sont des éléments des treillis. Les opérateurs de bornes sont définis par leurs applications un-à-un : $(a_1, b_1) \sqcup_C (a_2, b_2) = (a_1 \sqcup_{\mathcal{A}} a_2, (b_1 \sqcup_{\mathcal{B}} b_2))$; $(a_1, b_1) \sqcap_C (a_2, b_2) = (a_1 \sqcap_{\mathcal{A}} a_2, (b_1 \sqcap_{\mathcal{B}} b_2))$. Pareillement, il est possible de combiner les opérateurs d'élargissement et les fonctions de transfert en les appliquant aux éléments associés. On peut définir la fonction d'abstraction comme le produit des applications des deux fonctions d'abstraction sur un élément du domaine concret $\alpha_C(c) = (\alpha_{\mathcal{A}}(c), \alpha_{\mathcal{B}}(c))$. La fonction de concrétisation nécessite cependant d'effectuer l'intersection des résultats des applications : $\gamma_C(a, b) = \gamma_{\mathcal{A}}(a) \sqcap_\Gamma \gamma_{\mathcal{B}}(b)$. Se faisant, ce produit cartésien forme une correspondance de Galois avec le domaine concret Γ : $(\Gamma, \subseteq_\Gamma) \underset{\alpha_C}{\overset{\gamma_C}{\leftrightarrows}} (C, \sqsubseteq_C)$. Ainsi, la sémantique du produit cartésien est une sur-approximation sûre de la sémantique concrète.

Cependant, le produit cartésien peut contenir des éléments redondants. Par exemple, si l'on considère le produit de deux domaines numériques : celui des intervalles et celui de la parité, les éléments $([2; 4], \texttt{Impair})$, $([2; 3], \texttt{Impair})$, $([3; 4], \texttt{Impair})$ et $([3; 3], \texttt{Impair})$ peuvent être concrétisés en singleton 3.

Bien que le produit cartésien soit une manière efficace et peu coûteuse de combiner deux domaines abstraits, il est souhaitable que les domaines puissent échanger de l'information dans le but de se raffiner mutuellement. Ce mécanisme se nomme *produit réduit* [CC77, CCM11]. Soit (a, b) un élément d'un produit réduit composé de deux éléments de deux domaines abstraits différents et soit c_1 et c_2 les deux ensembles d'éléments concrets associés à a et b alors (a, b) représente l'ensemble $c_1 \cap c_2$. La réduction

essaie de trouver le plus petit élément (a', b') tel que $a' \sqsubseteq a$ et $b' \sqsubseteq b$ mais leur intersection reste égale à $c_1 \cap c_2$. La structure de treillis et la sémantique associée restent les mêmes que pour le produit cartésien mais, nous introduisons en plus un opérateur de réduction visant à raffiner les informations portées par les deux domaines et est utilisé après chaque application sémantique. L'opérateur de réduction est défini par : $\bigsqcap\{(a', b') \in \mathcal{A} \times \mathcal{B} \mid \gamma(a, b) \sqsubseteq \gamma(a', b')\}$. En général, cette définition n'est pas calculable et on cherche plutôt une version plus relâchée et peu coûteuse. Toutefois, un opérateur de réduction doit satisfaire deux propriétés : le résultat de la réduction doit être un élément abstrait plus précis $\rho(a, b) \sqsubseteq (a, b)$ et un élément abstrait et sa réduction conservent les mêmes propriétés $\gamma(\rho(a, b)) = \gamma(a, b)$.

2.2.6 Vérification des propriétés de sûreté

La finalité d'une analyse de programme est de vérifier le « bon comportement » d'un programme. Ce bon comportement peut se traduire par un ensemble de *propriétés de sûreté* que notre programme se doit de respecter afin d'assurer qu'il ne comporte pas de défaillances critiques.

En interprétation abstraite, le résultat d'une analyse est une sur-approximation de l'espace d'états atteignables. Ainsi, prouver le bon comportement d'un programme revient à montrer l'absence de *mauvais comportements* . En effet, l'abstraction engendrée peut contenir des comportements pouvant ne pas avoir lieu lors d'une exécution. En revanche, si l'abstraction ne contient pas de mauvais comportements alors nous sommes sûrs que notre programme est correct.

Formellement, soit Ω un ensemble de mauvaises configurations construites à partir des propriétés et S une sur-approximation de l'espace d'atteignabilité d'un programme :

- Si $S \cap \Omega = \emptyset$ alors les propriétés sont prouvées ;
- Sinon $S \cap \Omega \neq \emptyset$ alors deux possibilités :
  - L'abstraction contient des mauvaises configurations et celle-ci doit être raffinée, on parle alors de *fausse alarme* ;
  - L'implantation ne respecte pas la propriété et un bug est présent.

2.2.7 Outils industriels

La théorie de l'interprétation abstraite sort du cadre théorique pour concevoir de nombreux outils d'analyses statiques qui ont été développés et appliqués à des problèmes du « monde réel ». La plupart des outils existants se concentrent sur l'analyse de code C et sont généralement appliqués à des programmes de l'embarqué critique nécessitant un haut-niveau de certification.

Nous présentons quelques uns des outils reposant sur l'interprétation abstraite.

`Frama-C / EVA` — [KKP+15] Frama-C est une plate-forme extensible d'analyse statique et de spécification de programmes C. Frama-C dispose d'un système de greffons pouvant spécialiser différents types d'analyse. Parmi les greffons existants, EVA (*Evolved Value Analysis*), la nouvelle version de *Value*, utilise l'interprétation abstraite au travers d'une architecture modulaire comprenant un nombre conséquent de domaines abstraits. Frama-C est utilisé, par exemple, dans la vérification de code critique dans l'industrie du nucléaire. Par ailleurs, c'est sur cet outil que nous nous sommes basés pour réaliser l'implantation de notre prototype que nous présentons plus loin.

`Astrée` — [BCC+02, BCC+03, CCF+05, CCF+09]Astrée est un interpréteur abstrait initialement développé au sein du laboratoire d'informatique de l'École Normale Supérieure en 2001 puis acquis en 2009 par AbsInt. Astrée contient une grande variété de domaines abstraits permettant d'exprimer de

nombreux types de propriétés abstraites et d'établir des invariants numériques précis. Astrée possède un système de combinaisons de domaines abstraits basé sur des collaborations par passage de messages. Airbus compte parmi ses utilisateurs historiques.

`Verasco` — [JLB+15] Verasco est une analyseur par interprétation abstraite formellement prouvé grâce à l'assistant de preuve `Coq`. Il est intégré au compilateur pour C *CompCert* [Ler09], lui-même vérifié de la même manière. Ils collaborent de telle manière que les propriétés de sûreté établies par Verasco lors de la compilation peuvent être propagées au binaire généré.

`Polyspace` — [Deu03] Polyspace est un interpréteur abstrait de code C, C++ et ADA. Comme les outils présentés jusqu'ici, il est capable de détecter des erreurs à l'exécution et de valider des propriétés de sûreté. Il dispose d'une interface aboutie et est développé par MathWorks. Il est déployé dans différents contextes industriels (automobiles, ferroviaire, avionique, spatial, . . .).

`Clousot` — [FL10] Clousot détecte les erreurs à l'exécution et vérifie des contrats de spécifications de fonctions. Il analyse chaque fonction d'un programme de manière isolée et utilise des pré/post-conditions pour cela. Cet analyseur prend en entrée du code objet (*bytecode*) de la machine virtuelle `.Net` de Microsoft.

2.3 Vérification de programmes concurrents

La concurrence en informatique s'intéresse à l'exécution de plusieurs processus communiquant entre eux. Les processus peuvent être exécutés de manière parallèle (*i.e.* ayant chacun une unité de calcul dédiée) ou encore peuvent être agencés, par exemple, sur un seul processeur où un mécanisme d'ordonnancement leur permettra de prendre la main selon des critères spécifiés.

Aujourd'hui, la grande majorité des langages de programmation, tous paradigmes confondus, proposent des approches pour programmer de manière concurrente. Cela peut se manifester par une application web attribuant à chaque client un processus dédié pour répondre à ses requêtes, à des noyaux de calcul d'unité de calcul graphique (*Graphics Processing Unit*) pour afficher des images ou encore pour des calculs numériques divisés en sous-parties disposées sur plusieurs fils d'exécution à des fin d'accélération. Il existe aussi des propositions en vue de l'intégration de systèmes concurrents dans de l'embarqué critique [WW07].

Cependant, la concurrence est difficile à programmer et encore plus à vérifier. À la différence du séquentiel, on y trouve de nouvelles problématiques :

— L'orchestration des processus où le non-déterminisme dans la séléction des processus exécutés entraîne une explosion combinatoire du nombre d'états possibles ;
— Les situations de compétition (*data race*) où deux processus tentent d'accèder à une ressource partagée et où, au moins, l'un d'eux modifie celle-ci entraînant généralement des comportements indésirables ;
— L'interblocage où un ensemble de processus s'attendent mutuellement pour pouvoir reprendre leurs exécutions et entraînant ainsi un blocage global ;
— La famine de processus tentant d'accéder indéfiniment à une ressource « capturé » par d'autres et ne leur permettant plus de l'atteindre. Un exemple de cette situation est le dîner des philosophes formulé par Dijkstra [Dij83] : plusieurs philosophes sont disposés autour d'un repas. Il est nécessaire à un philosophe de posséder deux fourchettes pour manger son plat, or le nombre

de fourchettes est égal au nombre de philosophes ainsi si chaque philosophe en empoigne une, aucun ne sera capable de manger son plat.

Dans cette section, nous présentons quelques exemples de modèle de concurrence, puis nous introduisons quelques travaux de vérification par analyse statique, en particulier, ceux utilisant la méthodologie de l'interprétation abstraite.

2.3.1 Modèles de concurrence

Il est important de distinguer les différents traits caractérisant les modèles de concurrence. Premièrement, la mémoire peut être soit partagée, tous les processus disposent du même espace d'adressage, soit distincte (ou répartie), chaque processus possèdent un espace propre auquel lui-seul peut accéder. Le problème de la vérification pour la mémoire partagée est accru de par la présence des mémoires faiblement cohérentes [ABBM10] sur certaines architectures. Les modèles d'exécution doivent alors également prendre en compte : les optimisations du matériel, la cohérence des caches, etc. Il est nécessaire, pour une vérification sûre, de considérer ces comportements lors de l'analyse.

Les processus peuvent, ou non, communiquer entre eux. On distingue deux grandes familles de modèles de communication entre processus :

— Dans le modèle à mémoire partagée, les communications peuvent avoir lieu au travers d'une variable partagée. Il est alors important de protéger les accès en lecture et écriture pour éviter les situations de compétitions comme précédemment évoquées ;

— Il est également possible d'établir un protocole afin d'envoyer des *messages* contenant les données en travaillant par copie. Par ailleurs, il existe différentes manières d'implanter ce modèle comme, par exemple, l'utilisation d'une file d'attente pour stocker les messages en attente de réception.

Au sein du modèle de passage de messages, il est également possible de spécifier le comportement des envois et des réceptions. L'émetteur ou le récepteur peuvent attendre que l'autre participant soit prêt à effectuer la communication. On dit que les communications sont synchrones. Ou, au contraire, l'émetteur peut continuer son exécution après avoir envoyé son message. On parle alors d'asynchronisme. Ce mécanisme permet de minimiser les temps d'attente liés aux communications et d'améliorer ainsi les performances du système. Néanmoins, les communications synchrones ont pour avantage d'être conceptuellement moins complexes.

Un autre aspect de la concurrence à considérer est la possibilité qu'un programme a de créer de nouveaux processus que ceux présents à un état initial. En particulier, certains algorithmes peuvent se contenter d'un nombre fixe de processus pour effectuer leurs calculs mais il existe des situations où créer ou détruire des processus est utile. Par exemple, un serveur web peut démarrer un nouveau processus servant les requêtes d'un client lorsque celui-ci se connecte.

Lors de l'exécution, il est également important de spécifier quel processus doit avoir la priorité pour s'exécuter si leur nombre est supérieur aux unités de calculs attribuées au programme. Deux possibilités existent : le comportement préemptif, où un ordonnanceur attribue à chaque processus une portion de temps de calcul puis cède la main à un processus de manière arbitraire, généralement à l'aide d'une notion de priorité ; le comportement coopératif où chaque processus doit explicitement céder la main aux autres processus pour qu'ils puissent s'exécuter.

Nous présentons maintenant quelques uns des grands modèles de concurrences :

π-calcul

Le π-calcul [MPW92] est une algèbre de processus permettant de modéliser de manière formelle des systèmes concurrents. Un système écrit en π-calcul décrit un terme représentant un ensemble de processus $\{P, Q, R, \ldots\}$ capables de manipuler des *noms*. Ces noms décrivent des canaux de communications permettant lecture et écriture par tout processus les connaissant. Le π-calcul incorpore alors la notion de passage de message par file d'attente modélisé par les canaux. Les émissions sont, par définition, asynchrones. Il est également possible de paralléliser deux processus $P \mid Q$ ou d'effectuer un choix non-déterministe entre l'exécution de deux processus $P + Q$. Des extensions du π-calcul proposent la réplication de processus modélisant ainsi une forme de création dynamique. La vérification du π-calcul a été largement étudiée dans la littérature [Dam93, Fer05] où la méthode la plus utilisée repose sur le *model checking* que nous détaillerons par la suite.

BSP

Le modèle *Bulk Synchronous Parallel* [Val90] est un paradigme de concurrence permettant de représenter des calculs parallèles. Le programme exécuté est divisé en super étapes (*superstep*) où chacun des N processus (où N est fixé) va réaliser son calcul local, effectuer jusqu'à $N - 1$ communications (soit par passage message, soit par mémoire partagée) puis se mettre en attente via une barrière de synchronisation que tous les processus doivent atteindre pour conclure un superstep. Un tel partitionnement des calculs a pour avantage de ne pas pouvoir créer de situation de compétition puisque les barrières empêchent les dépendances circulaires entre données. De plus, l'intérêt de cette approche est de pouvoir prédire de manière relativement simple les performances d'un algorithme. En effet, le coût d'un superstep sera nécessairement le coût du processus le plus coûteux en temps. Il est également possible de modéliser des programmes BSP sous la forme d'automates [Hai17, TLHL17] introduisant alors une sémantique claire et facilitant la vérification de tels programmes.

Réseaux de Petri

Les réseaux de Petri [Pet62] sont des systèmes de transitions permettant de modéliser le comportement d'événements discrets. Plus précisément, un réseau de Petri est représenté par un graphe biparti où chaque nœud modélise soit une *place*, soit une *transition*. Deux places ou deux transitions ne peuvent être reliées entre elles. Les places contiennent des *jetons* que l'on considère comme des ressources disponibles et initialisés par une phase dite de *marquage*. Les réseaux de Petri permettent de représenter simplement et graphiquement des systèmes à la sémantique complexe. Ils ont été utilisés pour modéliser des chaînes de montages, des réseaux de transports ou encore de la télécommunication. Une des problématiques du domaine est le calcul de l'ensemble d'atteignabilité de tels systèmes : soit un réseau de Petri N et un marquage M (*i.e.* un état possible) : $M \overset{?}{\in} \text{Post}^*(N)$. Ce problème, pour un réseau de Petri classique, a été prouvé décidable [MM81] mais appartenant à la classe des problèmes EXPSPACE. C'est-à-dire, des problèmes qui peuvent être résolus en complexité $O(2^{p(n)})$ en espace avec $p(n)$ une fonction polynomiale.

Automates finis communicants

Les automates finis communicants [BZ83] sont un modèle de concurrence permettant de décrire des systèmes distribués composés de processus à états finis échangeant des messages via N files de messages. Pour illustrer, il est possible de modéliser avec des automates finis communicants un protocole

de communication entre deux ordinateurs : chaque comportement des deux ordinateurs est un processus à état fini et les files modélisent les canaux de communication. Formellement, on peut définir les automates communicants par un système de transition possédant un ensemble de localités C, un alphabet des messages $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_N$ où Σ_i désigne l'alphabet des messages pouvant être stockés dans le canal i et un ensemble de transition $\Delta \subseteq C \times A \times C$ où $A = \bigcup_i \{i\} \times \{!, ?\} \times \{\Sigma_i\}$ sont les actions (' !' pour une émission, ' ?' pour une réception). Ainsi, une configuration globale de ce type de système sera un n-uplet $\langle c, w_1, \ldots, w_N \rangle \in C \times \Sigma_1^* \times \ldots \Sigma_N^*$ où c est la localité courante et $w_i \in$ est un mot fini sur Σ_i représentant le contenu du canal i. Ainsi, une configuration $\langle c, w_1, \ldots, w_N \rangle$ est atteignable si $\langle c, \varepsilon, \ldots, \varepsilon \rangle \rightarrow^* \langle c, w_1, \ldots, w_N \rangle$. Il est cependant montré dans [BZ83] que la vérification de propriétés générales sur ce modèle est indécidable, ce qui n'empêche pas de vérifier certaines propriétés au moyen d'abstractions et/ou de semi-algorithmes [ADMP01, LGJJ06].

Systèmes paramétrés

Nous parlons de systèmes paramétrés [Tou01] pour désigner des systèmes composés d'un nombre arbitraire de processus. Ceux-ci peuvent être considérés comme des unions infinies de systèmes S_n composés de processus indistinguables organisés de manière linéaire. Pour vérifier des propriétés de sûreté sur de tels systèmes, il est nécessaire de calculer l'espace d'atteignabilité et vérifier qu'il ne contient aucune des « mauvaises » configurations. Ce problème a été montré, en général, indécidable [AK86] et des travaux se sont concentrés en l'abstraction de ce calcul ou en des semi-algorithmes. Un sous-cas de ces systèmes paramétrés se nomme *Regular Model Checking* et s'intéresse à exprimer des ensembles infinis de configurations d'états en tant que langage régulier (*c.f.* Section 2.5).

2.3.2 Analyse statique de programmes concurrents par interprétation abstraite

Dans cette section, nous nous intéressons plus particulièrement aux approches d'analyse statique utilisant la méthode de l'interprétation abstraite.

Un des premiers travaux s'intéressant à l'utilisation de l'interprétation abstraite pour la vérification de modèles de concurrence a été [Fer05] dans lequel l'auteur s'intéresse à la vérification de systèmes écrits en π-calcul. Pour ce faire, une sémantique opérationnelle du π-calcul est introduite afin de pouvoir utiliser la méthodologie de l'interprétation abstraite. Celle-ci permet de construire deux analyses distinctes : une analyse de flot de contrôle permettant de connaître les canaux de communications employés par chaque processus ; une analyse de comptage d'occurrences qui, dans ce modèle qui permet aux processus de se répliquer, permet de borner le nombre d'instances potentiellement créées. Ainsi, et grâce à la combinaison de ces deux analyses, il est par exemple possible après avoir modélisé un serveur *ftp* de conclure que le nombre maximum de processus travaillant sur des calculs ne dépassera jamais la limite imposé par le système.

Une des techniques les plus abouties et qui a mené à des prototypes utilisés industriellement est celle, proposée dans [Min12, Min13, Min14] qui a été auparavant proposée par [CH09]. L'idée développée dans ces travaux s'inspire de la méthode de preuve *rely-guarantee* [Jon83]. Celle-ci permet d'étendre la logique de Hoare aux programmes concurrents notamment en introduisant la possibilité de spécifier des attentes, sous la forme d'invariants, sur l'ensemble des fils d'exécution d'un programme.

Cette méthode s'intéresse à un modèle de concurrence comportant un ensemble fini de processus et un modèle à mémoire partagée. Pour donner une meilleure intuition du modèle, l'outil développé s'applique à la bibliothèque *POSIX Threads* [But97] sans création dynamique de processus (*i.e.* les threads

sont initialisés au début du programme dans une boucle explicitement bornée). Les communications sont donc réalisées par lectures et écritures de variables partagées. L'idée de cette méthode est de commencer par effectuer une analyse de valeurs classique (par interprétation abstraite) sur chaque thread de manière locale. Cependant, chaque écriture de variable partagée peut avoir un effet de bord sur le flot de contrôle d'un thread différent. On appelle cet effet une *interférence*. Ainsi, après cette première analyse, on collecte l'ensemble des interférences possibles en les intégrant à une nouvelle analyse de valeurs. Cependant, de nouveaux comportements peuvent être découverts et, de même, de nouvelles interférences sont possibles. Ce processus est donc réitéré jusqu'à obtenir un point-fixe. Pour illustrer ceci, considérons le programme multi-thread présenté par la Figure 2.4. Les deux threads s'exécutent de manière parallèle et influent l'un sur l'autre de par les écritures des variables partagées X et Y. En supposant $X = Y = 0$, une première analyse de valeurs, avec les intervalles pour domaine abstrait, donnerait les états finaux suivants : $\sigma_{t_1} = \{X = [0;0] ; Y = [0;0]\}$ et $\sigma_{t_2} = \{X = [0;0] ; Y = [0;5]\}$. Après cette itération, l'interférence capturée est « t_2 écrit dans Y la valeur abstraite $[1;5]$ ». En injectant cette interférence et en réitérant une analyse, nous obtenons : $\sigma_{t_1} = \{X = [0;5] ; Y = [0;5]\}$ et $\sigma_{t_2} = \{X = [0;0] ; Y = [0;5]\}$. La nouvelle interférence provient maintenant du thread t1 qui écrit dans X la valeur abstraite $[1;5]$. Après injection et nouvelle analyse, les états de threads n'évoluent plus et nous obtenons l'invariant de programme suivant : $X \in [0;5] \wedge Y \in [0;5]$. Il est à noter que nous n'obtenons pas l'invariant relationnel $X \leq Y$. Toutefois, il est possible d'obtenir cet invariant en utilisant des domaines relationnels (*e.g.* octogones, polyèdres, ...) dans le but de conserver ces relations entre variables.

Dans [CH09], les auteurs avaient proposé une approche similaire en intégrant notamment la création dynamique dans leur analyse. Cependant, leurs analyses souffraient d'une part de perte de relations dans leurs abstractions et d'autre part ne considérait pas d'ordonnancement temps-réel.

Thread t1	Thread t2
(1a) while random do	(1b) while random do
(2a) if X < Y then	(2b) if Y < 3 then
(3a) X := X + 1	(3b) Y := Y + [1,3]
(4a) endif	(4b) endif
(5a) done	(5b) done

FIGURE 2.4 – Exemple de programme concurrent

En pratique, cette méthode peut s'appliquer à des programmes de taille conséquente. Comme spécifié plus haut, il existe des prototypes utilisés dans l'industrie. Parmi ceux-ci, nous nous intéressons plus particulièrement à deux d'entre eux.

`AstréeA` — AstréeA est l'outil résultant du travail présenté ci-dessus. Il est construit comme une surcouche d'Astrée, l'analyseur statique séquentiel et profite ainsi de son analyse de valeurs préexistante. Actuellement, il est l'outil le plus abouti et inclut diverses optimisations liées à la performance et à la précision. Cet outil a été acquis par AbsInt en 2015 après l'acquisition d'Astrée en 2009.

`mThread` — mThread implante également cette méthode sous la forme d'un plug-in Frama-C. L'analyse de valeurs employée est celle de value et doit prochainement être adaptée pour pouvoir profiter des domaines relationnels d'EVA. L'outil est actuellement sous licence propriétaire.

Les analyseurs existants utilisant l'interprétation abstraite s'intéressent principalement au paradigme de mémoire partagée. Il existe cependant des méthodes étudiant les mémoires distribuées, c'est-à-dire,

aux systèmes dont chaque processus dispose de son propre espace mémoire. Dans le cadre de ce travail de thèse, nous nous sommes intéressés à la méthodologie employée par le *Regular Model Checking*. Cette technique permet de représenter des états de programme par des langages réguliers. Avant d'introduire le *Regular Model Checking*, nous présentons la théorie fondamentale des langages réguliers.

2.4 Langages réguliers et automates

Notre modélisation de systèmes concurrents va largement se baser sur une représentation sous forme d'automates. Ainsi, nous présentons maintenant, de manière non-exhaustive, la théorie des langages réguliers et des automates à états finis. Nous nous intéressons particulièrement à la définition et à deux opérations de ceux-ci : la déterminisation et la minimisation. [HMU06] propose une exploration plus large des langages réguliers et automates à états finis.

Soit Σ un ensemble fini que l'on appelle *alphabet* dont les éléments sont nommés *lettres*. Σ^* est le monoïde libre sur Σ, *i.e.* l'ensemble des *mots* finis (concaténations finies de lettres). ε est le mot vide et un mot w de taille n sera écrit : $w = a_1 \cdot a_2 \cdot \ldots \cdot a_n$. Nous nous intéressons aux langages les plus simples : les langages réguliers pouvant être reconnus par des automates à états finis.

Définition 7. *Un automate à états finis est un n-uplet $\mathcal{A} = \langle Q, \Sigma, Q_i, Q_f, \Delta \rangle$ où*

- *Q est l'ensemble fini des états de l'automate ;*
- *Σ est un ensemble fini appelé alphabet ;*
- *$Q_i \subseteq Q$ et $Q_f \subseteq Q$ sont respectivement les ensembles d'états initiaux et finaux ;*
- *$\Delta \subseteq Q \times \Sigma \times Q$ est l'ensemble des transitions.*

On note $q \xrightarrow{a} q'$ si $(q, a, q') \in \Delta$. Un mot fini $w = a_1 \cdot \ldots a_n \in \Sigma^*$ est reconnu par $\mathcal{A}$ si il existe une *séquence* de $n+1$ états : $q_0 \xrightarrow{a_1} q_1 \ldots \xrightarrow{a_n} q_n$ avec $q_0 \in Q_i$ et $q_n \in Q_f$. $L(\mathcal{A}) \subseteq \Sigma^*$ est le langage reconnu par l'automate $\mathcal{A}$. Un langage est *régulier* s'il peut être reconnu par un automate à états finis. Reg(Σ) est l'ensemble des langages réguliers.

Les opérations d'union ou d'intersection peuvent être facilement implémentées en termes d'opérations sur les automates. Par exemple, l'intersection de deux langages réguliers L_1 et L_2 reconnus par deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$ est l'automate $\mathcal{A}_3 = \langle Q^3, \Sigma, Q_i^3, Q_f^3, \Delta^3 \rangle$ tels que $Q^3 = Q^1 \times Q^2$, $Q_i^3 = Q_i^1 \times Q_i^2$, $Q_f^3 = Q_f^1 \times Q_f^2$ et $\Delta^3 = \{(p,p'), a, (q,q') \mid (p,p') \in Q^3 \wedge (q,q') \in Q^3 \wedge (p,a,q) \in \Delta^1 \wedge (p',a,q') \in \Delta^2\}$.

Les représentations d'automates finis ne sont pas canoniques, c'est-à-dire qu'un langage régulier peut être reconnu par plusieurs automates différents. Cependant, les langages réguliers possèdent une forme canonique en termes d'automates *déterministes minimaux*.

Définition 8. *Un automate à états finis $\mathcal{A} = \langle Q, \Sigma, Q_i, Q_f, \Delta \rangle$ est déterministe si :*

- *il existe un seul état initial : $|Q_i| = 1$;*
- *pour tout état $q \in Q$ et toute lettre $a \in \Sigma$, si $q \xrightarrow{a} q_1$ et $q \xrightarrow{a} q_2$ alors $q_1 = q_2$.*

Un algorithme classique [HMU06] permet de transformer un automate non déterministe $\mathcal{A} = \langle Q, \Sigma, Q_i, Q_f, \delta \rangle$ en un automate déterministe $\mathcal{A}' = \langle X, \Sigma, X_i, X_f, \Delta \rangle$ reconnaissant le même langage avec :

- $X = 2^Q$
- $X_i = \{Q_i\}$
- $X_f = \{E \subseteq Q \mid E \cap Q_f \neq \emptyset\}$

— $\Delta = \{(E_1, a, E_2) \mid \exists q_1 \in E_1, q_2 \in E_2, (q_1, a, q_2) \in \delta\}$

Cet algorithme est exponentiel puisque chaque état $x \in X$ est l'automate résultat correspondant à un ensemble d'états $x \in 2^Q$ de l'automate initial. Lorsqu'un automate est déterministe, son ensemble de transitions Δ définit une fonction de transition (nommé δ) :

$$\begin{aligned} \delta : Q \times \Sigma &\to \Sigma \\ (q, a) &\mapsto q' \text{ qui est l'unique état t.q. } (q, a, q') \in \Delta \end{aligned}$$

La clôture transitive de cette fonction est définie récursivement par :

$$\begin{aligned} \delta^* : Q \times \Sigma^* &\to \Sigma \\ (q, \varepsilon) &\mapsto q \\ (q, a, w) &\mapsto \delta^*(\delta(q, a), w) \text{ avec } a \in \Sigma \text{ et } w \in \Sigma^* \end{aligned}$$

Cette définition donne une caractérisation plus simple du langage reconnu par un automate fini déterministe ; un mot $w \in \Sigma^*$ est reconnu par un automate déterministe $\mathcal{A} = \langle Q, \Sigma, \{q_i\}, Q_f, \Delta \rangle$ si $\delta^*(q_0, w) \in Q_f$.

Les automates finis déterministes ne sont pas canoniques puisque, ici encore, un langage peut être reconnu par deux automates déterministes différents. La forme canonique est le résultat d'une opération nommée minimisation.

Un automate fini déterministe $\mathcal{A} = \langle Q, \Sigma, Q_i, Q_f, \Delta \rangle$ est *minimal* si tout autre automate déterministe $\mathcal{A}' = \langle Q', \Sigma, Q'_i, Q'_f, \Delta' \rangle$ reconnaissant le même langage $\mathcal{L}(\mathcal{A})$ possède plus d'états : $|Q| \leq |Q'|$. Pour un langage $\mathcal{L}$, il existe un unique automate déterministe minimal (à isomorphisme près) le reconnaissant. Il est possible de l'obtenir en *quotientant* l'automate par la relation d'équivalence de *Myhill-Nerode* [HMU06].

La théorie des langages réguliers est un domaine fondamental en informatique théorique. De nombreux travaux utilisent cette représentation pour, notamment, modéliser des systèmes concurrents.

2.5 *Regular Model Checking*

Le Regular Model Checking [BJNT00, AJNS04] est une technique de vérification algorithmique de systèmes paramétrés (*i.e.* possédant un nombre arbitraire de processus) où leurs configurations sont des langages réguliers sur un alphabet fini. Cette méthode utilise les langages réguliers comme une représentation d'ensembles d'états d'un système en encodant ceux-ci sous la forme d'un langage régulier pouvant être reconnu par un automate à états finis. Un mot d'un alphabet Σ représente une configuration d'un système où chaque lettre encode l'état d'un processus. Par conséquent, un ensemble de mots encode un ensemble de configurations. La relation de transition d'un système est représentée par un transducteur fini.

L'objectif de cette technique est de résoudre un problème d'atteignabilité afin de pouvoir vérifier des propriétés de sûreté. On cherche alors à calculer la clôture transitive et réflexive de la relation de transition $\mathcal{T}$ sur une configuration initiale I afin d'obtenir l'espace d'atteignabilité d'un système dans le but de vérifier des propriétés de sûreté.

$$\mathcal{T}^*(I) = \bigcup_{i \in \mathbb{N}} T^i(I)$$

Dans le cas général, ce problème est indécidable et les diverses travaux liés au domaine se concentrent sur la manière de produire des semi-algorithmes [BHV04] (algorithmes pouvant ne pas terminer) ou encore des abstractions de l'espace d'atteignabilité [AJNd02].

2.5.1 Transducteurs à états finis

Un transducteur à états finis est une extension des automates finis capable de produire une sortie. Intuitivement, un automate à états finis *reconnaît* un langage là où un transducteur à états finis va, en plus, le réécrire calculant un nouveau langage.

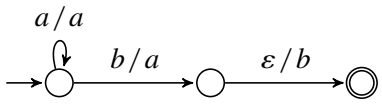
Définition 9. *Un transducteur à états finis est un n-uplet* $\mathcal{T} = \langle Q, \Sigma, \Gamma, Q_i, Q_f, \Delta \rangle$ *où*

- Q *est l'ensemble fini des états du transducteur ;*
- Σ *et* Γ *sont respectivement les alphabets d'entrée et de sortie ;*
- $Q_i \subseteq Q$ *et* $Q_f \subseteq Q$ *sont respectivement les ensembles d'états initiaux et finaux ;*
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times Q$ *est l'ensemble des transitions du transducteur.*

Par soucis de clarté, les transitions d'un transducteur seront notées σ/γ signifiant que le symbole σ lu sera réécrit en γ. Soit $w \in \Sigma^*$ un mot de l'alphabet d'entrée. On dit que w se réécrit en $w' \in \Gamma^*$, ce qui est noté $w' \in \mathcal{T}(w)$, s'il existe une séquence $q_0 \xrightarrow{w_1/w'_1} q_1 \xrightarrow{w_2/w'_2} \dots \xrightarrow{w_n/w'_n} q_n$ dans $\mathcal{T}$ telle que $w = w_1 \dots w_n$ $w' = w'_1 \dots w'_n$, $q_0 \in Q_i$ et $q_n \in Q_f$ et où w_i et w'_i peuvent être un mot vide. Par abus, on notera $\mathcal{T}(S)$ l'application du transducteur T à un ensemble de mots L, qui est formellement définie comme :

$$\mathcal{T}(L) = \bigcup_{w \in L} \mathcal{T}(w)$$

Exemple 3. *Soit le transducteur* $\mathcal{T}$ *muni d'un alphabet d'entrée/sortie* $\Sigma = \{a, b\}$. *Ce transducteur s'applique au langage régulier* $L = \{a^n b\}$, *où n est un entier fixé, en le réécrivant en* $\mathcal{T}(L) = \{a^{n+1} b\}$ *:*



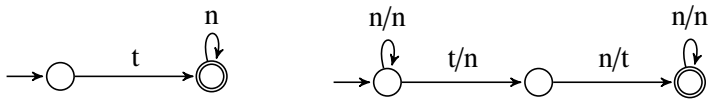
Soit $L = \{aab\}$, $\mathcal{T}(L) = \{aaab\}$, $\mathcal{T}^2(L) = \{aaaab\}$, *etc.*

Il est à noter que l'équivalence de deux transducteurs n'est pas décidable dans le cas général [Gur89]. Cependant, l'équivalence est décidable lorsque les deux transducteurs sont déterministes. Cette limitation provient du fait qu'une certaine classe de transducteurs non-déterministes ne peut être réduit à une forme déterministe.

2.5.2 Représentation des systèmes

Les configurations d'un système sont des langages usuellement représentés par des automates finis les reconnaissant. La fonction de transition d'un système est donc encodée par un transducteur. Pour illustrer cette modélisation, nous reprenons un exemple classique qui est la modélisation d'un système de passage successif d'un jeton unique dans un ensemble non-borné de processus. Les processus, ordonnés

de manière séquentielle peuvent ainsi être visualisés sous forme d'un mot et donc d'un automate (2.5a). L'alphabet considéré est : $\{n,t\}$. n codant pour l'absence de jeton et t pour la présence. Par exemple, une configuration où le 3ème processus du système disposerait du jeton serait le mot du langage $I = n{\cdot}n{\cdot}t{\cdot}n^*$. L'application du transducteur $\mathcal{T}$ (Figure 2.5b) à l'état initial I équivaut sémantiquement à passer le jeton au voisin de droite : on lit un processus possédant le jeton, on lui retire en le réécrivant en processus ne possédant pas le jeton puis on lit le processus suivant ne possédant pas de jeton pour le réécrire en un processus le possédant. Ainsi, $\mathcal{T}(I) = n \cdot n \cdot n \cdot t \cdot n^*$.



(a) Configuration initiale I (b) Transducteur $\mathcal{T}$

FIGURE 2.5 – Modélisation d'un système de passage de jeton

2.5.3 Calcul de l'espace d'atteignabilité

Comme énoncé plus haut, dans le cas général, calculer l'espace d'atteignabilité $\mathcal{T}^*(I)$ est indécidable. Même dans un cas particulier comme celui de notre exemple, l'espace d'atteignabilité exact du système (Figure 2.6) de la Figure 2.5 n'est pas calculable directement par itérations successives du transducteur. Il est donc nécessaire d'établir des *accélérations de calcul*.

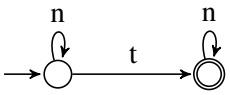


FIGURE 2.6 – Espace d'atteignabilité exact $\mathcal{T}^*(I)$

Dans la littérature, plusieurs solutions sont proposées, dont :

— L'abstraction des configurations [BHV04] permettant d'obtenir une sur-approximation calculable de l'espace d'atteignabilité. Les abstractions résultant d'une itération de point-fixe sont raffinées si la propriété de sûreté B que l'on cherche à vérifier est fausse : $\mathcal{T}^*(I) \cap B \neq \emptyset$. Si un contre-exemple est détecté, celui-ci est ôté de l'abstraction et le calcul se poursuit jusqu'à l'obtention du point-fixe.

— La technique du *quotienting* [AJNd02] s'intéresse à quotienter les transducteurs représentant des approximations de $\mathcal{T}^n$ pour des valeurs de n. Cela implique de trouver une relation d'équivalence $\simeq$ sur les états des approximations et de fusionner les états équivalents en permettant d'obtenir un *transducteur quotient*.

