Réduction paramétrée de spécifications formées d'automates communicants : algorithmes polynomiaux pour la réduction de modèles

Sébastien Labbé

UNIVERSITE PIERRE & MARIE CURIE
LA SCIENCE A PARIS

cea list

THÈSE DE DOCTORAT DE
L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité
INFORMATIQUE

Présentée par
M. Sébastien LABBÉ

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Réduction paramétrée de spécifications formées d'automates communicants : algorithmes polynomiaux pour la réduction de modèles

soutenue le 26/09/2007

devant le jury composé de :

M. Marc POUZET Professeur à l'Université Paris Sud XI	Directeur de thèse
M. Jean-Pierre GALLOIS Chercheur au CEA LIST	Encadrant de thèse
M. Jean-Claude FERNANDEZ Professeur à l'Université Joseph Fourier, Grenoble I	Rapporteur
M. Yves LE TRAON HDR, Maître de conférences à l'ENST Bretagne	Rapporteur
M. Marc AIGUIER Professeur à l'École Centrale Paris	Examinateur
Mme. Thérèse HARDIN Professeur à l'Université Pierre et Marie Curie, Paris VI	Examinateur

Université Pierre & Marie Curie - Paris 6
Bureau de la scolarité doctorat
Esc G, 2e étage
15 rue de l'école de médecine
75270 - PARIS CEDEX 06

Tél. Secrétariat : 01 42 34 68 35
Fax : 01 42 34 68 40
Tél. pour les étudiants de A à MF : 01 42 34 68 41
Tél. pour les étudiants de MG à Z : 01 42 34 68 51
E-mail : scolarite.doctorat@upmc.fr

Remerciements

Je tiens à remercier toutes les personnes qui ont contribué de diverses façons à l'aboutissement de ma thèse.

Tout d'abord, je n'oublierai pas la persévérance de Jean-Pierre Gallois (CEA LIST), qui avec la collaboration de Jean-François Tilman (Axlog Ingénierie), a été déterminant dans mon obtention d'un contrat de thèse au CEA. Je remercie aussi (et surtout) Jean-Pierre pour sa disponibilité, et son encadrement qui s'est avéré idéal pour moi, tant au niveau humain que scientifique.

Un grand merci à Marc Pouzet pour avoir accepté d'être mon directeur de thèse, pour avoir été disponible aux moments clés de la progression de mes travaux, et pour avoir contribué de façon significative à améliorer le manuscrit de thèse.

Je remercie très sincèrement Marc Aiguier, Jean-Claude Fernandez, Jean-Pierre Gallois, Thérèse Hardin, Yves Le Traon et Marc Pouzet de me faire l'honneur de participer au jury de ma soutenance ; merci particulièrement à Jean-Claude Fernandez et Yves Le Traon d'avoir accepté d'évaluer mon travail, et de m'avoir ainsi apporté nombre de commentaires constructifs.

Merci à François Terrier, chef du Laboratoire Logiciels pour la Sûreté des Procédés (CEA LIST), pour m'avoir accueilli dans son laboratoire durant ces trois ans.

Je salue tout particulièrement les personnes avec qui j'ai collaboré sur le projet Agatha du CEA LIST, pour leur gentillesse, leur convivialité et leurs compétences scientifiques de haut niveau : Daniel Mateus – envers qui je suis très reconnaissant pour les excellents souvenirs de nombreuses discussions (sérieuses ou non), et pour sa contribution importante à la formalisation de mes travaux – Arnault Lapitre et Laurent Philippe – merci pour votre aide précieuse sur le développement – Jean-

Pierre Gallois, Nicolas Rapin, Christophe Gaston et Diane Bahrami – merci pour vos relectures et vos conseils avisés quant à la rédaction – Alain Faivre, Jean-Yves Pierron, Céline Bigot, Assia Touil, Thomas Maître, Aurélien Ohayon, Mathieu Vidalies et Cyril Grepet. Merci à nouveau à Alain et Arnault pour m'avoir permis d'entrer au CEA à l'origine, et à Christophe pour m'avoir aiguillé sur la possibilité d'enseigner à l'Université d'Évry.

Je salue aussi toutes les personnes que j'ai côtoyé durant ces dernières années au CEA, et qui ont contribué à un cadre de travail des plus agréables, et notamment Safouan Taha, Frédéric Loiret, Nicolas Ravot, Sébastien Revol, Huascar Espinoza, Julio Medina, Arnaud Cuccuru, Frédéric Thomas, François Lagarde, Amanda Leblan, Ansgar Radermacher, Yann Tanguy, Sébastien Gérard, Joëlle Jouanest, Magali Genau, Annie Straboni, Jan Stransky, Patrick Vanuxeem, Vincent David, Géraud Canet et toutes les personnes du Service Outils Logiciels.

Salutations à Alex Summers, Jayshan Raghunandan, Mila Dalla Preda, Therrezinha Fernandes, Jim Woodcock, Augusto Sampaio, Ana Cavalcanti, Dominique Méry, Ridha Khedri, Mark Lawford, Pascale Le Gall, John Hatcliff, Venkatesh P. Ranganath et Jens Krinke.

Toute ma gratitude va à Jean-Michel et Claudine – sans qui, bien entendu, cette thèse n'existerait pas – ainsi qu'à tous les membres de ma famille, notamment Jean-Pierre, Maéva et Charline, Gisèle et Albert, Janine et Raymond.

Je salue et remercie Jean-Frédéric, Jean-Paul, Bruno et Meryem, Christèle et Romain, Brice, Bruno, Karl, Jeff, Franck, Ben, Johan, Vincent, Thomas et Magali, Maleck et Jessica, Christophe et Aurélie, Félix et Diane, Jérome, Laïka, Ursula, Mickaël, Thibault, Charles et Johanna, Marie-Claude, Floriane et Ludwig, Meng, Ludivine, Fabrice, Guillaume, Johan, Aurélien, Anne-Su, Sat, Arnaud, Thomas et Sophie, Thérèse et Jean-Pierre, Olivier, Émilie, pour leur amitié et tous les bons moments passés et à venir.

À Caroline.

Résumé

Laboratoire d'accueil CEA, DRT/LIST/DTSI, Service Outils Logiciels, Laboratoire Logiciels pour la Sûreté des Procédés, Saclay, F-91191 Gif-sur-Yvette, France.

Résumé Les travaux décrits dans ce manuscrit de thèse s'inscrivent dans le cadre des méthodes formelles pour les langages de spécifications formées d'automates communicants. Ce type de langage est largement utilisé dans les industries de pointe où le niveau de fiabilité requis est élevé (e.g. aéronautique, transports), car ils permettent d'améliorer la précision des spécifications et d'exploiter des outils de simulation, de test ou de vérification qui contribuent à la validation des spécifications. Un frein au passage à l'échelle industrielle de ces méthodes formelles est connu sous le nom de l'explosion combinatoire, qui est due à la fois à la manipulation de larges domaines numériques, et au parallélisme intrinsèque aux spécifications.

L'idée que nous proposons consiste à contourner ce phénomène en appliquant des techniques de réduction paramétrée, pouvant être désignées sous le terme anglo-saxon "*slicing*", en amont d'une analyse complexe. Cette analyse peut ainsi être effectuée *a posteriori* sur une spécification réduite, donc potentiellement moins sujette à l'explosion combinatoire. Notre méthode de réduction paramétrée est basée sur des relations de dépendances dans la spécification sous analyse, et est fondée principalement sur les travaux effectués par les communautés de la compilation et du *slicing* de programmes. Dans cette thèse nous établissons un cadre théorique pour les analyses statiques de spécifications formées d'automates communicants, dans lequel nous définissons formellement les relations de dépendances mentionnées ci-dessus, ainsi que le concept de *tranche* de spécification par rapport à un *critère* de réduction. Ensuite, nous décrivons et démontrons les algorithmes efficaces que nous avons mis au point pour calculer les relations de dépendances et les tranches de spécifications, et enfin nous décrivons notre mise en œuvre de ces algorithmes dans l'outil Carver, pour la réduction paramétrée de spécifications formées d'automates communicants.

Mots clés Réduction paramétrée de modèles, slicing, spécifications, automates communicants, analyse statique, analyse de flot de données, algorithmes polynomiaux.

Title Slicing Communicating Automata Specifications: Polynomial Algorithms for Model Reduction.

Abstract This PhD thesis is concerned with formal methods for languages of communicating automata specifications. In the industry, this kind of languages is mainly used in fields where the reliability requirements are high (e.g. aeronautical, transportation industries), as a mean of improving the precision of specifications, and exploiting simulation, testing and verification tools, for the purpose of specification validation. Still, on large scale industrial specifications, formal methods suffer from the combinatorial explosion phenomenon; this is notably due to the manipulation of wide numerical domains, and the specifications inner parallelism.

In our contribution, we suggest to bypass this phenomenon, in applying *slicing* techniques before the targeted complex analysis. This analysis can thus be performed *a posteriori* on a reduced (or *sliced*) specification, which is potentially less exposed to combinatorial explosion. Our slicing method is based on dependence relations, defined on the specification under analysis, and is mainly founded on the literature on compiler construction and program slicing. In this thesis, we state a theoretical framework for static analyses of communicating automata specifications, in which we formally define the aforementioned dependence relations, together with the concept of a *slice* of a specification with respect to a slicing *criterion*. Then, we describe and prove the efficient algorithms that we designed for calculating dependence relations and specification slices. Finally, we describe our implementation of these algorithms in the CARVER tool, for slicing communicating automata specifications.

Keywords Slicing, model reduction, specifications, communicating automata, static analysis, dataflow analysis, polynomial algorithms.

Sommaire

Chapitre 1

Introduction

Sommaire

1.1 Contexte et objectifs

Depuis plusieurs décennies, les acteurs de la recherche se sont évertués à relever des défis scientifiques et technologiques d'ampleur toujours plus importante, notamment dans les domaines de l'aérospatiale, de l'aéronautique et du nucléaire. On conçoit aisément que le coût d'une expérience dans un de ces domaines peut s'avérer considérablement élevé, aussi bien au niveau humain que matériel. Les impératifs économiques et humains ont ainsi imposé aux scientifiques de développer leur capacité à évaluer de façon précise les chances de succès de leurs expériences. Cette capacité a été notamment développée à travers l'étude de la *fiabilité*, en tant que méthode de mesure de la sûreté de fonctionnement[1]. Après plus de vingt ans d'expérimentation, cette démarche a fait ses preuves dans les domaines de l'aérospatiale

[1] Plus précisément, définition de la fiabilité donnée par l'Académie : "*Grandeur caractérisant la sûreté de fonctionnement, ou mesure de la probabilité de fonctionnement d'un appareillage selon des normes prescrites*" [Fou93].

[RHTR06], de l'aéronautique [ZRI+00] et du nucléaire [WL03] ; elle s'est imposée plus tardivement dans d'autres industries comme l'automobile [Pho01], cette dernière ayant eu tendance à considérer dans un premier temps que les approches développées par les industries de pointe citées précédemment, ne s'appliquent pas dans le domaine de la production en grande série. Aujourd'hui, l'omniprésence de l'informatique dans notre vie quotidienne (e.g. dans les secteurs des télécommunications, des transports et du médical) accroît encore la prépondérance de la sûreté de fonctionnement des logiciels : même si dans ce contexte des vies humaines ne sont pas toujours en jeu, les implications économiques, voire juridiques, de défaillances logicielles dans un produit sont potentiellement catastrophiques pour leurs concepteurs et/ou distributeurs. En effet, qu'adviendrait-il d'une banque qui perdrait subitement tous ses moyens de traitement d'information, d'une société commercialisant des téléphones cellulaires dont le système d'exploitation s'avère défaillant ?