— La technique dite d'accélération [BLW04] consiste à essayer de « deviner » $\mathcal{T}^*$ à partir de plusieurs itérés de $\mathcal{T}$. Ce genre de techniques ne fonctionne pas dans tous les cas, et s'appuie par exemple sur des relations d'équivalence entre les états des différents itérés $\mathcal{T}$, $\mathcal{T}^2$, $\mathcal{T}^3$ etc pour détecter une possible régularité dans cette séquence et en déduire alors $\mathcal{T}^*$.

2.6 Transducteurs symboliques

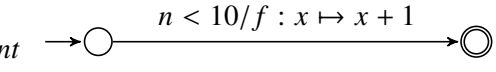
La principale limitation des transducteurs finis tels que présentés ci-dessus est la finitude de l'alphabet. Dans [VHL+12], les auteurs définissent une nouvelle structure de transducteurs capables de réécrire

des langages considérant des alphabets symboliques donc potentiellement infinis. Cette généralisation amène la nécessité de reconsidérer les transitions. En effet, les transitions des transducteurs à alphabet fini acceptent au plus une lettre et en réécrivent au plus une autre. Désormais, les gardes peuvent accepter un ensemble de valeurs et doivent donc être modélisées par un sous-ensemble de l'alphabet d'entrée. De même, la réécriture des valeurs reconnues peut générer un nouveau sous-ensemble de l'alphabet de sortie. Ainsi, la réécriture est modélisée par une fonction $f : \Sigma \mapsto \Gamma$ avec Σ et Γ respectivement les alphabets d'entrée et de sortie. Contrairement au cas des transducteurs finis, Σ et Γ ne sont pas nécessairement des ensembles finis.

Définition 10. *Un transducteur symbolique à états finis est un n-uplet $\mathcal{T}_s = \langle Q, Q_i, Q_f, \Sigma, \Gamma, \Delta \rangle$ où*

- *Q est l'ensemble fini des états du transducteur ;*
- *Σ et Γ sont respectivement les alphabets symboliques d'entrée et de sortie (ce sont des ensembles dénombrables, pas nécessairement finis) ;*
- *$Q_i \subseteq Q$ et $Q_f \subseteq Q$ sont respectivement les ensembles d'états initiaux et finaux ;*
- *$\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \mathcal{F} \times Q$ est l'ensemble des transitions du transducteur avec $\mathcal{F} : \Sigma \cup \{\varepsilon\} \to \Gamma^n$ où $n \in \mathbb{N}$.*

Les transitions sont notées $p \xrightarrow{g \ / \ f} q$ où $p, q \in Q$, g est la garde (ou condition d'entrée) et f est une fonction de réécriture pouvant générer n nouveaux symboles.

Exemple 4. *Soit $\mathcal{T}$ le transducteur suivant* 

sur l'alphabet d'entrée et de sortie $\mathbb{N}$ et soit un langage $L = \{2; 9\}$. Si l'on applique $\mathcal{T}$ à ce langage on obtient, $\mathcal{T}(L) = \{3; 10\}, \mathcal{T}^2(L) = \{4; 10\}$, etc.

Dans [VHL+12], les auteurs proposent un algorithme pour le test d'équivalence de deux transducteurs symboliques. Comme évoqué précédemment pour les transducteur à alphabet fini, l'équivalence n'est pas décidable dans le cas général mais l'est pour deux transducteurs déterministes. Ces restrictions s'appliquent également pour les transducteurs symboliques *déterministes*.

2.7 Automates de treillis

Les automates de treillis [LJ07, Le 08] définissent une structure d'automates capables de reconnaître des langages à alphabet infini. En particulier, les automates de treillis permettent de définir un domaine abstrait sur cette classe de langages avec comme élément minimal le mot vide, comme élément maximal l'automate qui reconnaît toutes les lettres, la borne inférieure est l'intersection des langages, la borne supérieure est l'union des langages et l'opérateur d'élargissement est défini plus loin dans cette section. Ce domaine abstrait agit comme un foncteur prenant en paramètre un domaine abstrait (*e.g.* un domaine numérique). Cela permet donc d'abstraire des mots mais permet également d'abstraire des structures de *files* et de *piles* [LJ07].

Définition 11. *Un automate de treillis est un n-uplet $\mathcal{A} = \langle \Lambda, Q, Q_i, Q_f, \Delta \rangle$ où :*

- *$(\Lambda, \sqsubseteq)$ est un treillis atomique ;*
- *Q est l'ensemble fini des états de l'automate ;*
- *$Q_i \subseteq Q$ et $Q_f \subseteq Q$ sont respectivement les ensembles d'états initiaux et finaux ;*
- *$\Delta \subseteq Q \times (\Lambda \backslash \{\bot\}) \times Q$ est la relation de transition.*

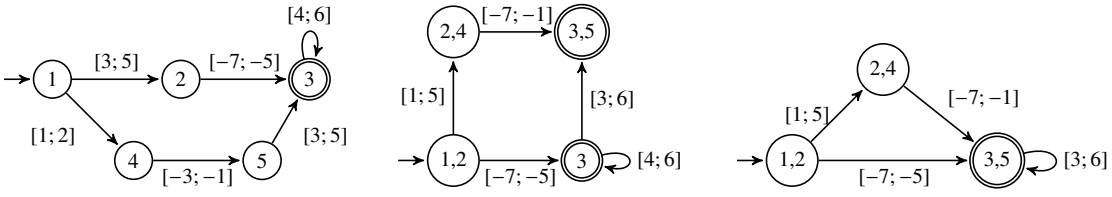
Un mot $w = a_1 \cdot \ldots \cdot a_n \in At(\Lambda)^*$ est accepté par $\mathcal{A}$ si il existe une séquence $q_0, \ldots q_n$ avec $q_0 \in Q_i$, $q_n \in Q_f$ et $\forall i \leq n, \exists (q_i, \lambda_i, q_{i+1}) \in \Delta : a_i \sqsubseteq \lambda_i$. L'ensemble des mots reconnus par $\mathcal{A}$ est noté $\mathcal{L}(A)$ et l'ensemble des langages reconnus par des automates de treillis est noté $\mathrm{Reg}(\Lambda)$. La relation d'inclusion entre langages engendre un ordre partiel sur les automates de treillis : $\mathcal{A} \sqsubseteq \mathcal{A}'$ si et seulement si $\mathcal{L}(\mathcal{A}) \sqsubseteq \mathcal{L}(\mathcal{A})'$.

Les transitions d'un automate de treillis sont étiquetées par des éléments d'un treillis atomique constituant son alphabet. Un élément $x \in \Lambda$ d'un treillis Λ est un atome si $\bot \sqsubset x \wedge \forall y \in \Lambda : \bot \sqsubset y \wedge y \sqsubseteq x \Rightarrow y = x$. L'ensemble des atomes d'un treillis est noté $At(\Lambda)$. Un treillis Λ est dit atomique si tout élément $x \in \Lambda$ est la plus petite borne supérieure des atomes plus petits que lui-même : $\bigsqcup\{a \mid a \in At(\Lambda) \wedge a \sqsubseteq x\}$. Une partition finie d'un treillis est un ensemble fini $(\lambda_i)_{0 \leq i < n}$ d'éléments tels que $\forall i \neq j : \lambda_i \sqcap \lambda_j = \bot$ et $\forall \lambda \in \Lambda : \lambda = \bigsqcup_{i=0}^{n-1}(\lambda \sqcap \lambda_i)$. Si le treillis est atomique, il existe un isomorphisme entre un élément $\lambda \in \Lambda$ et sa projection $\langle \lambda \sqcap \lambda_0, \ldots, \lambda \sqcap \lambda_{n-1} \rangle$ sur la partition. Une *fonction de partitionnement* finie : $\pi : \Sigma \to \Lambda$ est une fonction telle que $(\pi(\sigma)_{\sigma \in \Sigma})$ est une partition finie de Λ. En particulier, l'utilisation de partitions facilite la définition d'opérations sur les automates de treillis. Les transitions sont alors fusionnées pour ne conserver qu'une seule transition par classe d'équivalence. Cette fusion engendre une sur-approximation du langage reconnu.

Définition 12. *Un automate de treillis partitionné $\mathcal{A} = \langle \Lambda, \pi, Q, Q_i, Q_f, \Delta \rangle$ est un automate de treillis muni d'une fonction de partitionnement finie $\pi : \Sigma \to \Lambda$ telle que la relation de transition satisfait :* $(q, \lambda, q') \in \Delta, \exists \sigma \in \Sigma : \lambda \sqsubseteq \pi(\sigma)$.

On peut définir, shape($\mathcal{A}$), la forme d'un automate de treillis partitionné $\mathcal{A}$, qui est l'automate fini obtenu en remplaçant chaque transition par la classe d'équivalence à laquelle elle appartient.

La déterminisation des automates de treillis est semblable à celui des automates finis : on considère les ensembles états de l'automate de départ et, à chaque étape, on regarde pour chaque classe d'équivalence $\sigma \in \Sigma$ l'ensemble des états pouvant être atteints par une transition étiquetée par un élément de cette classe d'équivalence. La différence est que l'on fusionne les transitions appartenant à la même classe d'équivalence. L'algorithme de minimisation suit le même principe : on minimise la forme de l'automate de treillis, en fusionnant au besoin les transitions.



(a) Automate de départ (b) Automate déterministe (c) Automate minimal

FIGURE 2.7 – Déterminisation et minimisation d'un automate de treillis muni de la partition $\{]-\infty; 0], [0; +\infty[\}$

Ces deux opérations engendrent des sur-approximations (voir Figure 2.7). Les langages $\det(\mathcal{A})$ et $\min(\mathcal{A})$ reconnaissant des langages plus grands que $\mathcal{A}$. Cependant ces sur-approximations sont optimales au sens où pour tout automate fusionné et déterministe $\mathcal{A}'$ basé sur la même partition que $\mathcal{A}$, $\mathcal{A} \sqsubseteq \mathcal{A}' \Rightarrow \det(\mathcal{A}) \sqsubseteq \mathcal{A}'$. De même, il existe un unique automate minimal $\widehat{\mathcal{A}}$ basé sur la même partition π tel que : $\mathcal{A} \sqsubseteq \widehat{\mathcal{A}}$ et pour tout automate minimal $\mathcal{A}'$ basé sur la même partition π, $\mathcal{A} \sqsubseteq \mathcal{A}' \Rightarrow \widehat{\mathcal{A}} \sqsubseteq \mathcal{A}'$.

Cette forme, déterministe et minimale, permet de donner une représentation canonique des langages reconnus par un automate de treillis qui sera utilisée pour définir un opérateur d'élargissement.

Opérateur d'élargissement des automates de treillis

Les automates de treillis définissent un domaine abstrait lui-même paramétré par un treillis Λ. Nous supposons que ce dernier dispose d'un opérateur d'élargissement ∇_Λ.

Pour obtenir un opérateur d'élargissement des automates de treillis, nous considérons tout d'abord un opérateur ρ défini à partir d'une relation d'équivalence sur les états de l'automate notée $\approx$. Si $\mathcal{A} = \langle \Lambda, \pi, Q, Q_i, Q_f, \Delta \rangle$ est un automate de treillis partitionné $\rho(\mathcal{A}) = \langle \Lambda, \pi, \overline{Q}, \overline{Q_i}, \overline{Q_f}, \overline{\Delta} \rangle$ est l'automate défini par :

— $\overline{Q} = Q/\approx$ est l'ensemble des classes d'équivalence de $\approx$
— $\overline{Q_i} = \{\overline{q} \in \overline{Q} | \overline{q} \cup Q_i \neq \emptyset\}$
— $\overline{Q_f} = \{\overline{q} \in \overline{Q} | \overline{q} \cup Q_f \neq \emptyset\}$
— $\overline{\Delta} = \{(\overline{q_1}, \lambda, \overline{q_2}) \in \overline{Q} \times \Lambda \times \overline{Q} \mid (q_1, \lambda, q_2) \in \Delta\}$

C'est donc une opération de quotient assez classique. Le choix de la relation d'équivalence utilisée pour cette opération a été discuté dans [Le 08]. En particulier, on peut choisir une famille de relations d'équivalence $(\approx_k)_{k \in \mathbb{N}}$, comme la bisimulation bornée à la profondeur k et obtenir ainsi une famille d'opérateurs ρ_k dont la précision croît avec k.

Soient maintenant deux automates de treillis $\mathcal{A}_1$ et $\mathcal{A}_2$ normalisés et définis sur la même partition. On suppose $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$, et on définit l'opérateur d'élargissement ∇_k ainsi :

$$\mathcal{A}_1 \nabla_k \mathcal{A}_2 = \begin{cases} \widetilde{\rho_k(\mathcal{A}_2)} & \text{si } \text{shape}(\mathcal{A}_1) \neq \text{shape}(\mathcal{A}_2) \\ \mathcal{A}_1 \nearrow \mathcal{A}_2 & \text{sinon} \end{cases}$$

Autrement dit, si $\text{shape}(\mathcal{A}_1) \neq \text{shape}(\mathcal{A}_2)$, on se contente de quotienter le plus grand automate. Sinon, puisque les deux automates sont normalisés où $\mathcal{A}_1 \nearrow \mathcal{A}_2$ est l'automate de treillis déterministe minimal ayant le même ensemble d'états que $\mathcal{A}_1$ et $\mathcal{A}_2$ où on applique l'opérateur d'élargissement du treillis entre les étiquettes des transitions équivalentes des deux automates. La preuve que ∇_k est bien un opérateur d'élargissement est donnée dans [Le 08].

2.8 Conclusion

Nous avons vu, dans ce chapitre, quelques notions de base sur la vérification de systèmes communicants et les différentes techniques existant actuellement pour aborder cette question.

De manière générale, pour vérifier formellement une propriété de sûreté sur un système de transition, il faut être capable de calculer, d'une manière ou d'une autre, l'ensemble des états atteignables. Or ce calcul n'est pas faisable dans tous les cas, tant pour des raisons théoriques (indécidabilité) que pratiques (explosion du nombre d'états). Aussi, un certain nombre de techniques, dont l'interprétation abstraite, ont été utilisées pour calculer une sur-approximation de cet ensemble d'états atteignables qui est, elle, effectivement calculable. L'interprétation abstraite a été utilisée avec succès pour la vérification de programmes séquentiels, notamment ceux qui font du calcul numérique. Des travaux récents l'ont également utilisée pour certains programmes concurrents.

Cependant, d'autres techniques existent pour vérifier de tels systèmes communicants, comme le *Regular Model Checking*. Cette dernière technique, qui représente des ensembles d'états par des automates

à états finis, est toutefois limitée à la vérification de systèmes dont tous les processus sont modélisables par des machines à états finis, et ne permet pas le calcul arithmétique. Or le principal usage des programmes parallèles est justement de faire plus rapidement des calculs numériques.

C'est pourquoi nous proposons, dans les prochains chapitres, d'utiliser le même genre de technique que le *Regular Model Checking* mais en remplaçant les automates à états finis par les automates de treillis.

Chapitre 3

Analyse statique de processus communiquants par voisinage

Dans ce chapitre, nous introduisons la classe de systèmes concurrents que nous souhaitons traiter. Nous introduisons un langage relativement simple capable de décrire ces systèmes concurrents et nous détaillerons les choix de modélisation que nous avons faits. Nous explicitons ensuite la sémantique formelle de ce langage grâce à une extension des transducteurs symboliques dont nous donnons la définition. Pour pouvoir vérifier un tel système, nous nous basons sur l'interprétation abstraite et définissons une abstraction de la sémantique précédemment établie. Grâce à cela, nous construisons une analyse de valeurs calculant une sur-approximation de l'espace d'états du programme. Cette analyse s'appuie sur les concepts utilisés classiquement en interprétation abstraite : domaine abstrait (basé sur les automates de treillis), fonctions de transfert, opérateur d'élargissement, etc.

3.1 Modélisation du système

Dans un premier temps, il est nécessaire d'identifier la classe de problèmes que nous souhaitons traiter. En particulier, déterminer l'expressivité des programmes, le modèle mémoire, le modèle de communication ainsi que les aspects de réplication de processus. Notre choix s'est porté sur l'étude d'un système concurrent possédant les caractéristiques suivantes :

— Calcul algébrique – Les processus de notre système doivent être capables d'effectuer des opérations arithmétiques classiques. Cela nous donne des systèmes plus généraux que des machines à états finis ;

— Mémoire distribuée – Chaque processus possède sa propre zone mémoire, dans laquelle il peut stocker des valeurs numériques. Cela implique que chaque processus ne peut pas être modélisé par une machine à états finis ;

— Communications synchrones par messages – Les processus doivent être capables de communiquer entre eux. Nous faisons le choix de nous limiter à des communications synchrones : deux processus doivent être prêts à communiquer entre eux pour que la communication soit effective. De plus, les communications s'effectuent par passage d'un message contenant les données transmises, autrement dit nous autorisons la communication de valeurs numériques ;

— Création et destruction dynamique – Nous souhaitons que les processus soient à même, au cours de leurs exécutions, de lancer de nouveaux processus et également de s'arrêter.

En choisissant ces traits, nous nous donnons la possibilité de traiter une classe relativement générale de programmes. Nous pourrons ainsi traiter des programmes intéressants tels que des algorithmes de calculs parallèles, des serveurs web ou encore des protocoles de communications complexes. De plus, nous nous démarquons significativement de la littérature existante en proposant une modélisation permettant d'exprimer des processus effectuant communications et calculs numériques là où chaque processus était restreint à une machine à états finis. Nous souhaitons également permettre la création dynamique de nouveaux processus ce qui amène une explosion combinatoire supplémentaire et qui n'est généralement que peu traitée.

3.2 Langage de processus communiquants

Nous présentons, en Figure 3.1, la grammaire d'un langage concurrent. Ce langage impératif va permettre de décrire des systèmes concurrents où plusieurs processus s'exécutent en parallèle. Chacun de ces processus va pouvoir effectuer des calculs numériques, réaliser des communications synchrones au sein du système et créer/détruire dynamiquement de nouveaux processus.

Notre langage possède une structuration par *processus nommés* : chaque processus exécutera le code de son bloc nommé. Chaque programme débute avec un seul processus lancé et exécutant le bloc `main`. Ce processus peut alors créer de nouveaux processus grâce aux instructions `createL` et `createR` recevant des expressions permettant de les initialiser. La mémoire est *distribuée*, ainsi chaque processus possède sa propre zone mémoire. De ce fait, il n'existe pas de notion de variable partagée. Les communications sont *synchrones* et doivent être symétriques : deux processus peuvent communiquer si l'un est en état d'envoi et l'autre en état de réception. Une fois la communication effectuée, les processus peuvent progresser à nouveau. Chaque progression des processus est indépendante et entraîne donc des exécutions non-déterministes.

Nos programmes possèdent une *notion d'ordre* entre processus : chacun d'entre eux sera disposé sur un *vecteur* entraînant une notion de voisinage entre processus. Dans notre langage, nous limitons la création de nouveaux processus et les communications au voisinage de la position des processus : *i.e.* à gauche et/ou à droite. Ainsi, les nouveaux processus créés seront disposés à droite (resp. à gauche) de leur créateur qui vient d'exécuter l'instruction `createR` (resp. `createL`). Par exemple, un programme contenant initialement un processus P_0 qui va créer un processus P_1 à sa droite puis, de nouveau, un processus P_2 à sa droite sera respectivement dans l'état $\langle P_0 \rangle$, $\langle P_0 \cdot P_1 \rangle$, $\langle P_0 \cdot P_2 \cdot P_1 \rangle$. Le même mécanisme est appliqué aux communications qui limiteront leur portée aux processus situés dans leur voisinage. Les communications sont synchrones ce qui signifie que dans le cas où un processus tente de communiquer avec l'un de ses voisins qui n'est pas en attente de communication, il sera bloqué jusqu'à ce que son voisin soit prêt à communiquer. De plus, si aucun voisin n'est présent pour communiquer (*e.g.* un processus communiquant à sa gauche tout en étant au début du vecteur) le processus sera bloqué de même.

Nous donnons en exemple, Figure 3.2, un programme reprenant l'idée du passage de jeton (*c.f.* Figure 2.5) mais en y introduisant des calculs numériques. Dans cet exemple, le processus initial `main` crée 100 processus `token` à sa droite ($\langle M_0 \cdot T_{100} \cdot \ldots T_1 \rangle$) qu'il initialise avec des entiers aléatoires compris entre 0 et 9. Après la phase de création, il envoie la valeur 0 au processus situé à sa droite pour démarrer le calcul puis s'auto-détruit. Les processus `token_process` vont, quant à eux, se mettre en attente d'une communication provenant de leur gauche et conserver la valeur maximum entre la valeur reçue et celle passée à l'initialisation. Finalement, ils envoient à leur tour leur résultat à leur droite avant

⟨*program*⟩ : := `main` () : ⟨*instr*⟩ ⟨*proc_list*⟩

⟨*proc_list*⟩ : := ⟨*proc*⟩ ⟨*proc_list*⟩
| ε

⟨*proc*⟩ : := ⟨*proc_ident*⟩ (⟨*var_list*⟩) : ⟨*instr*⟩

⟨*instr*⟩ : := ⟨*instr*⟩ ; ⟨*instr*⟩
| ⟨*var*⟩ := ⟨*expr*⟩
| **if** ⟨*expr*⟩ **then** ⟨*instr*⟩ [**else** ⟨*instr*⟩] **endif**
| **while** ⟨*expr*⟩ **do** ⟨*instr*⟩ **done**
| **L !** ⟨*expr*⟩ Envoi synchrone vers le voisin *de gauche*
| **L ?** ⟨*var*⟩ Réception synchrone depuis le voisin *de gauche*
| **R !** ⟨*expr*⟩ Envoi synchrone vers le voisin *de droite*
| **R ?** ⟨*var*⟩ Réception synchrone depuis le voisin *de droite*
| **createL** ⟨*proc_ident*⟩ (⟨*expr_list*⟩) Création dynamique d'un nouveau processus *à gauche*
| **createR** ⟨*proc_ident*⟩ (⟨*expr_list*⟩) Création dynamique d'un nouveau processus *à droite*
| **stop** Terminaison du processus appelant

⟨*expr*⟩ : := (⟨*expr*⟩)
| ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩
| min (⟨*expr_list*⟩)
| max (⟨*expr_list*⟩)
| ⟨*int*⟩
| ⟨*bool*⟩
| ⟨*var*⟩
| random()

⟨*op*⟩ : := + | - | / | * | %
| < | > | <= | => | = | !=

⟨*var*⟩ : := [a-z][0-9a-zA-Z_]*

⟨*proc_ident*⟩ : := [a-z][0-9a-zA-Z_]*

FIGURE 3.1 – Grammaire

de s'arrêter via la primitive **stop**. Le 100ème processus sera cependant bloqué au dernier envoi à droite puisqu'aucun processus n'est disponible.

3.3 Sémantique du système

Nous avons vu qu'en *Regular Model Checking*, on représente chaque processus d'un système par une lettre d'un alphabet fini, et un état global du système par un mot (suite finie de lettres), et donc, un ensemble d'états de programme par un ensemble de mots, c'est-à-dire un langage. Si ce langage est régulier alors il peut être reconnu par un automate à états finis. La relation de transition est modélisée par un transducteur à états finis, qui en plus de reconnaître un langage à la manière d'un automate, va également réécrire ce langage. De cette manière, le transducteur permet d'encoder la sémantique du système. Il est donc possible de représenter des systèmes paramétrés, donc à états infinis, en utilisant des automates et transducteurs à états finis.

Nous nous inspirons de ces mêmes concepts : notre représentation des configurations d'un système

```
1  main () :
2    n := 0;
3    while n < 100 do
4      createR token_process(random() % 10)
5      n := n + 1
6    done;
7    R! 0;
8    stop
9
10 token_process (val) :
11   L? left_val;
12   val := max(left_val, val);
13   R! val;
14   stop
```

FIGURE 3.2 – Passage de jeton avec calcul numérique

sera des ensembles de mots de processus et notre relation de transition prendra la forme d'un transducteur. Cependant, dans notre modèle, nos processus peuvent prendre une infinité d'états. Ainsi, il nous est nécessaire d'encoder nos processus par des lettres d'un alphabet infini. De même, pour notre relation de transitions, les transducteurs à états finis utilisés en *Regular Model Checking* ne suffisent plus. Nous allons alors utiliser des transducteurs symboliques (*c.f.* Section 2.6), pouvant considérer des alphabets infinis, afin de réécrire nos configurations.

Pour notre langage, nos états de processus seront représentés par un couple $\mathcal{L} \times \mathcal{E}$ où :

— $\mathcal{L}$ est une *localité* – Les localités sont encodées par l'instruction que le processus va exécuter *lors de son prochain pas d'exécution*. Pour des soucis de clarté, nous nous référons à la ligne correspondante dans le programme. Ainsi, L_4 (ou $\mathbf{main}_{L_4}$) dans le programme de la Figure 3.2 est la localité d'un processus qui viendrait d'évaluer la condition de la boucle et qui s'apprête à effectuer une création dynamique ;

— $\mathcal{E}$ est un *environnement* – Les environnements sont des fonctions $\mathcal{E} : \mathbb{V} \to \mathbb{Z}$ où $\mathbb{V}$ représente le domaine des variables.

Un processus à la localité $\mathbf{main}_{L_4}$ possédant une variable x ayant pour valeur possible les entiers de 1 à 10 sera noté ainsi : $\langle \mathbf{main}_{L_4}, \{x \mapsto \{1 \dots 10\}\}\rangle$.

Pour illustrer le comportement de nos systèmes, nous prenons l'exemple du programme effectuant un passage de jeton avec calcul numérique (Figure 3.2) et dont les états concrets sont présentés dans la Figure 3.3. À l'étape 1, seul le processus **main** est présent à la localité L_2 (*i.e.* ligne 2 du programme) avec une variable n non-initialisée que nous supposerons comme pouvant prendre toutes les valeurs possibles. Puis, ce premier processus évolue *localement* jusqu'à atteindre la localité L_4 où un processus va être créé *à sa droite* dans le mot. Le nouveau processus **token_process** créé est à la localité L_{11} et attend une communication de son voisin de gauche ; il ne peut donc pas évoluer. Parallèlement, le processus **main** exécute la boucle jusqu'à avoir créé 100 processus **token_process** tous bloqués (*N.B.* les identifiants notés sur chaque **token_process** ne sont présents que pour une meilleure compréhension). Ensuite, à l'étape 14, le processus **main** envoie l'entier 0 à son voisin de droite simulant un passage de token et débloquant l'exécution de celui-ci avant de s'auto-détruire après l'étape 15. Puis, chaque processus **token_process** situé à la gauche du mot va, séquentiellement et après avoir reçu une valeur du processus anciennement à sa gauche, calculer le maximum entre sa valeur d'initialisation et celle reçue, envoyer ce résultat au processus de droite puis s'auto-détruire. Finalement, le dernier **token_process** restant sera

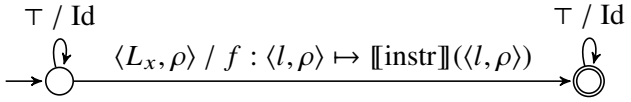
1. $\langle \mathbf{main}_{L_2}, \{n \mapsto \mathbb{Z}\}\rangle$
2. $\langle \mathbf{main}_{L_3}, \{n \mapsto 0\}\rangle$
3. $\langle \mathbf{main}_{L_4}, \{n \mapsto 0\}\rangle$
4. $\langle \mathbf{main}_{L_5}, \{n \mapsto 0\}\rangle \cdot \langle \mathbf{tok\_proc}^0_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle$
5. $\langle \mathbf{main}_{L_6}, \{n \mapsto 1\}\rangle \cdot \langle \mathbf{tok\_proc}^0_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle$
6. $\langle \mathbf{main}_{L_3}, \{n \mapsto 1\}\rangle \cdot \langle \mathbf{tok\_proc}^0_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle$
7. $\langle \mathbf{main}_{L_4}, \{n \mapsto 1\}\rangle \cdot \langle \mathbf{tok\_proc}^0_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle$
8. $\langle \mathbf{main}_{L_5}, \{n \mapsto 1\}\rangle \cdot \langle \mathbf{tok\_proc}^1_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle \cdot \langle \mathbf{tok\_proc}^0_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle$
9. $\dots$
10. $\langle \mathbf{main}_{L_5}, \{n \mapsto 99\}\rangle \cdot \langle \mathbf{tok\_proc}^{99}_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle \cdot \dots \cdot \langle \mathbf{tok\_proc}^0_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle$
11. $\langle \mathbf{main}_{L_5}, \{n \mapsto 99\}\rangle \cdot \langle \mathbf{tok\_proc}^{99}_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle \cdot \dots \cdot \langle \mathbf{tok\_proc}^0_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle$
12. $\langle \mathbf{main}_{L_5}, \{n \mapsto 99\}\rangle \cdot \langle \mathbf{tok\_proc}^{99}_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle \cdot \dots \cdot \langle \mathbf{tok\_proc}^0_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle$
13. $\langle \mathbf{main}_{L_3}, \{n \mapsto 100\}\rangle \cdot \langle \mathbf{tok\_proc}^{99}_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle \cdot \dots \cdot \langle \mathbf{tok\_proc}^0_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle$
14. $\langle \mathbf{main}_{L_7}, \{n \mapsto 100\}\rangle \cdot \langle \mathbf{tok\_proc}^{99}_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle \cdot \dots \cdot \langle \mathbf{tok\_proc}^0_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle$
15. $\langle \mathbf{main}_{L_8}, \{n \mapsto 100\}\rangle \cdot \langle \mathbf{tok\_proc}^{99}_{L_{12}}, \{n \mapsto \{0 \dots 9\}\ ;\ \texttt{left\_val} \mapsto 0\}\rangle \cdot \dots$
16. $\langle \mathbf{tok\_proc}^{99}_{L_{12}}, \{n \mapsto \{0 \dots 9\}\ ;\ \texttt{left\_val} \mapsto 0\}\rangle \cdot \langle \mathbf{tok\_proc}^{98}_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle \cdot \dots$
17. $\langle \mathbf{tok\_proc}^{99}_{L_{13}}, \{n \mapsto \{0 \dots 9\}\ ;\ \texttt{left\_val} \mapsto 0\}\rangle \cdot \langle \mathbf{tok\_proc}^{98}_{L_{11}}, \{n \mapsto \{0 \dots 9\}\}\rangle \cdot \dots$
18. $\langle \mathbf{tok\_proc}^{99}_{L_{14}}, \{n \mapsto \{0 \dots 9\}\ ;\ \texttt{left\_val} \mapsto 0\}\rangle \cdot \langle \mathbf{tok\_proc}^{98}_{L_{12}}, \{n \mapsto \{0 \dots 9\}\ ;\ \texttt{left\_val} \mapsto \{0 \dots 9\}\}\rangle \cdot \dots$
19. $\langle \mathbf{tok\_proc}^{98}_{L_{12}}, \{n \mapsto \{0 \dots 9\}\ ;\ \texttt{left\_val} \mapsto \{0 \dots 9\}\}\rangle \cdot \langle \mathbf{tok\_proc}^{97}_{L_{11}}, \{n \mapsto \{0 \dots 9\}\ ;\ \texttt{left\_val} \mapsto \{0 \dots 9\}\}\rangle \cdot \dots$
20. $\dots$
21. $\langle \mathbf{tok\_proc}^0_{L_{13}}, \{n \mapsto \{0 \dots 9\}\ ;\ \texttt{left\_val} \mapsto \{0 \dots 9\}\}\rangle$

FIGURE 3.3 – États concrets du programme de passage de token avec calcul numérique

bloqué à la localité L_{13} puisqu'il n'y a aucun processus à sa droite pour effectuer une communication avec dans son environnement la valeur du maximum de tous les processus.

Cet exemple met en évidence le comportement des fonctions de transition. Celles-ci agissent : soit localement (affections, branchements, ...) en modifiant un simple état de processus ou soit globalement (créations/destructions dynamiques, communications) en modifiant un état du système (*i.e.* un mot d'états de processus).

Ces fonctions de transitions $\mathcal{T} : (\mathcal{L} \times \mathcal{E})^* \to (\mathcal{L} \times \mathcal{E})^*$ peuvent être encodées de manière naturelle vers un transducteur symbolique. Par exemple :



Informellement, ce transducteur reconnaît des états de système composés de deux ensembles quelconques de processus qui vont se réécrire en l'identité lesquels entourent un processus dont l'état satisfait la garde $\langle L_x, \rho\rangle$ se réécrivant en modifiant son état selon l'évaluation sémantique d'une instruction $[\![\cdot]\!] : (\mathcal{L} \times \mathcal{E}) \to (\mathcal{L} \times \mathcal{E})$. Par ailleurs, l'évaluation d'une expression sera notée par la fonction $\texttt{eval} : \text{expr} \times \mathcal{E} \to \mathbb{Z}$ et le test conditionnel par $\texttt{assume} : \text{cond} \times \mathcal{E} \to \mathcal{E}$.

Cette forme simple de transducteur permet d'encoder de manière générale la sémantique des instructions locales de nos systèmes. Nous détaillerons par la suite comment encoder la sémantique des instructions globales.

3.3.1 Définition du transducteur

Nous donnons maintenant la définition du transducteur étendu que nous allons utiliser afin d'encoder la sémantique de nos systèmes concurrents sous la forme d'une relation de transition. En réécrivant les configurations de système modélisé par des automates. La différence avec la définition des transducteurs symboliques donnée dans la section 2.6 (ou avec celle donnée par Veanes et al. dans [VHL+12]), est la possibilité de construire des gardes et des fonctions de réécriture de longueur finie sur la même transition, que nous appelons *gardes multiples*. Ces gardes multiples permettent de reconnaître, via une seule et même transition, un nombre fini de lettres. En faisant de la sorte, nous donnons à nos fonctions de réécriture autant d'arguments que de lettres reconnues. Dans notre contexte, ceci permet aux transitions de reconnaître un nombre borné de processus et de les réécrire en considérant les états des autres processus acceptés. Ce mécanisme va nous servir, entre autres, à la modélisation de communications entre processus (*c.f.* 3.3.2).

Définition 13. *Un transducteur étendu est un n-uplet* $\mathcal{T} = \langle \Sigma, Q, Q_0, Q_f, \Delta \rangle$ *composé de :*

- Σ *l'alphabet d'entrée (qui est aussi l'alphabet de sortie) ;*
- Q *l'ensemble fini des états du transducteur ;*
- $Q_i \subseteq Q$ *l'ensemble fini des états initiaux ;*
- $Q_f \subseteq Q$ *l'ensemble fini des états finaux ;*
- $\Delta \subseteq Q \times \Sigma^n \times (\Sigma^n \to \Sigma)^m \times Q$ *l'ensemble fini des transitions avec* $n, m \in \mathbb{N}$.

Dans le cas de notre analyse, on aura $\Sigma = \mathcal{P}(\mathbb{Z}^d)$ où $\mathcal{P}(\mathbb{Z}^d)$ représente l'ensemble des états d'un processus à d variables entières. Les transitions des transducteurs étendus sont composées d'un état de départ q et de fin q', d'un ensemble de gardes $\gamma_1 \cdot \gamma_2 \cdot \ldots \cdot \gamma_n$, d'un ensemble de fonctions de réécriture $f_1(\sigma_1, \ldots, \sigma_n) \cdot \ldots \cdot f_m(\sigma_1, \ldots, \sigma_n)$. Les fonctions de réécriture associées reçoivent donc autant d'arguments que de gardes données. Dans la suite de ce document, nous noterons ces transitions ainsi :

$$q \xrightarrow{\gamma_1 \cdot \gamma_2 \cdot \ldots \cdot \gamma_n \;/\; f_1(\sigma_1, \ldots, \sigma_n) \cdot \ldots \cdot f_m(\sigma_1, \ldots, \sigma_n)} q'$$

La fonction de réécriture induite par ce transducteur est définie comme pour la Définition 9. Soit $w \in \Sigma^*$. On dit que w se réécrit en $w' \in \Sigma^*$, ce qui est noté $w' \in \mathcal{T}(w)$, s'il existe une séquence $q_0 \xrightarrow{w_1/w'_1} q_1 \xrightarrow{w_2/w'_2} \ldots \xrightarrow{w_n/w'_n} q_n$ dans $\mathcal{T}$ telle que $w = w_1 \ldots w_n$ $w' = w'_1 \ldots w'_n$, $q_0 \in Q_i$ et $q_n \in Q_f$. La seule différence est que les w_i et w'_i ne sont plus des lettres mais des mots. Par abus, on notera $\mathcal{T}(S)$ l'application du transducteur T à un ensemble de mots S, qui est formellement définie comme :

$$\mathcal{T}(S) = \bigcup_{w \in S} \mathcal{T}(w)$$

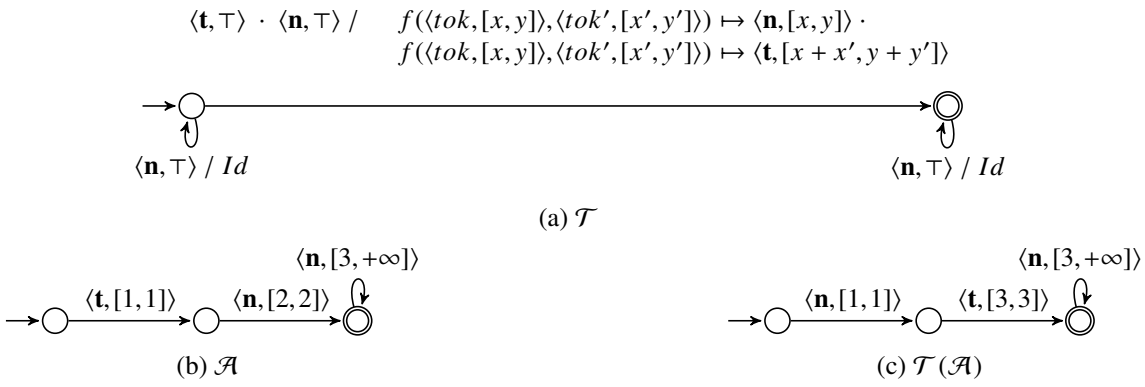


FIGURE 3.4 – Passage de jeton et addition de valeur

3.3.2 Modélisation des communications

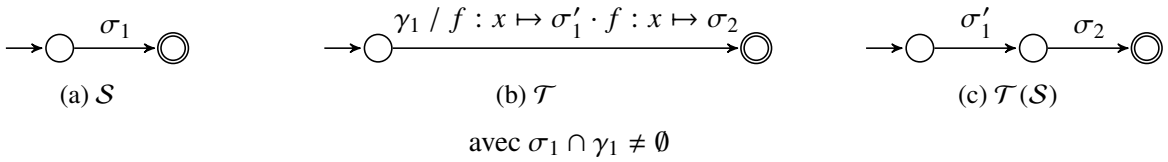
Habituellement, en *Regular Model Checking*, les communications dans un système sont exprimées via le transducteur par une suite de transitions définissant un chemin empruntable lorsque le système est dans un état particulier (*e.g.* échange de token Figure 2.5). Ce mécanisme est suffisant lorsque l'on manipule des structures à alphabet fini. Cependant, pour un alphabet infini, nous ne pouvons pas représenter ces communications sous cette forme. En effet, cela nécessiterait de traiter, pour chaque communication, toutes les valeurs possibles reconnues par les gardes et nécessiterait un nombre, potentiellement infini, d'états et/ou de transitions. Pour cette raison, nous encodons nos communications à l'aide de transitions possédant des gardes multiples. Se faisant, nos fonctions de réécriture associées reçoivent en argument chacun des éléments gardés. Ainsi, nous donnons aux fonctions de réécriture les contextes complets des processus effectuant une communication.

Nous illustrons ce mécanisme par la Figure 3.4 où l'on manipule des processus dont les états sont représentés par le produit de deux treillis : celui des jetons (*i.e.* un treillis composé de deux atomes : $\{\mathbf{t}, \mathbf{n}\}$) et celui des intervalles. Le transducteur étendu que nous donnons stipule qu'une communication n'est possible entre deux processus que lorsque le premier possède le jeton et est suivi par un second processus sans le jeton. La sémantique de cette communication est de transmettre le jeton au deuxième et d'ajouter sa valeur à celle du processus recevant le jeton. Comme les états des deux processus reconnus par la garde sont passés en paramètre de chaque fonction de réécriture, celles-ci sont alors capables de raisonner sur les deux états pour en calculer un nouveau. Ici, cela nous permet d'effectuer une addition des valeurs de deux processus.

3.3.3 Création et destruction dynamique

Dans notre représentation, nous encodons la création et destruction de processus par une réécriture d'un état de processus supplémentaire. Pour illustrer, si l'on donne une garde reconnaissant une lettre et en fournissant une réécriture générant une lettre supplémentaire, nous pouvons simuler une création dynamique d'un nouveau processus dans une configuration. Symétriquement, en reconnaissant une lettre que l'on décide de ne pas réécrire, nous modélisons la destruction dynamique de ce processus.

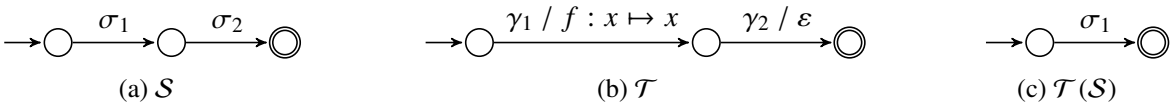
Nous donnons deux exemples de systèmes munis d'un transducteur effectuant ce mécanisme. Le premier, illustré par la Figure 3.5, présente un système $\mathcal{S}$ avec une configuration contenant un seul processus local dont l'état est σ_1. Sa fonction de transition, encodée par le transducteur $\mathcal{T}$, dont la sémantique est de reconnaître un processus satisfaisant la garde γ_1 et de réécriture deux nouveaux pro-



(a) $\mathcal{S}$ (b) $\mathcal{T}$ (c) $\mathcal{T}(\mathcal{S})$

avec $\sigma_1 \cap \gamma_1 \neq \emptyset$

FIGURE 3.5 – Création dynamique de procesus



(a) $\mathcal{S}$ (b) $\mathcal{T}$ (c) $\mathcal{T}(\mathcal{S})$

avec $\sigma_1 \cap \gamma_1 \neq \emptyset \wedge \sigma_2 \cap \gamma_2 \neq \emptyset$

FIGURE 3.6 – Destruction dynamique de procesus

cessus : σ'_1 et σ_2. En appliquant ce transducteur à la configuration initiale, nous obtenons $\mathcal{T}(\mathcal{S})$ qui présente maintenant un nouvel état supplémentaire. Nous avons ainsi créé un nouveau processus dans notre configuration. Dans le deuxième exemple, Figure 3.6, nous partons d'une configuration $\mathcal{S}$ à deux processus avec un transducteur $\mathcal{T}$ reconnaissant deux processus mais ne réécrivant que le premier (par l'identité). Le résultat de l'application $\mathcal{T}(\mathcal{S})$ est ainsi une configuration à un seul état.

3.3.4 Règles sémantiques

Nous exprimons maintenant la sémantique formelle exprimée sous la forme de règles d'inférence et de transducteurs. Nous mettons ainsi en évidence l'équivalence entre les deux écritures. Nous supposons que les états correspondants reconnus sont les mêmes pour chaque couple de règles de sémantique.

De cette manière, la sémantique formelle du système pourra être définie :

— par l'application successive des règles d'inférence, qui est une manière classique de présenter une sémantique opérationnelle ;

— ou, par l'application successive d'un transducteur $\mathcal{T}$.

Ce transducteur $\mathcal{T}$ est défini comme l'union des transducteurs que nous allons présenter pour chaque instruction du programme.

Affectation

L'instruction $x := e$ située à la localité l, et exécutée par le processus $k + 1$, modifiera l'état du système selon la règle :

$$\frac{\sigma_1 \dots \sigma_k \cdot \langle l, \rho\rangle \cdot \sigma_{k+2} \dots \sigma_n \qquad l \xrightarrow{x:=e} l' \qquad \rho' = \rho[x \mapsto \text{eval}(e, \rho)]}{\sigma_1 \dots \sigma_k \cdot \langle l', \rho'\rangle \cdot \sigma_{k+2} \dots \sigma_n} \text{ (aff.)}$$

On peut aussi définir la sémantique de cette instruction à l'aide du transducteur suivant :

$$\mathcal{T}_{l \xrightarrow{x:=e} l'} = \quad \top \,/\, \text{Id} \qquad \langle l, \top\rangle \,/\, f : \langle l, \rho\rangle \mapsto \langle l', \rho[x \mapsto \text{eval}(e, \rho)]\rangle \qquad \top \,/\, \text{Id}$$

Autrement dit, la valeur de l'expression e est évaluée en fonction de l'état mémoire ρ, qui est mis-à-jour localement. Les autres processus gardent leurs états inchangés. L'équivalence des deux sémantiques

est mise en évidence par le fait que si on a un état $\sigma_1 \dots \sigma_k \cdot \langle l, \rho\rangle \cdot \sigma_{k+2} \dots \sigma_n$ et que $\rho' = \rho[x \mapsto \text{eval}(e, \rho)]$, on a bien :

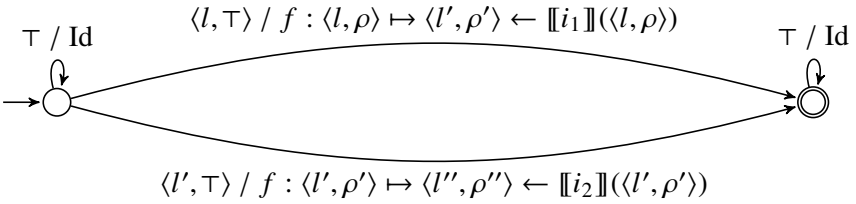
$$\mathcal{T}_{l \xrightarrow{x:=e} l'}(\sigma_1 \dots \sigma_k \cdot \langle l, \rho\rangle \cdot \sigma_{k+2} \dots \sigma_n) = \sigma_1 \dots \sigma_k \cdot \langle l', \rho'\rangle \cdot \sigma_{k+2} \dots \sigma_n$$

Séquence

La suite d'instructions $i_1; i_2$, i_1 à la localité l et i_2 à la localité localité l', et exécutée par le processus k, modifiera l'état du système comme il suit :

$$\frac{\sigma_1 \dots \sigma_k \dots \sigma_n \qquad l \xrightarrow{i_1} l' \xrightarrow{i_2} l'' \qquad \sigma'_k = [\![i_1]\!](\sigma_k) \qquad \sigma''_k = [\![i_2]\!](\sigma'_k)}{\sigma_1 \dots \sigma''_k \dots \sigma_n} \text{ (séq.)}$$

$\mathcal{T}_{l \xrightarrow{i_1} l' \xrightarrow{i_2} l''} =$



Pour le transducteur, la séquence se traduit par une simple union. Il faudra donc deux applications du transducteur pour avoir l'équivalence des deux sémantiques. Autrement dit, si on a un état $\sigma_1 \dots \cdot \langle l, \rho\rangle \cdot \dots \sigma_n$ et que $\langle l', \rho'\rangle = [\![i_1]\!](\langle l, \rho\rangle)$ et $\langle l'', \rho''\rangle = [\![i_2]\!](\langle l', \rho'\rangle)$, on a bien :

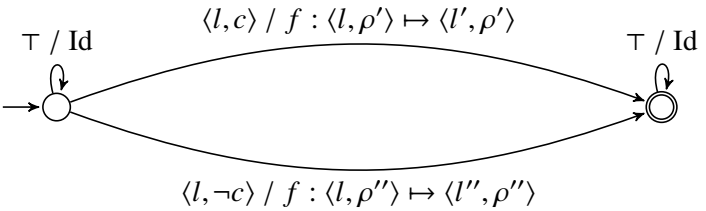
$$\begin{aligned} \mathcal{T}_{l \xrightarrow{i_1} l' \xrightarrow{i_2} l''}(\mathcal{T}_{l \xrightarrow{i_1} l' \xrightarrow{i_2} l''}(\sigma_1 \dots \langle l, \rho\rangle \dots \sigma_n)) &= \mathcal{T}_{l \xrightarrow{i_1} l' \xrightarrow{i_2} l''}(\sigma_1 \dots \langle l', \rho'\rangle \dots \sigma_n) \\ &= \sigma_1 \dots \langle l'', \rho''\rangle \dots \sigma_n \end{aligned}$$

Conditionnelle

$$\frac{\sigma_1 \dots \langle l, \rho\rangle \dots \sigma_n \qquad l \xrightarrow{\text{if } c} l' \qquad \dfrac{\text{assume}(c, \rho) \neq \bot}{\rho' = \text{assume}(c, \rho)}}{\sigma_1 \dots \langle l', \rho'\rangle \dots \sigma_n} \text{ (if}_\top\text{)}$$

$$\frac{\sigma_1 \dots \langle l, \rho\rangle \dots \sigma_n \qquad l \xrightarrow{\text{if } \neg c} l'' \qquad \dfrac{\text{assume}(\neg c, \rho) \neq \bot}{\rho'' = \text{assume}(\neg c, \rho)}}{\sigma_1 \dots \langle l'', \rho''\rangle \dots \sigma_n} \text{ (if}_\bot\text{)}$$

$\mathcal{T}_{l \xrightarrow{c} l' \mid l \xrightarrow{\neg c} l''} =$

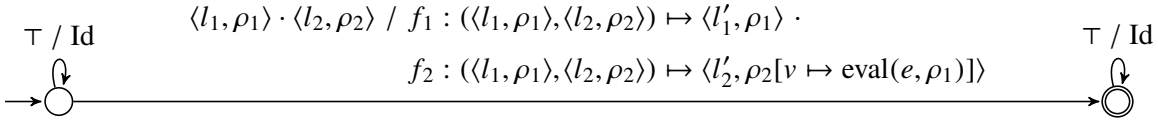


Un processus réalise une transition vers le conséquent d'une conditionnelle si la condition de la structure est vérifiée. Autrement, le processus évolue vers son alternant. De plus, l'environnement du processus est réduit selon la condition imposée. Ceci s'effectue grâce au test conditionnel *assume*. *e.g.* assume$(x > 10, \{x \in [-10; 20]\}) = \{x \in [11; 20]\}$).

Dans le transducteur, ce test peut être exprimé par la garde (partie gauche du transducteur). En effet, un état ne pourra être réécrit que si cette garde est vérifiée. Intrinsèquement, la garde réduira cet état en effectuant l'intersection entre celui-ci et la garde. *e.g.* Soit $\sigma = \langle L_{\text{if}}, \{x \in [-10; 20]\}\rangle$ et $g = \langle L_{\text{if}}, \{x \in [10; +\infty]\}\rangle$, il vient $g \cap \sigma = \{x \in [11, 20]\}$.

Communications

$$\frac{\sigma_1 \dots \sigma_k \cdot \langle l_1, \rho_1\rangle \cdot \langle l_2, \rho_2\rangle \cdot \sigma_{k+3} \dots \sigma_n \qquad l_1 \xrightarrow{R!\ e} l_1' \qquad l_2 \xrightarrow{L?\ v} l_2' \qquad \rho_2' = \rho_2[v \mapsto \text{eval}(e, \rho_1)]}{\sigma_1 \dots \sigma_k \cdot \langle l_1', \rho_1\rangle \cdot \langle l_2', \rho_2'\rangle \cdot \sigma_{k+3} \dots \sigma_n} \text{ (comm. } L \to R)$$



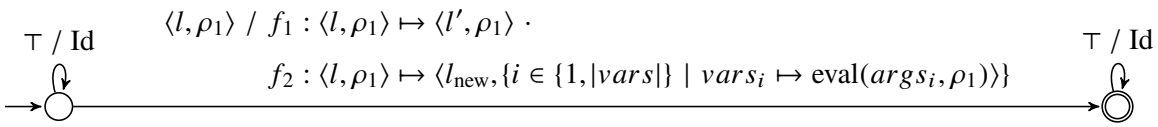
La communication synchrone ne peut s'effectuer que si deux processus adjacents sont respectivement en état d'envoi et de réception. Le cas échéant l'environnement du processus récepteur est mis à jour avec la valeur qui est le résultat d'une évaluation d'expression dans l'environnement de l'émetteur.

Grâce aux gardes multiples de notre structure de transducteur étendu, il est possible d'exprimer l'adjacence des processus au sein d'une même garde. Ce faisant, les deux fonctions de réécriture associées prennent en argument les deux différents états des processus connaissant ainsi le contexte de chacun d'entre-eux. Ainsi, il est possible de réécrire l'état du récepteur en prenant en compte celui de l'émetteur.

La règle de communication : L ← R est symétrique.

Création dynamique

$$\frac{\sigma_1 \dots \sigma_k \cdot \langle l, \rho_1\rangle \cdot \sigma_{k+2} \dots \sigma_n \quad l_{\text{new}} = \text{init <proc>}(vars) \quad l \xrightarrow{\text{create <proc>(args)}} l' \quad \dfrac{|vars| = |args|}{\rho_2 = \{i \in \{1, |vars|\} \mid vars_i \mapsto \rho_1(\text{eval}(args_i, \rho_1))\}}}{\sigma_1 \dots \sigma_k \cdot \langle l', \rho_1\rangle \cdot \langle l_{\text{new}}, \rho_2\rangle \cdot \sigma_{k+2} \dots \sigma_n} \text{ (create R)}$$



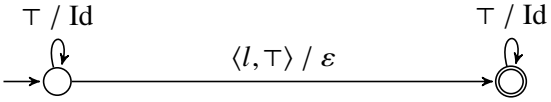
La sémantique de la création dynamique insère un nouvel état de processus adjacent au processus effectuant l'appel. Son état initial est composé de la localité l_{new} qui est la première localité du bloc des processus <proc> (supposée donnée par la fonction init). Son environnement est initialisé avec les variables définies à la déclaration du bloc <proc> avec pour valeurs les expressions évaluées dans le contexte de l'appelant.

Exprimer cette règle sémantique par un transducteur revient à réécrire plus de lettres que celles que l'on a lu. Dans ce cas précis, on reconnaît un état et on en réécrit deux ce qui modélise bien une création dynamique.

La création dynamique à « gauche » est symétrique.

Destruction dynamique

$$\frac{\sigma_1 \dots \sigma_k \cdot \langle l, \rho \rangle \cdot \sigma_{k+2} \dots \sigma_n \qquad l \xrightarrow{stop} \_}{\sigma_1 \dots \sigma_k \cdot \sigma_{k+2} \dots \sigma_n} \text{(stop)}$$



Symétriquement à la création, la destruction dynamique reconnaît un état qui ne sera pas réécrit.

3.4 Sémantique abstraite du système

Dans la section précédente, nous avons défini une modélisation des états de nos systèmes sous la forme de langages reconnaissables par des automates. La sémantique des systèmes peut être exprimée par des transitions de transducteur aux gardes étendues. Ainsi, la sémantique concrète est représentée par un transducteur étendu $\mathcal{T}$. Cependant, cette sémantique concrète n'est pas calculable. Nous devons donc déduire une sémantique abstraite. Se faisant, il sera possible de calculer un sur-ensemble des états atteignables d'un système. Cette sur-approximation devra cependant être suffisamment précise pour nous permettre la preuve des propriétés recherchées.

Dans cette section, nous donnons une sémantique abstraite sous forme d'une nouvelle définition de transducteur et montrons une correspondance de Galois avec la sémantique concrète précédemment définie.

3.4.1 Transducteurs de treillis

Un transducteur de treillis est un transducteur symbolique étendu où l'alphabet en entrée et sortie est un treillis atomique acceptant et réécrivant des lettres ou mots d'états abstraits de processus locaux modélisés par des automates de treillis.

Définition 14. *Un transducteur de treillis est un transducteur étendu où l'alphabet est un treillis atomique* Λ *:* $\mathcal{T}^\sharp = \langle \Lambda, Q, Q_0, Q_f, \Delta \rangle$.

Cette structure reprend la définition précédente des transducteurs étendus en considérant maintenant des états abstraits modélisés par des automates de treillis. Se faisant, nous sommes, d'une part, capables d'encoder une abstraction de la sémantique grâce à la structure de treillis que nous utilisons en tant

qu'alphabet. Et d'autre part, il est possible de considérer des sur-approximations d'ensembles d'états grâce à l'utilisation des automates de treillis auxquels les transducteurs de treillis s'appliquent.

Cette nouvelle définition va nous permettre d'encoder une abstraction de la sémantique de notre modèle.

Définition 15. *Un transducteur de treillis* $\mathcal{T}^\sharp = \langle \Lambda, Q, Q_0, Q_f, \Delta^\sharp \rangle$ *est une abstraction d'un transducteur étendu* $\mathcal{T} = \langle \Sigma, Q, Q_0, Q_f, \Delta \rangle$ *si :*

— $\Sigma \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \Lambda$;

— $\Delta^\sharp = \{(p, g^\sharp, f^\sharp, q) \mid (p, g, f, q) \in \Delta, g^\sharp = \alpha(g) \wedge f^\sharp = \alpha \circ f \circ \gamma\}$.

Lemme 1. *Soit* $\mathcal{T}$ *un transducteur étendu et son abstraction* $\mathcal{T}^\sharp$ *un transducteur de treillis, on a :*

$$\mathcal{T}(L) \subseteq \gamma(\mathcal{T}^\sharp(\alpha(L)))$$

Démonstration. Soit $w \in L$ et $w' \in \mathcal{T}(w)$, $\exists q_0 \dots q_n$, $q_0 \xrightarrow{g_1/f_1} q_1 \dots \xrightarrow{g_n/f_n} q_n$ avec $q_0 \in Q_0$, $q_f \in Q_f$ et $\forall i$, $w_i \sqsubseteq g_i \wedge w'_i = f_i(w_i)$ avec $w = w_1 \dots w_n$ et $w' = w'_1 \dots w'_n$.

Par monotonie de α, on a que $\alpha(w_i) \sqsubseteq g_i^\sharp$ et par définition de $T^\sharp$, $\exists q_0 \dots q_n$, $q_0 \xrightarrow{g_1^\sharp/f_1^\sharp} q_1 \dots \xrightarrow{g_n^\sharp/f_n^\sharp} q_n$ et $f_1^\sharp(\alpha(w_1)) \dots f_n^\sharp(\alpha(w_n)) \in T^\sharp(\alpha(w))$ avec $\alpha(w) = \alpha(w_1) \dots \alpha(w_n)$.

Ensuite, $\forall i, \alpha \circ f_i \circ \gamma(\alpha(w_i)) = f_i^\sharp(\alpha(w_i))$. De plus, on sait que par la correspondance de Galois, on a : $w_i \sqsubseteq \gamma \circ \alpha(w_i)$ et par monotonie de α et de f_i, on a : $\alpha \circ f_i(w_i) \sqsubseteq \alpha \circ f_i \circ \gamma(w_i)$. Donc $\alpha(f_1(w_1) \dots f_n(w_n)) \sqsubseteq f_1^\sharp(\alpha(w_1)) \dots f_n^\sharp(\alpha(w_n))$ et par monotonie de $\gamma \circ \alpha(f_1(w_1) \dots f_n(w_n)) \sqsubseteq \gamma(f_1^\sharp(\alpha(w_1)) \dots f_n^\sharp(\alpha(w_n)))$. On sait que $w' = f_1(w_1) \dots f_n(w_n)$ et donc $w' \sqsubseteq \gamma \circ \alpha(w')$. Donc, $\forall w' \in \mathcal{T}(L), w' \in \gamma(\mathcal{T}^\sharp(\alpha(L)))$. □

Lemme 2. *Soit* $\mathcal{T}$ *un transducteur étendu et* $\mathcal{T}^\sharp$ *son abstraction sous forme de transducteur de treillis, on a :*

$$\forall i,\ \mathcal{T}^i(L) \subseteq \gamma(\mathcal{T}^{\sharp^i}(\alpha(L))$$

Démonstration. On raisonne par récurrence :

— pour $i = 0$, $\mathcal{T}^0(L) = L \subseteq \gamma(\alpha(L))$;

— on suppose $\mathcal{T}^i(L) \subseteq \gamma(\mathcal{T}^{\sharp^i}(\alpha(L))$:

$$\begin{aligned}
\mathcal{T}^{i+1}(L) &= \mathcal{T}(\mathcal{T}^i(L)) \\
&\subseteq \gamma(\mathcal{T}^\sharp(\alpha(\mathcal{T}^i(L)))) && \text{(lemme)} \\
&\subseteq \gamma(\mathcal{T}^\sharp(\alpha(\gamma(\mathcal{T}^{\sharp^i}(\alpha(L)))))) && \text{(hypothèse de récurrence)} \\
&\subseteq \gamma(\mathcal{T}^\sharp(\mathcal{T}^{\sharp^i}(\alpha(L)))) && (\alpha \circ \gamma \subseteq Id) \\
&\subseteq \gamma(\mathcal{T}^{\sharp^{i+1}}(\alpha(L)))
\end{aligned}$$

□

Nous disposons maintenant d'une sémantique abstraite qui assure, par construction, que la sémantique concrète sera sur-approximée. En utilisant des configurations modélisées par des automates de treillis ainsi que ses opérateurs, en particulier l'opérateur d'élargissement, nous sommes capables de construire une analyse statique correcte et ainsi de vérifier notre modèle.

3.5 Vérification du système

Nous nous intéressons maintenant à l'analyse de cette modélisation. Dans la méthodologie du *Regular Model Checking*, l'objectif d'une l'analyse est de tester l'intersection entre les propriétés de sûreté, exprimées en tant que mauvaises configurations $\mathcal{B}$, et l'*espace d'atteignabilité* du programme : $\mathcal{T}^*(I)$ où I représente une configuration initiale d'un programme et $\mathcal{T}^*$ est la fermeture transitive et réflexive du transducteur $\mathcal{T}$. Comme évoqué dans le chapitre précédent (*c.f.* 2.5), la difficulté réside dans le calcul de $\mathcal{T}^*(I)$ qui, dans le cas général, n'est pas calculable. En *Regular Model Checking*, les techniques utilisées pour résoudre ce problème consistent à définir, entre autres, des semi-algorithmes afin de traiter le plus grand nombre de cas [AJNS04].

Une manière équivalente de définir ce $\mathcal{T}^*(I)$ est de calculer l'union de toutes les applications possibles du transducteur sur la configuration de départ I :

$$\mathcal{T}^*(I) = \bigcup_{i \in \mathbb{N}} \mathcal{T}^i(I)$$

Notre méthode diverge du *Regular Model Checking* au sens où, comme en interprétation abstraite, nous allons chercher à calculer une *sur-approximation* calculable $\mathcal{T}^\alpha(I)$ des états atteignables du programme : $\mathcal{T}^*(I) \subseteq \mathcal{T}^\alpha(I)$. Ainsi, $\mathcal{T}^\alpha(I)$ contient au moins tous les états atteignables possibles du programme. Certains états impossibles à atteindre seront également présents dans la sur-approximation que nous avons calculée. Ces derniers sont les résultats des diverses techniques d'accélération utilisées afin d'assurer la terminaison du calcul.

Se faisant, nous sommes capables de prouver des propriétés de sûreté en testant l'intersection : $\mathcal{T}^\alpha(I) \cap \mathcal{B} \stackrel{?}{=} \emptyset$. Si cette intersection est vide alors notre propriété est prouvée ; autrement, si cette intersection n'est pas vide alors soit la propriété est fausse ou soit la sur-approximation calculée est trop grossière et doit être raffinée afin d'éviter cette fausse alarme.

Afin de procéder à ce calcul, il est nécessaire d'établir dans un premier temps l'application d'un transducteur de treillis à un automate de treillis représentant un ensemble d'états. Ensuite, nous nous intéresserons à l'itération de ce calcul. Nous appliquerons ensuite cette méthodologie à un exemple, et, finalement nous discuterons de l'automatisation de cette analyse par la traduction d'un programme écrit dans notre langage vers un transducteur de treillis.

3.5.1 Algorithme d'application

Nous définissons l'opération (Algorithme 1) qui permet l'application d'un transducteur de treillis à un automate de treillis. Cet algorithme itère sur les transitions du transducteur, récupérant les gardes de longueur n, puis cherche dans l'automate les séquences (de longueur n) telles que toute intersection entre gardes et lettres correspondantes ne soit pas vide. À partir des transitions éligibles, nous introduisons de nouvelles séquences où chaque transition correspond à l'application d'une des fonctions de la transition du transducteur sur l'ensemble des intersections de $\gamma_i \sqcap \sigma_i$ où γ est une garde et σ, un état. *N.B.* Comme de nouvelles transitions peuvent être créées (*i.e.* $m > n$), il est nécessaire d'introduire de nouveaux états intermédiaires entre p_0 et p_n. Une fois ces transitions calculées, nous déterminons les états initiaux (resp. finaux) du nouvel automate en regardant si les deux états de départ (resp. d'arrivée) des nouvelles transitions appartiennent aux états initiaux (resp. finaux) du transducteur et de l'automate. Finalement, nous filtrons les transitions inatteignables par la fonction `trim` qui consiste en l'intersection des états

atteignables et co-atteignables [1] de l'automate avant de déterminiser l'automate résultat. En effet, dans la définition de notre transducteur, il est possible de créer des transitions de la forme $\gamma_1 \cdot \ldots \cdot \gamma_n \ / \ \varepsilon$. En appliquant notre algorithme, ce type de transition peut alors générer des ε-transitions. Les automates de treillis étant nécessairement déterministes, nous devons les éliminer. Pour cela, nous utilisons, classiquement [HMU06], un algorithme de fermeture transitive pour construire un équivalent déterministe.

Algorithme 1 : `Application`$(\mathcal{T},\mathcal{A})$

pour tout $q \xrightarrow{\gamma_1\cdot\ldots\cdot\gamma_n \ / \ f_1\cdot\ldots\cdot f_m} q' \in \Delta_{\mathcal{T}}$ **faire**
  pour tout $p_0 \xrightarrow{\sigma_1} p_1 \cdots \xrightarrow{\sigma_n} p_n \in \Delta_{\mathcal{A}}$ **faire**
    si $\{ \forall i \in \{1 \ldots n\} \mid \sigma_i \sqcap \gamma_i \neq \bot\}$ **alors**
      $Q_{\mathcal{T}'} \leftarrow Q_{\mathcal{T}'} \cup \{(p_0,q),(p_1,q),\ldots,(p_n,q')\}$
      args $\leftarrow \{\sigma_1 \sqcap \gamma_1,\ldots,\sigma_n \sqcap \gamma_n\}$
      $\Delta_{\mathcal{T}'} \leftarrow \Delta_{\mathcal{T}'} \cup \{(p_0,q) \xrightarrow{f_1(args)} (p_1,q) \cdots \xrightarrow{f_m(args)} (p_n,q')\}$
    fin si
  fin
fin
pour tout $(p,q) \in Q^{\mathcal{T}'}$ **faire**
  si $p \in Q_i^{\mathcal{A}} \wedge p \in Q_i^{\mathcal{T}}$ **alors**
    $Q_i^{\mathcal{T}'} \leftarrow Q_i^{\mathcal{T}'} \cup \{(p,q)\}$
  fin si
  si $p \in Q_f^{\mathcal{A}} \wedge p \in Q_f^{\mathcal{T}}$ **alors**
    $Q_f^{\mathcal{T}'} \leftarrow Q_f^{\mathcal{T}'} \cup \{(p,q)\}$
  fin si
fin
$\mathcal{T}' \leftarrow \langle \Sigma^A, Q^{\mathcal{T}'}, Q_i^{\mathcal{T}'}, Q_f^{\mathcal{T}'}, \Delta_{\mathcal{T}'}\rangle$
retourner ε-`removal(trim(`$\mathcal{T}'$`))`

Nous prouvons désormais la correction de cet algorithme. Ceci revient à montrer que le résultat de l'application du transducteur d'un mot du langage de l'automate est contenu dans le langage de l'application du transducteur à l'automate.

Lemme 3. *Soit $T = \langle \Lambda, Q^T, Q_0^T, Q_f^T, \Delta^T\rangle$, un transducteur de treillis et $A = \langle \Lambda, Q^A, Q_0^A, Q_f^A, \Delta^A\rangle$, un automate de treillis, nous avons :*

$$T(\mathcal{L}(A)) \subseteq \mathcal{L}(T(A))$$

Démonstration. Par définition, $w' \in T(w)$ avec $w \in \mathcal{L}(A)$ si $p_0 \xrightarrow{w_1/w'_1}\!\!\!\!\rightarrow\ p_1 \xrightarrow{w_2/w'_2}\!\!\!\!\rightarrow\ \ldots \xrightarrow{w_n/w'_n}\!\!\!\!\rightarrow\ p_n$ [2] avec $p_0 \in Q_0^T, p_n \in Q_f^T$, $w = w_1.w_2 \ldots w_n$, $w' = w'_1.w'_2 \ldots w'_n$ et $\forall i = 1..n, \exists (p_{i-1}, G_i, F_i, p_i) \in \Delta^T$ avec $w_i \sqsubseteq G_i$ et $w'_i = F(w_i)$.

Puisque $w \in \mathcal{L}(A)$, il vient $\exists q_0 ... q_n \in Q^A$ avec $q_0 \in Q_0^A, q_n \in Q_f^A$ tel que, $q_0 \xrightarrow{w_1}\!\!\!\!\rightarrow\ q_1 \xrightarrow{w_2}\!\!\!\!\rightarrow\ \ldots \xrightarrow{w_n}\!\!\!\!\rightarrow\ q_n$.

En d'autres mots, $\forall i = 1 \ldots n$ il y a une séquence de transitions telle que : $q_{i-1} \xrightarrow{\lambda_{(i,1)}} \ldots \xrightarrow{\lambda_{(i,m_i)}} q_i$ avec $w_i \sqsubseteq \lambda(i,1) \ldots \lambda_{(i,m_i)}$.

1. Ensemble des états atteignables dans l'automate depuis les états initiaux (resp. finaux)
2. $p \xrightarrow{w/w'}\!\!\!\!\rightarrow\ p'$ est une suite de transitions allant de l'état p à p' et reconnaissant le mot w qui sera réécrit en w'.

Ainsi, $w_i \subseteq G_i \sqcap \lambda(i,1) \dots \lambda_{(i,m_i)}$ et $F(w_i) = w'_i \sqsubseteq F(G_i \sqcap \lambda(i,1) \dots \lambda_{(i,m_i)})$.

Par définition de $T(A)$,$(p_0, q_0) \xrightarrow{w'_1}\!\!\!\!\rightarrow \dots \xrightarrow{w'_n}\!\!\!\!\rightarrow (p_n, q_n)$ est un chemin accepté de $T(A)$, donc $w' \in \mathcal{L}(T(A))$.

□

Complexité

La complexité de notre algorithme d'application de transducteur dépend de la taille de la partition π de l'automate de treillis utilisée et de la longueur maximum N des gardes utilisées dans les transitions du transducteur. Ainsi, la complexité est :

$$O(|\Delta_T| \cdot |\pi|^N \cdot N + |Q_A| \cdot |Q_T|)$$

Démonstration. Dans l'algorithme 1, il y a deux imbrications de boucles « pour tout » :

1. Première imbrication : le premier niveau sera exécuté au plus $|\Delta_T|$ fois ; pour le second il y a tout au plus $|\pi|^n$ séquences à considérer dans l'automate A, car celui-ci est partitionné, ce qui implique qu'il y a au plus $|\pi|$ transitions sortant de chaque état, et comme N est défini comme la longueur maximale des séquences apparaissant dans les gardes, on a bien toujours $n \geq N$ quelle que soit la séquence, d'où une majoration par $|\pi|^N$. Enfin, dans la conditionnelle, on effectue tout au plus $O(n)$ calculs, ce qui donne, pour cette première implication, une complexité $O(|\Delta_T| \cdot |\pi|^N \cdot N)$
2. Seconde imbrication : il y a au plus $|Q_{T(A)}| \leq |Q_A| \cdot |Q_T|$ paires à considérer

La complexité au pire cas est donc bien de l'ordre de $O(|\Delta_T| \cdot |\pi|^N \cdot N + |Q_A| \cdot |Q_T|)$. □

En pratique, le cardinal des partitions de l'automate est en relation avec la taille du code cependant, dans notre sémantique, nous n'avons pas de séquences de gardes de longueur supérieure à 2. En outre, le nombre d'états du transducteur est, par construction, toujours égal à 1.

3.5.2 Calcul de l'espace d'atteignabilité

Comme énoncé plus haut, nous cherchons à calculer une sur-approximation de l'ensemble des états atteignables. Dans la littérature, cette sur-approximation du $\mathcal{T}^*(I)$ a été étudiée dans [BHV04] où les auteurs calculent des abstractions entre deux ensembles d'états successifs jusqu'à obtenir un point fixe. Notre méthode est similaire à ceci près que nous manipulons des automates de treillis et pouvons utiliser des opérations d'élargissement $\triangledown$ en tant que fonction d'abstraction. Notre calcul de point fixe prend la forme d'une itération de Kleene présentée par la Figure 3.7. Cet algorithme termine car la suite calculée sera stationnaire au delà d'un certain rang, du fait des propriétés de l'opérateur d'élargissement détaillé plus bas.

En théorie, ce calcul itératif peut donner un résultat très imprécis. Cependant, notre analyse utilise un treillis contenant les localités de chaque processus. Ainsi, en pratique, nous pouvons effectuer une première amélioration sur la précision de notre analyse en appliquant l'opérateur $\triangledown$ seulement à certaines localités (*e.g.* au point d'entrée des boucles) sachant que l'opérateur d'élargissement n'est nécessaire que pour casser les cycles de dépendance. Dans [Bou93], l'auteur donne une étude détaillée sur les choix des localités où appliquer cet opérateur.

$$\mathcal{A}_0 = I$$

$$\mathcal{A}_{n+1} = \begin{cases} \mathcal{A}_n & \text{si } \mathcal{T}(\mathcal{A}_n) \sqsubseteq \mathcal{A}_n \\ \mathcal{A}_n \triangledown (\mathcal{A}_n \sqcup \mathcal{T}(\mathcal{A}_n)) & \text{sinon} \end{cases}$$

$$\mathcal{T}^{\alpha}(I) = \lim_{n \to +\infty} \mathcal{A}_n$$

FIGURE 3.7 – Calcul de $\mathcal{T}^{\alpha}(I)$

Comme décrit dans [LJ07], l'opérateur d'élargissement appliqué à deux automates de treillis consiste d'abord à calculer les *formes* des deux automates (*shape automata*) et à tester leur isomorphisme. En effet, la forme d'un automate de treillis partitionné $\mathcal{A}$ est obtenue en remplaçant toute transition (q, λ, q') de $\mathcal{A}$ par une transition $(q, \sigma^{-1}(\lambda), q')$, autrement dit on remplace tout élément du treillis par le représentant de sa classe d'équivalence. Comme il n'y a qu'un nombre fini de classes d'équivalence, shape($\mathcal{A}$) est bien un automate fini.