Face à cette problématique, les *méthodes formelles* – méthodes fondées sur la logique mathématique – présentent des avantages indéniables. D'une part, la syntaxe et la sémantique des langages de spécification formels étant définies mathématiquement, ces langages permettent de spécifier un système de manière non ambiguë et non contradictoire ; d'autre part, les outils formels habituellement associés à ces langages sont basés sur des cadres théoriques (e.g. analyse statique) et sur des algorithmes dont la correction, et aussi de préférence la complétude et la complexité, peuvent être démontrées mathématiquement. Une fois qu'un système est spécifié formellement à partir des exigences de son commanditaire (i.e. à partir d'un cahier des charges), et réalisé (e.g. à l'aide de techniques de génération automatique de code), l'évaluation de la correction globale du système se découpe en deux activités différentes et complémentaires : la *validation* consiste à s'assurer que la spécification formelle exprime bien le système souhaité (e.g. en employant des méthodes de test et de simulation), et la *vérification* consiste à s'assurer que le système réalisé se comporte conformément à la sémantique de la spécification formelle (e.g. preuve de propriétés). Ainsi, les méthodes formelles permettent de spécifier et de développer un système tout en permettant de s'assurer que les niveaux requis de fiabilité, de sûreté, *etc.* sont atteints.

Cependant, comme les activités de validation et de vérification peuvent nécessiter l'examen de toutes les exécutions possibles du système, elles sont confrontées au phénomène de l'*explosion combinatoire* – ce phénomène correspond à la démultiplication du nombre de comportements possibles d'une partie d'un système, par exemple lorsqu'elle est exécutée de façon concurrente avec une autre partie de la spé-

cification. Il en résulte une complexité généralement élevée des méthodes formelles, ce qui constitue un frein au passage à l'échelle industrielle de ces méthodes.

Les langages de spécification à base d'automates communicants, comme UML Statecharts, SDL, Stateflow charts, sont bien adaptés pour spécifier les systèmes informatiques produits par l'industrie, toujours plus complexes et parallélisés. L'approche AGATHA [Lap02, Pie03, Rap04, RGLG03, BFG+03a, LBV+04] est une solution proposée par le CEA LIST[2], pour la validation de systèmes à base d'automates communicants. Cette approche est fondée sur l'*exécution symbolique* des systèmes sous analyse. L'idée qui sous-tend l'exécution symbolique part du constat qu'habituellement, il y a dans une spécification des traitements identiques qui sont prévus pour plusieurs entrées différentes. Autrement dit, les données d'entrée ne se distinguent pas toutes du point de vue des traitements que la spécification leur applique. En somme, les traitements d'une spécification induisent des classes d'équivalences sur les entrées, selon la relation "induit les mêmes traitements que". Le graphe d'exécution symbolique, produit par ce calcul, représente l'ensemble des comportements du système : dans ce graphe, chaque chemin symbolique représente une classe d'équivalence de chemins numériques. Le caractère compact et exhaustif de cette représentation est avantageux dans l'optique de la validation et de la vérification.

Les travaux présentés dans ce manuscrit ont vocation à être complémentaires du projet AGATHA. De façon synthétique, le but de ce projet est de travailler sur l'analyse et la validation de systèmes réactifs, éventuellement embarqués, qui proviennent des industries de pointe évoquées précédemment. Les spécifications formées d'automates communicants qui modélisent ces systèmes sont particulièrement sujettes au phénomène d'explosion combinatoire, de par leur caractère concurrent ainsi que par la quantité et le domaine des données manipulées. En outre, les systèmes réactifs étant spécifiés pour interagir indéfiniment, ils peuvent avoir un nombre infini de comportements, même symboliques. Aussi, l'approche AGATHA propose des procédures de réduction qui interviennent dans la construction du graphe d'exécution symbolique. Néanmoins dans certains cas (e.g. systèmes de taille industrielle), la complexité de la technique mise en œuvre demeure bloquante, soit parce que le graphe produit est trop volumineux pour permettre une exploitation par l'utilisateur, soit à cause de temps de réponse trop important.

[2]CEA : Commissariat à l'Énergie Atomique / LIST : Laboratoire d'Intégration des Systèmes et des Technologies.

1.2 Contributions de la thèse

La solution proposée dans cette thèse consiste à réduire les spécifications en amont d'analyses formelles complexes, telles que l'exécution symbolique avec AGATHA (mais on pourrait aussi citer la vérification de propriétés avec un model checker). Cette réduction s'opèrera en fonction de critères bien choisis vis-à-vis de l'application souhaitée. Pour cela, nous nous appuyons sur l'état de l'art des méthodes de réduction paramétrée (ou *slicing*), qui ont été introduites à l'origine pour réduire des programmes impératifs (et plus précisément une représentation de ces programmes, nommée *graphe de flot de contrôle*), afin d'en faciliter la compréhension et le débogage. Nous verrons que les caractéristiques des spécifications formées d'automates communicants nécessitent des solutions adaptées, dans l'optique de la réduction paramétrée – notamment l'absence de point de sortie unique, l'occurrence de communications sur les canaux.

Le manuscrit de thèse est organisé comme suit. La fin de ce chapitre 1 est consacrée à la définition des graphes de flot de contrôle et à l'introduction du concept original de réduction paramétrée. Le chapitre 2 décrit le cadre théorique des analyses de flot de données, que nous utiliserons comme fondement des techniques de réduction paramétrée. Le chapitre 3 décrit la syntaxe et la sémantique du formalisme des automates communicants, sur lequel sont basés tous les travaux de cette thèse. Les chapitres 4 et 5 sont consacrés à la description précise de notre solution. Plus précisément, le chapitre 4 présente l'analyse de dépendances dans les spécifications formées d'automates communicants. Ce chapitre forme la pierre angulaire de notre méthode de réduction paramétrée, décrite au chapitre 5. Les analyses de dépendances, ainsi que notre méthode proprement dite de réduction paramétrée sont décrites par des définitions formelles et des algorithmes, qui s'inscrivent dans le cadre théorique des chapitres 2 et 3. Les chapitres 4 et 5 comprennent aussi les démonstrations de la complexité polynomiale de ces algorithmes, et de leur correction et complétude par rapport aux définitions correspondantes ; de plus, ils complètent l'état de l'art sur la réduction paramétrée, initié en section 1.4. Enfin, le chapitre 5 présente la mise en œuvre que nous avons faite de toutes les techniques décrites dans ce manuscrit, sous la forme d'un outil logiciel (nommé CARVER). On précise que le contenu des chapitres 3, 4 et 5 a fait l'objet d'une publication courte et d'une présentation au *Doctoral Symposium* de la conférence *Formal Methods 2006*[3] [LG06], d'une présentation et d'une soumission pour les actes de la conférence *International*

[3] À cette occasion, les organisateurs ont décerné à l'auteur le prix *FACJ Outstanding Young Researcher on Formal Methods Award.*

Symposium on Leveraging Applications of Formal Methods 2006 [LL06], d'une publication et d'une présentation lors de la conférence *Australian Software Engineering Conference 2007* [LGP07b], ainsi que d'une soumission au journal *Formal Aspects of Computing* [LG07]. Au moment de finaliser ce manuscrit, une version longue du papier [LGP07b] vient d'être soumise au journal *Journal of Software* [LGP07a], et un brevet est en cours de dépôt[4]. De façon usuelle, le manuscrit se terminera par une conclusion, qui récapitule les contributions de nos travaux.

1.3 Graphe de flot de contrôle (CFG)

Pour analyser un programme, un travail important consiste à comprendre le *flot de contrôle* – les suites d'exécutions possibles dans le programme. Ce travail est relativement simple pour des programmes écrits dans un langage bien structuré. Cependant, si des constructions comme les instructions de branchement (`goto` ...) sont autorisées dans le programme, cela introduit des sauts quelconques dans le contrôle du programme, que l'on appelle alors programme non-structuré, et l'identification des flots de contrôle possibles devient non-triviale. Pour cette raison, on utilise souvent une représentation à base de graphes.

Cette approche consiste à associer à chaque programme un graphe, dénommé *graphe de flot de contrôle*, CFG en abrégé[5] [AK02, Muc97], dans lequel les nœuds représentent les affectations du programme, et les arcs représentent le flot de contrôle entre ces nœuds.

Définition 1.1 (Graphe de flot de contrôle) *Un* graphe de flot de contrôle *est un graphe orienté étiqueté* $G = (X, N, A, n_e, n_s)$ *:*

- X *est l'ensemble des variables qui interviennent dans* G *;*
- N *est l'ensemble des nœuds de* G *;*
- A *est l'ensemble des arcs de* G *;*
- *Les affectations et conditions sont représentés par les nœuds* $n \in N$ *et le flot de contrôle entre les nœuds de* N *est représenté par les* arcs de flot de contrôle $(n, m) \in A$*, où* $n \in N$ *et* $m \in N$ *;*
- *Deux nœuds spéciaux sont distingués : le nœud* ENTRÉE*, noté* n_e*, et le nœud* SORTIE*, noté* n_s*. Ces nœuds représentent respectivement le début et la fin du programme.*

[4] Le dépôt du brevet BD1775 (MCF 9799) "Réduction paramétrée de modèles d'automates communicants" à été demandé par le CEA en avril 2007.

[5] CFG : *Control Flow Graph*.

Notations

- Soit (n, m) un arc dans A, alors on dira que n est un prédécesseur de m, et m est un successeur de n.
- On note $C_G[n, m]$ l'ensemble de tous les chemins finis dans G allant de n à m , et $C_G[n, m[$ l'ensemble de tous les chemins finis dans G allant de n à un prédécesseur de m.
- On note $c = [n_1, \ldots, n_k]$ un chemin $c \in C_G[n_1, n_k]$ tel que $\forall\ 1 \leq i \leq k, n_i \in N$ et $\forall\ 1 \leq j < k, (n_j, n_{j+1}) \in A$.
- Soit E un ensemble, on note $\wp(E)$ l'ensemble des parties finies de E, et on note $|E|$ le nombre d'éléments (ou cardinal) de E.

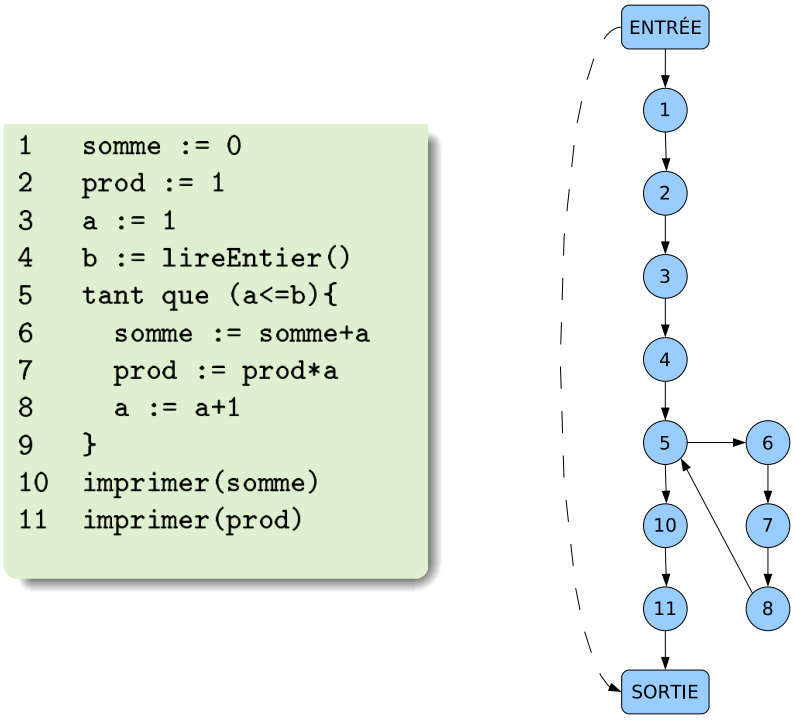


FIG. 1.1 – Exemple de programme et le CFG correspondant.

Exemple 1.2 La figure 1.1 montre un petit programme et son graphe de flot de contrôle. Les nœuds sont désignés par le numéro de ligne correspondant à l'instruction représentée. Le programme en question calcule, étant donné une entrée b, la somme et le produit des b premiers entiers.

Définition 1.3 (Définitions, utilisations) *Soit $G = (X, N, A, n_e, n_s)$ un graphe de flot; les ensembles def_n et ref_n, pour tout n dans N, sont définies par :*

- *ref_n est l'ensemble des variables utilisées au nœud n ;*
- *def_n est l'ensemble des variables définies (autrement dit, sur lesquelles on opère une affectation) en n.*

Exemple 1.4 Reprenons l'exemple de la figure 1.1. Le nœud `2` représente l'affectation `prod=1` ; le nœud `6` représente l'affectation `somme:=somme+a` ; le nœud `8` représente l'affectation `a:=a+1`. Le tableau de la figure 1.2 donne alors la valeur des ensembles def_n et ref_n, pour $n \in \{2, 6, 8\}$.

n	def_n	ref_n
2	{prod}	{}
6	{somme}	{somme, a}
8	{a}	{a}

FIG. 1.2 – def_n et ref_n, pour $n \in \{2, 6, 8\}$ dans le CFG de la figure 1.1.

Remarques Comme on peut le constater dans l'exemple 1.4, pour un nœud n donné, il se peut qu'une variable x soit simultanément dans les deux ensembles def_n et ref_n. Selon la sémantique usuelle des programmes, on considère dans ce cas que le nœud n correspond consécutivement à une utilisation puis une définition de la variable x.

On peut relier plus formellement les deux fonctions définies ci-dessus aux nœuds du graphe en étendant la définition du CFG à $G = (X, N, A, n_e, n_s, \mu)$, où $\mu : N \mapsto \wp(X) \times \wp(X)$ est une fonction qui fait la correspondance entre les nœuds et leurs étiquettes [Kri03]. On la définit ainsi : $\mu(n) = (def_n, ref_n)$.

Exemple 1.5 À nouveau, reprenons l'exemple de la figure 1.1. Pour donner $\mu(6)$, on peut écrire soit $\mu(6) = (\{\texttt{somme}\}, \{\texttt{somme}, \texttt{a}\})$.

1.4 Réduction paramétrée de programmes : *program slicing*

Le concept de *réduction paramétrée de programmes* (ou *program slicing*) fut introduit par Weiser [Wei81] en tant que méthode pour abstraire un programme par

rapport à des points d'intérêt dans ce même programme, ces points particuliers formant un *critère de réduction*. Dans ce cadre d'analyse, un programme est abstrait par une selection, dans le programme lui-même, des éléments qui ont une influence potentielle sur le critère. Les éléments ainsi sélectionnés forment un programme appelé *tranche arrière*, qui sera souhaitablement plus petit et plus facile à comprendre que le programme original. En effet, on attend d'un algorithme de réduction paramétrée qu'il retire un maximum des éléments qui n'ont aucune relation avec le critère. De manière analogue, un programme formé à partir d'une sélection, dans le programme lui-même, des éléments potentiellement influencés par le critère, est appelé *tranche avant*.

Dans un de ses travaux, Weiser [Wei82] indique que les programmeurs construisent mentalement des tranches d'un programme lorsqu'ils déboguent celui-ci, et par conséquent qu'un outil capable d'extraire automatiquement des tranches d'un programme – un *slicer* – permettrait d'améliorer l'efficacité et la sûreté du processus de débogage de programmes. Les travaux sus-nommés ont apporté la première principale motivation au développement d'un tel outil. Depuis lors, des techniques de réduction paramétrée ont été développées pour prendre en compte des constructions et structures de données plus complexes, comme les procédures [HRB88], les flots de contrôle non-structurés [BH93], les tableaux et pointeurs [Kri03], et la concurrence [HCD+99, Kri98, NR00, MT00]. Des techniques de réduction paramétrée ont aussi été proposées pour des langages ou formalismes plus modernes, comme le langage Z [CR94, BW05], les langages synchrones [GR02], et les machines à états hiérarchiques [HW97, WDQ03]. Aussi dérivé des travaux de Weiser, le concept de réduction paramétrée *dynamique* [KL88] consiste à réduire un programme, non seulement par rapport à un critère, mais aussi par rapport à une exécution particulière (ou un ensemble d'exécutions) du programme. Plusieurs auteurs ont publié leur version de l'état de l'art sur les techniques de réduction paramétrée [Tip95, XQZ+05]. En outre, il a été démontré que l'utilisation de techniques de réduction paramétrée peut être bénéfique dans des domaines de recherche importants, comme par exemple le model checking [DHH+06]. La quasi-totalité des travaux, dont nous connaissons l'existence, qui permettent de mettre en œuvre la réalisation d'une méthode de réduction paramétrée (indépendamment du langage ou formalisme sur lequel cette méthode opère), peuvent être regroupés dans l'une des deux grandes classes d'approches que nous allons introduire dans le reste de cette section.

1.4.1 Réduction paramétrée *à la* Weiser

La première approche peut être appelée réduction paramétrée *à la Weiser* (*Weiser-style* slicing, comme le suggère Krinke dans [Kri03]). La méthode consiste à résoudre des équations qui définissent des ensembles de variables et d'éléments du programme qui sont pertinents pour un critère de réduction donné. Les équations peuvent être résolues par recherche incrémentale d'un point fixe pour l'ensemble des parties pertinentes du programme ; lorsqu'il est trouvé, ce point fixe constitue la tranche recherchée.

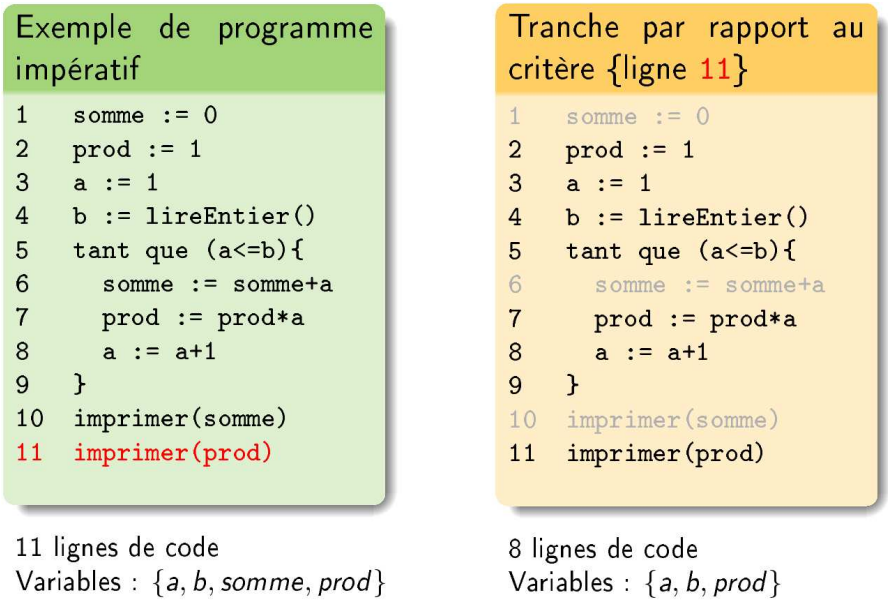


FIG. 1.3 – Exemple de réduction paramétrée de programme *à la* Weiser.

Exemple 1.6 Le lecteur pourra se référer à l'exemple de la figure 1.3, où le programme sur la gauche est le programme que l'on a déjà pris en exemple dans la section 1.3, et le programme sur la droite est une tranche arrière, extraite du programme original, par rapport au critère {ligne 11}. L'exemple de la figure 1.3 montre bien l'intérêt quantitatif que peut avoir cette méthode de réduction paramétrée, puisque la tranche calculée est plus petite que le programme original, en terme de nombre de variables, ainsi qu'en terme de nombre de lignes de code. D'autre part, cet exemple illustre aussi l'intérêt sémantique de la réduction paramétrée : ici, le critère correspond à la "sortie" du calcul du produit, or on voit bien que dans la tranche résultante tous les calculs indépendants sont retirés (notamment ceux qui interviennent seulement dans le calcul de la somme). Par conséquent la réduction

paramétrée permet sur cet exemple de focaliser le programme, de manière sûre, sur une fonctionalité donnée.

1.4.2 Réduction paramétrée basée sur des relations de dépendances

La seconde principale approche, que nous désignerons par réduction paramétrée *basée sur des relations de dépendances*, fait intervenir le calcul de relations de dépendances entre les différentes parties d'un programme. Dans ce type d'approche, une représentation additionnelle du programme – un *graphe de dépendances* – est usuellement contruite à partir des relations de dépendances. Dans un graphe de dépendances, les nœuds représentent des éléments du programme (e.g affectations, conditionnelles), tandis que les arcs représentent les relations de dépendances : deux éléments du programme sont mis en relation par une dépendance si et seulement si il existe un arc reliant les nœuds correspondants dans le graphe de dépendances. Lorsque toutes les relations de dépendances sont transitives[6], le problème de la réduction paramétrée se ramène à un problème d'atteignabilité dans le graphe de dépendances [Kri03, FOW87]. Les tranches obtenues de cette façon sont plus précises que les tranches obtenues par le biais des précédentes méthodes [FOW87]. Il existe dans la littérature différentes définitions de graphes de dépendances, chacune adaptée pour les utilisations souhaitées par leurs auteurs, et pour la plupart dérivées du *Program Dependence Graph* d'Ottenstein (PDG) [OO84, FOW87] ; citons le *System Dependence Graph* d'Horwitz *et al.* [HRB88], le *Threaded Program Dependence Graph* de Krinke [Kri98], le *Parallel Program Graph* de Sarkar [Sar97], et le *Multithreaded Dependence Graph* de Zhao *et al.* [ZCU96].

Exemple 1.7 La figure 1.4 reprend le programme de la figure 1.1, et montre une tranche de ce programme par rapport au critère {nœud `11`}. Par rapport à l'exemple de la figure 1.3, cet exemple montre donc une autre manière de calculer la même tranche, cette fois-ci par l'intermédiaire d'une méthode basée sur des relations de dépendances. Les nœuds et arcs non inclus dans la tranche sont représentés en pointillés ; la tranche ainsi calculée est $S(\texttt{11}) = \{\texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{7}, \texttt{8}, \texttt{11}\}$.