Nous rappelons la définition de l'application de l'opérateur d'élargissement appliqué à deux automates de treillis. Soient deux automates de treillis $\mathcal{A}_1$ et $\mathcal{A}_2$, que l'on suppose normalisés (*i.e.* déterministes minimaux).

— Si shape($\mathcal{A}_1$) et shape($\mathcal{A}_2$), qui sont eux aussi déterministes et minimaux, reconnaissent le même langage, alors ils sont isomorphes ; appelons ι : shape($\mathcal{A}_1$) $\to$ shape($\mathcal{A}_2$) cet isomorphisme, l'automate de treillis $\mathcal{A}_1 \nabla \mathcal{A}_2$ est obtenu ainsi : c'est un automate qui a le même ensemble d'états que $\mathcal{A}_1$ et, pour toute transition (q_1, λ_1, q_1') de $\mathcal{A}_1$, il existe une transition $(\iota(q_1), \lambda_2, \iota(q_1'))$ dans $\mathcal{A}_2$ et donc on ajoute une transition $(q_1, \lambda_1 \nabla_\Lambda \lambda_2, q_1')$ dans $\mathcal{A}_1 \nabla \mathcal{A}_2$.

— Sinon, on considère l'automate $\mathcal{A}_1 \cup \mathcal{A}_2$. Soit $\equiv$ une relation d'équivalence, dont le nombre de classes d'équivalences est borné par un entier N (par exemple : la bisimulation bornée à l'ordre k [LJ07]). Alors $\mathcal{A}_1 \nabla \mathcal{A}_2$ est défini comme le quotient $(\mathcal{A}_1 \cup \mathcal{A}_2) \setminus \equiv$.

[LJ07] a prouvé qu'il s'agit bien d'un opérateur d'élargissement, qui est paramétrable par le choix de l'élargissement sur le treillis numérique ∇_Λ et le choix de la relation d'équivalence $\equiv$.

3.5.3 Exemple : *Moonrobot*

Nous donnons maintenant un exemple de programme écrit dans notre langage et une vérification détaillée de propriétés de sûreté numérique. Notre exemple (Figure 3.8) modélise une *base lunaire* munie d'un robot destiné à l'exploration et à la récolte d'informations. Ce robot doit communiquer ses résultats à la base lunaire au fur et à mesure qu'il les récolte. En retour, la base s'acquitte de cette bonne réception. Cependant, la distance parcourue peut être grande et le lien entre la base et le robot peut potentiellement être rompue. Pour résoudre ce problème, le robot peut déposer des antennes-relais qui serviront de répéteurs aux messages. En créant une antenne, le robot y transfère un peu de son énergie pour lui donner une autonomie.

Notre programme possède trois types de processus :

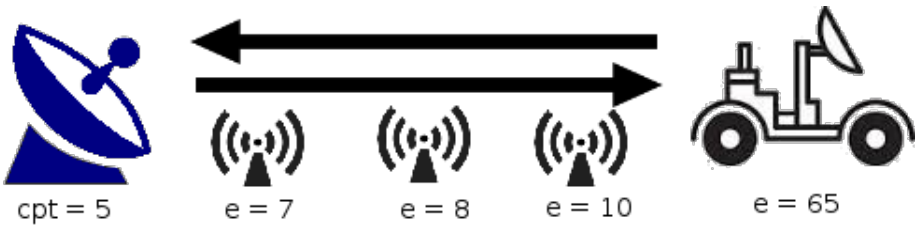
— La base, créée initialement et qui commence par déployer le robot-sonde. Ensuite, la base se met en attente de potentiels messages et, le cas échéant, effectue l'acquittement avant de se remettre en attente ;

```
1  main(): // base
2    createR robot(100);
3    while true do
4      R?(msg);
5      cpt := cpt + 1;
6      R!(); // envoi de l'acquittement
7      ... // traitement du message
8    done
9
10 robot(energie):
11   while energie > 0 then
12     action = random() % 3;
13     if action = 0 then
14       createL antenne(10) // crée une antenne relai
15       energie = energie - 10
16     else if action = 1 then
17       msg = scan();
18       L!(msg); // envoie les résultats à la base
19       L?(); // attente de l'acquittement de la base
20       energie = energie - 1
21     else
22       // attente
23     endif
24   done
25
26 antenne(energie):
27   // relaie les résultats à la base
28   R?(msg);
29   L!(msg);
30   // relaie l'acquittement depuis la base
31   L?();
32   R!();
33   energie := energie - 1
```

FIGURE 3.8 – Programme *Moonrobot*

— le robot, capable d'effectuer deux types d'opérations : la création d'une antenne-relai et le prélèvement d'informations avec envoi de message vers la base. La création d'une antenne lui prélève 10 unités d'énergie et l'envoi d'un message, 1 unitée d'énergie. Les messages envoyés passent nécessairement par les antennes-relais de façon antéchronologique ;

— les antennes quant à elles, attendent un message à relayer de la part du robot puis transmettent celui-ci soit à l'antenne précédente, soit à la base si elle est la plus ancienne. Puis, attendent le message d'acquittement de la base vers le robot.



La première étape de notre analyse est de traduire la sémantique du programme en transducteur de treillis. Dans ce type de programme, le chemin des transitions empruntées importent peu et, ainsi, il est possible de modéliser le transducteur par un « transducteur tournesol « ; c'est-à-dire, un transducteur à un seul nœud (initial et final) composé uniquement de transitions cycliques. La méthode pour géné-

rer automatiquement un tel transducteur est de parcourir les transitions du graphe de flot de contrôle du programme et d'associer pour chaque opération sémantique la fonction de transfert correspondante modélisée en forme de règle gardes/fonctions de réécriture annotée à chaque transition. Pour faciliter la lecture, chaque localité de notre programme correspond à la ligne d'une instruction (L_1, L_2, ...). Voici les gardes et fonctions de réécriture pour cet exemple :

Base :

— $L_2 \times \top \;/\; f_1 : (l,\rho) \mapsto L_3 \times \rho \;\cdot\; f_2 : (l,\rho) \mapsto L_{11} \times \{\texttt{energie} \leftarrow 100\}$

— $L_3 \times \top \;/\; f : (l,\rho) \mapsto L_4 \times \rho$

— ~~$L_3 \times \bot \;/\; f : (l,\rho) \mapsto L_9 \times \rho$~~ *N.B.* Chaque instruction de communication avec un voisin doit être appareillée à une instruction symétrique (**L**! avec **R**?, **L**? avec **R**!, etc.). Ces transitions sont générées au moment de traduire les envois (**R**! et **L**!). C'est pour cela qu'ici, nous ne traduisons pas encore l'instruction **R**!(`msg`).

— $L_5 \times \top \;/\; f : (l,\rho) \mapsto L_6 \times \rho[\texttt{cpt} \mapsto \rho(\texttt{cpt}) + 1]$

— $L_6 \times \top \;\cdot\; L_{22} \times \top \;/\; f_1 : ((l_{\texttt{base}},\rho_{\texttt{base}}),(l_{\texttt{robot}},\rho_{\texttt{robot}})) \mapsto L_3 \times \rho_{\texttt{base}} \;\cdot$
$f_2 : ((l_{\texttt{base}},\rho_{\texttt{base}}),(l_{\texttt{robot}},\rho_{\texttt{robot}})) \mapsto L_{23} \times \rho_{\texttt{robot}}$

— $L_6 \times \top \;\cdot\; L_{34} \times \top \;/\; f_1 : ((l_{\texttt{base}},\rho_{\texttt{base}}),(l_{\texttt{ant}},\rho_{\texttt{ant}})) \mapsto L_3 \times \rho_{\texttt{base}} \;\cdot$
$f_2 : ((l_{\texttt{base}},\rho_{\texttt{base}}),(l_{\texttt{ant}},\rho_{\texttt{ant}})) \mapsto L_{35} \times \rho_{\texttt{ant}}$

Robot :

— $L_{11} \times \{\texttt{energie} \in [1;+\infty]\} \;/\; f : (l,\rho) \mapsto L_{12} \times \rho$

— $L_{11} \times \{\texttt{energie} \in [-\infty;0]\} \;/\; f : (l,\rho) \mapsto L_{28} \times \rho$

— $L_{12} \times \top \;/\; f : (l,\rho) \mapsto L_{13} \times \rho[action \mapsto \top]$ *N.B.* Dans cet exemple, le `random`() est simplement utilisé afin de modéliser le choix non-déterministe. Nous pourrions utiliser une domaine abstrait de congruences pour conserver la propriété numérique mais cela ne nous sert pas pour le reste du programme. Ainsi, l'évaluation du % donne $\top$.

— $L_{13} \times \{\texttt{action} \in [0;0]]\} \;/\; f : (l,\rho) \mapsto L_{15} \times \rho$

— $L_{13} \times \{\texttt{action} \in [-\infty;-1] \;\vee\; [1;+\infty]]\} \;/\; f : (l,\rho) \mapsto L_{17} \times \rho$

— $L_{15} \times \top \;/\; f_1 : (l,\rho) \mapsto L_{30} \times \{\texttt{energie} \mapsto [10;10]\} \;\cdot\; f_2 : (l,\rho) \mapsto L_{16} \times \rho$

— $L_{16} \times \top \;/\; f : (l,\rho) \mapsto L_{11} \times \rho[\texttt{energie} \mapsto \rho(\texttt{energie}) - 10]$

— $L_{17} \times \{\texttt{action} \in [1;1]]\} \;/\; f : (l,\rho) \mapsto L_{18} \times \rho$

— $L_{17} \times \{\texttt{action} \in [-\infty;0] \;\vee\; [2;+\infty]]\} \;/\; f : (l,\rho) \mapsto L_{25} \times \rho$

— $L_{18} \times \top \;/\; f : (l,\rho) \mapsto L_{20} \times \rho[\texttt{msg} \mapsto \top]$

— $L_4 \times \top \;\cdot\; L_{20} \times \top \;/\; f_1 : ((l_{\texttt{base}},\rho_{\texttt{base}}),(l_{\texttt{robot}},\rho_{\texttt{robot}})) \mapsto L_5 \times \rho_{\texttt{base}}[msg \mapsto \rho_{\texttt{robot}}(msg)] \;\cdot$
$f_2 : ((l_{\texttt{base}},\rho_{\texttt{base}}),(l_{\texttt{robot}},\rho_{\texttt{robot}})) \mapsto L_{22} \times \rho_{\texttt{robot}}$

— $L_{31} \times \top \;\cdot\; L_{20} \times \top \;/\; f_1 : ((l_{\texttt{ant}},\rho_{\texttt{ant}}),(l_{\texttt{robot}},\rho_{\texttt{robot}})) \mapsto L_{32} \times \rho_{\texttt{ant}}[msg \mapsto \rho_{\texttt{robot}}(msg)] \;\cdot$
$f_2 : ((l_{\texttt{ant}},\rho_{\texttt{ant}}),(l_{\texttt{robot}},\rho_{\texttt{robot}})) \mapsto L_{22} \times \rho_{\texttt{robot}}$

— $L_{23} \times \top \;/\; f : (l,\rho) \mapsto L_{11} \times \rho[\texttt{energie} \mapsto \rho(\texttt{energie}) - 1]$

— $L_{25} \times \top \;/\; f : (l,\rho) \mapsto L_{11} \times \rho$

Antennes :

— $L_4 \times \top \;\cdot\; L_{32} \times \top \;/\; f_1 : ((l_{\texttt{base}},\rho_{\texttt{base}}),(l_{\texttt{ant}},\rho_{\texttt{ant}})) \mapsto L_5 \times \rho_{\texttt{base}}[msg \mapsto \rho_{\texttt{ant}}(msg)] \;\cdot$
$f_2 : ((l_{\texttt{base}},\rho_{\texttt{base}}),(l_{\texttt{ant}},\rho_{\texttt{ant}})) \mapsto L_{34} \times \rho_{\texttt{ant}}$

— $L_{31} \times \top \;\cdot\; L_{32} \times \top \;/\; f_1 : ((l_{\texttt{ant}_1},\rho_{\texttt{ant}_1}),(l_{\texttt{ant}_2},\rho_{\texttt{ant}_2})) \mapsto L_5 \times \rho_{\texttt{ant}_1}[msg \mapsto \rho_{\texttt{ant}_2}(msg)] \;\cdot$
$f_2 : ((l_{\texttt{ant}_1},\rho_{\texttt{ant}_1}),(l_{\texttt{ant}_2},\rho_{\texttt{ant}_2})) \mapsto L_{34} \times \rho_{\texttt{ant}_2}$

— $\mathrm{L}_{35} \times \top \cdot \mathrm{L}_{22} \times \top \,/\, f_1 : ((l_{\mathtt{ant}}, \rho_{\mathtt{ant}}), (l_{\mathtt{robot}}, \rho_{\mathtt{robot}})) \mapsto \mathrm{L}_{36} \times \rho_{\mathtt{ant}} \cdot$
$f_2 : ((l_{\mathtt{ant}}, \rho_{\mathtt{ant}}), (l_{\mathtt{robot}}, \rho_{\mathtt{robot}})) \mapsto \mathrm{L}_{23} \times \rho_{\mathtt{robot}}$

— $\mathrm{L}_{35} \times \top \cdot \mathrm{L}_{34} \times \top \,/\, f_1 : ((l_{\mathtt{ant}_1}, \rho_{\mathtt{ant}_1}), (l_{\mathtt{ant}_2}, \rho_{\mathtt{ant}_2})) \mapsto \mathrm{L}_{36} \times \rho_{\mathtt{ant}_1} \cdot$
$f_2 : ((l_{\mathtt{ant}_1}, \rho_{\mathtt{ant}_1}), (l_{\mathtt{ant}_2}, \rho_{\mathtt{ant}_2})) \mapsto \mathrm{L}_{35} \times \rho_{\mathtt{ant}_2}$

Identité :

— $\top \,/\, f : x \mapsto x$

N.B. Afin de modéliser le non-déterminisme, nous ajoutons une règle « identité » qui va permettre à chaque état local de processus de ne pas évoluer.

Nous donnons maintenant l'état initial du programme, sous la forme d'un automate de treillis, qui correspond à un processus `main` à la première localité (L_2) avec un environnement initialisé à $\top$. Nous donnons également les premières itérations de l'algorithme de calcul de l'espace d'atteignabilité.

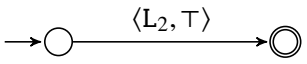


FIGURE 3.9 – Configuration initiale : $\mathcal{I}$

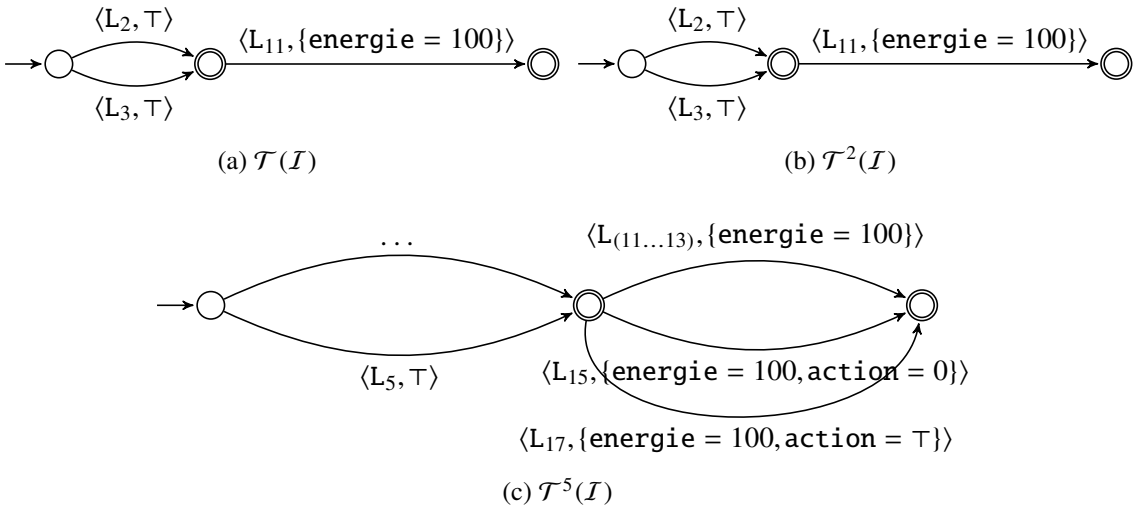


FIGURE 3.10 – Premières itérations de $\mathcal{I} \circ \mathcal{T}^{\alpha}$

Il est à noter que la présence de la transition **Identité** entraîne par nature une union entre les différentes applications.

Vérification de propriétés

Nous souhaitons prouver la propriété numérique suivante : au maximum 10 antennes ont été déposées par le robot. Pour se faire, nous devons tester l'intersection entre l'espace d'atteignabilité obtenu et la négation de la propriété pouvant être modélisée par un automate (Figure 3.11) établissant une *mauvaise configuration* contenant, par exemple, 11 processus **antenne**.

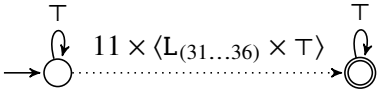


FIGURE 3.11 – Mauvaise configuration C_1 : 11 antennes déposées

Nous souhaitons également tester que la base ne peut recevoir plus de 100 messages. Ici encore, nous définissons une mauvaise configuration qui contredit la propriété (Figure 3.12).

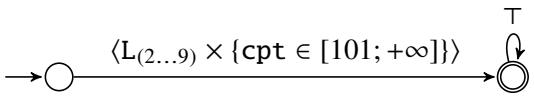


FIGURE 3.12 – Mauvaise configuration C_2 : plus de 100 messages reçus

Pour cet exemple, le domaine des automates de treillis est paramétré avec le domaine numérique des intervalles. L'analyse a terminé en 17 itérations en 1,32s. Notre expérimentation a montré que les deux propriétés sont vraies en testant que l'intersection entre les mauvaises configurations et l'espace d'états atteignables est vide.

3.6 Conclusion

Nous avons présenté dans cette section un modèle de concurrence ainsi qu'une manière d'encoder sa sémantique à l'aide de transducteurs étendus. Par définition, ces transducteurs ne peuvent pas avoir de gardes possédant une arité variable et donc incapables de lire un nombre arbitraire de processus. Bien que ce mécanisme nous permette d'encoder des communications entre processus voisins, il n'est pas possible, avec ce formalisme, d'exprimer de communications point-à-point, c'est-à-dire, entre deux processus « distants » dans le mot de processus. Il nous faut donc un modèle plus expressif capable d'encoder ce type de comportement.

Chapitre 4

Sémantique de communications point-à-point et collectives

Nous avons vu que les transducteurs de treillis définis dans le chapitre précédent permettent d'encoder des systèmes communicants manipulant des valeurs numériques qui réalisent des créations et destructions dynamiques de processus. De plus, ces processus sont capables d'effectuer des communications à leur voisinage (*i.e.* à leur gauche et droite selon leur position dans le *mot* de processus). Ces contraintes de position limitent grandement l'expressivité d'un tel modèle. Avec ceci en tête, nous définissons dans ce chapitre une nouvelle manière de procéder à la réécriture de processus. Nous proposons des règles capables d'accepter un ensemble de mots potentiellement non-bornés en entrée et de les réécrire en un ensemble de mots non-bornés. Nous montrons que, grâce à ce mécanisme, nous sommes désormais capables d'encoder des communications point-à-point ou collectives quel que soit la position des processus et ainsi pouvons nous abstraire, au niveau de la programmation, de la structure sous-jacente de « mots » nous permettant alors de proposer un langage plus expressif se rapprochant d'un langage *à la* `C` muni de primitives de communications généralistes.

4.1 Langage du modèle

Nous raffinons ici le langage défini dans le chapitre précédent pour nous rapprocher d'un langage *à la* `C`. Contrairement au langage présenté dans le chapitre précédent, tous les processus exécutent le même code. Cependant, l'existence d'un identifiant pour chaque processus va permettre d'utiliser les instructions conditionnelles pour générer des comportement différents entre les processus. La grammaire de ce nouveau langage est donnée par la Figure 4.1.

Dans ce nouveau langage, chaque processus contient un identifiant unique, un entier, attribué séquentiellement à l'initialisation *en partant de l'entier 0* ou lors de la création dynamique de celui-ci. Nous supposons, cette fois-ci, une donnée globale partagée par tous les processus : la cardinalité des processus. Cette taille peut être obtenue par la primitive **size**. De plus, les processus peuvent connaître leur identifiant via la primitive **get_id**. Les communications restent synchrones. Nous introduisons également la possibilité de communiquer avec un processus quelconque grâce à l'identifiant spécial : `ANY`.

Les programmes démarrent avec un nombre arbitraire de processus choisi par l'utilisateur qui commence à la première instruction de la fonction **main**. La primitive **create** retourne désormais l'identifiant du nouveau processus créé mais ne prend plus d'argument. Leur initialisation peut se faire par commu-

⟨*program*⟩ : := `main` () { ⟨*instr*⟩ }

⟨*instr*⟩ : := ⟨*instr*⟩ ; ⟨*instr*⟩
| ⟨*var*⟩ := ⟨*expr*⟩
| **if** ⟨*expr*⟩ **then** ⟨*instr*⟩ [**else** ⟨*instr*⟩] **endif**
| **while** ⟨*expr*⟩ **do** ⟨*instr*⟩ **done**
| **send**(⟨*expr*⟩, ⟨*proc_id*⟩) — envoi point-à-point
| **receive**(⟨*var*⟩, ⟨*proc_id*⟩) — réception point-à-point
| **broadcast**(⟨*expr*⟩, ⟨*proc_id*⟩) — envoi collectif
| **reduce**(⟨*reduce_op*⟩, ⟨*var*⟩, ⟨*var*⟩, ⟨*proc_id*⟩) — réduction collective
| **return** [⟨*expr*⟩]

⟨*proc_id*⟩ : := ⟨*int*⟩ | ANY | ⟨*expr*⟩

⟨*expr*⟩ : := (⟨*expr*⟩)
| ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩
| min (⟨*expr_list*⟩)
| max (⟨*expr_list*⟩)
| ⟨*int*⟩
| ⟨*float*⟩
| ⟨*bool*⟩
| ⟨*var*⟩
| **random** ()
| **create** (⟨*expr*⟩) — création dynamique
| **size** () — taille des processus
| **get_id** () — accesseur à l'identifiant

⟨*op*⟩ : := + | - | / | * | % | <| >| <= | =>| = | !=

⟨*reduce_op*⟩ : := + | - | * | min (⟨*expr*⟩, ⟨*expr*⟩) | max (⟨*expr*⟩, ⟨*expr*⟩)

⟨*var*⟩ : := [a-z][0-9a-zA-Z_]*

FIGURE 4.1 – Grammaire

nications ultérieures. Chaque nouveau processus démarre son exécution à la première instruction de la fonction `main`, et, tout processus termine son exécution après un appel à une instruction **return**.

4.1.1 Communications point-à-point

Dans ce nouveau langage, nous disposons de deux primitives permettant d'effectuer des communications point-à-point synchrones : **send** et **receive**. Ces instructions n'ont pas de restrictions de position et ne nécessitent que de connaître l'identifiant du processus avec lequel l'on souhaite communiquer.

Une communication sera effectuée si deux processus font respectivement un appel à **send**, portant l'identifiant du processus récepteur, et **receive**, portant l'identifiant du processus émetteur.

Nous illustrons ce mécanisme par la Figure 4.2 présentant un programme où plusieurs processus communiquent. Le premier processus, d'identifiant 0, envoie à chaque autre processus, d'identifiants ≥ 1, un entier dont la valeur est le double de leur identifiant.

```
1   main() {
2     id = get_id();
3     world_size = size();
4     x = 0;
5
6     if id = 0 then
7       i = 1;
8       while (i < world_size)
9         send(i+i, i);
10        i = i + 1
11      done
12    else
13      receive(x, 0)
14    endif;
15
16    return
17  }
```

FIGURE 4.2 – Échanges de messages entre au moins deux processus

4.1.2 Communications collectives

Nous introduisons, dans ce langage, des communications collectives. Ce type de communications s'effectue en consensus avec tous les processus. Nous présentons deux types d'opérations :

— la communication globale, effectuée par la primitive **broadcast** consistant à envoyer une valeur calculée par un processus, dont l'identifiant est passé en paramètre, à tous les autres.

— l'opération de réduction, par la primitive **reduce** prenant un opérateur (commutatif), et récoltant chaque valeur, présente dans la première variable des arguments, des processus en appliquant à chaque réception l'opérateur associé au résultat accumulé pour finalement transmettre le résultat au processus dont l'identifiant est passé en paramètre. Ce résultat se trouvera dans la deuxième variable passée en paramètre dans l'environnement du processus désigné.

Ces communications ne sont effectuées que si tous les processus sont prêts à communiquer. Ainsi, les processus atteignant une communication collective se mettent en attente jusqu'à ce que les autres processus les rejoignent. Les processus doivent d'exécuter la communication se trouvant à la même localité : si deux processus sont en attente sur deux communications collectives différentes, ils s'en trouvent irrémédiablement bloqués.

Nous montrons, dans la Figure 4.3a, un programme dont un processus est chargé de recevoir une entrée utilisateur pour la transmettre aux autres processus. Le processus 0 attend donc l'entrée utilisateur puis rejoint le **broadcast** où les autres processus sont censés attendre pour effectuer la communication qui va propager la valeur de `user_input`.

Dans la Figure 4.3b, nous réalisons la réduction par une somme de tous les identifiants des processus. Une fois tous les identifiants et tailles obtenus, les processus exécutent l'opération **reduce**. Cette opération somme tous les identifiants et envoie le résultat au processus d'identifiant 0. Puis, le processus 0 propage ce résultat aux autres processus grâce à un appel à **broadcast**. *N.B.* Ce type de communication n'a aucun effet si le système ne comprend pas au moins deux processus.

Il est important de noter que ces deux communications pourraient facilement être encodées à l'aide de **send/receive** et de boucles. Nous avons fait le choix d'ajouter ces primitives principalement pour faciliter et améliorer la précision de l'analyse. Nous verrons par la suite comment ces primitives sont traitées.

```
1  main() {
2    id = get_id();
3    user_input = -1;
4
5    if id = 0 then
6      // get input...
7      user_input = ...;
8    endif;
9
10   broadcast(user_input, 0);
11
12   return
13 }
```

(a) Envoi collectif

```
1  main() {
2    id = get_id();
3    n = get_size();
4    res = 0;
5
6    reduce(+, id, res, 0);
7    broadcast(res, 0);
8
9    // assert (res = n * (n-1) / 2)
10
11   return;
12 }
```

(b) Réduction

FIGURE 4.3 – Exemples de communications collectives

4.2 Sémantique des règles

Nous allons maintenant décrire une nouvelle structure de réécriture dont le but est de nous permettre d'encoder les différents traits de langage présentés. Cette structure se doit d'être suffisamment générale pour nous permettre de modéliser des comportements haut-niveau tel que les communications point-à-point et collectives.

Comme nous l'avons vu, un des principaux désavantages des transducteurs de treillis présentés dans le chapitre précédent est de ne pas être capable de réécrire deux processus espacés, dans le mot, d'un ensemble arbitraire de processus. Notre idée est donc de proposer une réécriture où des gardes spéciales sont capables de reconnaître une séquence de processus de longueur arbitraire, a priori non bornée. Grâce à la notion de *gardes étoilées* g^*, nous sommes capables de reconnaître des sous-mots de processus satisfaisant une condition g.

Cependant, il nous faut également un moyen de réécrire ce sous-ensemble reconnu. Pour se faire, nous introduisons également des fonctions de réécriture permettant de réécrire tous les processus reconnus par une garde étoilée g^*. Ces fonctions de réécriture spéciales auront besoin d'un argument supplémentaire : le processus à réécrire dans le sous-mot. Une fois appliquées, ces fonctions permettront de réécrire tous les processus présents dans le sous-mot en modifiant l'argument pour chaque processus.

4.2.1 Définition

Nous donnons ici une définition formelle des règles de réécriture.

Définition 16. *Soit Λ un treillis, une règle de réécriture $\mathcal{R} = (G, F)$ sur Λ est donnée par deux séquences $G = (g_0)^* \cdot w_1 \cdot (g_1)^* \cdot w_2 \ldots w_n \cdot (g_n)^*$ et $F = f_0 \cdot F_0^* \cdot f_1 \cdot F_1^* \cdot \ldots \cdot F_n^* \cdot f_{n+1}$ telles que :*

- $\forall\ 1 \leq i \leq n, w_i \in \Lambda^*$ *and* $|w_i| > 0$ *;*
- $\forall\ 0 \leq i \leq n, g_i \in \Lambda$ *;*

  On note $N = |w_1| + |w_2| + \ldots + |w_n|$
- $\forall\ 0 \leq i \leq n+1,\ f_i : \Lambda^N \to \Lambda^*$ *;*
- $\forall\ 0 \leq i \leq n,\ F_i^* : \Lambda^{N+1} \to \Lambda$.

Avec cette règle, un mot fini $w \in \Lambda^*$ est réécrit en $w' \in \Lambda^*$ si :

- w peut être réécrit comme une concaténation $w = u_0 \cdot v_1 \cdot u_1 \cdot \ldots \cdot v_n \cdot u_n$ avec :

— $\forall\, 0 \leq i \leq n, u_i = \lambda_0 \dots \lambda_{|u_i|}$ et $\forall\, 0 \leq j \leq |u_i|, \lambda_j \sqsubseteq g_i$,
— $\forall\, 1 \leq i \leq n, v_i \sqsubseteq w_i$;

— $w' = v'_0 \cdot u'_0 \cdot v'_1 \cdot u'_1 \cdot \ldots \cdot v'_n \cdot u'_n \cdot v'_{n+1}$ avec :
— $\forall\, 0 \leq i \leq n, u'_i = \lambda'_0 \cdot \lambda'_1 \cdot \ldots \cdot \lambda'_{|u_i|}$ et $\forall\, 0 \leq j \leq |u_i|, \lambda'_j = F^*_i(\lambda_j, v_1, \ldots, v_n)$,
— $\forall\, 0 \leq i \leq n+1, v'_i = f_i(v_1, \ldots, v_n)$.

Un algorithme d'application d'une règle $\mathcal{R}$ à un automate $\mathcal{A}$ est donné en section 4.3.1.

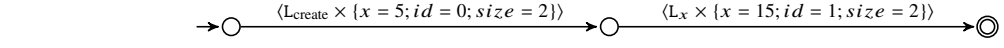
4.2.2 Sémantique abstraite des communications

Nous donnons maintenant une nouvelle sémantique pour la création dynamique de processus considérant désormais la taille ainsi que la sémantique des primitives de communication définies dans le langage sous la forme de règles de réécriture ainsi que quelques exemples d'application de celles-ci.

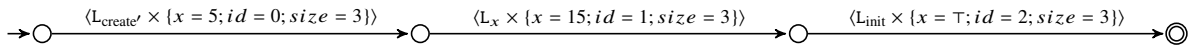
Création dynamique de processus avec taille

Dans notre langage, nous considérons désormais une variable de taille. Celle-ci va permettre de conserver une information du nombre de processus présents lorsque nous effectuons des abstractions sur la forme des automates. Cette variable doit se comporter comme une variable globale. Ainsi, lorsque nous effectuons une création de processus, nous devons mettre à jour cette taille dans les mots de processus qui vont en créer un nouveau. Nous définissons ainsi une nouvelle sémantique pour cette création dynamique intégrant cette mise-à-jour de la taille. Nous faisons le choix d'attribuer au nouveau processus créé la taille précédente en tant que nouvel identifiant avant d'incrémenter globalement celle-ci. Finalement, nous attribuons en tant qu'état initial au nouveau processus créé, la localité initiale du programme (*i.e.* celle associée à la première instruction) avec un environnement initialisé à $\top$.

$$
\begin{aligned}
&\top^* \cdot \langle \mathrm{L}_{\mathrm{create}} \times \top \rangle \cdot \top^* \,/ \\
&F^*_0 : (\langle l_{\mathrm{self}}, \rho_{\mathrm{self}} \rangle, \langle l_{\mathrm{create}}, \rho_{\mathrm{create}} \rangle) \mapsto \langle l_{\mathrm{self}}, \rho_{\mathrm{self}}[\mathrm{size} \mapsto \rho_{\mathrm{self}}(\mathrm{size}) + 1] \rangle \cdot \\
&f_1 : (l_{\mathrm{create}}, \rho_{\mathrm{create}}) \mapsto \langle \mathrm{L}_{\mathrm{create}'}, \rho_{\mathrm{create}}[\mathrm{size} \mapsto \rho_{\mathrm{create}}(\mathrm{size}) + 1] \rangle \cdot \\
&F^*_1 : (\langle l_{\mathrm{self}}, \rho_{\mathrm{self}} \rangle, \langle l_{\mathrm{create}}, \rho_{\mathrm{create}} \rangle) \mapsto \langle l_{\mathrm{self}}, \rho_{\mathrm{self}}[\mathrm{size} \mapsto \rho_{\mathrm{self}}(\mathrm{size}) + 1] \rangle \cdot \\
&f_2 : (l_{\mathrm{create}}, \rho_{\mathrm{create}}) \mapsto \langle \mathrm{L}_{\mathrm{init}}, \rho_{init}[id \mapsto \rho_{\mathrm{create}}(\mathrm{size}), \mathrm{size} \mapsto \rho_{\mathrm{create}}(\mathrm{size}) + 1] \rangle
\end{aligned}
$$



(a) $\mathcal{A}$



(b) $\mathcal{R}(\mathcal{A})$

FIGURE 4.4 – Exemple de création dynamique avec mise-à-jour de taille

La Figure 4.4 illustre ce mécanisme. Nous partons d'une configuration à 2 processus respectivement d'identifiants 0 et 1 où le premier processus est à une localité de création dynamique et le deuxième à une localité quelconque. En appliquant la règle de réécriture, nous obtenons un nouvel état de programme composé d'un nouveau processus où toutes les variables de taille des processus reconnus par la règle ont

été mises à jour. Également, le nouveau processus obtient en identifiant l'ancienne valeur de la taille du mot. Il est à noter que l'on concatène le nouveau processus à la droite du mot reconnu mais tout autre position (*e.g.* au début, à gauche ou à droite de l'appelant, ...) serait possible.

Communications point-à-point

Cette communication s'effectue par la synchronisation entre deux processus exécutant respectivement une instruction **send** et une instruction **receive**. La sémantique de cette communication d'un processus d'identifiant i vers un autre d'identifiant j aux localités respectives l_{send} et l_{recv} allant vers l'_{send} et l'_{recv} transmettant la valeur d'une expression e évaluée dans l'environnement du premier processus vers la variable y du second se présente ainsi :

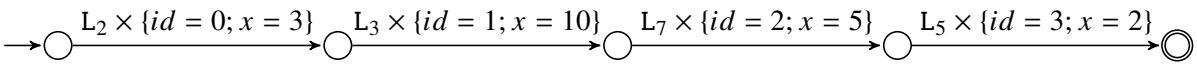
$$
\begin{aligned}
&\top^* \cdot \langle l_{\text{send}}, \{id = i\}\rangle \cdot \top^* \cdot \langle l_{\text{recv}}, \{id = j\}\rangle \cdot \top^* \,/\\
&F_0^* : (\sigma_{\text{self}}, \sigma_{\text{send}}, \sigma_{\text{recv}}) \mapsto \sigma_{\text{self}} \cdot\\
&f_1 : (\langle l_{\text{send}}, \rho_{\text{send}}\rangle, \langle l_{\text{recv}}, \rho_{\text{recv}}\rangle) \mapsto \langle l'_{\text{send}}, \rho_{\text{send}}\rangle \cdot\\
&F_1^* : (\sigma_{\text{self}}, \sigma_{\text{send}}, \sigma_{\text{recv}}) \mapsto \sigma_{\text{self}} \cdot\\
&f_2 : (\langle l_{\text{send}}, \rho_{\text{send}}\rangle, \langle l_{\text{recv}}, \rho_{\text{recv}}\rangle) \mapsto \langle l'_{\text{recv}}, \rho_{\text{recv}}[y \mapsto \text{eval}(e, \rho_{\text{send}})]\rangle \cdot\\
&F_2^* : (\sigma_{\text{self}}, \sigma_{\text{send}}, \sigma_{\text{recv}}) \mapsto \sigma_{\text{self}}
\end{aligned}
$$

Étant donné qu'aucune supposition n'est faite sur la disposition des processus dans le mot, il est nécessaire d'effectuer l'union avec la règle symétrique plaçant l'émetteur après le récepteur.