La réduction paramétrée basée sur des relations de dépendances est un principe très adaptable : on peut définir les relations de dépendances appropriées pour le langage ciblé par l'approche, puis construire un graphe de dépendances (comme

[6] Transitivité : notion de précision des relations de dépendances, sur laquelle nous reviendrons au chapitre 5.

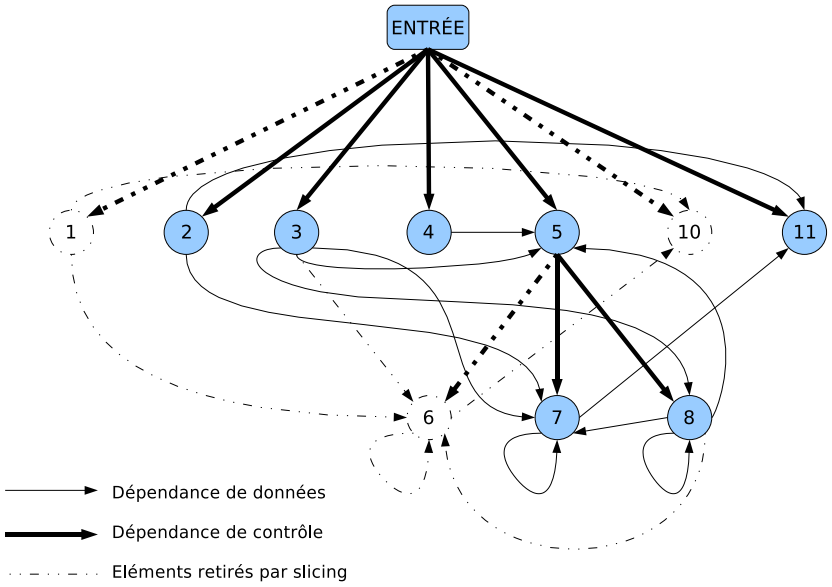


FIG. 1.4 – PDG du programme de la figure 1.1, et tranche par rapport au critère {nœud `11`}.

illustré par la figure 1.4), et finalement calculer des tranches par résolution de problèmes d'atteignabilité dans le graphe de dépendances. Le précision d'une approche de réduction paramétrée basée sur des relations de dépendances dépend fortement de deux facteurs : les définitions des relations de dépendances, et les caractéristiques du langage ciblé. Par exemple, dans le contexte de la réduction paramétrée de programmes concurrents, Krinke [Kri98] a introduit la notion de *dépendance d'interférence*, pour dénoter les dépendances qui sont induites par les lectures et les écritures concurrentes de variables partagées, et la notion de *dépendance transitive*, pour évaluer la précision des relations de dépendances.

Grâce à son adaptabilité, sa précision et sa relative simplicité pour la compréhension et l'implémentation, le principe de réduction paramétrée basée sur des relations de dépendances a souvent été mis en œuvre dans la littérature, en particulier lorsqu'il s'agit de prendre en compte des constructions complexes [Kri03, NR00, BH93, HRB88, RAB+05, ZCU96].

Chapitre 2

Analyse de flot de données

Ce chapitre introduit le cadre théorique des analyses de flot de données, que nous utiliserons comme fondement des techniques de réduction paramétrée.

Sommaire

2.1 Introduction

Une analyse de flot de données (ou analyse *dataflow*) est une forme traditionnelle d'analyse statique de programmes, qui a pour objectif de trouver des informations sur le comportement du programme analysé, sans effectivement l'exécuter. Dans ce chapitre, la présentation adoptée ne consiste pas à analyser des programmes directement, mais plutôt leur graphe de flot de contrôle (CFG, cf. section 1.3 page 15). Pour résoudre un problème spécifique de flot de données, il n'est pas forcément nécessaire de considérer la sémantique complète du système ; l'idée est donc de considérer une sémantique abstraite, et plus simple, adaptée au problème (sur ce plan, on peut dire que les analyses de flot de données et l'*interprétation abstraite*[1] [CC77] sont des sujets connexes).

2.2 Notions de théorie des treillis

Dans cette section, nous décrivons les notions de théorie des treillis qui permettent de définir un cadre formel pour fonder les analyses de flot de données.

Définition 2.1 (Ordre partiel) *Un ordre partiel sur un ensemble $\mathcal{L}$ est une relation $\sqsubseteq$ entre des couples d'éléments de $\mathcal{L}$:*

- *Réflexive :* $\forall l \in \mathcal{L}, l \sqsubseteq l$;
- *Antisymétrique :* $\forall l_1, l_2 \in \mathcal{L}, l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$;
- *Transitive :* $\forall l_1, l_2, l_3 \in \mathcal{L}, l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$

Nous allons définir les bornes inférieure et supérieure de deux éléments sur un ensemble muni d'un ordre partiel. Intuitivement, la borne inférieure de deux éléments l_1 et l_2 est un minorant de ces deux éléments, et un majorant de tous les autres minorants de l_1 et l_2 (autrement dit, le plus grand minorant de l_1 et l_2) ; la borne supérieure de deux éléments l_1 et l_2 est un majorant de ces deux éléments, et un minorant de tous les autres majorants de l_1 et l_2 (autrement dit, le plus petit majorant de l_1 et l_2).

Définition 2.2 (Borne inférieure, borne supérieure) *Soit $\mathcal{L}$ un ensemble muni d'un ordre partiel $\sqsubseteq$, soient $l_1, l_2 \in \mathcal{L}$:*

- *Soit $m \in \mathcal{L}$ tel que $m \sqsubseteq l_1$ et $m \sqsubseteq l_2$. Si $\forall l \in \mathcal{L}, l \sqsubseteq l_1 \wedge l \sqsubseteq l_2 \Rightarrow l \sqsubseteq m$ alors m est la borne inférieure de (l_1, l_2) ;*

[1] L'interprétation abstraite est une théorie de l'approximation discrète de sémantiques de systèmes informatiques principalement utilisée pour l'analyse et la vérification statique de logiciels.

- *Soit* $M \in \mathcal{L}$ *tel que* $l_1 \sqsubseteq M$ *et* $l_2 \sqsubseteq M$. *Si* $\forall l \in \mathcal{L}, l_1 \sqsubseteq l \wedge l_2 \sqsubseteq l \Rightarrow M \sqsubseteq l$ *alors* M *est la borne supérieure de* (l_1, l_2).

On remarque que par définition, pour chaque couple d'éléments, la borne supérieure et la borne inférieure sont uniques.

Définition 2.3 (Treillis) *Un treillis* $\mathcal{T}$ *est un uple* $(\mathcal{L}, \sqsubseteq, \sqcap, \sqcup)$ *où :*

- $\mathcal{L}$ *est un ensemble muni de l'ordre partiel* $\sqsubseteq$*, tel que tout couple d'éléments de* $\mathcal{L}$ *possède une unique borne inférieure et une unique borne supérieure ;*
- $\sqcap$ *et* $\sqcup$ *sont des opérations sur les éléments de* $\mathcal{L}$ *telles que pour tous* $l_1, l_2 \in \mathcal{L}$*,* $l_1 \sqcap l_2$ *est la borne inférieure de* (l_1, l_2)*, et* $l_1 \sqcup l_2$ *est la borne supérieure de* (l_1, l_2).

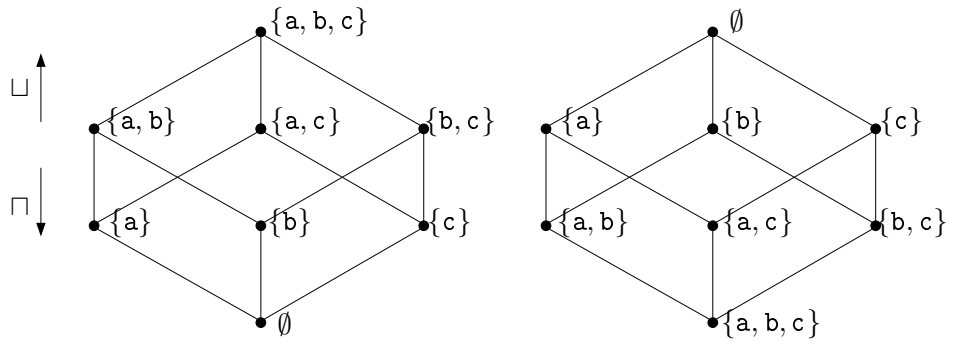
Définition 2.4 (Treillis complet) *Un uple* $\mathcal{T} = (\mathcal{L}, \sqsubseteq, \sqcap, \sqcup, \bot, \top)$ *est un treillis complet si* $(\mathcal{L}, \sqsubseteq, \sqcap, \sqcup)$ *est un treillis, et tout sous-ensemble* χ *de* $\mathcal{L}$ *possède une borne supérieure* $\bigsqcup \chi$ *et une borne inférieure* $\bigsqcap \chi$*. En particulier,* $\mathcal{L}$ *admet une borne inférieure et une borne supérieure ; on notera respectivement* $\bot$ *et* $\top$ *ces deux éléments particuliers de* $\mathcal{T}$*, tels que :*

- $\bot$ *est appelé* élément 0 *de* $\mathcal{T}$ *et vérifie :* $\forall l \in \mathcal{L}, \bot \sqsubseteq l$ *;*
- $\top$ *est appelé* élément 1 *de* $\mathcal{T}$ *et vérifie :* $\forall l \in \mathcal{L}, l \sqsubseteq \top$.

Remarque Par définition, un treillis $(\mathcal{L}, \sqsubseteq, \sqcap, \sqcup)$ est tel qu'il existe une borne supérieure et une borne inférieure pour tout couple d'éléments de $\mathcal{L}$: $\forall l_1, l_2 \in \mathcal{L}, \exists m, M \in \mathcal{L} \mid (m = l_1 \sqcap l_2) \wedge (M = l_1 \sqcup l_2)$. Si le treillis est fini (i.e. $\mathcal{L}$ est un ensemble fini), cela implique qu'il existe une borne supérieure et une borne inférieure pour tout sous-ensemble de $\mathcal{L}$, et en particulier pour $\mathcal{L}$.

Tout treillis fini est donc un treillis complet, en particulier tout treillis fini possède un élément 0 et un élément 1 (cela n'est pas vrai pour un treillis quelconque, cf. exemple 2.5).

Exemple 2.5 Prenons l'exemple d'un treillis non fini : si $\mathcal{L}$ est l'ensemble des entiers relatifs $\mathbb{Z}$, et $\sqsubseteq$ est la relation d'ordre sur les entiers $\leq$, alors tout couple d'éléments de $\mathcal{L}$ possède une borne supérieure et une borne inférieure, mais il n'est pas vrai que tout sous-ensemble de $\mathcal{L}$ possède une borne supérieure ou une borne inférieure (un contre-exemple est l'ensemble $\mathbb{Z}$ lui-même). Ce treillis n'est donc pas complet.



2.1 (a) 2.1 (b)

	Figure 2.1 (a)	Figure 2.1 (b)
Ordre partiel ($\sqsubseteq$)	$\subseteq$	$\supseteq$
Borne inférieure ($\sqcap$)	$\cap$	$\cup$
Borne supérieure ($\sqcup$)	$\cup$	$\cap$
Élément 0 ($\bot$)	$\emptyset$	$\{\mathtt{a},\mathtt{b},\mathtt{c}\}$
Élément 1 ($\top$)	$\{\mathtt{a},\mathtt{b},\mathtt{c}\}$	$\emptyset$

FIG. 2.1 – Un exemple de treillis complet $(\wp(\{\mathtt{a},\mathtt{b},\mathtt{c}\}), \sqsubseteq, \sqcap, \sqcup, \bot, \top)$.