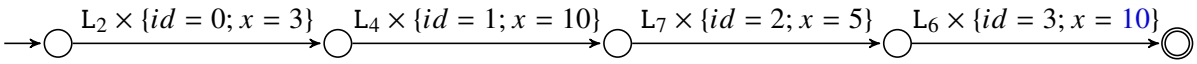
Pour illustrer cette sémantique, nous donnons un exemple où 2 processus communiquent aux localités $\mathtt{L}_3$ (send) et $\mathtt{L}_5$ (receive). La règle de communication, donnée par la Figure 4.5a est appliquée une configuration (Figure 4.5b). Le résultat de l'application de cette règle calcule un nouvel automate (Figure 4.5c) où la variable x du processus d'identifiant 1 est envoyée à la variable x du processus d'identifiant 3.

$$\mathcal{R} = \top^* \cdot \langle \mathrm{L}_3 \times \{id = 1\}\rangle \cdot \top^* \cdot \langle \mathrm{L}_5 \times \{id = 3\}\rangle \cdot \top^* \,/$$
$$F_0^* : (\sigma_{\text{self}}, \sigma_{\text{send}}, \sigma_{\text{recv}}) \mapsto \sigma_{\text{self}} \cdot$$
$$f_1 : (\langle \mathrm{L}_3, \rho_{\text{send}}\rangle, \langle \mathrm{L}_5, \rho_{\text{recv}}\rangle) \mapsto \langle \mathrm{L}_4, \rho_{\text{send}}\rangle \cdot$$
$$F_1^* : (\sigma_{\text{self}}, \sigma_{\text{send}}, \sigma_{\text{recv}}) \mapsto \sigma_{\text{self}} \cdot$$
$$f_2 : (\langle \mathrm{L}_3, \rho_{\text{send}}\rangle, \langle \mathrm{L}_5, \rho_{\text{recv}}\rangle) \mapsto \langle \mathrm{L}_6, \rho_{\text{recv}}[x \mapsto \rho_{\text{send}}(x)]\rangle \cdot$$
$$F_2^* : (\sigma_{\text{self}}, \sigma_{\text{send}}, \sigma_{\text{recv}}) \mapsto \sigma_{\text{self}}$$

(a) Règle de communication point-à-point



(b) Configuration $\mathcal{A}$



(c) Configuration $\mathcal{R}(\mathcal{A})$

FIGURE 4.5 – Exemple de communication Point-à-point

Communications collectives

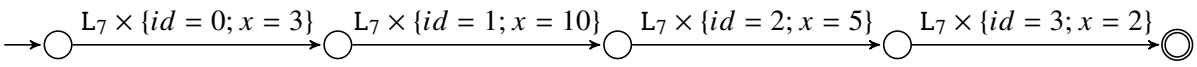
La première primitive de communication collective dont nous exprimons la sémantique est la communication multi-point **broadcast**. La sémantique de cette primitive se présente ainsi : tous les processus sont à la localité l_{bcast} allant vers l'_{bcast}, et un processus d'identifiant i envoie la valeur d'une expression e, évaluée dans son environnement, dans la variable y de tous les autres processus du système se présente ainsi :

$$\langle l_{\text{bcast}}, \top\rangle^* \cdot \langle l_{\text{bcast}}, \{id = i\}\rangle \cdot \langle l_{\text{bcast}}, \top\rangle^* \,/$$
$$F_0^* : (\langle l_{\text{bcast}}, \rho_{\text{self}}\rangle, \langle l_{\text{bcast}}, \rho_{\text{send}}\rangle) \mapsto \langle l'_{\text{bcast}}, \rho_{\text{self}}[y \mapsto \text{eval}(e, \rho_{\text{send}})]\rangle \cdot$$
$$f_1 : \langle l_{\text{bcast}}, \rho_{\text{send}}\rangle \mapsto \langle l'_{\text{bcast}}, \rho_{\text{send}}\rangle \cdot$$
$$F_1^* : (\langle l_{\text{bcast}}, \rho_{\text{self}}\rangle, \langle l_{\text{bcast}}, \rho_{\text{send}}\rangle) \mapsto \langle l'_{\text{bcast}}, \rho_{\text{self}}[y \mapsto \text{eval}(e, \rho_{\text{send}})]\rangle$$

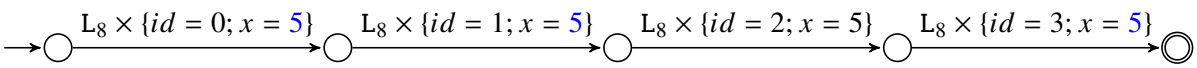
Comme pour la règle de communication point-à-point, nous donnons ici un exemple où un processus d'identifiant 2 envoie sa variable x à tous lorsque les processus sont à la localité L_7. La règle, présentée par la Figure 4.6a, est appliquée à une configuration (Figure 4.6b) pour donner comme résultat l'automate de la Figure 4.6c.

$$\mathcal{R} = \langle \mathrm{L}_7 \times \top \rangle^* \cdot \langle \mathrm{L}_7 \times \{id = 2\} \rangle \cdot \langle \mathrm{L}_7 \times \top \rangle^* \,/$$
$$F_0^* : (\langle \mathrm{L}_7, \rho_{\text{self}} \rangle, \langle \mathrm{L}_7, \rho_{\text{send}} \rangle) \mapsto \langle \mathrm{L}_8, \rho_{\text{self}}[x \mapsto \rho_{\text{send}}(x)] \rangle \cdot$$
$$f_1 : (\mathrm{L}_7, \rho_{\text{send}}) \mapsto \langle \mathrm{L}_8, \rho_{\text{send}} \rangle \cdot$$
$$F_1^* : (\langle \mathrm{L}_7, \rho_{\text{self}} \rangle, \langle \mathrm{L}_7, \rho_{\text{send}} \rangle) \mapsto \langle \mathrm{L}_8, \rho_{\text{self}}[x \mapsto \rho_{\text{send}}(x)] \rangle$$

(a) Règle de communication multi-point



(b) Configuration $\mathcal{A}$



(c) Configuration $\mathcal{R}(\mathcal{A})$

FIGURE 4.6 – Exemple de communication multi-point

La deuxième primitive de communication collective est une opération de réduction. Celle-ci est une opération complexe, il est nécessaire de la décomposer en plusieurs règles. L'idée est d'introduire un processus *fantôme* « collecteur » chargé de récolter les valeurs des variables de chaque processus et d'effectuer au fur et à mesure les opérations. Nous avons donc besoin de trois règles : la création de ce processus fantôme, le parcours de chaque processus et la transmission du résultat final au processus récepteur en même temps que la destruction de ce collecteur.

Ainsi, la sémantique d'une opération **reduce**, où tous les processus sont à une localité l_{reduc} suivie par l'_{reduc} et où un processus d'identifiant i reçoit le résultat de la réduction de toutes les variables v des processus avec l'opérateur commutatif op se présente comme :

1.
$$\langle l_{\text{reduc}}, \top \rangle^* \,/ f_0 : \emptyset \mapsto \langle l_{\text{coll}}, \{v \mapsto e_0\} \rangle \cdot F_0^* : \langle \emptyset, \rho_{\text{self}} \rangle \mapsto \langle l_{\text{lock}}, \rho_{\text{self}} \rangle$$

2.
$$\top^* \cdot \langle l_{\text{lock}}, \top \rangle \cdot \langle l_{\text{coll}}, \top \rangle \cdot \top^* \,/$$
$$Id^* \cdot$$
$$f_1 : (\langle l_{\text{coll}}, \rho_{\text{coll}} \rangle, \langle l_{\text{lock}}, \rho_{\text{lock}} \rangle) \mapsto \langle l_{\text{lock}}, \rho_{\text{lock}} \rangle \cdot$$
$$f_2 : (\langle l_{\text{coll}}, \rho_{\text{coll}} \rangle, \langle l_{\text{lock}}, \rho_{\text{lock}} \rangle) \mapsto \langle l_{\text{coll}}, \rho_{\text{coll}}[v \mapsto \text{eval}(v \; [\![op]\!] \; \rho_{\text{lock}}(v))] \rangle \cdot$$
$$Id^*$$

3.
$$\langle l_{\text{lock}}, \top \rangle^* \cdot \langle l_{\text{lock}}, \{id = i\} \rangle \cdot \langle l_{\text{lock}}, \top \rangle^* \cdot \langle l_{\text{coll}}, \top \rangle \,/$$
$$F_0^* : (\langle l_{\text{lock}}, \rho_{\text{self}} \rangle, \langle l_{\text{lock}}, \rho_{\text{recv}} \rangle, \langle l_{\text{coll}}, \rho_{\text{coll}} \rangle) \mapsto \langle l'_{\text{reduc}}, \rho_{\text{self}} \rangle \cdot$$
$$f_1 : (\langle l_{\text{lock}}, \rho_{\text{recv}} \rangle, \langle l_{\text{coll}}, \rho_{\text{coll}} \rangle) \mapsto \langle l'_{\text{reduc}}, \rho_{\text{recv}}[v \mapsto \rho_{\text{coll}}(v)] \rangle \cdot$$
$$F_1^* : (\langle l_{\text{lock}}, \rho_{\text{self}} \rangle, \langle l_{\text{lock}}, \rho_{\text{recv}} \rangle, \langle l_{\text{coll}}, \rho_{\text{coll}} \rangle) \mapsto \langle l'_{\text{reduc}}, \rho_{\text{self}} \rangle$$

où l_{coll} est la localité fantôme du collecteur, l_{lock} est une localité fantôme permettant de bloquer l'évolution des processus en attente de la complétion de la collecte, e_0 est l'élément neutre de l'opérateur op et Id^* est la fonction réécrivant les processus d'un sous-mot par l'identité.

4.3 Vérification du système

4.3.1 Algorithme d'application des règles

Pour appliquer une règle de réécriture au langage reconnu par un automate de treillis, nous commençons par identifier le sous-ensemble de mots reconnus par la garde $(g_0)^* \cdot w_1 \cdot (g_1)^* \cdot w_2 \dots w_n \cdot (g_n)^*$. Dans ce type de garde, il est plus simple de commencer par chercher les séquences de l'automate qui vérifient les gardes $w_1, w_2, \dots, w_n$. Dans l'automate $\mathcal{A}$, une séquence qui satisfait w_1 commence à l'état q_b^1 et finit à l'état q_e^1. Puis, nous devons chercher l'ensemble des sous-automates qui satisfait $(g_0)^*$, *i.e.* tous les états atteignables depuis q_0 (l'état initial) et les états co-atteignables depuis q_b^1 en ne considérant que les transitions dont les labels sont des éléments λ tels que $g_0 \sqcap \lambda \neq \bot$. En appliquant itérativement cette méthode, nous identifions un sous-automate de $\mathcal{A}$ auquel nous appliquons les fonctions de réécriture correspondantes calculant un nouvel automate $\mathcal{A}'$. Le résultat de cet algorithme est ainsi l'union de tous les automates $\mathcal{A}'$ construits de cette manière. Toutefois, cette reconnaissance n'étant pas déterministe, nous devons considérer toutes les séquences possibles.

Nous introduisons quelques notations avant de donner l'algorithme. Soit $w = \lambda_1 \dots \lambda_n \in \Lambda^n$ et $\mathcal{A}$ un automate de treillis. Nous notons par matches$(w, \mathcal{A})$ la fonction reconnaissant l'ensemble des séquences dans un automate $\mathcal{A}$ satisfaisant le mot w.

$$\begin{aligned}\text{matches}(w,\mathcal{A}) = \{(q_b, v_1 \dots v_n, q_e) \mid \exists q_0 \xrightarrow{\lambda'_1} q_1 \xrightarrow{\lambda'_2} \dots \xrightarrow{\lambda'_n} q_n \in \mathcal{A},\\ q_0 = q_b \wedge q_n = q_e \wedge \forall i = 1..n, v_i = \lambda_i \sqcap \lambda'_i \neq \bot\}\end{aligned}$$

Soit (q_b, q_e) un couple d'états d'un automate de treillis $\mathcal{A} = \langle \Lambda, Q, Q_0, Q_f, \delta \rangle$ et soit $\lambda \in \Lambda$. Nous notons $\mathcal{A}_{q_b \to q_e}$ le sous-automate $\langle \Lambda, Q, \{q_b\}, \{q_e\}, \delta \rangle$. Pour un automate de treillis $\mathcal{A}$ et une fonction $f : \Lambda \to \Lambda$, nous notons map$(f, \mathcal{A})$ l'automate $\langle \Lambda, Q, Q_0, Q_f, f(\delta) \rangle$ où $f(\delta) = \{(q, f(\lambda), q') | (q, \lambda, q') \in \delta \wedge f(\lambda) \neq \bot\}$.

Nous pouvons maintenant donner l'algorithme d'application d'une règle de réécriture $\mathcal{G}$ à un auto-

mate $\mathcal{A}$.

Algorithme 2 : `Application`$(\mathcal{R},\mathcal{A})$

Étant donné $\mathcal{R} = (g_0)^* \cdot w_1 \cdot (g_1)^* \dots w_n \cdot (g_n)^* \ / \ f_0 \cdot F_0^* \cdot f_1 \dots F_n^* \cdot f_{n+1}$;
pour tout $(q_b^1, v^1, q_e^1) \in \text{matches}(w_1, \mathcal{A})$, …, $(q_b^n, v^n, q_e^n) \in \text{matches}(w_n, \mathcal{A})$ **faire**
  pour tout $q_0 \in Q_0^{\mathcal{A}} \times q_f \in Q_f^{\mathcal{A}}$ **faire**
    Soit $\mathcal{A}_0 = \text{map}(x \mapsto g_0 \sqcap x, \mathcal{A}_{q_0 \to q_b^1})$,
    $\mathcal{A}_1 = \text{map}(x \mapsto g_1 \sqcap x, \mathcal{A}_{q_e^1 \to q_b^2})$,
    …,
    $\mathcal{A}_n = \text{map}(x \mapsto g_n \sqcap x, \mathcal{A}_{q_e^n \to q_f})$.
    pour tout $i \in 0 \dots n$ **faire**
      $\mathcal{A}'_i = \text{map}(x \mapsto F_i^*(x, v^1, \dots, v^n), \mathcal{A}_i)$.
    fin
    pour tout $i \in 0 \dots n+1$ **faire**
      $w'_i = f_i(v^1, \dots, v^n)$
    fin
    Soit q_{-1} et q_{n+1} deux états frais (non-présents dans $\mathcal{A}'_i$)
    Soit $\delta^{seq} = \{(q_{-1}, w'_0, q_0)(q_b^1, w'_1, q_e^1)(q_b^1, w'_1, q_e^1) \dots (q_b^n, w'_n, q_e^n)(q_b^n, w'_{n+1}, q_{n+1})\}$
    Soit $\mathcal{A}' = \langle \Lambda, Q \cup \{q_{-1}, q_{n+1}\}, \{q_{-1}\}, \{q_{n+1}\}, \delta^{\mathcal{A}'} \rangle$ avec $\delta^{\mathcal{A}'} = \delta^{seq} \cup \delta^{\mathcal{A}'_0} \cup \dots \cup \delta^{\mathcal{A}'_n}$
    Result ← Result ∪ $\mathcal{A}'$
  fin
fin
retourner ε-`removal(trim(`*Result*`))`

Nous illustrons cet algorithme, par la Figure 4.7, où nous définissons une règle de communication multi-point s'appliquant à un état de programme où les processus sont représentés par une variable numérique abstraite grâce au domaine des intervalles. La règle de réécriture est composée d'une garde reconnaissant un ensemble non-borné de processus dont la valeur est comprise entre $\varphi_1 \in [1,9]$ suivi d'un processus dont la valeur est $\gamma_1 \in [10,20]$ et se terminant par un ensemble non-borné de processus entre $\varphi_2 \in [1,9]$. Notre algorithme commence par identifier les gardes simples γ_1 donnant les deux états abstraits de processus $[11,18]$ et $[10,12]$ puis cherche les sous-automates partant du nœud initial allant à ceux-ci. Cette opération en identifie deux et ignore l'état $[-9,-5]$ qui n'est pas reconnu par la garde. La même opération est effectuée pour la partie de droite à ceci près qu'il n'y a plus de garde simple à atteindre : on doit ainsi chercher les automates atteignant un état final. Dans notre cas, le sous-automate reconnu est le même pour les deux états. On applique maintenant les fonctions de réécriture correspondantes aux différents éléments discriminés de notre état abstrait. Les deux sous-automates de gauche reçoivent respectivement les valeurs $[11,18]$ et $[10,12]$ selon les chemins reconnus. Cependant, les états de processus du sous-automate de droite se réécrivent soit en $[11,18]$, soit en $[10,12]$. Notre algorithme effectue alors l'union de ces deux sous-automates possibles résultant en des processus pouvant prendre l'intervalle de valeur $[10,18]$. Les états reconnus par la garde simple sont réécrits en l'identité comme spécifié par la fonction de réécriture associée.

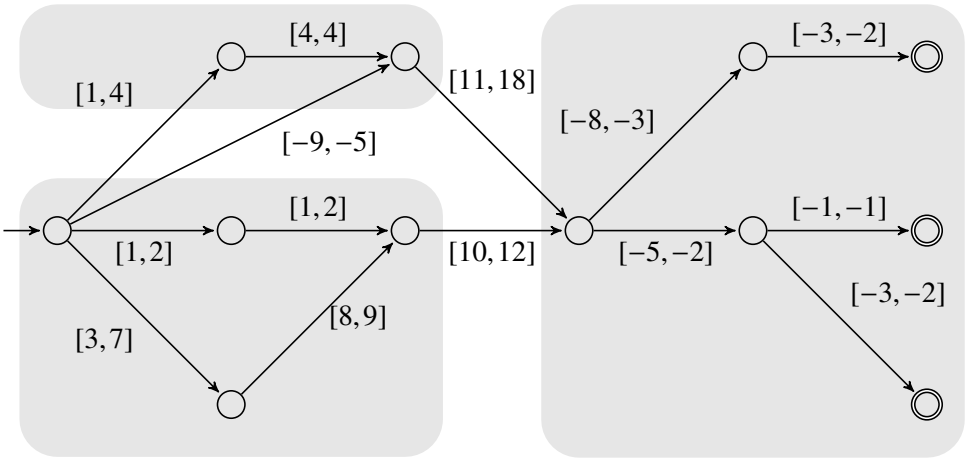
Complexité

La complexité de cet algorithme est :

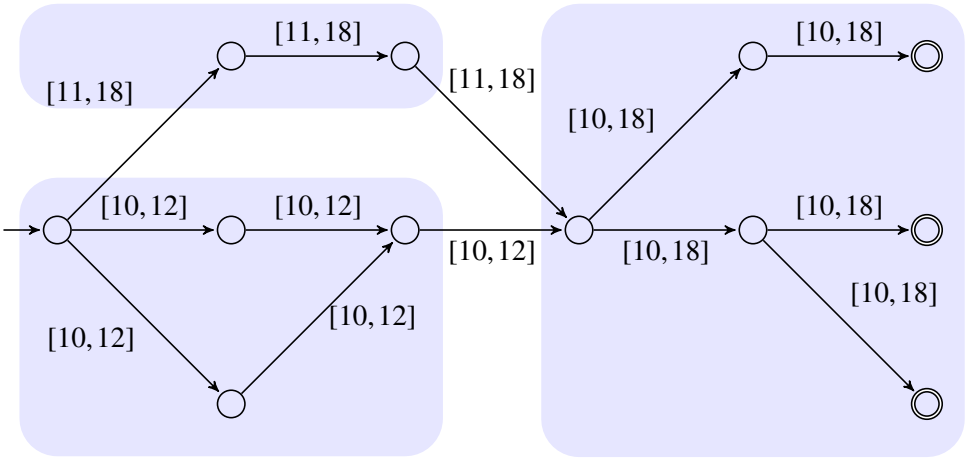
$$\varphi_1^* \cdot \gamma_1 \cdot \varphi_2^* \,/\, \mathbf{F}_1^*(\boldsymbol{\sigma_c}, \boldsymbol{\sigma_1}) \mapsto \sigma_c \xleftarrow{val} \sigma_1 \cdot \mathbf{f}(\boldsymbol{\sigma_1}) \mapsto \sigma_1 \cdot \mathbf{F}_2^* \,(\equiv \mathbf{F_1})$$

avec $\varphi_1 = [1,9]$, $\gamma_1 = [10,20]$ et $\varphi_2 = [-10,0]$.

(a) $\mathcal{R}_{bcast}$ = règle de communication multi-point



(b) $\mathcal{A}$



(c) $\mathcal{R}_{bcast}(\mathcal{A})$

FIGURE 4.7 – Application de règle sur un état de programme abstrait

$$O\left(Q_{\mathcal{A}}^{2n+2} \cdot |\pi|^N \cdot ((n+1) \cdot |\Delta_{\mathcal{A}}| + (n+2) \cdot c(f))\right)$$

où $|Q_{\mathcal{A}}|$ est la cardinalité des états de l'automate de treillis, n a la même signification que dans l'Algorithme 2, $|\pi|$ est la taille de la partition de l'automate de treillis (i.e. le nombre de localités différentes dans le programme), N est $|w_1| + |w_2| + \cdots + |w_n|$ et $c(f)$ représente la complexité maximum de toutes les fonctions de réécriture données.

Démonstration. Dans la première boucle de l'algorithme, il y a au plus $Q_{\mathcal{A}} \times |\pi|^{|w_1|} \times Q_{\mathcal{A}}$ séquences de transitions dans matches$(w_1, \mathcal{A})$, puisque chaque état de $\mathcal{A}$ a au plus $|\pi|$ transitions sortantes. Ce qui fait au plus $Q_{\mathcal{A}}^{2n} \times |\pi|^{|w_1|+\cdots+|w_n|} = Q_{\mathcal{A}}^{2n} \times |\pi|^N$ itérations possibles de la première boucle. Avec les choix de $q_0 \in Q_0^{\mathcal{A}} \times q_f \in Q_f^{\mathcal{A}}$, on a donc $Q_{\mathcal{A}}^{2n+2} \times |\pi|^N$ itérations possibles, dans lesquelles on fait :

— $n+1$ calculs d'automates $\mathcal{A}_0, \mathcal{A}_n$, chacun ayant une complexité en $O(|\Delta_{\mathcal{A}}|)$ puisqu'on applique des opérations à potentiellement toutes les transitions de $\mathcal{A}$

— $n+2$ applications des fonctions de réécriture, chacune ayant une complexité d'au plus $c(f)$

D'où une complexité totale de l'ordre de :

$$O\left(Q_{\mathcal{A}}^{2n+2} \cdot |\pi|^N \cdot ((n+1) \cdot |\Delta_{\mathcal{A}}| + (n+2) \cdot c(f))\right)$$

□

Correction

Pour un automate $\mathcal{A}$ et une règle de réécriture $\mathcal{R}$, on note $\mathcal{R}(\mathcal{A}) = ApplyRule(\mathcal{R}, \mathcal{A})$. Avec ces notations, on a :

Théorème 4. *Soit $\mathcal{R} = (g_0)^* \cdot w_1 \cdot (g_1)^* \cdot w_2 \ldots w_n \cdot (g_n)^* \;/\; f_0 \cdot F_0^* \cdot f_1 \cdot F_1^* \cdot \ldots \cdot F_n^* \cdot f_{n+1}$ une règle de réécriture et $\mathcal{A}$ un automate de treillis. On a :*

$$\mathcal{R}(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{R}(\mathcal{A}))$$

Démonstration. Soit $\mathcal{R} = (g_0)^* \cdot w_1 \cdot (g_1)^* \cdot w_2 \ldots w_n \cdot (g_n)^* / f_0 \cdot F_0^* \cdot f_1 \cdot \ldots \cdot F_n^* \cdot f_{n+1}$

Soit $w \in \mathcal{L}(A)$. Si w est reconnu par la garde $(g_0)^* \cdot w_1 \cdot (g_1)^* \cdot w_2 \ldots w_n \cdot (g_n)^*$, cela signifie que nous pouvons le décomposer en $w = u_0.v_1.u_1.v_2.u_2 \ldots v_n.u_n$ avec :

— Pour $i = 1..n$, $v_i \sqsubseteq w_i$

— Pour $i = 0..n$, chaque lettre u_i est plus petite que g_i

Puisque $w \in \mathcal{L}(\mathcal{A})$, nous considérons un chemin $q_0 \xrightarrow{u_0}\!\!\!\!\rightarrow q_b^1 \xrightarrow{v_1}\!\!\!\!\rightarrow q_e^1 \ldots \xrightarrow{v_n}\!\!\!\!\rightarrow q_e^n \xrightarrow{u_n}\!\!\!\!\rightarrow q_f$ dans $\mathcal{A}$, i.e. il y a des séquences acceptantes $(q_b^1, v^1, q_e^1) \in \text{matches}(w_1, \mathcal{A})$, $(q_b^2, v^2, q_e^2) \in \text{matches}(w_2, \mathcal{A})$, …, $(q_b^n, v^n, q_e^n) \in \text{matches}(w_n, \mathcal{A})$, telles que $\forall i, v_i \sqsubseteq v^i$.

Dans l'algorithme `ApplyRule`, ces séquences acceptantes génèrent un automate $\mathcal{A}'$. En appliquant les fonctions de réécriture $f_0 \cdot F_0^* \cdot f_1 \cdot F_1^* \cdot \ldots \cdot F_n^* \cdot f_{n+1}$ à w, nous obtenons $\mathcal{R}(w)$ qui est reconnaissable par l'automate $\mathcal{A}'$. Ainsi, $\mathcal{R}(w) \in \mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{R}(A))$. □

Toutefois, il est à noter que nous n'avons pas la relation $\mathcal{R}(\mathcal{L}(\mathcal{A})) \supseteq \mathcal{L}(\mathcal{R}(\mathcal{A}))$ comme nous le montrons dans cet exemple :

Exemple 5. *Soit $\mathcal{A}$ l'automate de treillis reconnaissant le langage $L = \{[0,0],[1,1]\}$ (*i.e. *un seul processus avec une variable pouvant être 0 ou 1) et soit $\mathcal{R}$ la règle de réécriture $\top \;/\; f_1 \cdot f_2$ où $f_1 : x \mapsto 2x$ et $f_2 : x \mapsto 4x$ alors $\mathcal{R}(\mathcal{L}(\mathcal{A})) = \{[0,0].[0,0], [2,2].[4,4]\}$. Mais $\mathcal{R}(\mathcal{A})$ est un automate de treillis reconnaissant 4 mots :*

$\mathcal{L}(\mathcal{R}(\mathcal{A})) = \{[0,0] \cdot [0,0], [2,2] \cdot [4,4], [0,0] \cdot [4,4], [2,2] \cdot [0,0]\}$.

4.3.2 Initialisation de la vérification

Configuration initiale

Avant de débuter la phase d'analyse, il est nécessaire de définir la classe de système que l'on souhaite analyser. Nous en distinguons deux principales : les configurations à nombre de processus fixe (*e.g.* 4.8a) et les configurations à processus non-bornés (*e.g.* 4.8b où $N = +\infty$). Ce choix dépend principalement du type de propriétés à prouver. La première classe permet d'obtenir un résultat généralement plus précis puisqu'aucune abstraction n'est faite au départ sur le nombre de processus traités. En revanche, la deuxième permet de *généraliser une propriété à tout nombre de processus* ce qui est utile pour certaines

classes de programmes comme, par exemple, des algorithmes numériques pouvant être exécutés sur différentes plates-formes de calcul. Ce choix est donc laissé à l'utilisateur selon le type de propriété cherchant à être prouvée.

(a) Configuration à 3 processus (b) Configuration à $1 + N$ processus

FIGURE 4.8 – Configurations modulaires

Sélection des domaines abstraits

Les automates de treillis pouvant être paramétrés par un domaine abstrait, nous pouvons établir selon la nécessité de précision et de performance le domaine à utiliser. Dans la suite de ce chapitre, *nous emploierons le domaine des polyèdres* pour nous intéresser à la précision atteignable pour notre type d'analyse. Les parties séquentielles de nos systèmes profiteront ainsi des relations que ceux-ci apportent et nous verrons par la suite que certaines de ces relations peuvent être propagées à l'état global notamment lorsque la taille évolue dynamiquement.

Traduction vers le modèle

Avant de débuter la phase de vérification, il est également nécessaire de traduire nos programmes vers notre modèle. Nous donnons ici l'idée générale de cette traduction que nous développerons en détail dans le chapitre suivant. Comme pour notre précédent langage, nous partons du graphe de flot de contrôle et considérons toutes les transitions de celui-ci. Pour chaque transition labellisée par une instruction séquentielle (*i.e.* affectation, boucle, conditionnelle, etc.), nous générons une règle qui reconnaît un processus et en réécrit un. Par exemple, l'affectation : `x := 5` de la localité L_{10} à L_{11} sera traduit par la règle *e.g.* $L_{10} \times \top \;/\; f : (l, \rho) \mapsto \langle L_{11}, \rho[x \mapsto 5]\rangle$. Pour les instructions globales (*i.e.* communications, créations de processus), des règles seront générées selon les squelettes présentés dans la section 4.2. Pour les communications point-à-point, il est important de noter qu'il est nécessaire de générer une règle pour chaque couple du produit cartésien des instructions **send/receive**. Par la suite, nous discuterons des optimisations que nous pouvons effectuer pour réduire cet ensemble de règles.

Nous distinguons cependant la manière dont nous traitons les instructions séquentielles et les instructions globales. Les règles de réécriture générées à partir des instructions séquentielles sont disposées sous forme d'un *transducteur tournesol* $\mathcal{T}$ exactement comme présenté dans le chapitre 3. Les règles traduites des instructions globales sont considérées comme un ensemble et appliquées à chaque itération par une fonction que l'on nommera également $\mathcal{R}$.

Une dernière opération est nécessaire : la génération des *partitions de treillis* qui vont composer l'alphabet de nos automates de treillis. Pour cela, nous parcourons l'ensemble des instructions et générons pour chacune une partition contenant la localité de l'instruction ainsi qu'un environnement défini à $\top$. Nous effectuons ceci pour l'ensemble des instructions hormis les conditionnelles où nous générons, pour chaque instruction suivante, une partition où l'environnement satisfait la garde.

4.4 Calcul de l'espace d'atteignabilité

Premièrement, il est nécessaire d'adapter l'algorithme de calcul d'espace d'atteignabilité afin de maintenant considérer l'application les règles de réécriture. Nous définissons l'algorithme présenté par la Figure 4.9 comme notre nouvelle itération de point-fixe. Ce nouvel algorithme intègre l'application des différentes règles de réécriture générées à partir du programme en effectuant l'union entre les transitions séquentielles et globales.

$$\mathcal{A}_0 = I$$

$$\mathcal{A}_{n+1} = \begin{cases} \mathcal{A}_n & \text{si } \mathcal{T}(\mathcal{A}_n) \sqcup \mathcal{R}(\mathcal{A}_n) \sqsubseteq \mathcal{A}_n \\ \mathcal{A}_n \triangledown (\mathcal{A}_n \sqcup \mathcal{T}(\mathcal{A}_n) \sqcup \mathcal{R}(\mathcal{A}_n)) & \text{sinon} \end{cases}$$

$$\mathcal{T}^{\alpha}(I) = \lim_{n \to +\infty} \mathcal{A}_n$$

FIGURE 4.9 – Calcul de l'espace d'atteignabilité avec règles de réécriture

Nous nous intéressons désormais à un programme exemple (Figure 4.10) où le processus d'identifiant 0 crée dynamiquement 5 nouveaux processus qui stockent dans une variable x le double de leur identifiant. Puis, les variables `x` des différents processus sont additionnées et le résultat est stocké dans la variable `res` du processus 0 qui est finalement transmise à tous les autres processus par un `broadcast`.

Dans cet exemple, nous nous intéressons à deux propriétés :

— il y a au maximum 6 processus présents à tout instant ;

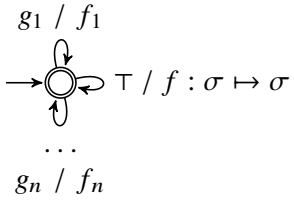
— aux états terminaux du programme, le résultat du calcul, contenu dans la variable res, est borné.

En effectuant l'analyse, nous obtenons un invariant décrit par la Figure 4.11. Pour des raisons de

```
1  main():
2    id := get_id ();
3    res := 0;
4
5    if id = 0 then
6      i := 0;
7      while i < 5 do
8        create (child_id);
9        i := i + 1
10     done;
11     x := 0
12   else
13     x := id + id
14   endif;
15
16   reduce(+, x, res, 0);
17   broadcast(res, 0);
18
19   return
```

(a) Programme source

Transitions locales



$\mathcal{R}_{\text{create}} = \top^* \cdot \langle \mathrm{L}_8 \times \top \rangle \cdot \top^* /$

$F_1 : ((l_s, \rho_s), (l_{cr}, \rho_{cr}) \mapsto \langle l_s, \rho_s[\text{size} \mapsto \rho_s(size) + 1] \rangle \cdot$

$f_1 : (l_{cr}, \rho_{cr}) \mapsto \langle \mathrm{L}_9, \rho_{cr}[\text{size} \mapsto \rho_{cr}(size) + 1] \rangle \cdot$

$F_2 : ((l_s, \rho_s), (l_{cr}, \rho_{cr}) \mapsto \langle l_s, \rho_s[\text{size} \mapsto \rho_s(size) + 1] \rangle \cdot$

$f_2 : (l_{cr}, \rho_{cr}) \mapsto \langle \mathrm{L}_2, \rho_{\text{init}}[\text{ id} \mapsto \rho_{cr}(\text{size});$
$\text{size} \mapsto \rho_{cr}(\text{size}) + 1] \rangle$

$\mathcal{R}_{\text{broadcast}} = \ldots$

$\mathcal{R}_{\text{reduce}} = \ldots$

(b) Règles traduites

FIGURE 4.10 – Création dynamique de 5 processus avec calcul numérique

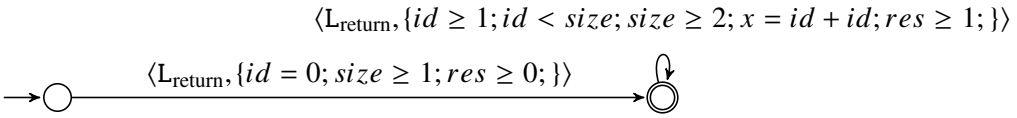


FIGURE 4.11 – États terminaux de l'invariant

lisibilité, nous ne nous intéressons qu'aux états terminaux (*i.e.* les états à la localité du `return`). Nous observons que l'invariant obtenu ne permet pas de prouver les propriétés énoncées. En effet, nous n'obtenons de bornes ni sur la taille, ni sur le résultat. Pour expliquer ce résultat, nous nous intéressons à la boucle de création dynamique des processus. Intuitivement, le point-fixe que l'on pourrait vouloir après analyse de la boucle serait celui présenté par la Figure 4.12a. Cependant, selon la sémantique de notre création dynamique, cet automate n'est pas un point-fixe. En effet, il est encore possible d'appliquer la règle de création dynamique puisque $i \in [1; 6]$ et celle-ci ajouterait un nouvel état en queue du mot en augmentant la taille de tous les processus présents sur ce mot. Une solution pour pallier ce problème serait de séparer les mots de tailles différentes comme présenté par la Figure 4.12b. Cette fois-ci, cet automate est un point-fixe correct : chaque nouveau processus génère un nouveau chemin dans l'automate et appliquer la règle reviendrait à créer un nouveau chemin. Cependant, pour obtenir un tel point-fixe, nous devons empêcher les processus ayant la même localité (*e.g.* le point d'entrée de la boucle) de se regrouper. Pour cela, il est nécessaire de diviser la partition associée à la localité en n partitions correspondant à chaque valeur possible de la variable i (où $i = size - 1$) itérée dans la boucle. Autrement dit, nous avons besoin d'une partition infinie ce qui, par propriété des automates de treillis, n'est pas possible. Finalement, le point-fixe que l'on peut obtenir est celui illustré par la Figure 4.12c où les tailles de processus ne sont pas bornées et où nous avons regroupé tous les nouveaux processus créés dynamiquement. La relation avec la taille n'est plus vraie puisqu'elle peut encore croître tandis que $i \in [0; 5]$ de par la condition de boucle.

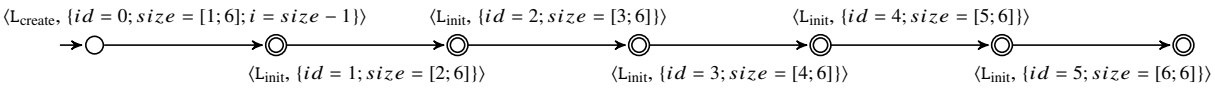
Un effet de bord à cette perte d'information sur le taille des mots est la perte de la possibilité de déterminer une borne supérieure du calcul effectué par le `reduce` puisque nous ne sommes pas capable de discerner le nombre de processus présents et, par extension, leur identifiant. Nous ne pouvons donc pas obtenir de résultats intéressants vis-à-vis de nos propriétés.