Exemple 2.6 La figure 2.1 donne deux représentations possibles, figure 2.1 (a) et figure 2.1 (b), d'un treillis complet $(\mathcal{L}, \sqsubseteq, \sqcap, \sqcup, \bot, \top)$, où $\mathcal{L} = \wp(\{\mathtt{a},\mathtt{b},\mathtt{c}\})$.

Les nœuds du graphe représentent les éléments de $\mathcal{L}$ et les arcs représentent l'ordre partiel $\sqsubseteq$: deux éléments sont comparables selon $\sqsubseteq$ seulement si les nœuds associés sont reliés par un arc. Pour illustrer le fait que $\sqsubseteq$ est un ordre partiel, notons que $\{\mathtt{b},\mathtt{c}\}$ et $\{\mathtt{a},\mathtt{c}\}$ sont deux éléments incomparables de ce treillis. Ces deux représentations sont associées au choix de l'ordre partiel $\sqsubseteq$ sur $\mathcal{L}$.

Si les éléments du treillis sont ordonnés selon l'inclusion d'ensembles $\subseteq$ (cf. figure 2.1 (a)), alors un élément l_1 est inférieur à un autre élément l_2 selon $\sqsubseteq$ si et seulement si $l_1 \subseteq l_2$. Selon l'ordre partiel $\sqsubseteq$ (ici, $\subseteq$), $l_1 \cap l_2$ est inférieur à l_1 et à l_2 et tout autre minorant de l_1 et l_2 est inférieur à $l_1 \cap l_2$; plus formellement, $\forall m \in \mathcal{L}, m \subseteq l_1 \wedge m \subseteq l_2 \Rightarrow m \subseteq (l_1 \cap l_2)$. Alors la borne inférieure $l_1 \sqcap l_2$ de deux éléments l_1 et l_2 de $\mathcal{L}$, est par définition $l_1 \cap l_2$, l'intersection de ces deux éléments (cf. définition 2.2). L'opérateur dual, $\sqcup$ est l'union d'ensembles, $\cup$, i.e. la borne supérieure de deux éléments l_1 et l_2 est $l_1 \cup l_2$, l'union de ces deux éléments.

Par contre, si les éléments du treillis sont ordonnés selon l'inclusion de surensembles $\supseteq$ (cf. figure 2.1 (b)), alors un élément l_1 est inférieur à un autre élément l_2

selon $\sqsubseteq$ si et seulement si $l_1 \supseteq l_2$. De façon symétrique au cas précédent, la borne inférieure $\sqcap$ de deux éléments est l'union de ces deux éléments. L'opérateur dual $\sqcup$ est alors l'intersection d'ensembles, et la borne supérieure de deux éléments est l'intersection de ces deux éléments.

Enfin, pour tout l dans $\mathcal{L}$ on a $\emptyset \subseteq l$, donc l'élément 0 pour l'ordre $\subseteq$ est $\emptyset$; pour tout l dans $\mathcal{L}$ on a $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \supseteq l$, donc l'élément 0 pour l'ordre $\supseteq$ est $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ (cf. définition 2.4). De manière analogue, l'élément 1 pour $\subseteq$ est $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$; et l'élément 1 pour $\supseteq$ est $\emptyset$.

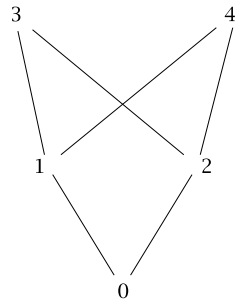


FIG. 2.2 – Un contre-exemple de treillis.

Exemple 2.7 On ne peut construire un treillis à partir de tout ensemble muni d'un ordre partiel : la figure 2.2 représente un ensemble fini $\mathcal{L} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}\}$, muni d'un ordre partiel $\preceq$, or il n'existe pas de borne supérieure pour $(\mathtt{3}, \mathtt{4})$ dans $\mathcal{L}$. $(\mathcal{L}, \preceq)$ ne possède donc pas les propriétés d'un treillis.

Définition 2.8 (Chaîne) *Soit $\mathcal{T} = (\mathcal{L}, \sqsubseteq, \sqcap, \sqcup)$ un treillis. Une chaîne de $\mathcal{T}$ est un sous-ensemble de $\mathcal{L}$, totalement ordonné. Plus formellement, χ est une chaîne de $\mathcal{T}$ si et seulement si $\chi \subseteq \mathcal{L}$ et $\forall l_1, l_2 \in \chi, (l_1 \sqsubseteq l_2) \vee (l_2 \sqsubseteq l_1)$.*

Soit $\chi = \{l_i \mid i \geq 0\}$ une chaîne de $\mathcal{T}$. On dit que :

- *χ est une chaîne ascendante si $\forall i, j \geq 0, i < j \Rightarrow l_i \sqsubseteq l_j$;*
- *χ est une chaîne descendante si $\forall i, j \geq 0, i < j \Rightarrow l_j \sqsubseteq l_i$;*
- *χ se stabilise si il existe $n \in \mathbb{N}$ tel que $\forall i \geq n, l_n = l_i$.*

Exemple 2.9 $\chi = \{\emptyset, \{\mathtt{c}\}, \{\mathtt{a}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \ldots\}$ est une chaîne ascendante du treillis (a) de la figure 2.1, en effet on a : $\emptyset \sqsubseteq \{\mathtt{c}\} \sqsubseteq \{\mathtt{a}, \mathtt{c}\} \sqsubseteq \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \sqsubseteq \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \sqsubseteq \ldots$. De plus χ se stabilise car $\forall i \geq 3, l_3 = l_i$.

Définition 2.10 (Fonction monotone) *Soit $\mathcal{L}$ un ensemble, une fonction $f : \mathcal{L} \mapsto \mathcal{L}$ est dite* monotone *si et seulement si $\forall l_1, l_2 \in \mathcal{L}, l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2)$ (les images par f des arguments sont ordonnées de la même façon que les arguments).*

Définition 2.11 (Points fixes) *Soit $f : \mathcal{L} \mapsto \mathcal{L}$ une fonction monotone sur un treillis complet $\mathcal{T} = (\mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$.*

- *$l \in \mathcal{L}$ est un* point fixe *de f si et seulement si $f(l) = l$;*
- *f est* réductive *en l si et seulement si $f(l) \sqsubseteq l$;*
- *f est* extensive *en l si et seulement si $f(l) \sqsupseteq l$;*
- *On définit $Fix(f) = \{l \mid f(l) = l\}$, $Red(f) = \{l \mid f(l) \sqsubseteq l\}$, et $Ext(f) = \{l \mid f(l) \sqsupseteq l\}$ les ensembles associés à cette définition.*

Notations Soit $f : \mathcal{L} \mapsto \mathcal{L}$ une fonction monotone, l'expression $f^n(x)$ dénote x si $n = 0$, et $f(f^{n-1}(x))$ si $n > 0$.

Soit $\mathcal{T} = (\mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ un treillis complet ; par définition d'un treillis complet, l'ensemble $Fix(f)$ a une borne supérieure et une borne inférieure. Ainsi, on note $lfp(f) = \bigsqcap Fix(f)$ et $gfp(f) = \bigsqcup Fix(f)$.

Exemple 2.12 La figure 2.3 donne une représentation des notions introduites par la définition 2.11 et les notations ci-dessus.

Le résultat suivant est connu comme le théorème du point fixe de Tarski, et montre que $lfp(f)$ est le plus petit point fixe de f, et que $gfp(f)$ est le plus grand point fixe de f.

Théorème 2.13 (Théorème du point fixe [Tar55, NNH99]) *Soit $\mathcal{T}$ un treillis complet de la forme $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$, soit $f : \mathcal{L} \mapsto \mathcal{L}$ une fonction sur $\mathcal{T}$. Si f est monotone, alors $lfp(f)$ et $gfp(f)$ vérifient :*

$$\begin{aligned} lfp(f) &= \bigsqcap Red(f) \in Fix(f) \\ gfp(f) &= \bigsqcup Ext(f) \in Fix(f) \end{aligned}$$

2.3 Application aux analyses de flot de données

Nous allons voir dans cette partie comment les notions de théorie des treillis présentées dans la partie précédente (section 2.2), ainsi que les résultats associés, peuvent être utilisés comme fondement d'une méthode de résolution des analyses de flot de données. Dans cette partie on traitera des analyses de flot de données

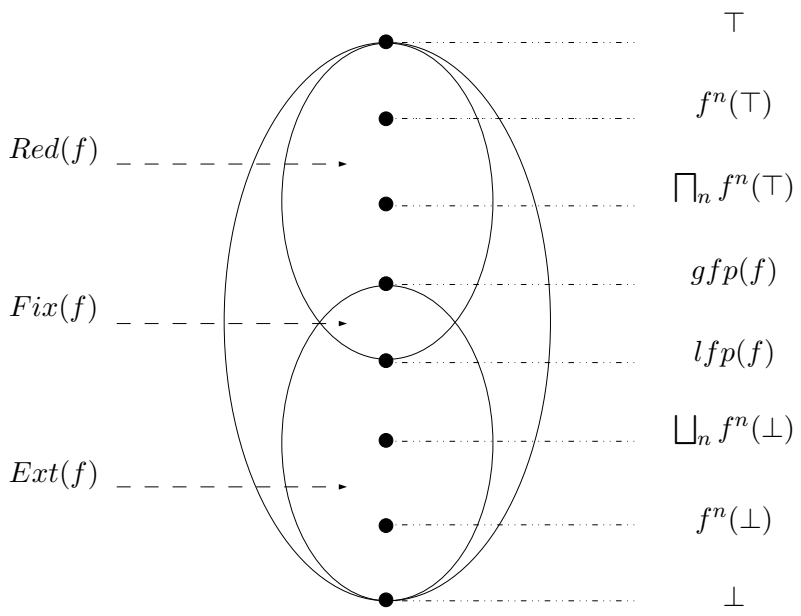


FIG. 2.3 – Points fixes et images d'une fonction monotone.

en avant, ce qui regroupe les analyses dans lesquelles l'information est toujours propagée dans le sens du flot de contrôle, e.g. la *propagation des définitions*. Étant donné les caractéristiques des graphes[2] employés comme base d'une analyse dans cette présentation, la reformulation pour les analyses de flot de données *en arrière*[3] est assez immédiate, mais ne sera pas décrite explicitement.

2.3.1 Théorie des treillis et analyses de flot de données

On peut présenter une analyse de flot de données comme la résolution d'équations de flot de données, dans un cadre d'analyse constitué d'un treillis complet et de fonctions de transfert. Il est souhaitable que les fonctions de transfert qui caractérisent une analyse de flot de données soient monotones, i.e. pour toute paire (l_1, l_2) d'éléments du domaine de définition de la fonction, $l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2)$ (cf. définition 2.10). Intuitivement cela signifie que l'augmentation de notre connaissance sur les informations de flot de données en entrée d'un nœud doit produire une augmentation de notre connaissance sur les informations de flot de données en sortie

[2]Notamment, unique point d'entrée et unique point de sortie.