4.5 Optimisation par raffinement de partitions

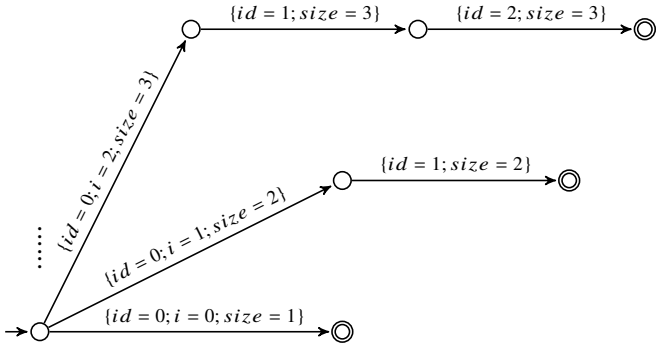
Dans cette section, nous partons de l'exemple de la Figure 4.10 dont nous avons vu que le résultat est une sur-approximation trop imprécise pour nous permettre d'établir la preuve des propriétés nous intéressant.

Nous présentons ici une optimisation majeure nous permettant d'affiner l'analyse afin d'obtenir une abstraction suffisamment précise. La première optimisation consiste à effectuer une pré-analyse nous permettant d'affiner les partitions de treillis constituant l'alphabet de nos automates afin d'être capable de projeter nos états locaux vers des sous-ensembles plus précis.

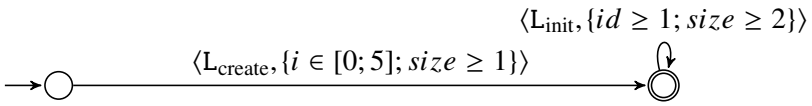
Un moyen d'affiner nos résultats est de générer des partitions d'automates de treillis plus précises qu'une simple association $\mathtt{Loc}_x \times \top$ vers lesquels nous projetons nos états de processus. Notamment, en sortie de boucle où nous ne sommes pas capables d'établir une borne supérieure. Se faisant, nos états de processus obtiendraient un gain en précision.



(a) Faux point-fixe



(b) Point-fixe avec partition infinie



(c) Point-fixe calculable

FIGURE 4.12 – Point-fixe des créations dynamiques

Afin de restreindre ces partitions, nous effectuons sur le programme une pré-analyse. Le programme est traité comme *séquentiel* où la sémantique des instructions globales est simplifiée : les valeurs reçues par communications sont abstraites. Nous perdons alors toute information sur ces variables. Cette restriction entraîne, de fait, des pertes de précision notamment dans le cas où le flot du programme dépend de la valeur de ces variables. Nous définissons alors comme partition d'automate de treillis l'ensemble des états obtenus par cette analyse.

Pour illustrer ceci, nous donnons dans la Figure 4.13 les partitions dont nous disposions auparavant et les partitions résultant de cette *pré-analyse* du programme. Il est à noter que nous utilisons des heuristiques pour tenter d'améliorer la précision telles que la technique d'élargissement avec « seuil » [1] [CC77, Hal93] d'où la borne exacte $\{i = 6\}$ calculée pour la localité [`x := 0`] correspondant à la sortie de boucle.

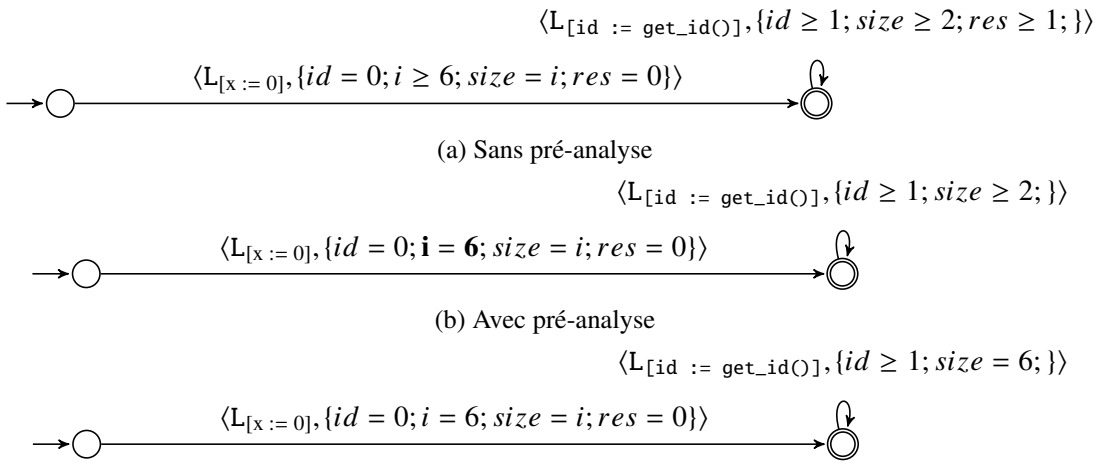
Si nous reprenons notre exemple en utilisant la nouvelle partition d'automate de treillis, nous constatons un invariant plus précis notamment en sortie de boucle comme illustré par la Figure 4.14. Nous notons qu'après cette pré-analyse, nous sommes désormais à même de trouver une borne exacte à la taille des processus pour le processus $id = 0$. Toutefois, les processus dynamiquement créés ne disposent pas de cette borne. Nous pouvons cependant réaliser l'intersection de toutes les tailles d'un mot afin de mettre à jour celle-ci de manière sûre : les mots de tailles différentes de 6 ne sont pas possibles puisqu'ils ne disposeraient pas de préfixe disposant d'un état initial.

Cet invariant $size = 6$ se propage tout au long de l'analyse puisqu'aucune autre instruction ne peut modifier la taille. Nous sommes alors capables de prouver la première propriété : à tout instant, au maximum 6 processus sont présents. Cela est vrai, même à l'intérieur de la boucle puisque i est en

1. Plutôt que de projeter les valeurs abstraites directement vers $+\infty$, l'opérateur d'élargissement commence par essayer de projeter vers des valeurs significatives telles que la post-condition de la borne de boucle

Localité	Env. sans pré-analyse	Env. avec pré-analyse
[id := get_id ()]	$\top$	$\top$
[res := 0]	$\top$	$\top$
[if id = 0]	$\top$	$\{res = 0\}$
[i := 0]	$\{id = 0\}$	$\{res = 0; id = 0\}$
[while i < 5 do]	$\top$	$\{res = 0; id = 0; i \in [0; 6]\}$
[create(child_id)]	$\{i \in [0; 5]\}$	$\{res = 0; id = 0; i \in [0; 5]\}$
[i := i + 1]	$\top$	$\{res = 0; id = 0; i \in [0; 5]\}$
[x := 0]	$\{i \geq 6\}$	$\{res = 0; id = 0; i = 6\}$
[x := id + id]	$\{id \geq 1\}$	$\{res = 0; id \geq 1\}$
[reduce(...)]	$\top$	$\{res = 0\}$
[broadcast(...)]	$\top$	$\top$
[return]	$\top$	$\top$

FIGURE 4.13 – Partitions sans et avec pré-analyse



(a) Sans pré-analyse

(b) Avec pré-analyse

(c) Avec pré-analyse et intersection des tailles

FIGURE 4.14 – Invariants après analyse de la boucle

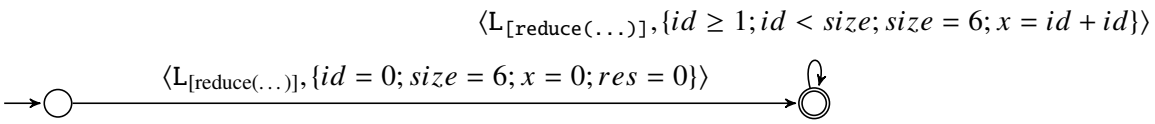


FIGURE 4.15 – État avant analyse du `reduce`

relation avec size et $i \in [0; 5]$.

La propagation de l'invariant de taille entraîne une conséquence supplémentaire : à l'entrée du `reduce`, les processus connaissant leur taille possèdent l'état suivant : $\{x = id + id; id \in [1; size - 1]; size = 6\}$ ce qui nous donne $x \in [2; 10]$. Nous obtenons alors l'état décrit par la Figure 4.15.

Un problème subsiste : la sémantique de notre opération de réduction n'est pas capable de déterminer avec le nombre de processus ayant transmis leur valeur au collecteur. Ce qui, en l'état, entraîne l'incapacité à borner le résultat de la somme. Pour pallier ce problème, nous introduisons au collecteur une variable fantôme k incrémentée à chaque communication avec un processus. Celle-ci va potentiellement détecter une relation entre les résultats que nous effectuons et cette variable « compteur ». Dans notre exemple, nous pouvons déduire une relation linéaire entre le résultat et le compteur explicité par

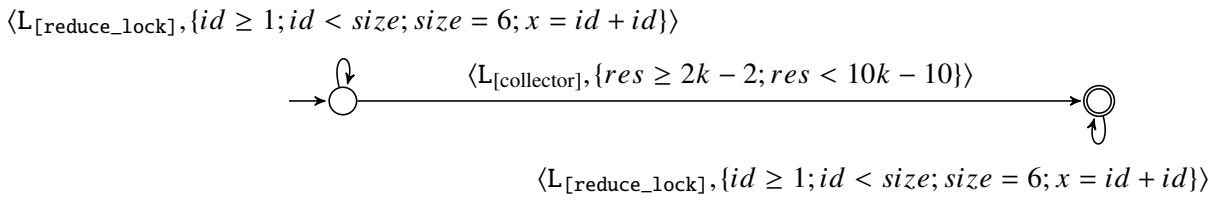


FIGURE 4.16 – Invariant après analyse de la *collecte* du `reduce`

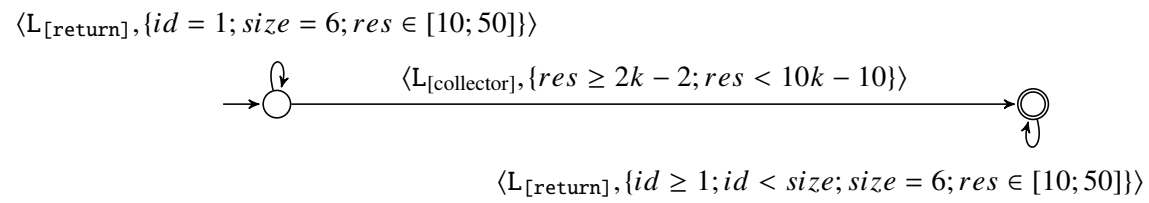


FIGURE 4.17 – Invariant des états terminaux avec optimisation

la Figure 4.16.

Nous savons que cette variable fantôme k représente le nombre de communications effectuées et que, sémantiquement, ce nombre représente la taille des processus. Ainsi, nous pouvons instancier cette variable par la taille du mot et obtenir le résultat suivant pour le processus $id = 0 : \{id = 0; size = 6; res \in [10; 50]\}$ qui sera ensuite transmis à tous les processus par un `broadcast` qui vont finalement se terminer. Nous obtenons alors une borne supérieure sur le résultat du calcul. *N.B.* la valeur exacte $res = 30$ n'est pas obtenable puisque l'abstraction générée par l'application de l'opérateur d'élargissement unifiant les processus créés dynamiquement n'est pas suffisamment précise.

Grâce à cette optimisation, nous sommes désormais capables d'obtenir un invariant des états terminaux (Figure 4.17) suffisamment précis pour nous permettre de conclure sur les propriétés que nous avons énoncées.

4.6 Conclusion

Dans ce chapitre, nous avons vu que l'introduction de règles de réécriture autorisant (à l'aide de gardes étoilées) la reconnaissance et la réécriture de séquences arbitrairement longues permet de modéliser les communications point-à-point et les communications collectives entre processus, et lève donc la restriction de « communications restreintes au voisinage » que nous avions dans le modèle du chapitre 3.

Nous avons ensuite donné un algorithme d'application de ces règles à un ensemble d'états représentés par un automate de treillis. Nous pouvons calculer l'ensemble d'atteignabilité d'un système communiquant, et par conséquent vérifier des propriétés de sûreté.

Nous avons maintenant tous les outils pour modéliser et vérifier des programmes utilisant des primitives de communication standards comme celles que l'on trouve dans la bibliothèque `MPI`, ce que nous allons faire dans les chapitres 5 et 6.

Chapitre 5

Application à MPI

Message Passing Interface (MPI) [For15] est un standard pour architectures parallèles dédié au calcul haute performance (*High Performance Computing* ou HPC). Cette spécification propose un modèle de processus distribués communicant par passage de messages. MPI emploie le paradigme SPMD (*Single Program Multiple Data*) : chaque processus partage le même code source en possédant sa propre zone mémoire. MPI se veut portable : les machines déployées pour effectuer un calcul parallèle peuvent avoir des architectures matérielles hétérogènes.

Des implantations de ce standard, pour la plupart *open-source* (MPICH [GLS99], OpenMPI [GFB+04], ...), fournissent des interfaces de programmation pour différents langages parmi lesquels on retrouve le plus souvent C, C++ et Fortran. À l'heure actuelle, MPI est considéré comme une référence dans le domaine du calcul distribué et jouit d'un succès industriel.

De nombreux travaux portant sur la vérification et l'analyse de programmes MPI existent mais relativement peu se sont intéressés à la vérification formelle de tels programmes [VVD+09, GKS+11, SZ11, SCB13]. Un des outils les plus aboutis appartenant à cette catégorie est ISP[1] [VVD+09] (*In-Situ Partial-order*). ISP effectue une analyse *dynamique* d'un programme MPI en utilisant des techniques de *model checking* afin de détecter de potentiels interblocages dans un programme MPI. Cependant, à notre connaissance, il n'existe pas de propositions s'attaquant à une analyse de valeurs réalisée de manière statique. Ainsi, dans le but de valider notre approche et de répondre à ce manque, nous nous sommes intéressés au développement d'un analyseur permettant de vérifier formellement ce type de programmes. L'aspect SPMD et le paradigme de passage de messages de MPI en font un bon candidat pour appliquer notre méthodologie et pour fournir une preuve de concept de notre modèle.

Nous proposons donc, dans ce chapitre, un outil prototype d'analyse statique avec pour cible l'analyse des programmes écrits dans un sous-ensemble de MPI/C . Notre prototype prend la forme d'un greffon de *Frama-C* [KKP+15], plate-forme d'analyse statique de code C. Nous commençons par présenter, succinctement, les spécificités de MPI puis nous décrirons le sous-ensemble des primitives que notre prototype est, à ce jour, capable de traiter et expliciterons la traduction automatique de celles-ci vers notre modèle. Pour finir, nous présenterons, dans les grandes lignes, l'implantation et quelques optimisations réalisées permettant d'accroître l'efficacité de notre analyse.

1. http://formalverification.cs.utah.edu/ISP-release/

5.1 Introduction à MPI

5.1.1 Notions générales

MPI est un standard de programmation d'applications distribuées. La programmation en MPI passe par des appels explicites à des primitives de communication et de synchronisation. Ces communications s'effectuent par passage de messages. Les primitives de communication présentes peuvent être synchrones ou asynchrones. Une communication synchrone entre deux processus ne peut s'effectuer que si les deux processus sont respectivement prêts à émettre et à recevoir. Lorsque l'un des deux n'est pas prêt, l'autre doit effectuer une attente active jusqu'à la synchronisation nécessaire à l'envoi de message. Dans une communication asynchrone, cette attente n'est pas plus nécessaire et le processus arrivé en premier peut continuer son fil d'exécution. Toutefois, si la réception est asynchrone, il est nécessaire d'effectuer une synchronisation explicite avant utilisation de la valeur reçue. Les messages envoyés de manière asynchrone sont disposés dans une file de type *First In First Out* garantissant l'ordre d'arrivée des messages.

Comme évoqué ci-dessus, chaque processus possède sa propre zone mémoire mais exécute le même code source (modèle SPMD). Chaque processus possède un certain ensemble de valeurs privées auquel il peut accéder à tout moment de l'exécution. Parmi celles-ci, on peut y trouver son *rang* (*i.e.* l'identifiant du processus représenté par un entier), le nombre de processus en exécution ou encore la description de son architecture matérielle. La plupart des programmes MPI organisent leurs calculs en utilisant un processus « maître » agissant en tant qu'orchestrateur répartissant les donnéees entre les différents processus et recevant les différents résultats. Par convention, le processus utilisé pour cela est celui d'identifiant 0.

La norme MPI recommande de débuter les programmes par la primitive `MPI_Init()` et de le conclure par `MPI_Finalize()` avant d'effectuer le `return` terminant la procédure principale (*e.g.* Figure 5.1 [2]). Le prélude (`L.6` à `L.14`) permet d'allouer et d'initialiser des variables d'environnement et la conclusion ((`L.21`) de les désallouer. Le comportement de toute autre instruction en dehors de ce bloc est indéfini.

En MPI, les communications étant explicites, il est nécessaire de spécifier à quel processus l'on souhaite s'adresser. Chaque primitive de communication possède un argument permettant de spécifier le récepteur ou le destinataire. Certaines primitives acceptent également l'étiquette `MPI_ANY_SRC` en place de l'identifiant d'émetteur pour signifier qu'elles peuvent recevoir d'une source non spécifiée.

Dans l'exemple présenté ci-dessus, l'exécution du programme est effectuée sur une seule machine. Afin d'exécuter un programme MPI sur différentes machines physiques, il est au préalable nécessaire de compiler, en utilisant la même implantation de MPI, les programmes pour chaque architecture que l'on souhaite employer. La manière la plus courante de démarrer les programmes sur les différentes machines est de passer par un script de lancement accédant aux machines via SSH (*Secure SHell*) en établissant les droits adéquats.

Parmi les différentes manières d'optimiser les performances de son application distribuée, il existe le paradigme de *programmation hybride*. En plus de distribuer le calcul sur un ensemble de machines, chacune d'entre-elles peut paralléliser localement son calcul à l'aide des différents cœurs de son processeur, de son GPU (*Graphics Processing Unit*) ou tout autre matériel. L'approche la plus couramment utilisée dans le domaine est de combiner l'exécution de MPI à l'outil OpenMP [3] offrant des directives de compilation principalement destinées à paralléliser les boucles de calcul.

2. http://mpitutorial.com/tutorials/mpi-hello-world/
3. http://www.openmp.org/

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    // Initialisation de l'environnement MPI
    MPI_Init(&argc, &argv);

    // Recupere le nombre de processus
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Recupere l'identifiant (ou rang) du processus
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    printf("Bonjour monde depuis le processus de rang %d\
            sur %d processeurs\n",
            world_rank, world_size);

    // Finalisation de l'environnement MPI.
    MPI_Finalize();
    return 0;
}
```

```
> mpicc hello_world.c -o hello_world
> mpirun -n 5 hello_world
Bonjour monde depuis le processus de rang 2 sur 5 processeurs
Bonjour monde depuis le processus de rang 0 sur 5 processeurs
Bonjour monde depuis le processus de rang 4 sur 5 processeurs
Bonjour monde depuis le processus de rang 1 sur 5 processeurs
Bonjour monde depuis le processus de rang 3 sur 5 processeurs
```

FIGURE 5.1 – Exemple de programme MPI/C – `hello_world.c`

5.1.2 Sous-ensemble des primitives et du langage

À cette date, le standard MPI, présentement à sa 3ème version majeure [4], distingue environ 300 primitives. Parmi celles-ci, on retrouve certaines permettant de factoriser la syntaxe (*e.g.* `MPI_ReduceAll(...)` est équivalent à `MPI_Reduce(...)` ; `MPI_Bcast(...)`), d'autres de déclarer des types de données portables, ou encore de gérer des entrées/sorties, ...

Dans le cadre de notre prototype, nous avons ciblé un sous-ensemble de MPI et du langage C qui nous a semblé, dans un premier temps, suffisamment expressif pour décrire des programmes intéressants. Ainsi, les primitives traitées par notre prototype ressemblent fortement aux primitives présentes dans le langage donné par la Figure 4.1 du chapitre précédent. Dans un soucis d'établir une preuve de concept, nous nous sommes limités à des domaines abstraits relativement simples. De ce fait, nous simplifions le langage C en, notamment, n'autorisant pas d'allocation dynamique, d'appels de fonctions et de types de données non-scalaires. Nous reviendrons sur ces choix par la suite en proposant des pistes permettant de lever ces contraintes.

Le standard MPI décrit de manière détaillée le comportement attendu des différentes primitives mais n'établit pas de sémantique formelle. Les programmes MPI peuvent alors avoir des comportements indéfinis selon les implantations de la bibliothèque employées. Certaines implantations de MPI, lors d'un envoi de message synchrone, s'autorisent de ne pas attendre la confirmation de réception d'un message

4. http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

pour reprendre leur exécution. Les travaux présentés par Guodong Li et al. [LDGK08] proposent la formalisation d'un sous-ensemble des primitives permettant de désambiguïser celles-ci.

Nous présentons ci-dessous les différentes primitives que notre prototype, en l'état, traite. La sémantique que nous associons à chacune d'entre elles est celle que nous décrivons dans le chapitre précédent.

Communications point-à-point

Les deux premières primitives de communication que nous traitons sont :

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

La sémantique de ces deux primitives est similaire à celle des instructions `send` et `receive` présentées dans le chapitre précédent. Ces communications sont synchrones et on les suppose bloquantes tant que la valeur n'est pas reçue par le processus récepteur.

Toutes les primitives MPI retournent un entier correspondant à un code d'erreur, 0 si tout s'est déroulé correctement.

- `buf` est l'adresse pointant vers la donnée à envoyer ;
- `count` est le nombre d'éléments que l'on souhaite envoyer ;
- `datatype` est le type des éléments traités ;
- `dest` et `source` sont les identifiants respectifs du destinataire et du récepteur. Il est possible de donner MPI_ANY_SRC en identifiant de réception pour recevoir de n'importe quel émetteur ;
- `tag` est un marqueur spécifiant le type d'un message. Pour effectuer une communication, les deux instructions doivent porter le même marqueur ;
- `comm` est un contexte de communication (par défaut : `MPI_COMM_WORLD`) ;
- `status` est la structure qui sera remplie lorsque que la communication sera effectuée et qui contiendra les informations liées à la communication.

Communications collectives

Les deux communications collectives que nous traitons sont, comme dans notre langage défini dans le chapitre précédent, la communication globale et la réduction globale.

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root,
              int tag, MPI_Comm comm)
```

La sémantique est, ici encore, semblable à celle précédemment donnée. Les arguments sont les mêmes que pour la communication point-à-point outre que :

- `root` est l'identifiant du processus envoyant sa valeur à tous les autres ;
- `buf` est l'adresse contenant la valeur que le processus d'id `root` va envoyer mais également l'adresse où sera stockée la valeur reçue.

L'opération de réduction se présente ainsi :

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

En MPI, l'opération de réduction ne respecte pas d'ordre dans les réductions effectuées. Ainsi, MPI suppose l'opérateur associatif et ne contredit donc pas notre sémantique attribuant, elle, un ordre aux réductions.

- `root` est l'identifiant du processus qui reçoit la valeur réduite ;
- `sendbuf` est l'adresse contenant la valeur que tous les processus vont envoyer pour effectuer la réduction ;
- `recvbuf` est l'adresse à laquelle sera stockée le résultat du calcul. Cette valeur ne sera accessible que par le processus d'identifiant `root`.

Création dynamique de processus

En MPI, il existe une primitive permettant de créer dynamiquement des processus. Cependant, le comportement de cette primitive « haut-niveau » est différent de la sémantique que nous avons donnée : cette primitive crée des processus fils *associés* à l'appelant. Par la suite, ces processus fils ne seront capables de ne communiquer qu'avec leur parent.

```
MPI_Comm_spawn(char *command, char *argv[], int maxprocs,
               MPI_info info, int root, MPI_Comm comm,
               MPI_Comm *intercomm, int array_of_errcodes[])
```

- `command` est la chaîne correspondant au fichier exécutable que les processus-fils vont lancer ;
- le tableau `argv` contient les arguments avec lequel les processus-fils vont démarrer leurs programmes ;
- `maxprocs` est le nombre de processus que le processus parent va tenter de créer (selon le nombre de machines disponibles)
- `info` est un ensemble de paires clé/valeur (string/string) permettant de *configurer* l'exécution des processus-fils ;
- `root` est l'identifiant du processus dont le contexte va être utilisé pour évaluer les précédents arguments ;
- `comm` communicateur interne contenant le groupe des processus-fils ;
- `intercomm`, après l'appel, contiendra le communicateur externe qui sera associé à tous les processus et dans lequel ils seront contraints de communiquer ;
- `array_of_errcodes` contiendra l'ensemble des codes d'erreurs retournés par les processus.

Pour manipuler cette primitive, il est nécessaire de comprendre le concept des *communicators*. Les communicateurs sont des partitionnements d'espace de communications. Ils permettent de définir une topologie des différents processus. En utilisant les communicateurs, il est donc possible d'effectuer une communication multi-point avec un sous-ensemble de processus en instanciant un nouveau communicateur et en remplaçant la constante `MPI_COMM_WORLD` vue jusqu'à présent.

Cette primitive est complexe à utiliser et ne s'applique qu'à certains cas particuliers. Par ailleurs, le standard MPI déconseille l'utilisation de la création dynamique en MPI pour des raisons de performances.

Pour ces raisons, nous avons fait le choix de nous abstraire de ces notions haut-niveau et de définir une nouvelle primitive de création dynamique afin de pouvoir valider nos résultats théoriques :

```
void MPI_Create(int *new_id)
```

où `new_id` contiendra l'identifiant du nouveau processus créé. La sémantique de cette primitive est équivalente à celle donnée au cours du chapitre précédent. Bien que n'appartenant pas à l'ensemble des primitives officielles, nous avons fait le choix d'ajouter le préfixe "MPI_" pour la distinguer des primitives *C*.

5.2 Implantation du prototype

Nous présentons maintenant l'implantation, nommée `MPAI` (Message Passing Abstract Interpreter), que nous avons réalisée de l'analyseur de code `MPI/C`. Notre outil prend en entrée un programme `MPI`, un nombre de processus et calcule une sur-approximation de l'espace d'atteignabilité du programme sous forme d'automate de treillis. Cette étape est complètement automatique à partir d'un programme C écrit dans le sous-ensemble présenté.

Notre prototype est implanté en OCaml comptant, à l'heure actuelle, environ 10.000 lignes de code et se présente sous la forme d'un greffon (*plug-in*) Frama-C. Frama-C est une plate-forme en licence libre (GPL) d'analyse de code C principalement utilisée à des fins de vérification de programmes séquentiels. Frama-C utilise un système de *plug-in* afin de spécialiser ses analyses. On trouve, par exemple, le plug-in `EVA` (Evolved Value Analysis, anciennement `value`) effectuant une analyse statique de flot de données par interprétation abstraite de programmes séquentiels, `mthread` (non-libre) qui, comme AstréeA (non-libre), est une analyse de valeurs de threads POSIX ou encore `WP` qui implante une vérification déductive par un calcul de la plus faible pré-condition permettant de générer à partir d'annotations de code des obligations de preuves pouvant être transmises à des assistants de preuves (`Coq`, `Alt-Ergo`, ...). En regroupant tous ces *plug-ins* au sein d'un seul et même outil, il est aisé, lorsque l'on analyse un programme d'appliquer plusieurs types d'analyses dont certaines sont capables de coopération raffinant ainsi leurs résultats.

L'architecture de notre plug-in se décompose en trois parties principales :

— La phase de traduction permettant, à partir du graphe de flot de contrôle calculé par Frama-C, de générer les règles de réécriture selon nos squelettes de traduction ;

— L'implantation des domaines abstraits destinés à paramétrer la bibliothèque d'automate de treillis ;

— L'analyse nous permettant d'obtenir notre espace d'atteignabilité implantant nos stratégies d'itérations de point fixe.

Nous décrivons graphiquement, par la Figure 5.2, le fonctionnement de notre outil. À partir du programme source, dont Frama-C a réalisé une analyse syntaxique, nous extrayons les localités du programme afin de générer un treillis (plat) de celles-ci. De plus, ces localités seront raffinées à l'aide d'une pré-analyse comme nous l'avons détaillé au chapitre précédent (*c.f.* Section 4.5). Nous paramétrons nos automates de treillis par le produit d'un domaine des localités et d'un domaine numérique (par défaut, les polyèdres). Parallèlement, nous effectuons la traduction de la sémantique du programme source vers notre modèle (transducteur et règles) qui sera paramétré par les automates de treillis précédemment créés. Finalement, le calcul du point fixe s'effectue en appliquant les diverses stratégies d'itération et optimisations. Le résultat de ce calcul donne l'espace d'atteignabilité que nous affichons à l'utilisateur.

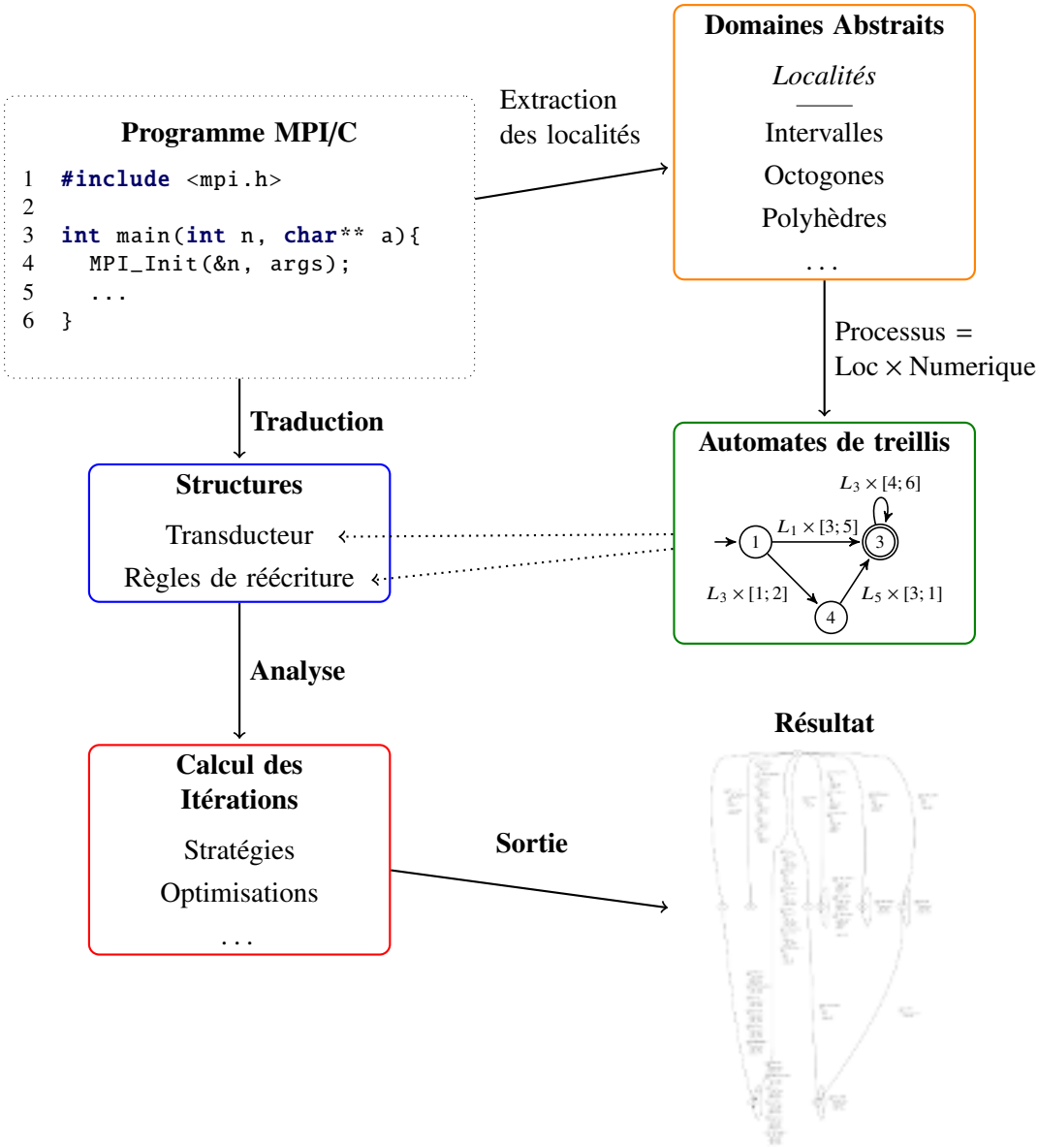


FIGURE 5.2 – Architecture globale de MPAI

Notre analyse est entièrement automatisée et accepte en entrée un programme MPI/C satisfaisant notre sous-ensemble traité. Notre prototype fournit en résultat l'automate de treillis correspondant à la sur-approximation de l'espace d'états du programme considéré et cet automate peut être affiché au format `DOT`. En l'état, le prototype peut être testé [5]. L'accès au code-source est, pour le moment, privé pour cause de licence propriétaire. Toutefois, la distribution du code en licence libre est prévue. L'outil est présenté au sein d'une machine virtuelle *Virtualbox*.

5.2.1 Traduction automatique des primitives MPI/C

Nous présentons en détails notre approche permettant de traduire la sémantique des différentes instructions C en des fonctions de transfert applicables à notre modèle présenté lors des chapitres précédents.

5. `https://www-apr.lip6.fr/~botbol/mpai/`

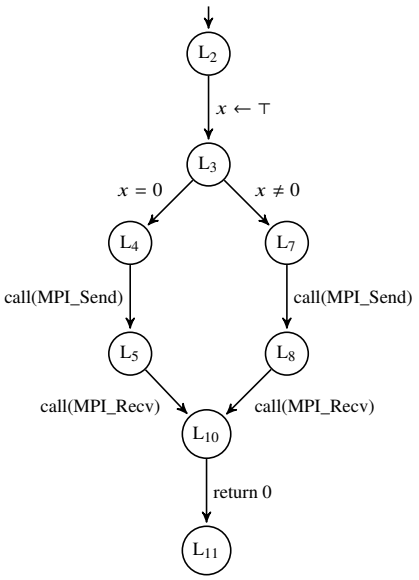
À partir du graphe de flot de contrôle obtenu par l'intermédiaire de Frama-C, nous générons, pour les parties séquentielles, des équivalents aux fonctions de transfert classiques et pour les parties concurrentes des règles de réécriture selon nos squelettes de traductions.

Le graphe de flot de contrôle (CFG) est obtenu grâce à Frama-C qui intègre une analyse syntaxique et grammaticale de code C transformant le code vers une représentation haut-niveau inspirée de CIL [6] (*C Intermediate Langage*). Celle-ci prend la forme d'un arbre de syntaxe abstraite (AST) duquel Frama-C est capable d'extraire un CFG (*e.g.* Figure 5.3). Notre prototype récupère alors ce CFG et génère pour chaque transition présente la fonction de transfert associée selon les squelettes de traduction que nous avons définies.

Comme stipulé dans les sections précédentes, la traduction des instructions séquentielles génère des fonctions de transfert classiques. Nous représentons celles-ci par un transducteur (Figure 5.3c). Pour des raisons de lisibilité, les transitions du transducteur sont regroupées. Cependant chaque fonction de transfert dispose d'une transition propre. Il est à noter qu'une transition supplémentaire $\top \,/\, f : x \mapsto x$ est présente ce qui nous permet de modéliser le non-déterminisme.

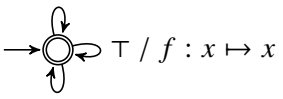
```
1  main(){
2    int x = random();
3    if (id == 0){
4      MPI_Send(&x,1);
5      MPI_Recv(&x,1);
6    } else {
7      MPI_Send(&x,0);
8      MPI_Recv(&x,0);
9    }
10   return 0;
11 }
```

(a) Code source



(b) Graphe de Flot de Contrôle

$$\mathrm{L}_2 \times \top \,/\, f : (l,\rho) \mapsto \mathrm{L}_3, \rho[x \leftarrow \top]$$
$$\mathrm{L}_{10} \times \top \,/\, f : (l,\rho) \mapsto \mathrm{L}_{11}, \rho$$



$$\mathrm{L}_3 \times \{id \in [0,0]\} \,/\, f : (l,\rho) \mapsto \mathrm{L}_4, \rho$$
$$\mathrm{L}_3 \times \{id \in [1,+\infty]\} \,/\, f : (l,\rho) \mapsto \mathrm{L}_7, \rho$$

(c) Traduction des instructions séquentielles

FIGURE 5.3 – Traduction d'un programme MPI/C

Les squelettes d'instructions de communication nécessitent des traitements supplémentaires. Nous présentons maintenant les détails techniques nécessaires à celles-ci.

6. https ://github.com/cil-project/cil

Traductions des communications point-à-point

Avant de pouvoir générer la règle de réécriture, il est nécessaire de considérer toutes les paires `send`/`recv` possibles. En effet, lors de la traduction, nous ne pouvons pas faire de suppositions générales sur quelles communications vont être effectuées. Ainsi, la méthode naïve consiste à calculer le produit cartésien de chaque communication point-à-point afin de considérer toutes les permutations possibles. Puis, pour chaque paire générer deux règles de réécriture encodant cette communication : nous devons générer une première règle si le processus émetteur précède le processus récepteur dans le mot de processus et une règle symétrique dans le cas où le récepteur précède. Nous obtenons un ensemble large de règles pour les programmes effectuant un grand nombre de communications « distinctes ». Pour répondre à ce problème, nous effectuons une optimisation lorsque ce nombre de règles est très grand : avant d'effectuer une analyse précise et coûteuse, nous procédons à une première analyse imprécise mais rapide à l'aide d'un domaine numérique très simple (intervalles) et des abstractions fortes. Se faisant, nous détectons les paires qui ne sont jamais « activées » et pouvons ainsi les filtrer du résultat du produit cartésien.

Il survient un deuxième problème : dans le cas où les identifiants de l'émetteur et/ou du récepteur ne sont pas constants, il devient nécessaire d'évaluer l'expression lors de l'analyse. Par exemple, pour ce type de construction :

```
1  if (id == 0){
2    ...
3    for (i = 0; i < size; i++){
4      MPI_Send(&x,i);
5      ...
6    }
7    ...
8  } else {
9    ...
10   MPI_recv(&y, 0);
11   ...
12 }
```

Il nous est impossible d'évaluer l'identifiant du récepteur lors du test de la garde générée :

$$\top^* \cdot \langle \mathrm{L}_{send} \times [id = 0] \rangle \cdot \top^* \cdot \langle \mathrm{L}_{recv} \times [id = eval(i, \rho_{send})] \rangle \cdot \top^* \ / \ \ldots$$

En effet, il est nécessaire de connaître l'état de l'émetteur. Pour pallier ce problème, nous effectuons le test au sein des fonctions de réécriture, là où le contexte de chaque communicateur sera connu. Ainsi, nos deux règles de réécriture sont modifiées de la façon suivante :

$$\top^* \cdot \langle \mathrm{L}_{send} \times \top \rangle \cdot \top^* \cdot \langle \mathrm{L}_{recv} \times \top \rangle \cdot \top^* \ / \ F \ \cdot \ f_{send} \ \cdot \ F \ \cdot \ f_{recv} \cdot \ F$$
$$\top^* \cdot \langle \mathrm{L}_{recv} \times \top \rangle \cdot \top^* \cdot \langle \mathrm{L}_{send} \times \top \rangle \cdot \top^* \ / \ F \ \cdot \ f_{recv} \ \cdot \ F \ \cdot \ f_{send} \cdot \ F$$

où

$$F : (\sigma_{self}, \_) \mapsto \sigma_{self}$$

$$\begin{aligned} f_{send} : \langle (l_{send}, \rho_{send}), (l_{recv}, \rho_{recv}) \rangle \mapsto \\ \text{si } \rho_{send}(id) \sqcap eval(0, \rho_{recv}) = \bot \\ \text{alors } \bot \\ \text{sinon } \langle \mathrm{L}_{send'}, \rho_{send} \rangle \end{aligned}$$

$$\begin{aligned} f_{recv} : \langle (l_{send}, \rho_{send}), (l_{recv}, \rho_{recv}) \rangle \mapsto \\ \text{si } \rho_{recv}(id) \sqcap eval(i, \rho_{send}) = \bot \\ \text{alors } \bot \\ \text{sinon } \langle \mathrm{L}_{recv}, \rho_{recv}[y \mapsto eval(x, \rho_{send})] \rangle \end{aligned}$$

Il est à noter que cette transformation reste équivalente à la sémantique définie lors du chapitre précédent. Finalement, pour notre exemple présenté par la Figure 5.3, les règles de communications générées, avant optimisation et sans les symétriques, sont :

$$\begin{aligned} \top^* \cdot \langle \mathrm{L}_4 \times \top \rangle \cdot \top^* \cdot \langle \mathrm{L}_5 \times \top \rangle \cdot \top^* \,/\, F \cdot f_{send} \cdot F \cdot f_{recv} \cdot F \\ \top^* \cdot \langle \mathrm{L}_4 \times \top \rangle \cdot \top^* \cdot \langle \mathrm{L}_8 \times \top \rangle \cdot \top^* \,/\, F \cdot f_{send} \cdot F \cdot f_{recv} \cdot F \\ \top^* \cdot \langle \mathrm{L}_7 \times \top \rangle \cdot \top^* \cdot \langle \mathrm{L}_5 \times \top \rangle \cdot \top^* \,/\, F \cdot f_{send} \cdot F \cdot f_{recv} \cdot F \\ \top^* \cdot \langle \mathrm{L}_7 \times \top \rangle \cdot \top^* \cdot \langle \mathrm{L}_8 \times \top \rangle \cdot \top^* \,/\, F \cdot f_{send} \cdot F \cdot f_{recv} \cdot F \end{aligned}$$

Après optimisation, toutes les transitions ont été ôtées puisqu'aucune n'a pu être empruntée lors de l'analyse naïve : les deux processus sont bloqués par leur `send` respectif.

Pour les deux communications collectives (`bcast` et `reduce`), aucun traitement supplémentaire n'a eu à être considéré hormis le même mécanisme d'évaluation des identifiants qu'auparavant.

Au terme de cette étape, notre traduction construit un système ayant le type suivant :

```
1 type system =
2   { transducer          : Process.t Transducer.t ;
3     rules               : Process.t Rules.rule Map.t ;
4     initial_automaton   : Process.t Automaton.t ;
5     partition           : Process.t Map.t }
```

Le transducteur est utilisé pour l'analyse des instructions locales ; *i.e.* les instructions n'ayant d'effets que sur un seul processus. Les règles (*rules*) correspondent à la traduction des instructions globales comme les primitives de communication. L'automate initial est déduit selon les paramètres passés à l'analyseur correspondant au nombre de processus que l'on souhaite modéliser : fixe ou infini. L'état initial de chaque processus comprend notamment un identifiant unique et la taille globale du système. Finalement, la partition, avec laquelle on paramètre les automates de treillis est conservée jusqu'à la phase d'analyse et servira à paramétrer certains opérateurs d'élargissements des automates de treillis. Chaque structure employée est polymorphe et est paramétrée par le type du treillis des processus `Process.t`.

5.2.2 Domaines abstraits

Comme évoqué précédemment, nos états abstraits de processus sont encodés par un treillis résultant du produit d'un treillis de localités et d'un treillis numérique. Le treillis des localités est un treillis plat (*i.e.* un treillis atomique) engendré par les différentes localités présentes du programme. Les éléments composant ce treillis sont encodés par les nœuds de l'arbre de syntaxe abstraite CIL représentant les instructions et dont chacun contient un identifiant unique.

Notre prototype utilise la bibliothèque de domaines abstraits numériques `Apron` [7]. Cette bibliothèque est écrite en `C` et propose une interface vers `OCaml`. `Apron` dispose de 4 domaines : les intervalles, les octogones et les polyèdres. Cette bibliothèque est implanté en `C` et il existe une interface `OCaml`. Nous pouvons paramétrer nos automates de treillis selon la précision désirée pour effectuer nos analyses. Cependant, les valeurs abstraites que manipulent les domaines présents dans `Apron` sont limités aux scalaires.

En MPI, et plus généralement en calcul distribué, les données manipulées sont souvent des vecteurs ou ensembles de valeurs contenus dans des *buffers* encodés sous la forme de tableaux `C`. De ce fait, il est intéressant pour notre analyseur d'être capable d'abstraire ces structures. Dans le domaine de l'interprétation abstraite, il existe de nombreux travaux proposant des abstractions les représentant. Par exemple, Gopan et al. [GDD+04] proposent une extension de la méthode consistant à représenter l'ensemble des valeurs d'un tableau comme une unique valeur abstraite (*smashing*) en y ajoutant la possibilité de conserver des relations ; certaines analyses proposent de partitionner les tableaux dynamiquement lors de l'analyse pour permettre d'établir des propriétés non-uniformes [CCL11, GRS05, HP08].

Au sein de Frama-C, le plug-in EVA dispose également de domaines abstraits intéressants. Historiquement, le plug-in `value` (ancêtre d'EVA) ne disposait que de domaines non-relationnels (*i.e.* intervalles et congruences) mais disposant d'une abstraction de la mémoire efficace permettant, par exemple, de détecter des accès invalides. Grâce aux travaux de David Bühler [BBY17], le plug-in EVA incorpore désormais, en plus des domaines historiques de `value`, des domaines relationnels permettant d'accroître la précision de l'analyse et apporte la possibilité d'ajouter de manière modulaire de nouveaux domaines. Les domaines ajoutés reposent principalement sur la bibliothèque `Apron` et, à l'aide d'un mécanisme de coopération entre domaines permettent d'accroître la précision des analyses.

Pour améliorer la précision des domaines numériques d'Apron, il nous a semblé intéressant de réutiliser les domaines d'EVA. En particulier, le domaine mémoire abstraite [KKP+15] permet de traiter tableaux et pointeurs. Cependant, le plug-in EVA est construit de telle manière qu'il n'est pas directement possible de manipuler les domaines abstraits implantés et optimisés pour des analyses séquentielles. Cependant, grâce à l'aide de David Bühler, un des développeurs principaux d'EVA, il nous a été possible d'accéder aux domaines de manière ad hoc. Ainsi, nous disposons à l'heure actuelle d'un interfaçage rudimentaire capable de supporter les tableaux C. Cet interfaçage est donc le résultat d'une première tentative de « branchement » vers EVA et n'a pas vocation à être étendu. Dans le futur, nous envisageons de nous intégrer à EVA en implantant notre modèle directement dans son architecture.

5.2.3 Analyse et stratégie d'itération

Lors de nos expérimentations, nous avons constaté que l'algorithme d'itération de point fixe décrit dans le chapitre 4, qui consiste en l'application des règles et du transducteur à l'espace d'états à chaque

7. `http://apron.cri.ensmp.fr/`

itération, était, pour les programmes effectuant un grand nombre de communications, peu efficace. Ceci s'explique par la complexité de l'algorithme d'application des règles (*c.f.* Algorithme 2) nécessitant de calculer des sous-automates comparativement à la complexité d'application d'un transducteur. Pour améliorer les performances et appliquer notre prototype à des programmes conséquents, nous avons remarqué que notre itération de point fixe pouvait être modifiée pour minimiser le nombre d'applications de règles.

Nous avons constaté empiriquement que les transitions locales étaient bien plus appliquées que les règles des réécritures globales en particulier en début des programmes où peu de communications ont lieu. Toutefois, notre algorithme d'application de règles est appelé même si aucune d'entre-elles n'est applicable. Ainsi, pour réduire le nombre d'appels, notre idée est que : plutôt que d'appliquer une seule fois les transitions locales avant d'appliquer les règles globales, nous allons commencer par chercher un premier point fixe des transitions locales avant d'appliquer les règles globales. Autrement, dit, nous calculons la limite de la suite croissante définie par :

$$\begin{aligned}
\mathcal{A}_0 &= I \\
\mathcal{A}_{n+1} &= \begin{cases} \mathcal{A}_n & \text{si } \mathcal{T}(\mathcal{A}_n) \sqcup \mathcal{R}(\mathcal{A}_n) \sqsubseteq \mathcal{A}_n \\ \mathcal{A}_n \triangledown (\mathcal{A}_n \sqcup \mathcal{T}^*_{\texttt{local}}(\mathcal{A}_n) \sqcup \mathcal{R}(\mathcal{A}_n)) & \text{sinon} \end{cases} \\
\mathcal{T}^\alpha(I) &= \lim_{n\to+\infty} \mathcal{A}_n
\end{aligned}$$

où, pour tout automate $\mathcal{A}$, $\mathcal{T}^*_{\texttt{local}}(\mathcal{A})$ est défini comme la limite de la suite croissante :

$$\begin{aligned}
\mathcal{A}_{\texttt{local},0} &= \mathcal{A} \\
\mathcal{A}_{\texttt{local},n+1} &= \begin{cases} \mathcal{A}_{\texttt{local},n} & \text{si } \mathcal{T}(\mathcal{A}_{\texttt{local},n}) \sqsubseteq \mathcal{A}_{\texttt{local},n} \\ \mathcal{A}_{\texttt{local},n} \triangledown (\mathcal{A}_{\texttt{local},n} \sqcup \mathcal{T}(\mathcal{A}_{\texttt{local},n})) & \text{sinon} \end{cases} \\
\mathcal{T}^*_{\texttt{local}}(\mathcal{A}) &= \lim_{n\to+\infty} \mathcal{A}_{\texttt{local},n}
\end{aligned}$$

Sur nos cas de test, cette nouvelle version de l'algorithme permet d'accélérer jusqu'à 5 fois la vitesse de l'analyse. Cette optimisation est particulièrement intéressante lorsque les communications et les entrelacements sont nombreux.

Théoriquement, cette itération n'est pas équivalente à l'ancienne. Notamment à cause de l'absence de monotonie de l'opérateur d'élargissement. Toutefois, nous obtenons une sur-approximation sûre de l'espace d'atteignabilité.

Afin d'assurer une précision satisfaisante, il est nécessaire d'appliquer l'opérateur d'élargissement avec parcimonie. Pour une analyse séquentielle, il est possible d'appliquer la technique des *itérations chaotiques* présentée par François Bourdoncle [Bou93]. On commence par établir un ensemble minimal de points de programme sur lesquels il est nécessaire d'appliquer l'opérateur d'élargissement pour augmenter la précision de l'analyse tout en conservant sa terminaison.

Dans notre formalisme, nous ne pouvons pas appliquer cette heuristique directement. En effet, nos états de programmes sont représentés par des automates de treillis et considèrent plusieurs points de programmes. L'opérateur d'élargissement défini pour les automates de treillis possède un comportement différent : lorsqu'on l'applique à deux automates, on commence par regarder leurs « formes » (*i.e.* on efface les étiquettes des transitions), et, si elles diffèrent, une abstraction est générée et les transitions, pouvant être associées dans les deux automates, sont fusionnées (*c.f.* Section 2.7). Autrement, les deux automates sont isomorphes et, pour chaque paire de transitions des deux automates, l'opérateur d'élargissement du treillis paramétrant les automates de treillis est appliqué.

Pour pouvoir utiliser la technique des *itérations chaotiques*, nous avons défini un nouvel opérateur d'élargissement pouvant être paramétré. Cet opérateur aura un comportement différent si les deux automates sont isomorphes. Plutôt que d'appliquer l'opérateur d'élargissement sous-jacent entre deux transitions, nous « surchargeons » celui-ci pour qu'il ne l'applique que si les localités des processus correspondent à un sommet sélectionné du CFG ; autrement, l'union entre les deux états locaux sera effectuée.

5.2.4 Conclusion

Nous avons présenté, MPAI, notre prototype d'analyse d'un sous-ensemble de MPI/C sous la forme d'un plug-in Frama-C. Notre outil est capable d'analyser automatiquement les programmes donnés et peut paramétrer la précision de son analyse, en particulier, via le choix de domaines abstraits utilisés et par l'opérateur d'élargissement des automates de treillis. Le résultat de notre outil est l'ensemble des états atteignables présenté sous la forme d'un automate affichable.

Malgré nos efforts, nous n'avons malheureusement pas réussi à intégrer de manière satisfaisante les domaines d'EVA, l'analyse de valeurs de Frama-C, et par conséquent, en l'état, nous ne pouvons pas réutiliser ses capacités à traiter pointeurs et tableaux C.

Une autre perspective pour notre outil serait l'ajout d'un langage de spécification. Ceci permettrait d'exprimer et de vérifier de manière directe et automatique des propriétés séquentielles, pour chaque processus, et concurrentes en considérant l'ensemble des processus. Nous avons commencé à nous pencher sur la réutilisation du plug-in `E-ACSL` [DKS13], un autre plug-in Frama-C proposant, entre autre, son propre langage de spécification pour code C séquentiel. Notre idée serait d'adapter ce langage à notre cadre : la vérification de programmes concurrents. Par exemple, introduire la notion de multi-processus disposant d'identifiants.

Chapitre 6

Expérimentations

Comme mentionné dans le chapitre précédent, à notre connaissance, il n'existe pas d'outil effectuant d'analyse de valeurs de programmes `MPI/C` de manière statique. Un des outils le plus abouti et s'intéressant à la vérification de `MPI` est `ISP` [VVD+09] (*In-Situ Partial order*). Ce dernier se concentre sur la détection d'interblocage des programmes dont l'analyse est effectuée de manière *dynamique*.

Bien qu'ayant des vocations différentes, notre outil `MPAI` et `ISP` peuvent être comparés. En effet, en supposant que le comportement attendu des programmes auxquels on s'intéresse est l'absence d'interblocage (*i.e.* tous les états non-finaux des processus peuvent progresser), il est possible d'utiliser notre prototype pour vérifier que les programmes analysés n'en contiennent pas. Pour effectuer cette comparaison, nous considérons, entre autres, un sous-ensemble d'une base de tests de ISP en comparant les résultats sur les différents cas d'études.

Nous proposons ensuite d'étudier les résultats sur divers exemples exploitant les possibilités de notre prototype, c'est-à-dire, l'analyse de valeurs de programmes `MPI/C` en l'utilisant sur des exemples disposant d'un nombre de processus non-borné et comportant potentiellement de la création dynamique de processus.

Nos expérimentations sont effectuées sur une distribution Linux (3.13.0) Ubuntu 16.04 LTS (64-bits) disposant d'un processeur Intel(R) Core(TM) i7-4790 CPU à 3.60GHz avec 16Go de mémoire DDR3. Les temps d'exécution sont établis à l'aide de la commande `Unix time`. Ces temps comprennent donc d'éventuels sur-coûts liés à l'initialisation ou la fermeture des fichiers, mais nous estimons que ces sur-coûts sont suffismamment proches entre les différents outils pour que notre comparaison reste valide.

6.1 Comparaison avec ISP

Dans cette section, nous avons repris certains exemples de programmes utilisés dans la base de tests de ISP [1]. Ces exemples ont, dans la plupart des cas, été adaptés pour qu'ils correspondent au sous-ensemble de `MPI` que nous traitons. Ces programmes adaptés sont disponibles dans le dossier `tests/deadlock/` de notre outil [2].

Les tests que nous avons repris de cette base sont des variantes du programme présenté par la Figure 6.1. En outre, tous les programmes-tests que nous présentons disposent d'un interblocage possiblement atteignable à l'exécution dû au non-déterminisme provenant de l'ordre dans lequel l'ordonnanceur

1. `http://formalverification.cs.utah.edu/ISP_Tests/`
2. `https://www-apr.lip6.fr/~botbol/mpai/`

effectue ses communications, il y a des exécutions bloquantes comme des exécutions non-bloquantes. Ainsi, dans ce programme MPI/C, le processus de rang 0 commence par recevoir une valeur d'un autre processus (L. 15), qui n'est pas déterminé a priori par l'utilisation de la primitive MPI_ANY_SOURCE. Si le processus de rang 0 reçoit une communication du processus 2 (L. 22) alors le processus 0 et processus 1 ne peuvent plus progresser puisqu'ils sont mutuellement en train d'essayer de s'envoyer un message (L. 16 et L. 29). En revanche, si le processus de rang 0 reçoit d'abord une communication du processus 1, tous les processus peuvent terminer leur exécution.

```
1  #include <mpi.h>
2
3  int main (int argc, char **argv){
4    int nprocs, rank, x, y;
5
6    MPI_Init (&argc, &argv);
7    MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
8    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
9
10   if (rank < 1) {
11     MPI_Recv (&y, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, NULL);
12     MPI_Send (&y, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
13     MPI_Recv (&x, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, NULL);
14   } else if (rank < 2) {
15     x = 1;
16     MPI_Send (&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
17     MPI_Recv (&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, NULL);
18   } else if (rank < 3) {
19     y = 2;
20     MPI_Send (&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
21   }
22
23   MPI_Finalize ();
24   return 0;
25 }
```

FIGURE 6.1 – Programme interbloquant dépendant de l'ordre des communications

6.1.1 Détection d'interblocages

Rappelons que MPAI n'a pas été conçu dans le but de détecter la présence ou l'absence d'interblocage, mais pour calculer une sur-approximation de l'espace d'états atteignables. Cependant, il nous est possible, à partir de cette sur-approximation, d'en détecter certains cas.

Par exemple, si l'on pose comme hypothèse que tous les processus du programmes sont censés terminer, c'est-à-dire qu'ils doivent tous atteindre la localité de l'instruction `return`, et que la sur-approximation des états atteignables ne contient aucune configuration dans laquelle tous les processus sont dans cette localité, alors nous sommes sûrs que le programme analysé ne terminera jamais. Si au contraire il existe une telle configuration dans l'espace que MPAI a calculé, alors nous ne pouvons rien conclure. En effet, du fait des sur-approximations, nous ne savons pas si cette configuration est effectivement atteignable par le programme.

Pour la détection d'interblocages proprement dite, nous avons employé la méthode suivante pour ces premiers jeux de tests : pour tout chemin simple, *i.e.* sans transition cyclique, présent de l'espace des états atteignables, nous avons appliqué la fonction de transition à ces sous-automates : si le résultat n'est pas vide ($\neq \bot$), alors cela signifie que la configuration représentée par ce chemin simple n'est

pas bloquante. En revanche, si le résultat cette application est vide, cela signifie que l'on a trouvé une configuration bloquante. Une alarme est déclenchée et nous affichons la configuration en cause à l'utilisateur. Ici encore, l'alarme peut être un faux positif dû à une abstraction trop peu précise. Si tel est le cas, l'abstraction doit être raffinée. Notons aussi que cette méthode peut échouer à détecter un interblocage réellement présent : en effet, un chemin simple représente un ensemble de configurations. En appliquant le transducteur à ce sous-automate A, on pourrait constater que $T(A)$ est non vide alors même qu'il contient au moins une configuration bloquante.

Nous avons implanté cette méthode dans `MPAI`, qu'il est possible d'activer via l'option `-check-liveness`. Nous évaluons ensuite les performances des différents outils sur les programmes issus de la base de tests de `ISP`. Nous comparons, pour `ISP`, sa détection d'interblocage, et pour notre outil, le déclenchement d'alarme lié à la présence potentielle d'interblocages.

Les commandes exactes effectuées pour les mesures de `ISP` sont :

```
$ ispCC fichier.c -o prog
$ time isp -n NB_PROC -b prog
```

où `ispCC` est le compilateur de `ISP` instrumentant le programme `MPI/C` afin de capturer des informations supplémentaires lors de l'exécution, `NB_PROC` est le nombre de processus attendu et `-b` est une option forçant la sémantique des émissions de messages à être bloquante : en effet, certaines implantations de `MPI` poursuivent leurs exécutions après un envoi sans attendre de message d'acquittement du récepteur. Notre sémantique suppose la synchronisation de l'émetteur et du récepteur : nous activons cette option pour une cohérence entre les résultats. Il est à noter que sans cette option, sur certains programmes, `ISP` n'est pas capable de détecter un interblocage.

La commande utilisée pour les mesures de `MPAI` est :

```
$ time mpai -np NB_PROC -only-tstar -check-liveness fichier.c
```

Par défaut, `MPAI` affiche l'état de chaque itération, l'option `-only-tstar` n'affiche que la sur-approximation de l'espace d'états et permet de ne pas perdre du temps d'exécution en affichage superflu ; `-check-liveness` est l'option activant la vérification évoquée plus haut.

Nous avons donc sélectionné les exemples de ISP pouvant également être analysés par notre outil. Les exemples utilisant extensivement des tableaux ou encore utilisant des primitives de communications asynchrones n'ont pas été considérés. Les résultats des tests de détection d'interblocages sont présents dans le tableau de la Figure 6.2. Ce banc d'essai nous montre, qu'en moyenne, sur les exemples traités, les performances de nos outils sont relativement équivalents. Sur les exemples de la base de tests de ISP, celui-ci est capable de détecter tous les interblocages présents. `MPAI` est, quant à lui, également capable, dans la plupart des cas, de déclencher une alarme. Après vérification, les interblocages détectés sur ces exemples sont effectivement ceux attendus.

Le premier test `send_recv.c` correspond au programme de la Figure 6.1. Les deux tests suivants `send_recv_barrier.c` et `send_recv_barrier2.c` sont similaires au premier test mais font varier le nombre de communications et emploient également des barrières de synchronisation. Dans l'exemple `send_recv_interleaving.c` augmente encore le nombre de communications. Dans cet exemple, il nous a été nécessaire d'approfondir la recherche de chemin bloquant. En effet, notre algorithme de fusion d'états de processus locaux rassemble certains états similaires (*i.e.* plusieurs états sur une même transition) ne permettant pas à l'heuristique de détection d'interblocages de détecter l'interblocage. Pour y parvenir, nous avons développé une version plus coûteuse de l'heuristique éclatant ces états en de nou-

Programme	# proc.	ISP		MPAI	
		Détection	Temps d'exéc.	Alarme	Temps d'exéc.
send_recv.c	3	✓	2,28s	✓	1,55s
send_recv_barrier.c	3	✓	2,3s	✓	1,94s
send_recv_barrier2.c	3	✓	2,2s	✓	1,84s
send_recv_interleaving.c	3	✓	2,2s	✓*	2,89s
broadcast.c	3	✓	1,02s	✓	0.29s
for_loop.c	10	✓	1,06s	✓	0,77s
long_computation_loop.c	2	✓	47m12s	✓	2,25s
diffusion_2D_no_deadlock.c	4	✗	1.24s	✗	43,77s
diffusion_2D.c	4	✓	1,05s	✓	43,51s

* : Pour certains exemples, notre heuristique d'exploration de branches de l'automate ne suffit pas à montrer l'absence de progrès ; nous nous autorisons l'utilisation d'une exploration plus précise et plus coûteuse (`-check-deep-liveness`).

FIGURE 6.2 – Détection d'interblocage

veaux chemins qu'il est alors possible d'analyser. Dans cet exemple, le temps d'analyse varie d'environ 25% entre la version simple et la version coûteuse permettant d'obtenir cette alarme. `broadcast.c` est un exemple simple où le premier processus n'exécute pas l'instruction de communication globale `MPI_bcast`. Dans `for_loop.c`, le processus maître d'identifiant 0 envoie à chaque processus une valeur à l'aide d'une boucle *for* puis essaye d'envoyer à un processus d'identifiant inexistant un message générant un interblocage. `long_computation_loop.c` reprend l'exemple précédent mais utilise la valeur reçue pour effectuer un calcul local coûteux. La différence du temps d'exécution des deux outils provient du fait que `ISP` est dépendant de l'exécution des calculs du programme tandis que notre exécution est, elle, pûrement symbolique et grâce aux techniques d'accélérations de l'interprétation abstraite notre analyse est quasi-instantanée. Le dernier exemple, `diffusion_2D.c`, propage des informations en communicant entre processus d'identifiants voisins. La première version de cet exemple ne comprend pas d'interblocages. L'écart entre les deux temps d'exécution s'explique de par le fait que notre analyse ne parvient pas à abstraire efficacement le grand nombre de communications. En réduisant la précision de l'analyse, nous sommes capables de retrouver des temps comparables à `ISP` mais en générant des états inatteignables comprenant des interblocages. Dans la deuxième version, nous avons ajouté des communications susceptibles de causer un interblocage. Ici encore, notre analyse souffre du trop grand nombre d'entrelacements que nous ne pouvons pas abstraire si nous souhaitons obtenir l'interblocage significatif. Il est à noter que nous pourrions encoder ce programme sous la forme d'un transducteur comme décrit dans le chapitre 3 et gagnerions sensiblement en performance. Malgré ce manque d'efficacité sur ce dernier exemple, nous obtenons bien les résultats attendus.

6.1.2 Passage à l'échelle

Nous nous sommes également intéressés à l'évolution des performances des deux outils lorsque le nombre de processus croit (passage à l'échelle). Notons que ces expérimentations ont été réalisées sur la même machine que précédemment, qui ne dispose pas de centaines, ni même de dizaines de processus. Si cette différence ne change pas grand chose pour `MPAI`, qui procède par analyse statique, cela peut ralentir `ISP`. De plus, `MPI` ne permet pas de lancer ainsi plus de 200 processus en local, ce qui limite d'autant la taille des exemples. Aussi le but de ces expérimentations est plus de voir comment les performances de chaque outil évoluent plutôt que de prétendre que ces performances seraient les mêmes sur un cas

d'étude réel.

Programme	# proc.	ISP		MPAI	
		Détection	Temps d'exéc.	Alarme	Temps d'exéc.
send_recv.c	10	✓	2,62s	✓	9,6s
send_recv.c	100	✓	10,17s	✓	10,57s
send_recv.c	200	✓	17,13s	✓	11,33s
for_loop.c	10	✓	1,06s	✓	0,77s
for_loop.c	100	✓	4,412s	✓	1,19s
diffusion_2D.c	4	✓	1,04s	✓	43,51s
diffusion_2D.c	8	✓	1,22s	✓	35m23s
diffusion_2D.c	16	✓	1.47s	?	$+\infty$
long_loop_computation.c	$+\infty$			✓	4,53s
send_recv.c	$+\infty$			✓	17,93s

FIGURE 6.3 – Performances (grand nombre de processus)

Dans un premier temps, nous avons cherché à mesurer la différence entre les analyses reçevant un grand nombre de processus. Pour les deux derniers exemples, nous avons choisi de considérer un nombre infini de processus, chose qu'il est possible de modéliser avec MPAI mais pas avec ISP.

Dans le premier exemple send_recv.c, la plupart des processus ne font rien après leur création. Cependant, l'aspect dynamique de ISP l'oblige à exécuter la totalité des processus même si ceux-ci ne font rien. Nous constatons que pour ISP, l'augmentation du nombre de processus se traduit par un accroissement linéaire du temps de calcul alors que notre prototype conserve un temps relativement constant. Cela s'explique par le fait que les processus inutilisés sont très rapidement fusionnés (après la première itération). Notre analyse détecte que ceux-ci n'ont pas d'influence dans le calcul et les abstrait en une transition cyclique.

L'exemple suivant, for_loop.c, illustre également ce même comportement à la différence qu'ISP est capable de détecter plus rapidement l'interblocage que pour l'exemple précédent. En effet, dans cet exemple, le premier processus est rapidement bloqué et celui-ci se détecte aux premiers itérations des deux analyses. La différence de temps est donc principalement liée à l'instanciation des nœuds de calcul.

Le troisième exemple, diffusion_2d.c, est plus complexe et nous constatons que le nombre de processus influe grandement sur les performances. En effet, il y a beaucoup de communications entre les processus, et donc un grand nombre d'entrelacements. Comme nous l'avons évoqué dans la section précédente, cette complexité exponentielle provient du fait que dans notre analyse, dans certains cas, peut entraîner un sur-coût de l'analyse en explorant certains états issus de l'abstraction. Au contraire, ISP n'a pas de problème de performances particulier.

Les deux derniers exemples illustrent le fait que notre outil est capable de considérer des analyses avec un nombre de processus arbitraire ce dont ISP n'est pas capable.

Toutes ces expérimentations sont résumées par la Figure 6.3. Globalement, MPAI passe bien à l'échelle lorsque l'on augmente le nombre de processus. Le seul contre-exemple est diffusion_2d.c qui montre que, dans notre prototype, la génération automatique des règles pour les communications est encore perfectible.

6.1.3 Avantages de l'analyse statique

Comme nous l'avons dit, MPAI procède par analyse statique tandis que ISP procède de manière dynamique. L'avantage de l'analyse statique est de pouvoir donner une sur-approximation possible, d'explorer « tous les chemins » alors que l'analyse dynamique est tributaire d'une exécution particulière.

```
1  #include <mpi.h>
2
3  int main (int argc, char **argv) {
4    int rank;
5    MPI_Init (&argc, &argv);
6    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
7
8    if (rank == 0) {
9      if (random() % 2 > 0){
10       MPI_Send (&rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
11     } else {
12       MPI_Recv (&rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, NULL);
13     }
14   }
15   else {
16       MPI_Recv (&rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, NULL);
17   }
18
19   MPI_Finalize ();
20   return 0;
21 }
```

FIGURE 6.4 – Interblocage aléatoire : `random.c`

Ainsi, pour des programmes provoquant des interblocages selon un non-déterminisme issu d'un calcul numérique (*e.g.* un branchement selon une condition aléatoire), ISP n'est capable de détecter la présence d'un interblocage seulement si celui-ci advient à l'exécution. Cela donne donc un caractère aléatoire à la détection, et n'offre aucune garantie d'absence de blocage. Autrement dit, même si ISP ne détecte pas d'interblocage, cette situation peut quand même se produire lors d'une autre exécution que celle analysée par ISP.

Au contraire, l'analyse statique effectuée par MPAI, est capable de traiter ce type de cas comme le montre l'exemple du programme `random_deadlock.c` (Figure 6.4). Dans cet exemple, le premier processus opère un choix non-déterministe. Si la condition du `if`, est respectée alors cela conduit à une situation d'interblocage. L'analyse statique va nous permettre de considérer tous les cas possibles et donc d'être capable de détecter la présence d'un interblocage (Figure 6.5).

Programme	**# proc.**	**ISP**		**MPAI**	
		Détection	Temps d'exéc.	Alarme	Temps d'exéc.
`random.c`	2	✓/✗	1,09s	✓	0,56s

FIGURE 6.5 – Comparaison de MPAI et de ISP sur l'exemple `random.c`

Comme expliqué précédemment, la méthode de détection des interblocages utilisée par MPAI est loin d'être parfaite, car elle repose elle-même sur des approximations. Cependant, en supposant que l'ensemble des états bloquants est connu et représentable sous forme d'automate de treillis, on peut vérifier que cet ensemble n'est pas atteignable à partir des états initiaux, en utilisant l'algorithme général de calcul de point-fixe. Si, effectivement, cet ensemble n'est pas atteignable alors nous avons la garantie

formelle qu'il n'y aura pas d'interblocage à l'exécution. Cette garantie formelle ne peut pas être établie par des analyses du type de celles utilisées par `ISP`.

Enfin, `ISP` étant dépendant de l'exécution du programme, si le programme effectue des calculs coûteux, les performances en sont affectées, comme nous l'avons vu sur la question du passage à l'échelle. Au contraire, l'analyse statique, notamment grâce à l'utilisation d'un opérateur d'élargissement, peut se révéller bien plus économe.

Tous ces avantages ne sont bien sûr pas propres à `MPAI` : n'importe quelle autre analyse statique aurait les mêmes avantages comparée à `ISP`. Cependant, `MPAI` est l'un des très rares outils développés pour faire l'analyse statique de programmes `MPI`.

Il est maintenant temps de s'intéresser aux invariants numériques qui peuvent être découverts automatiquement par `MPAI`.

6.2 Analyses de valeurs

Nous nous intéressons à des programmes effectuant du calcul numérique pour lesquels nous souhaitons prouver des propriétés de sûreté. A ce titre, nous voulons montrer, à partir de quelques exemples, quels sont les types d'invariants que notre analyse est capable de découvrir.

Dans ces exemples, nous considérons également des programmes effectuant des créations dynamiques. Sauf mention du contraire, le domaine abstrait numérique sous-jacent que nous employons pour nos analyse est celui des polyèdres (voir section 2.2.4).

6.2.1 Calcul de séries

Les deux premiers exemples que nous développons sont des calculs de séries, dont la forme générale est $\Sigma_{i\in\{0..N\}} f(i)$, pour une certaine fonction f. Ce genre de calcul peut facilement se paralléliser à l'aide de `MPI` en utilisant un squelette de la forme *map-reduce*. En effet, si nous avons p processus, chacun peut calculer une partie de la série, puis le processus 0 collecte la somme de tous ces résultats partiels grâce à la primitive `reduce`. Les propriétés que nous souhaitons prouver sur ces programmes sont des encadrements de la valeur du résultat (après `reduce`). Pour plus de simplicité, chaque processus est chargé de calculer une étape du calcul de la série. Ainsi, leurs identifiants correspondront à l'indice de la somme.

Le premier programme implémente une approximation de π où nous souhaitons encadrer la valeur de π entre $[3,00; 3,30]$. Ce calcul se fait grâce à la série $\frac{4}{N} \times \Sigma_{i\in\{0..N\}} \frac{1}{1+(i/N)^2}$ qui tend (lentement) vers π lorsque N tend vers l'infini. Le programme exact que nous analysons est celui de la Figure 6.6.

Le deuxième calcule la série $\Sigma_{i\in\{1..N\}} \frac{1}{2^i}$ qui tend vers 1 lorsque N tend vers l'infini [3]. La propriété que nous souhaitons montrer est que le résultat est inférieur à 1.