[3]Analyses dans lesquelles l'information est toujours propagée dans le sens inverse du flot de contrôle, e.g. l'*analyse des variables vivantes*.

de ce nœud (tout au moins, cette connaissance ne doit pas décroître).

L'analyse de flot de données d'un programme peut être spécifiée en extrayant des équations de ce programme, et plus particulièrement, de son graphe de flot de contrôle (CFG, cf. section 1.3 page 15). Soit $G = (X, N, A, n_e, n_s)$ le CFG du programme à analyser ; on compte deux classes d'équations qui peuvent être extraites de G. Une classe d'équations de flot de données donne l'expression des informations en entrée d'un nœud en fonction des informations en sortie des prédécesseurs de ce nœud, i.e. l'information d'entrée est donnée par l'ensemble des informations de sortie des nœuds desquels le flot de contrôle peut provenir. Ces équations sont de la forme :

$$Info_{\text{entrée}}(n) = \begin{cases} \iota & \text{si } n = n_e \\ \bigsqcup\{Info_{\text{sortie}}(m) \mid (m,n) \in A\} & \text{sinon} \end{cases}$$

où ι spécifie l'information de flot de données initiale.

L'autre classe d'équations de flot de données donne l'expression des informations en sortie d'un nœud en fonction des informations en entrée de ce même nœud. Ces équations sont de la forme :

$$Info_{\text{sortie}}(n) = [\![n]\!](Info_{\text{entrée}}(n))$$

où $[\![n]\!]$ est la fonction de transfert associée au nœud n.

Les treillis sont utilisés dans le cadre d'une analyse de flot de données pour :

- représenter les informations de flot de données partiellement ordonnées,
- démontrer l'existence d'une solution qui peut être trouvée par recherche itérative de point fixe [KU77, NNH99, KSV96].

En démontrant l'existence d'un point fixe, le théorème de Tarski (cf. théorème 2.13) garantit l'existence d'algorithmes itératifs qui terminent, pour les analyses de flot de données exprimées sous forme de treillis et fonctions ayant les bonnes propriétés.

2.3.2 Choix d'un semi-treillis : union ou intersection ?

Dans le cadre d'une analyse de flot de données avec un treillis $\mathcal{T}$, un graphe $G = (X, N, A, n_e, n_s)$ et une fonction monotone f, les valeurs associées à chaque nœud de N évoluent dans le treillis $\mathcal{T}$ dans un seul sens – c'est une conséquence de la monotonie de f. La résolution d'une analyse de flot de données correspond alors à la recherche soit du plus grand point fixe de f, soit du plus petit point fixe de f dans le treillis. Pour cette raison il suffit de se donner un treillis muni d'un seul opérateur,

$\sqcup$ ou $\sqcap$; on parle alors de semi-treillis. On ne redonnera pas les définitions explicites de semi-treillis et semi-treillis complet, qui découlent directement des définitions 2.3 et 2.4.

Pour chaque analyse de flot de données, on peut désigner un opérateur "naturel" de résolution, typiquement l'union ou l'intersection d'ensembles ; e.g. $\cup$ pour la propagation des définitions, et $\cap$ pour l'analyse des expressions disponibles. Dans le premier cas on préférera comme base de l'analyse un semi-treillis de la forme $(\mathcal{L}, \sqsubseteq, \sqcup, \bot)$, et dans le second cas un semi-treillis de la forme $(\mathcal{L}, \sqsubseteq, \sqcap, \top)$. La formulation symétrique est aussi efficace, mais nous l'éviterons car elle conduit à des représentations contre-intuitive comme celle de la figure 2.1 (b).

Une analyse de flot de données, qui a pour opérateur naturel de résolution l'union d'ensembles, a pour but de trouver les plus grands ensembles qui résolvent les équations de flot de données (i.e. le plus grand point fixe). Les propriétés démontrées par cette analyse sont vérifiées par au moins un chemin d'exécution atteignant l'entrée d'un nœud. Si l'opérateur naturel de résolution est l'intersection d'ensembles, alors l'analyse a pour but de trouver les plus petits ensembles qui résolvent les équations de flot de données (i.e. le plus petit point fixe). Les propriétés démontrées par cette analyse sont vérifiées par tous les chemins d'exécution atteignant l'entrée d'un nœud.

2.3.3 Choix d'un semi-treillis : ensemble des informations de flot de données

Afin de résoudre une analyse de flot de données dans le cadre théorique décrit dans ce chapitre, on commence par construire un semi-treillis à partir d'un ensemble d'informations de flot de données. L'ensemble des informations de flot de données est caractéristique du programme sous analyse, aussi cet ensemble est en général fini (cet ensemble peut, par exemple, être construit à partir des variables du programme, qui sont en nombre fini). Dans ce cas, on peut dire que le semi-treillis des données dataflow est complet (cf. définition 2.4 page 25, ainsi que la remarque associée). Ensuite, on détermine des fonctions de transfert sur ce treillis, pour notre analyse de flot de données. Comme on l'a vu, ces fonctions seront de préférence monotones, afin que la connaissance ne diminue pas au cours de l'analyse, et afin d'être en mesure d'appliquer le théorème du point fixe de Tarski (théorème 2.13).

Étant donné une analyse de flot de données dans un semi-treillis $\mathcal{T} = (\mathcal{L}, \sqsubseteq, \sqcup, \bot)$, l'ensemble $\mathcal{L}$ est l'ensemble de toutes les possibilités pour l'information de flot de données qui peut être associée à un nœud du graphe de flot, dans le cadre de

l'analyse que l'on souhaite effectuer. La réalisation de l'analyse de flot de données consiste à associer à chaque nœud du graphe de flot un sous-ensemble de l'ensemble – que l'on note L – qui contient toute l'information de flot de données possible. Tous les sous-ensembles de L doivent par conséquent être des éléments de $\mathcal{L}$. L'ensemble $\mathcal{L}$ est alors naturellement l'ensemble des parties finies de l'ensemble L qui contient toute l'information de flot de données. Le treillis $\mathcal{T}$ représente les évolutions possibles de l'information de flot de données associée à un nœud du graphe de flot, au cours de la résolution de l'analyse de flot de données.

Exemple 2.14 Le treillis représenté par la figure 2.1 page 26 est construit sur l'ensemble $\wp(\{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$. Supposons qu'une analyse de flot de données d'un graphe G soit basée sur l'ensemble $\mathcal{L} = \wp(\{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$. Alors $\mathcal{L}$ représente toutes les possibilités pour l'information de flot de données associée à un nœud de G, et $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ est l'ensemble que l'on a appelé L, qui contient toute l'information de flot de données.

2.3.4 Produit cartésien de treillis

La correction de cette approche basée sur la théorie des treillis pour la résolution d'analyses de flot de données repose en partie sur le fait que le produit cartésien de treillis est un treillis, muni d'un ordre partiel induit. Notamment, le treillis qui représente l'ensemble des possibilités pour le calcul de l'information de flot de données du graphe de flot $G = (X, N, A, n_e, n_s)$ est basé sur le produit cartésien $\mathcal{L} \times \ldots \times \mathcal{L} = \mathcal{L}^{|N|}$, muni de l'ordre partiel induit $\preceq$, défini par :

$$\forall \overrightarrow{\ell_1}, \overrightarrow{\ell_2} \in \mathcal{L}^{|N|},$$
$$\overrightarrow{\ell_1} = (l_{11}, l_{12}, \ldots, l_{1n}) \preceq \overrightarrow{\ell_2} = (l_{21}, l_{22}, \ldots, l_{2n})$$
$$\text{si et seulement si } l_{11} \sqsubseteq l_{21} \text{ et } l_{12} \sqsubseteq l_{22}, \ldots, \text{ et } l_{1n} \sqsubseteq l_{2n}$$

Cet ordre partiel induit définit les opérations $\sqcap$ et $\sqcup$ sur le treillis produit cartésien (cf. définition 2.2).

Lorsque le treillis produit cartésien ainsi construit vérifie les conditions du théorème du point fixe, on sait alors qu'il existe un point fixe au calcul des informations de flot de données pour tout le graphe de flot, i.e. pour chaque nœud du graphe.

2.3.5 Introduction d'une sémantique abstraite

Comme on l'a vu précédemment, les analyses de flot de données consistent à considérer une sémantique abstraite, adaptée au problème considéré, plutôt que

la sémantique complète du système (cf. section 2.1). Pour définir la sémantique abstraite du système sous analyse, on définit une *fonction sémantique de transfert*, notée $[\![]\!]$. Cette fonction décrit un espace de fonctions de transfert, qui contient une fonction pour chaque nœud du graphe :

$$[\![]\!] : N \mapsto (\mathcal{L} \mapsto \mathcal{L})$$

où N est l'ensemble des nœuds du graphe de flot, et $\mathcal{L}$ est l'ensemble des informations de flot de données qui concernent l'analyse en cours. Cette fonction donne une sémantique abstraite à chaque point du programme (et donc à chaque nœud du graphe de flot), en terme d'une transformation du semi-treillis $(\mathcal{L}, \sqsubseteq, \sqcup, \bot)$ dans lui-même.

Les fonctions de transfert d'une analyse de flot de données sont souvent de la forme $\forall l \in \mathcal{L}, [\![n]\!](l) = (l \backslash nonp_n) \cup gen_n$, où n est un nœud du graphe, et $gen_n, nonp_n \in \mathcal{L}$ représentent les informations de flot de données générées – respectivement non propagées (ou bloquées) – par le nœud n. La proposition suivante montre que les fonctions de ce type sont monotones.

Proposition 2.15 *Soit $\mathcal{L}$ un ensemble, soient $b, g \in \mathcal{L}$. Les fonctions de la forme $\forall l \in \mathcal{L}, f(l) = (l \backslash b) \cup g$ sont monotones.*

Démonstration. Soit $(\mathcal{L}, \sqsubseteq, \sqcup, \bot)$ un semi-treillis, soient b, g deux éléments de $\mathcal{L}$, soit f une fonction définie par $\forall l \in \mathcal{L}, f(l) = (l \backslash b) \cup g$. Pour rester dans le cadre général, on considère que $\sqsubseteq$ peut être soit $\subseteq$ soit $\supseteq$ (cf. section 2.3.2) ; on se place dans le premier cas. Soient l_1, l_2 deux éléments de $\mathcal{L}$ tels que $l_1 \sqsubseteq l_2$. On a :

$$\begin{array}{lll} l_1 \sqsubseteq l_2 & \Rightarrow & l_1 \subseteq l_2 \\ & \Rightarrow & (l_1 \backslash b) \subseteq (l_2 \backslash b) \\ & \Rightarrow & (l_1 \backslash b) \cup g \subseteq (l_2 \backslash b) \cup g \\ & \Rightarrow & f(l_1) \subseteq f(l_2) \qquad\qquad \Rightarrow f(l_1) \sqsubseteq f(l_2) \end{array}$$

Le cas symétrique où $\sqsubseteq$ est interprété par $\supseteq$ se démontre de manière analogue. Par définition d'une fonction monotone (cf. définition 2.10), la proposition 2.15 est démontrée. △