Les résultats donnés par la Figure 6.7 indiquent que, pour le premier calcul, la propriété est vérifiée (en vert) pour un N suffisamment grand ce qui correspond à la précision du calcul de l'approximation de π. Pour la deuxième série, nous obtenons bien cette propriété pour tout nombre de processus fixé. Cependant, nos analyses sont plus lentes lorsque le nombre de processus augmente. Cela s'explique par le fait que pour ces tests en particulier, *nous désactivons l'élargissement* lors du calcul de point-fixe. En effet, si nous utilisons l'élargissement, l'analyse va fusionner des processus avec différents

3. On fait commencer la série à 1, et non à 0. Si on commençait à 0, la limite de la série serait 2

```
1
2       #include <mpi.h>
3
4       int main(int argc, char** argv){
5         int rank, size;
6         double x2, pi, result;
7
8         MPI_Init(&argc, &argv);
9
10        MPI_Comm_rank (MPI_COMM_WORLD, &rank);
11        MPI_Comm_size (MPI_COMM_WORLD, &size);
12
13        x2 = ((double)rank / (double)size);
14        result = 1.0/ ( 1.0 + (x2 * x2));
15        MPI_Reduce(&result, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
16        pi = (double)4 / (double)size * pi;
17        MPI_Bcast(&pi, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
18
19        MPI_Finalize();
20        return 0;
21      }
```

FIGURE 6.6 – Code `MPI` : Approximation distribuée de π

Programme	# proc.	Résultats	Temps d'exéc.
`pi.c`	2	$\pi_{approx} = 3.60$	0,43s
`pi.c`	4	$\pi_{approx} = 3.38$	1,5s
`pi.c`	8	$\pi_{approx} = 3.26$	30,82s
`pi.c`	10	$\pi_{approx} = 3.23$	58,4s
`pi.c`	∞	$\pi_{approx} \geq 0$	5,94s
`sum.c`	2	$3/4$	0,52s
`sum.c`	4	$(2^4 - 1)/2^4$	1,85s
`sum.c`	8	$(2^8 - 1)/2^8$	5,25s
`sum.c`	16	$(2^{16} - 1)/2^{16}$	47,21s
`sum.c`	$+\infty$	$\top$	2,77s

FIGURE 6.7 – Analyse de calculs parallèles de séries non-linéaires

identifiants, et va aboutir à une approximation qui calculera la somme de termes identiques. Dans notre second exemple, on obtiendrait (au mieux) pour chaque processus, une abstraction donnée par : $\texttt{id} \in [0; \texttt{size}]$ et $\texttt{res} = \frac{1}{2^{\texttt{id}+1}}$. Ainsi, si nous sommons ces valeurs abstraites avec $\texttt{size} = N$, nous obtenons : $\texttt{res}_{\texttt{total}} = \frac{1}{[1;2^{\texttt{size}}]} + \frac{1}{[1;2^{\texttt{size}}]} + \ldots + \frac{1}{[1;2^{\texttt{size}}]}$. Cela donne une somme de N termes identiques qui diverge lorsque N augmente.

6.2.2 Calculs numériques avec création dynamique

Nous nous intéressons maintenant à la capacité de notre prototype à découvrir des invariants numériques de programmes effectuant de la création dynamique de processus. Nous reprenons l'exemple développé dans la section 4.4 écrit cette fois en `MPI` présent sur la Figure 6.8. Pour rappel, dans ce programme, un processus initial crée N nouveaux processus en leur communiquant une valeur numérique (correspondant à leur identifiant). Ces processus nouvellement créés vont alors calculer le double de cette valeur reçue avant de synchroniser tous les processus entre eux pour calculer la somme des calculs effectués. Finalement, ce total est communiqué à l'ensemble des participants.

```
1   #include <mpi.h>
2
3   int main(int argc, char **argv) {
4     int id, i, new_id, x, res;
5
6     MPI_Init(&argc, &argv);
7     MPI_Comm_rank(MPI_COMM_WORLD, &id);
8
9     if (id < 1) {
10      x = 0;
11      for (i = 1; i <= 5; i++) {
12        MPI_Create(&new_id);
13        MPI_Send (&i, 1, MPI_INT, new_id, 0, MPI_COMM_WORLD);
14      }
15    } else {
16      MPI_Recv(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, NULL);
17      x = x + x;
18    }
19
20    MPI_Reduce(&x, &res, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
21    MPI_Bcast(&res, 1, MPI_INT, 0, MPI_COMM_WORLD);
22
23    MPI_Finalize();
24    return 0;
25  }
```

FIGURE 6.8 – Code `MPI` : création dynamique de processus

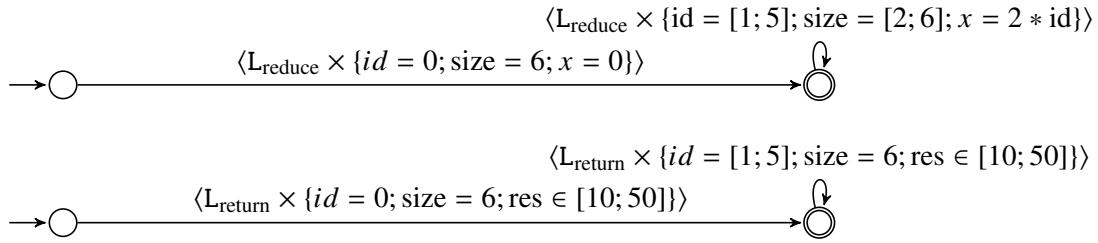


FIGURE 6.9 – Ensemble d'atteignabilité

Programme	**# proc.**	Propriétés prouvées	Temps d'exéc.
`create_and_sum.c`	$1 \Rightarrow 6^*$	✓	4,45s
`create_and_sum.c`	$1 \Rightarrow 101^*$	✓	4,819

* : $x \Rightarrow y$ indique que les programmes sont initialisés avec x processus et créent dynamiquement $(y - x)$ processus.

FIGURE 6.10 – `create_and_sum.c` : Résultats obtenus

Les invariants montrés appartiennent à l'espace d'atteignabilité calculé à l'aide de notre analyse Figure 6.9 et permettent de conclure sur les propriétés :

— Le premier processus crée 5 nouveaux processus ;

— Le résultat du calcul est borné.

Nous pouvons noter que dans le premier invariant, notre analyse a découvert la relation $x = 2 * id$. La taille des processus créés à ce moment n'est pas bornée puisque ces processus peuvent atteindre cette localité avant le processus 0. Après application du `reduce`, la taille des processus est réduite à 6 : la sémantique du `reduce` nécessite que tous les processus présents exécutent cette opération. Ainsi, nous avons ajouté un trait à l'application de la réduction stipulant que la taille des processus dans les mots concernés doivent être cohérents. Pour ce faire, nous calculons l'intersection de toutes les tailles.

Nous obtenons alors, en sortie de la réduction, l'invariant `size` = 6 pour l'ensemble des processus. En revanche, notre analyse n'est pas capable de donner un encadrement du résultat plus précis : les états des processus id $\neq 0$ étant abstraits avec la valeur $x \in [2; 10]$, la somme résultante est alors comprise entre $[10; 50]$ Figure 6.10.

Le dernier exemple que nous présentons est également un programme effectuant de la création dynamique de processus Figure 6.11. Comme pour l'exemple précédent, ce programme démarre avec un seul processus. Celui-ci initialise une variable x puis crée un nouveau processus avant d'effectuer un calcul $(x+4)$ et d'envoyer ce résultat au processus nouvellement créé avant de se terminer. Ce deuxième processus va à son tour créer un nouveau processus, incrémenter x et envoyer cette nouvelle valeur à son processus enfant.

```
1   #include <mpi.h>
2
3   int main(int argc, char **argv) {
4     int id, x, child_id;
5
6     MPI_Init(&argc, &argv);
7     MPI_Comm_rank(MPI_COMM_WORLD, &id);
8
9     if (id < 1){
10      x = 1;
11    } else {
12      MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, NULL);
13    }
14
15    MPI_Create(&child_id);
16    x = x + 4;
17    MPI_Send (&x, 1, MPI_INT, child_id, 0, MPI_COMM_WORLD);
18
19    MPI_Finalize();
20    return 0;
21  }
```

FIGURE 6.11 – Création dynamique non-bornée avec passage de valeur

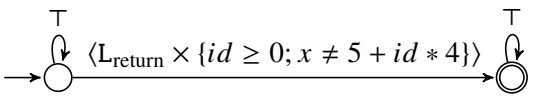


FIGURE 6.12 – $\mathcal{B}$: Mauvaise configuration

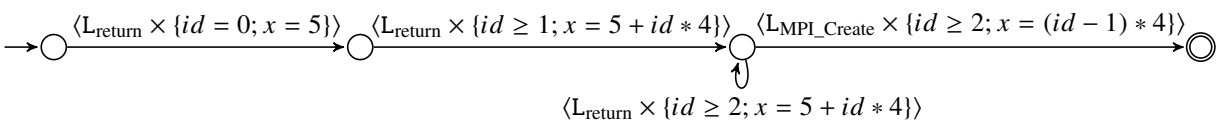


FIGURE 6.13 – Invariant présent dans l'espace d'atteignabilité (calculé en 30,6s)

La propriété que nous souhaitons montrer est que tout processus ayant atteint la localité du `return` a pour valeur x = 1 + (`id` + 1) ∗ 4. Nous exprimons cette propriété sous la forme d'une mauvaise configuration Figure 6.12. En effectuant notre analyse, nous obtenons l'invariant Figure 6.13 qui vérifie effectivement notre propriété. Cette analyse s'effectue en 30,6s.

6.3 Conclusion

Ces expérimentations nous ont permis de comparer `MPAI` avec `ISP`, qui est l'un des outils les plus aboutis pour vérifier des programmes `MPI`. `ISP` étant un outil de détection d'interblocage, nous avons dû légèrement modifier `MPAI` qui n'était pas prévu au départ pour faire spécifiquement ce type de vérification. Ces comparaisons nous ont permis de constater que `MPAI` se révèle souvent aussi voire plus performant que `ISP` sur de nombreux exemples tirés de la base de tests de ISP. `MPAI` passe également mieux à l'échelle sur la plupart des exemples. En tant qu'outil d'analyse statique, il offre en plus certaines garanties suplémentaires, comme la couverture de tous les chemins d'exécution et la preuve formelle que les états potentiellement en situation d'interblocage sont inaccessibles.

Dans un deuxième temps, nous avons évalué la qualité des invariants trouvés par notre analyse de valeurs, à la fois sur le calcul de séries mathématiques et pour l'analyse de programmes numériques avec création dynamique. Ces derniers exemples sont ceux pour lesquels notre analyse a été conçue, et les résultats sont encourageants. L'utilisation de domaines abstraits relationnels comme celui des polyèdres, ainsi que la possibilité de faire référence à certaines variables globales comme le nombre de processus nous permet de d'établir des contraintes linéaires entre la valeur d'une variable et le rang (identifiant) du processus. À notre connaissance, aucun autre outil de vérification de programme `MPI` permet de découvrir de tels invariants.

Chapitre 7

Conclusion

7.1 Contributions

Dans ce manuscrit, nous avons présenté une nouvelle méthode de vérification logicielle par analyse statique pour les programmes concurrents effectuant des calculs numériques. Cette méthode repose sur l'usage de transducteurs similaires à ceux utilisés pour le *Regular Model Checking*. Nous avons adapté ce modèle théorique en utilisant des abstractions issues de la méthodologie de l'interprétation abstraite afin de considérer des états de processus potentiellement infinis. Cette extension nous permet alors de vérifier plus aisément des programmes de calcul parallèle.

Notre première contribution a été de formaliser un modèle semblable à celui du *Regular Model Checking* mais en utilisant à la place des automates à alphabet fini, des automates à alphabet symbolique d'un type particulier appelés automates de treillis. Ces automates nous permettent d'encoder des ensembles de mots (autrement dit, des langages) sur un alphabet infini, chacun de ces mots étant une abstraction des états possibles de chaque processus du système distribué.

Pour encoder la fonction de transition (et donc la sémantique) d'un tel système, nous avons défini un nouveau type de transducteurs à alphabet symbolique capables de réécrire les langages encodés par les automates de treillis : les transducteurs de treillis. Nous avons également donné un algorithme pour appliquer ces transducteurs à un langage et montré sa correction. Munis de ces deux objets, nous avons alors pu construire une première analyse en réutilisant les techniques d'interprétation abstraite pour calculer une sur-approximation de l'espace des états atteignables. Les états de programmes étant encodés sous forme d'automates de treillis, nous pouvons utiliser l'opérateur d'élargissement nous assurant une terminaison de notre analyse.

Comme les transducteurs de treillis peuvent réécrire des mots de taille arbitraire, cette propriété nous permet de considérer des créations et destructions dynamiques de processus. Nous avons donc étudié un langage de programmation pour décrire des systèmes distribués facilement traduisibles vers notre modèle et possédant les caractéristiques suivantes : structures de contrôle, calculs numériques complexes sur chaque processus, communications entre processus voisins et création et destruction dynamique de processus. Les processus partagent le même code source mais possèdent chacun leur propre zone mémoire. De plus, les communications sont synchrones. Munis de ce langage et d'une analyse de notre modèle, nous montrons que nous sommes capables de construire des programmes expressifs et d'en générer des invariants nous permettant de prouver des propriétés non triviales.

Nous avons pu alors vérifier des programmes distribués générés à partir de ce langage. Cependant,

la communication limitée aux voisins est une restriction forte qui ne permet pas de modéliser de nombreux exemples classiques de programmes parallèles. Cela provient de la nécessité de reconnaître les deux processus souhaitant communiquer pour accéder aux deux contextes lors de la réécriture. Ainsi, les transitions du transducteurs doivent reconnaître un sous-mot comportant les processus souhaitant communiquer. Étant donné que la longueur des sous-mots reconnus par les transducteurs est bornée, il n'est pas possible de faire communiquer deux processus situés à une distance arbitraire. Ainsi, nous ne pouvons pas considérer de communications plus expressives comme, par exemple, du point-à-point. Pour résoudre ce problème, nous avons introduit de nouvelles règles capables de reconnaître puis de réécrire des mots de longueur arbitraire. Plus précisément, ces règles sont capables de reconnaître des expressions simplement régulières de processus. Avec ce nouveau type de règle de réécriture, il est désormais possible d'encoder une sémantique de communications point-à-point (*send*/*receive*) mais également celle de communications globales (*broadcast*). Ces règles peuvent également être composées pour encoder la sémantique d'opérations plus complexes comme la réduction d'un ensemble de valeurs contenues dans chacun des processus (*reduce*). Nous ajoutons la contrainte que chaque processus doit posséder une information d'identité afin de pouvoir discriminer les processus et une information globale (à un mot) de la taille des processus pour pouvoir découvrir des invariants plus précis.

Avec cette nouvelle formalisation, nous avons défini un langage impératif disposant, entre autres, de communications point-à-point et globales et conservant les traits du précédent langage en ôtant les communications par voisinage. Nous avons donné un algorithme d'application sur un langage de ces règles de réécriture en donnant la preuve de sa correction et en analysons la complexité. La méthodologie pour construire une analyse reste la même. Finalement, nous avons proposé certaines optimisations applicables à ce modèle nous permettant d'atteindre une précision plus fine et de meilleures performances.

Dans le but d'éprouver notre modèle, nous avons considéré plusieurs applications possibles pour l'implantation d'un prototype. Nous avons porté notre choix vers la vérification de programmes `MPI`. `MPI` est une norme définissant une bibliothèque de primitives permettant d'exécuter des programmes de manière distribuée. Pour notre prototype, nous avons sélectionné un sous-ensemble de ces primitives (communications synchrones point-à-point, multi-point, barrières de synchronisation et opérations de réduction) et un sous-ensemble de `C` comme langage de programmation supporté. Nous étendons notre sous-ensemble de `MPI` traité avec deux nouvelles primitives fictives simulant la création et destruction de processus : la primitive `MPI` de création dynamique existante reposant sur des concepts de partitionnement d'ensemble de processus qui demanderait à l'heure actuelle un travail d'ingénierie conséquent. Notre prototype supporte également la possibilité de considérer un nombre symbolique de processus permettant d'obtenir des invariants indépendamment du nombre de processus utilisés. Ce prototype est entièrement automatisé et ne requiert pas de paramétrage particulier autre que de spécifier le nombre de processus à simuler (fixe ou non-borné) ou encore la précision de l'opérateur d'élargissement utilisé par les automates de treillis.

Notre prototype, `MPAI`[1] (*Message-Passing Abstract Interpreter*) prend la forme d'un greffon `Frama-C` [KKP+15]. Dans le chapitre 5, nous avons présenté les caractéristiques techniques principales de l'implantation de notre prototype. Nous avons montré la méthodologie employée pour traduire les différentes primitives retenues. Nous avons donné une description plus technique du domaine abstrait que nous employons et ses divers domaines numériques possibles paramétrables. Finalement, nous avons raffiné les stratégies d'itérations de notre modèle théorique en vue d'optimiser notre prototype pour une utilisation

1. `https://www-apr.lip6.fr/~botbol/mpai/`

efficace et donc réaliste.

Enfin, nous avons cherché à montrer les capacités et les performances de notre prototype. Nous avons rencontré quelques difficultés à comparer notre outil aux analyseurs statiques de `MPI` existants. En effet, notre prototype s'intéresse à calculer l'espace d'atteignabilité de programmes `MPI`. Les outils existants les plus robustes et éprouvés emploient une méthodologie basé sur le *model-checking* pour vérifier statiquement des propriétés de vivacité sur l'utilisation des communications dans un programme. L'exécution local des processus (calcul numérique, flot de contrôle local, etc.) est, quant à elle, déléguée à un *runtime* instrumenté et suppose ainsi un comportement correct et déterministe. Cette méthodologie est notamment employé par l'outil `ISP` [VVD+09] (*In-Situ Partial-order*). Malgré ces différences, nous avons pu ajouter à notre prototype la possibilité de découvrir des états atteignables pouvant conduire à des interblocages. Nous avons repris des exemples de la base de test de `ISP` et les avons adaptés pour correspondre au sous-ensemble de `MPI` que nous traitons. Nous montrons que nous obtenons, sur la majorité de ces exemples, des résultats satisfaisants en terme de performance tout en détectant les mêmes interblocages que ceux détectés par `ISP`. Nous nous montrons également capables de détecter des interblocages sur des programmes ayant un nombre « infini » de processus (*i.e.* sans borne supérieure du nombre de processus), ce que `ISP` ne permet pas de faire. Sur des programmes localement non-déterministes, également, nous obtenons toujours le "bon" résultat là où le résultat obtenu par `ISP`, qui repose sur une exécution dynamique, peut varier d'une exécution à l'autre.

En plus de la comparaison avec `ISP`, nous avons donné quelques exemples des possibilités de notre prototype notamment sur des programmes effectuant du calcul numérique et en y ajoutant des créations dynamiques de processus calculant des invariants numériques relationnels. Les expérimentations présentées illustrent le fait que notre analyse est particulièrement efficace lorsque les processus effectuent des tâches similaires ce qui correspond généralement à la méthodologie employée lors d'écriture de programmes `MPI` réalisant du calcul numérique. Le processus « maître » commence par distribuer des entrées à tous les processus « esclaves » qui effectuent un calcul équivalent où seuls les identifiants varient. Les comportements de ceux-ci peuvent alors facilement être abstraits par notre analyse en établissant des relations en fonction de leurs identifiants. Cela nous permet alors d'obtenir un invariant suffisamment précis pour prouver les propriétés recherchées.

7.2 Perspectives

Une des limitations actuelles de notre prototype est l'incapacité à considérer les tableaux et pointeurs. Ceci est important à ajouter puisqu'un grand nombre de programmes `MPI` les utilisent. La solution que nous avions commencé à implanter mais qui a nécessité un plus grand coût d'ingénierie qu'initialement supposé est l'interopérabilité avec le greffon `EVA`[BBY17] (*Evolved Value Analysis*) de `Frama-C`. `EVA`, successeur du greffon `Value`, est l'outil permettant de réaliser des analyses de valeur de programmes C par interprétation abstraite. Cette nouvelle version apporte notamment le support de domaines abstraits relationnels en utilisant de manière sous-jacente la bibliothèque de domaines abstraits `Apron`. Une des grandes forces de `EVA` est son abstraction efficace des zones mémoires rendant l'analyse de pointeurs et de tableaux très précises. Cependant, cette abstraction ne permet pas de découvrir de relations au sein des valeurs d'un tableau. Une première étape serait donc de terminer cette interopérabilité ce qui nous permettrait, en plus de supporter le langage `C` dans sa quasi totalité (tableaux inclus), de déléguer à `EVA` l'analyse locale de chaque processus et de n'avoir à considérer que les opérations

globales que notre modèle nous permet de traiter. De plus, cette coopération permettrait de réutiliser la détection automatique existante de RTE (*RunTime Error*) au sein de notre programme comme la détection des dépassements d'entiers, division par zéro ou encore des accès à des zones mémoires illicites ce qu'à l'heure actuelle ne peut être déduit que manuellement à partir de l'invariant généré.

Pour aller plus loin, nous souhaiterions également pouvoir disposer de relations au sein des abstractions de tableaux. À l'heure actuelle, l'utilisation de EVA ne permet pas de découvrir de relation entre scalaires du programmes et la valeur des cases de zone mémoire. Pour notre cas d'application, disposer de ce type de résultat permettrait d'augmenter sensiblement la précision d'une analyse où plusieurs processus travailleraient chacun sur des sous-parties non-partagées de tableaux en amenant des relations avec, par exemple, les identifiants des processus.

Actuellement, notre prototype ne considère qu'un sous-ensemble de primitives MPI essentielles mais pas suffisantes pour pouvoir traiter la majorité des programmes MPI existants sans avoir à réécrire une version équivalente avec les primitives de base que nous supportons. Un certain nombre d'entre-elles peuvent être considérées comme du sucre syntaxique (*e.g.* MPI_All_Reduce qui combine la réduction et l'envoi global du résultat) ce qui nous semble très direct à intégrer ou encore les primitives MPI_Scatter et MPI_Gather qui sont des communications globales de sous-parties de tableaux de données et où là encore la sémantique reste aisément encodable à travers notre formalisme. D'autres sont cependant plus difficiles à traiter. En particulier, les communications asynchrones nécessitent encore un travail théorique non-négligeable. Nos pistes pour traiter ce type de communications envisagent d'intégrer à notre modèle des files d'attentes de messages. La possibilité d'intégrer des files aux automates de treillis existe déjà [Le 08]. Nous pourrions alors formaliser une sémantique non-standard permettant de modéliser ce type de communications et ainsi d'intégrer les primitives à notre prototype.

Au sein de Frama-C, il est possible d'annoter les programmes analysés à l'aide du langage de spécification ACSL (*ANSI/ISO C Specification Langage*). Une fois définies, ces spécifications peuvent être validées à l'aide de greffons tels que WP (*Weakest Precondition*) ou même EVA. Un autre greffon, E-ACSL (*Executable ACSL*) permet, quant à lui, de transformer des programmes pour vérifier à l'exécution la validité des prédicats décrits par les annotations E-ACSL. L'ajout d'un langage de spécification permettrait de générer les mauvaises configurations définies par ces annotations et rendrait ainsi notre outil totalement automatique. Cependant, la réutilisation du langage de spécification ACSL serait suffisante pour encoder des propriétés locales aux processus mais insuffisantes pour en exprimer des globales. Il nous faudrait l'étendre pour pouvoir, par exemple, être capable de discriminer différents processus. D'autre part, les invariants que nous obtenons prennent la forme d'un langage régulier. ACSL n'est pas suffisamment expressif pour établir des propriétés sur de tels objets.

Une autre voie d'exploration possible serait d'appliquer notre modèle à d'autres formalismes et paradigmes de concurrence. Nous avions, dans un premier temps, considéré d'implanter notre prototype pour vérifier statiquement des programmes BSP[Val90] (*Bulk-Synchronous Parallel*). Il serait intéressant de concrétiser cette idée pour observer si le formalisme de BSP nous permettrait d'obtenir des résultats plus précis que dans le modèle plus général qu'est MPI. Par ailleurs, nous aimerions également étudier la possibilité d'appliquer ces résultats en vue de faciliter la construction d'une analyse de coût.

Un autre paradigme que nous souhaiterions également pouvoir traiter est le modèle à mémoire partagée. Pour cela, nous avons imaginé ajouter aux automates de treillis une notion de mémoire pouvant permettre de modéliser les états de variables partagées. La mémoire partagée serait considérée comme un processus particulier qui ne peut pas évoluer de manière autonome. L'opération de lecture/écriture

dans cette mémoire partagée serait alors modélisée comme une communication point-à-point avec ce processus particulier. Nous avons donc déjà tous les outils pour modéliser ce type de mémoire partagée ; cependant, le fait de rajouter beaucoup de règles de communication du fait de ces lectures/écritures va probablement nuire fortement à l'efficacité de l'analyse.

Une autre approche serait d'effectuer le produit de plusieurs analyses. Par exemple, cumuler notre méthodologie à celle utilisée par [Min14] sur les programmes concurrents utilisant les interférences pour découvrir les interactions entre processus légers `POSIX` par les variables partagées. Une manière de combiner ces deux analyses serait de traiter cette mémoire partagée de manière assez grossière dans notre analyse et de faire un produit réduit avec une analyse du type de celle de [Min14] pour récupérer plus de précision sur ces variables partagées. Une autre possibilité de combinaison de ces deux analyses serait de considérer des programmes parallèles « hybrides », c'est-à-dire utilisant localement les *threads* du processeur (ou autre unité de calcul locale) avec un modèle de mémoire partagée et utilisant un ensemble de nœuds de calcul distribués communiquant via `MPI`. Notre analyse permettrait de couvrir les interactions entre les nœuds et les calculs locaux utilisant des threads `POSIX` seraient couverts par l'analyse [Min14].

Bibliographie

[ABBM10] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the Verification Problem for Weak Memory Models. *ACM Sigplan Notices*, 45(1) :7–18, 2010. 21

[AdBO09] Krzysztof R. Apt, Frank de Boer, and Ernst-Rdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 3rd edition, 2009. 4

[ADMP01] Pablo Argón, Giorgio Delzanno, Supratik Mukhopadhyay, and Andreas Podelski. Model Checking Communication Protocols. In *Proceedings of the 28th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'2001)*, pages 160–170. Springer Berlin Heidelberg, 2001. 23

[AJNd02] Parosh Aziz. Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular Model Checking Made Simple and Efficient. In *Proceedings of 13th International Conference on Concurrency Theory (CONCUR'02)*, pages 116–130. Springer, 2002. 27, 28

[AJNS04] Parosh A. Abdulla, Bengt Jonsson, Marcus Nilsson, and Mayank Saksena. A Survey of Regular Model Checking. In *Proceedings of 15th International Conference on Concurrency Theory (CONCUR'04)*, pages 35–48. Springer, 2004. 8, 26, 45

[AK86] Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6) :307–309, 1986. 23

[BBY17] Sandrine Blazy, David Bühler, and Boris Yakobowski. Structuring Abstract Interpreters Through State and Value Abstractions. In *Proceedings of 18th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'17)*, pages 112–130. Springer, 2017. 81, 99

[BCC+02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. *Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software*, pages 85–108. Springer, 2002. 19

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-critical Software. *SIGPLAN Notices*, 38(5) :196–207, 2003. 19

[Bei90] Boris Beizer. *Software Testing Techniques (2nd Ed.)*. International Thomson Computer Press., 1990. 5

[Ber79] Jean Berstel. *Transductions and context-free languages*. Springer, 1979. 8

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract Regular Model Checking. In *Proceedings of 16th International Conference on Computer Aided Verification, (CAV'04)*, pages 372–386. Springer, 2004. 27, 28, 47

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular Model Checking. In *Proceedings of 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855, pages 403–418. Springer, 2000. 26

[BLW04] Bernard Boigelot, Axel Legay, and Pierre Wolper. Omega-Regular Model Checking. In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, pages 561–575, 2004. 28

[Bou93] François Bourdoncle. Efficient Chaotic Iteration Strategies With Widenings. In *International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993. 47, 82

[But97] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997. 23

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2) :323–342, 1983. 22, 23

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977. 6, 14, 16, 18, 68

[CCF+05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *Proceedings of 14th European Symposium on Programming Languages and Systems (PLS'05)*, pages 21–30. Springer, 2005. 19

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does Astrée scale up ? *Formal Methods in System Design*, 35(3) :229–264, 2009. 19

[CCF13] Agostino Cortesi, Giulia Costantini, and Pietro Ferrara. A survey on product operators in abstract interpretation. In Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff, editors, *Semantics, Abstract Interpretation, and Reasoning about Programs*, volume 129 of *Electronic Proceedings in Theoretical Computer Science*. Open Publishing Association, 2013. 18

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. *SIGPLAN Notices*, 46(1) :105–118, 2011. 81

[CCM11] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *International Conference on Foundations of Software Science and Computational Structures*, pages 456–472. Springer, 2011. 18

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999. 6

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of 5th ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96. ACM, 1978. 17

[CH09] Jean-Loup Carre and Charles Hymans. From Single-thread to Multithreaded : An Efficient Static Analysis Algorithm. *Computing Research Repository*, abs/0910.5833, 2009. 23, 24

[Dam93] Mads Dam. Model Checking Mobile Processes. In *Proceedings of 4th International Conference on Concurrency Theory (CONCUR'93)*, pages 22–36. Springer, 1993. 22

[Deu03] Alain Deutsch. Static verification of dynamic properties. *Polyspace white paper*, 2003. 20

[Dij83] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications ACM*, 26(1) :21–22, 1983. 20

[DKS13] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. Common specification language for static and dynamic analysis of c programs. In *Proceedings of 28th Annual ACM Symposium on Applied Computing (SAC'13)*, pages 1230–1235. ACM, 2013. 83

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transaction of Computer-Aided Design of Integrated Circuits and Systems*, 27(7) :1165–1178, 2008. 5

[Fer05] Jérôme Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming*, 63(1) :59–130, 2005. 7, 22, 23

[FL10] Manuel Fähndrich and Francesco Logozzo. Static Contract Checking with Abstract Interpretation. In *Proceedings of 2nd International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10)*, pages 10–30. Springer, 2010. 20

[For15] Message Passing Interface Forum. MPI : A Message-Passing Interface Standard. `http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`, 2015. 71

[GDD+04] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. *Numeric Domains with Summarized Dimensions*, pages 512–529. Springer, 2004. 81

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, 2004. 71

[GKS+11] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. De Supinski, Martin Schulz, and Greg Bronevetsky. Formal Analysis of MPI-based Parallel Programs. *Commun. ACM*, 54(12) :82–91, 2011. 71

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI : portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999. 71

[GP12] Gabriella Gigante and Domenico Pascarella. Formal Methods in Avionic Software Certification : The DO-178C Perspective. In *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, pages 205–215. Springer, 2012. 5

[GRS05] Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. *SIGPLAN Notices*, 40(1) :338–350, 2005. 81

[GS13] Laure Gonnord and Peter Schrammel. Abstract Acceleration in Linear relation analysis (extended version). Technical report, February 2013. 16

[Gur89] Eitan M. Gurari. *An introduction to the theory of computation*, volume 338. Computer Science Press Rockville, 1989. 27

[Hai17] Gaetan Hains. *Enumerated BSP Automata*, pages 233–267. Springer, 2017. 22

[Hal93] Nicolas Halbwachs. *Delay analysis in synchronous programs*, pages 333–346. Springer, 1993. 68

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Automata theory, languages, and computation. *International Edition*, 24, 2006. 8, 25, 26, 46

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering Properties About Arrays in Simple Programs. *SIGPLAN Notices*, 43(6) :339–348, 2008. 81

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. *SIGPLAN Notices*, 50(1) :247–259, 2015. 20

[Jon83] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS'83)*, 5(4) :596–619, 1983. 23

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C : A Software Analysis Perspective. *Formal Aspects of Computing*, 27(3), 2015. 9, 19, 71, 81, 98

[LDGK08] Guodong Li, Michael Delisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Formal Specification of the MPI-2.0 Standard in TLA+. In *Proceedings of 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*. ACM, 2008. 74

[Le 97] Gérard Le Lann. An analysis of the Ariane 5 flight 501 failure-a system engineering perspective. In *Workshop on Engineering of Computer-Based Systems (ECBS'97)*, pages 339–246, 1997. 2

[Le 08] Tristan Le Gall. *Abstract Lattices for the Verification of Systems with Queues and Stacks*. PhD thesis, Université de Rennes 1, 2008. 8, 29, 31, 100

[Ler03] Jérôme Leroux. *Algorithmique de la vérification des systèmes à compteurs : approximation et accélération, implémentation de l'outil FAST*. PhD thesis, 2003. Thèse de doctorat dirigée par Finkel, Alain Informatique Cachan, Ecole normale supérieure 2003. 16

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009. 20

[LGJJ06] Tristan Le Gall, Bertrand Jeannet, and Thierry Jéron. Verification of Communication Protocols using Abstract Interpretation of FIFO queues. In *Proceedings of 11th International Conference on Algebraic Methodology and Software Technology (AMAST '06)*, volume 4019 of *LNCS*, pages 204–219. Springer, 2006. 23

[LJ07] Tristan Le Gall and Bertrand Jeannet. Lattice Automata : A Representation for Languages on Infinite Alphabets, and Some Applications to Verification. In *Proceedings of 14th International Symposium on Static Analysis (SAS'07)*, pages 52–68. Springer, 2007. 29, 48

[LS07] Jérôme Leroux and Grégoire Sutre. Accelerated data-flow analysis. In *Static Analysis (SAS'07)*, pages 184–199. Springer-Verlag, 2007. 16

[LT93] Nancy G. Leveson and Clark Savage Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7) :18–41, 1993. 2

[Min06] Antoine Miné. The Octagon Abstract Domain. *Higher Order Symbolic Computation*, 19(1) :31–100, 2006. 17

[Min12] Antoine Miné. Static analysis by abstract interpretation of sequential and multi-thread programs. In *10th School Edition on Modelling and Verifying Parallel Processes (MOVEP'12)*, pages 35–48, 2012. 7, 23

[Min13] Antoine Miné. *Static analysis by abstract interpretation of concurrent programs*. Habilitation à diriger des recherches, Ecole Normale Supérieure de Paris - ENS Paris, 2013. 23

[Min14] Antoine Miné. Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In *Proceedings of 15th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'14)*, pages 39–58. Springer, 2014. 23, 101

[MM81] Ernst W. Mayr and Albert R. Meyer. The complexity of the finite containment problem for petri nets. *Journal of the ACM*, 28(3) :561–576, 1981. 22

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1) :1 – 40, 1992. 22

[Pet62] Carl A. Petri. Fundamentals of a theory of asynchronous information flow. In *International Federation for Information Processing, 2nd International congress (IFIP'62)*, pages 386–390, 1962. 22

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. 6

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982. 6

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2) :358–366, 1953. 5

[SCB13] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. Combining static and dynamic validation of mpi collective communications. In *Proceedings of the 20th European MPI Users' Group Meeting (EuroMPI '13)*, pages 117–122, New York, NY, USA, 2013. ACM. 71

[SOHL+98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1 : The MPI Core*. MIT Press, 2nd. edition, 1998. 9

[SZ11] Stephen F. Siegel and Timothy K. Zirkel. TASS : The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science*, 5(4) :395–426, 2011. 71

[TLHL17] Thibaut Tachon, Chong Li, Gaétan Hains, and Frédéric Loulergue. Automated generation of bsp automata. *Parallel Processing Letters*, 27(01) :1740002, 2017. 22

[Tou01] Tayssir Touili. Regular Model Checking using Widening Techniques. *Electronic Notes in Theoretical Computer Science*, 50(4) :342 – 356, 2001. 23

[Tre08] Jan Tretmans. *Model Based Testing with Labelled Transition Systems. Formal Methods and Testing*. Springer, 2008. 5

[Val90] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8) :103–111, 1990. 22, 100

[VHL+12] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjørner. Symbolic Finite State Transducers : Algorithms and Applications. In *Proceedings of 39th ACM Symposium on Principles of Programming Languages (POPL'12)*, pages 137–150. ACM, 2012. 28, 29, 38

[VVD+09] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal verification of practical MPI programs. *ACM Sigplan Notices*, 44(4) :261–270, 2009. 10, 71, 85, 99

[WW07] Christopher B. Watkins and Randy Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, 2007. 20

Résumé

La vérification de systèmes distribués est un problème complexe pour de nombreuses raisons tant théoriques que pratiques, en particulier lorsque ces systèmes sont capables d'effectuer localement des calculs numériques. Le but de cette thèse est de proposer une méthode de vérification formelle de tels systèmes.

Nous présentons un modèle général, basé sur l'interprétation abstraite, permettant de construire des analyses statiques pour des systèmes de processus communiquants. Notre méthodologie s'inspire du *Regular Model Checking* en représentant des ensembles d'états de programmes concurrents sous la forme d'automates de treillis et la sémantique de ces programmes comme la réécriture des langages reconnus par ces automates. Ce modèle nous permet notamment d'exprimer des communications entre processus et des créations/destructions dynamiques de processus. L'utilisation de l'interprétation abstraite nous permet d'obtenir une sur-approximation sûre de l'espace d'atteignabilité des programmes nous permettant ainsi de vérifier des propriétés de sûreté numériques. Nous avons implanté cette méthode permettant d'analyser automatiquement des programmes utilisant la bibliothèque logicielle de calculs distribués `MPI/C`.

Abstract

Verifying distributed systems is a difficult problem on both theoretical and practice levels, in particular when systems are capable of local numerical computations. The goal of this thesis is to provide a formal verification method of such systems.

We present a general model, based on abstract interpretation, allowing the construction of static analyses for systems of communicating processes. Our methodology is inspired by *Regular Model Checking* where the set of program states are represented as lattice automata and the program semantics are encoded using rewriting systems applied on the language recognized by the automata. This model offers the possibility of expressing communications between processes as well as dynamic creation/destruction of process. Using the abstract interpretation methodology, we are able to provide a sound over-approximation of the reachability set of programs allowing us to verify numerical safety properties. We implemented this method allowing us to automatically analyse programs that use the distributed computation library `MPI/C`.