Si un nœud représente l'instruction vide ϵ (comme c'est habituellement le cas des nœuds n_e et n_s), alors on lui associe l'identité sur $\mathcal{L}$, notée $Id_{\mathcal{L}}$. Une fonction sémantique de transfert $[\![]\!]$ peut être étendue naturellement pour couvrir les chemins

finis. Pour tout chemin $c = [n_1, \ldots, n_k] \in C_G[n_1, n_k]$, où $n_1, \ldots, n_k \in N$, on définit :

$$[\![c]\!] = \begin{cases} Id_{\mathcal{L}} & \text{si } c = [] \\ [\![[n_2, \ldots, n_k]]\!] \circ [\![n_1]\!] & \text{sinon} \end{cases}$$

2.4 Résolution d'une analyse de flot de données

2.4.1 Introduction

L'approche communément adoptée dans la littérature pour résoudre une analyse de flot de données dans de cadre de la théorie des treillis consiste à définir une solution "idéale" mais indécidable dans le cas général, ainsi qu'une solution algorithmique qui approxime la solution "idéale" de manière conservative, c'est à dire sans introduire d'erreur. Kildall [Kil73], Kam et Ullman [KU77] introduisent les solutions MOP (*Meet Over all Paths*[4]) et MFP (*Maximum Fixed Point*) d'une analyse de flot de données dans un semi-treillis $(\mathcal{L}, \sqsubseteq, \sqcap, \top)$; c'est la présentation la plus couramment employée [Muc97, KSV96, CK98, NNH99, Kri03]. Historiquement, la solution MFP est appelée "Maximum Fixed Point", bien qu'elle corresponde pour certaines analyses à rechercher le plus petit point fixe de la fonction sémantique de transfert. La raison en est que dans la littérature classique, les analyses de flot de données sont le plus souvent formulées dans des semi-treillis de la forme $(\mathcal{L}, \sqsubseteq, \sqcap, \top)$, et comme on l'a remarqué en section 2.3.2, cela conduit à une représentation contre-intuitive pour les analyses dont l'opérateur naturel de résolution est $\cup$.

Nous adoptons la présentation duale, comme S. Graf *et al.* dans [GSL96] : les solutions JOP (*Join Over all Paths*[5]) et MFP (ici, MFP signifie *Minimum Fixed Point*) d'une analyse de flot de données dans un semi-treillis $(\mathcal{L}, \sqsubseteq, \sqcup, \bot)$. Cette stratégie consiste à partir d'un élément initial du semi-treillis (typiquement, $\bot$), et atteindre le plus grand point fixe de la fonction sémantique de transfert par application de $\sqcup$.

La figure 2.4 page 38 donne une illustration, pour les analyses de flot de données, des deux formulations MOP/MFP et JOP/MFP.

2.4.2 La solution JOP d'une analyse de flot de données

La stratégie JOP [GSL96], pour *Join Over all Paths* (cf. section 2.4.1), définit la solution de référence, i.e. l'information de flot de données recherchée pour un

[4] L'opérateur *borne inférieure* $\sqcap$ est appelé *meet* en anglais.
[5] L'opérateur *borne supérieure* $\sqcup$ est appelé *join* en anglais.

point donné du programme. L'idée consiste à rassembler l'information que l'on peut trouver sur tous les chemins qui mènent au nœud en considération, et ainsi prendre en compte toutes les exécutions possibles.

Étant donnés un graphe de flot $G = (X, N, A, n_e, n_s)$, un semi-treillis $(\mathcal{L}, \sqsubseteq, \sqcup, \bot)$, et une fonction sémantique de transfert $[\![]\!] : N \mapsto (\mathcal{L} \mapsto \mathcal{L})$, la solution JOP est donnée par :

$$\forall n \in N, \forall l_0 \in \mathcal{L},$$
$$JOP_{(G,[\![]\!])}(n)(l_0) = \bigsqcup\{[\![c]\!](l_0) \mid c \in C_G[n_e, n[\}$$

Cette solution nous donne directement les informations souhaitées, cependant Kam et Ullman [KU77] ont montré que cette solution est indécidable dans le cas général : l'existence d'un algorithme général pour calculer la solution JOP impliquerait la décidabilité d'un problème d'identification de sous-chaîne[6], connu pour être indécidable.

D'autre part, on peut remarquer que le calcul de la solution JOP pour un nœud n ne peut terminer que pour des graphes comportant un nombre fini de chemins de n_e à n, i.e. $|C_G[n_e, n[|$ doit être fini. Or, comme les graphes que l'on souhaite analyser représentent des programmes, ils comportent le plus souvent des boucles ; cela peut induire un nombre infini de chemins entre des nœuds du graphe.

Exemple 2.16 Soit $G = (X, N, A, n_e, n_s)$ le graphe de la figure 1.1 page 16 ; supposons que $\mathcal{T} = (\mathcal{L}, \sqsubseteq, \sqcup, \bot)$ et $[\![]\!] : N \mapsto (\mathcal{L} \mapsto \mathcal{L})$ forment le cadre théorique d'une analyse de flot de données que l'on souhaite opérer sur G. Alors :

$$\forall l_0 \in \mathcal{L}, JOP_{(G,[\![]\!])}(\texttt{10})(l_0) = \bigsqcup\{[\![c]\!](l_0) \mid c \in C_G[\text{ENTRÉE}, \texttt{10}[\}$$

Or, $C_G[\text{ENTRÉE}, \texttt{10}[$ est un ensemble infini de chemins de G qui correspondent à l'expression régulière $[\text{ENTRÉE}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}(, \texttt{6}, \texttt{7}, \texttt{8})^*, \texttt{10}]$; la solution $JOP_{(G,[\![]\!])}(\texttt{10})$ n'est donc pas calculable directement.

Dans le cas général, la stratégie JOP ne mène pas à l'élaboration d'algorithmes pour le calcul d'informations de flot de données ; on s'oriente donc vers une autre stratégie, appelée MFP.

[6] *Modified Post Correspondence Problem*

2.4.3 La solution MFP d'une analyse de flot de données

La stratégie MFP [GSL96], pour *Minimum Fixed Point* (cf. section 2.4.1), a pour principe d'approximer de manière itérative la plus grande solution du système d'équations de flot de données décrit en section 2.3.1.

Étant donnés un graphe de flot $G = (X, N, A, n_e, n_s)$, un semi-treillis $(\mathcal{L}, \sqsubseteq, \sqcup, \bot)$, et une fonction sémantique de transfert $[\![]\!] : N \mapsto (\mathcal{L} \mapsto \mathcal{L})$, la solution MFP est donnée par :

$$\forall n \in N, \forall l_0 \in \mathcal{L},$$
$$MFP_{(G,[\![]\!])}(n)(l_0) = \begin{cases} l_0 & \text{si } n = n_e \\ \bigsqcup\{[\![m]\!](MFP_{(G,[\![]\!])}(m)(l_0)) \mid (m,n) \in A\} & \text{sinon} \end{cases}$$

Pour une analyse de flot de données spécifique, si le semi-treillis est complet, et si pour tout $n \in N$ les fonctions $[\![n]\!]$ sont monotones, alors cette approche fournit une description algorithmique dont la convergence est justifiée par le théorème du point fixe de Tarski (théorème 2.13).

2.4.4 Comparaison des solutions JOP et MFP

Les analyses de flot de données pour lesquelles l'approche JOP/MFP est naturelle (e.g. la propagation des définitions, cf. section 2.5), sont des problèmes pour lesquels on recherche la plus grande solution aux équations de flot de données (cf. section 2.3.2). Pour ces analyses, si une solution x surestime la solution recherchée, i.e. telle que $JOP \sqsubseteq x$, alors on dit que x est une solution *sûre* ; une solution qui sous estime la solution JOP est fausse (cf. figure 2.4).

De façon duale, dans le cadre des analyses de flot de données pour lesquelles l'approche MOP/MFP est naturelle, les solutions sûres sont les solutions qui sous-estiment la solution recherchée (x tel que $x \sqsubseteq MOP$, cf. figure 2.4).

Définition 2.17 (Fonction distributive) *Soit $\mathcal{T} = (\mathcal{L}, \sqsubseteq, \sqcup)$ un semi-treillis ; une fonction $f : \mathcal{L} \mapsto \mathcal{L}$ est dite* distributive *sur $\mathcal{T}$ si et seulement si :*

$$\forall L \subseteq \mathcal{L}, f(\bigsqcup\{l \mid l \in L\}) = \bigsqcup\{f(l) \mid l \in L\}$$

La distributivité est une propriété plus forte que la monotonie, dans le sens où l'on peut montrer [KU77, Ryd03] que la définition suivante de la monotonie est équivalente à la définition 2.10 page 28 : une fonction $f : \mathcal{L} \mapsto \mathcal{L}$ est monotone si et seulement si $\forall L \subseteq \mathcal{L}, f(\bigsqcup\{l \mid l \in L\}) \sqsubseteq \bigsqcup\{f(l) \mid l \in L\}$.

Si une fonction f est distributive, alors on peut calculer $\sqcup$ sur le domaine de définition de f et ensuite appliquer f, ou alors appliquer f et ensuite calculer $\sqcup$ sur le domaine image de f : dans les deux cas on calcule le même résultat, i.e. si $x_1, x_2, x_3 \in Dom(f)$ alors $f(x_1 \sqcup x_2 \sqcup x_3) = f(x_1) \sqcup f(x_2) \sqcup f(x_3)$. Dans le domaine des analyses de flot de données, cela signifie intuitivement que si les fonctions sémantiques associées à une analyse de flot de données sont distributives, les informations de flot de données peuvent être rassemblées sur tous les chemins qui mènent au nœud qui intéresse cette analyse, sans perdre de précision.

Cette idée est à l'origine du théorème de coïncidence de Kam et Ullman [KU77], qui montre que les solutions JOP et MFP coïncident lorsque les fonctions sémantiques sont distributives[7] (cf. théorème 2.18). Kam et Ullman ont aussi montré que cela n'est pas le cas pour toutes les fonctions monotones : pour les fonctions monotones qui ne sont pas distributives, la solution MFP est seulement une approximation de la solution JOP.

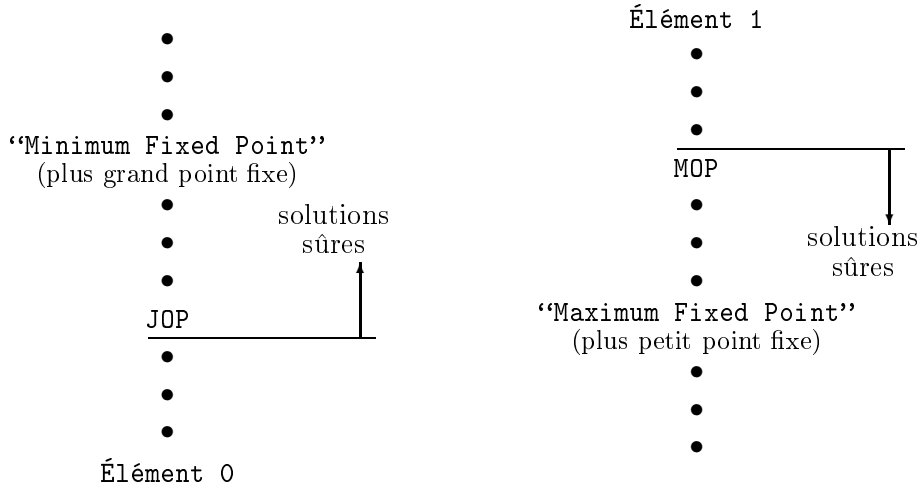
Théorème 2.18 (Théorème de coïncidence [KU77]) *Soient un graphe de flot $G = (X, N, A, n_e, n_s)$, et un semi-treillis $\mathcal{T} = (\mathcal{L}, \sqsubseteq, \sqcup, \bot)$. Si pour tout $n \in N$, les fonctions sémantiques $[\![n]\!] : \mathcal{L} \mapsto \mathcal{L}$, sont distributives, alors la solution MFP et la solution JOP coïncident, i.e. :*

$$\forall n \in N, \forall l_0 \in \mathcal{L}, MFP_{(G,[\![]\!])}(n)(l_0) = JOP_{(G,[\![]\!])}(n)(l_0)$$

Exemple 2.19 La figure 2.4 illustre la recherche du plus grand point fixe dans le cadre d'une stratégie JOP/MFP et la recherche du plus petit point fixe dans le cadre d'une stratégie MOP/MFP. Le théorème de coïncidence nous dit que si les fonctions de transfert de l'analyse de flot de données sont distributives, alors la solution MFP est égale à la solution JOP. En ce sens, on peut dire qu'un algorithme qui calcule la solution MFP résout l'analyse de flot de données de façon optimale.

Soit $\mathcal{T} = (\mathcal{L}, \sqsubseteq_{(a)}, \cup, \bot)$ le semi-treillis (a) de la figure 2.4, où $\sqsubseteq_{(a)}$ est l'inclusion d'ensembles $\subseteq$. Il est possible de traiter une même analyse de flot de données avec les deux différentes stratégies (cf. section 2.3.2). Supposons que les semi-treillis (a) et (b) de la figure 2.4 représentent la même analyse de flot de données, alors le semi-treillis (b) est $(\mathcal{L}, \sqsubseteq_{(b)}, \cap, \top)$, où $\sqsubseteq_{(b)}$ est l'inclusion de sur-ensemble $\supseteq$.

[7] Le théorème de coïncidence original de Kam et Ullman [KU77] se plaçait dans le cadre dual MOP/MFP ; le théorème établit que la solution *Maximum Fixed Point* est égale à la solution MOP lorsque les fonctions sémantiques sont distributives.



(a) *Stratégie JOP/MFP (semi-treillis avec $\sqcup$ et $\bot$)*

(b) *Stratégie MOP/MFP (semi-treillis avec $\sqcap$ et $\top$)*

FIG. 2.4 – Comparaison des solutions d'une analyse de flot de données.

Ainsi les deux approches duales sont cohérentes car la solution JOP de (a) et la solution MOP de (b) sont identiques, et les solutions sûres sont les l tels que $JOP \sqsubseteq_{(a)} l$ et $l \sqsubseteq_{(b)} MOP$, i.e. tels que $JOP \subseteq l$ et $l \supseteq MOP$.

2.4.5 Vecteurs de bits

Une analyse de flot de données typique opère sur $\wp(L)$ où L est un ensemble fini (cf. section 2.3.3). Cette propriété permet une formulation, dûe à M. Hecht [Hec77], basée sur des vecteurs de bits : on associe à chaque élément de L une unique position de bit i dans chaque vecteur ($1 \leq i \leq |L|$). Un sous-ensemble l de L (autrement dit, un élément du treillis) est alors représenté par un vecteur de $|L|$ bits :

- si l'élément de L auquel est associé la position i dans le vecteur est dans l, alors le i-ème bit du vecteur est positionné à 1 ;
- dans le cas contraire, le i-ème bit du vecteur est positionné à 0.

Cette formulation à base de vecteurs de bits permet une implémentation efficace de l'union d'ensembles, comme *ou* logique bit à bit des vecteurs qui représentent les ensembles dont on veut calculer l'union ; et de l'intersection d'ensembles, comme *et* logique bit à bit des vecteurs qui représentent les ensembles dont on veut calculer l'intersection.

Exemple 2.20 Soient $L = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, $l_1 = \{\mathtt{a}, \mathtt{b}\}$, et $l_2 = \{\mathtt{a}, \mathtt{d}\}$. Dans une for-

mulation à base de vecteurs de bits, l_1 est représenté par le vecteur $[1,1,0,0]$, et l_2 est représenté par $[1,0,0,1]$. Pour calculer l'union et l'intersection des ensembles l_1 et l_2 dans cette formulation, on utilise respectivement le *ou* logique et le *et* logique bit à bit : $(l_1 \cup l_2)$ est représenté par $[1,1,0,1]$, et $(l_1 \cap l_2)$ est représenté par $[1,0,0,0]$.

On retrouve bien les propriétés de l'union et de l'intersection d'ensembles : un élément qui est dans l_1 et dans l_2 est aussi dans l'union et dans l'intersection (`a` dans l'exemple) ; un élément qui est seulement dans un des ensembles l_1 et l_2 est dans l'union mais pas dans l'intersection (`b`, `d` dans l'exemple) ; enfin un élément qui n'est ni dans l_1 ni dans l_2 n'est ni dans l'union ni dans l'intersection (`c` dans l'exemple).

On dit qu'une analyse de flot de données est adaptée à une formulation à base de vecteurs de bits, si dans le cadre de cette analyse on a :

- un graphe de flot de contrôle,
- un semi-treillis complet basé sur l'ensemble des parties d'un ensemble fini,
- une fonction de transfert pour chaque nœud du graphe de flot, et pour chaque $n \in N$, la fonction de transfert associée est monotone, et de la forme $[\![n]\!](l) = (l \backslash nonp_n) \cup gen_n$, où l, $nonp_n$ et gen_n sont des éléments du treillis.

Proposition 2.21 *Les fonctions de transfert d'une analyse de flot de données adaptée à une formulation à base de vecteurs de bits sont distributives.*

Démonstration. Considérons une analyse de flot de données adaptée à une formulation à base de vecteurs de bits. Soit $(\mathcal{L}, \sqsubseteq, \sqcup, \bot)$ le semi-treillis complet associé à cette analyse, et χ un ensemble tel que $\chi \subseteq \mathcal{L}$. Notons $G = (X, N, A, n_e, n_s)$ le CFG à analyser. Pour rester dans le cadre général, on considère que $\sqcup$ peut être soit $\cup$ soit $\cap$ (cf. section 2.3.2). Notons $[\![]\!]$ la fonction sémantique de transfert associée à l'analyse de flot de données. Cette analyse étant adaptée à une formulation à base de vecteurs de bits, pour chaque $n \in N$, $[\![n]\!]$ est de la forme $[\![n]\!](l) = (l \backslash nonp_n) \cup gen_n$, avec $l, nonp_n, gen_n \in \mathcal{L}$ (cf. ci-dessus). Soient $b, g \in \mathcal{L}$, posons $[\![n]\!](l) = (l \backslash b) \cup g$ pour tout $l \in \mathcal{L}$. Supposons que $\sqsubseteq$ est interprété par $\subseteq$; on a :

$$\begin{aligned}
[\![n]\!](\bigsqcup\{l \mid l \in L\}) &= [\![n]\!](\bigcup\{l \mid l \in L\}) \\
&= ((\bigcup\{l \mid l \in L\}) \backslash b) \cup g \\
&= (\bigcup\{(l \backslash b) \mid l \in L\}) \cup g \\
&= \bigcup\{((l \backslash b) \cup g) \mid l \in L\} \\
&= \bigcup\{[\![n]\!](l) \mid l \in L\} \\
[\![n]\!](\bigsqcup\{l \mid l \in L\}) &= \bigsqcup\{[\![n]\!](l) \mid l \in L\}
\end{aligned}$$

Le cas symétrique où $\sqsubseteq$ est interprété par $\supseteq$ se démontre de manière analogue. Par définition d'une fonction distributive (cf. définition 2.17), on a montré que les fonctions de transfert de l'analyse de flot de données sont distributives. $\triangle$

Par application de la proposition 2.21 et du théorème du point fixe (théorème 2.13), la solution MFP calculée pour une analyse de flot de données adaptée à une formulation à base de vecteurs de bits coïncide avec la solution JOP de cette analyse.

2.5 Une application : Propagation des définitions

Le cadre théorique défini aux sections précédentes de ce chapitre permet de résoudre de nombreuses analyses de flot de données en avant, citons la propagation des définitions, l'analyse des expressions disponibles, la propagation des copies, la propagation des constantes. Comme nous l'avons mentionné précédemment, le cadre théorique de ce chapitre est aussi adapté à la résolution d'analyses de flot de données en arrière (e.g. variables vivantes), moyennant quelques adaptations des sections 2.3 et 2.4. Dans cette section, on se focalisera uniquement sur le problème de la propagation des définitions.

2.5.1 Définition du problème RD

On appelle définition d'une variable une instruction qui modifie la valeur associée à cette variable (ce sont typiquement les affectations de variables). Un exemple classique d'analyse de flot de données est le calcul des nœuds atteints par les définitions de variables dans un graphe : intuitivement, il s'agit de calculer pour chaque définition l'ensemble des sites (e.g. points de contrôle d'un programme, nœuds d'un graphe de flot) où peuvent être utilisées la (les) variable(s) modifiée(s) par cette définition. On appellera *propagation des définitions* cette analyse ; en anglais, on parle de *reaching definitions*[8] [ASU86, CK98, Muc97, NNH99], d'où l'abréviation "problème RD". La propagation des définitions permet de construire des liens entre les sites d'un programme ou d'un graphe qui produisent des valeurs et les sites qui utilisent ces valeurs. Les informations apportées par cette analyse sont notamment utiles pour réaliser des optimisations dans le cadre de la compilation, et pour calculer des dépendances dans le cadre d'analyses statiques.

[8] Comme le suggèrent Nielson *et al.* dans [NNH99], cette analyse pourrait être appelée *reaching assignments* (i.e. propagation des affectations).

On peut représenter les définitions de variables par des couples (n, x), signifiant que la variable x est définie au nœud n. L'ensemble des définitions qui apparaissent dans un graphe $G = (X, N, A, n_e, \mu)$ sera noté $\mathcal{D}_G$ et est défini par :

$$\mathcal{D}_G = \{(n, x) \in N \times X \mid x \in def_n\}$$

Définition 2.22 (Propagation des définitions) *Soit $G = (X, N, A, n_e, \mu)$ un graphe de flot. Pour tout nœud $n \in N$, pour toute variable $x \in X$ telle que x est définie au nœud n (i.e. $x \in def_n$), on dit que la définition de la variable x au nœud n, $(n, x) \in \mathcal{D}_G$,* atteint *un nœud m, s'il existe un chemin $c = [n_1, \ldots, n_k] \in C_G[n_1, n_k]$, où $n_1, \ldots, n_k \in N$, tel que les conditions suivantes soient vérifiées :*

1. $k > 1$
2. $n_1 = n \wedge n_k = m$
3. $\forall i$ *tel que* $1 < i < k$, $x \notin def_{n_i}$

Exemple 2.23 Soit G le graphe de la figure 1.1 page 16. La définition (2,`prod`) atteint le nœud `5` car il existe un chemin $c \in C_G[2, 5]$, $c = [2, 3, 4, 5]$ et `prod` $\notin def_3$ et `prod` $\notin def_4$.

2.5.2 Définition du treillis

La propagation des définitions – ou problème RD – est une analyse de flot de données, aussi une approche fondée sur la théorie des treillis fournit un cadre théorique sûr pour la résolution de ce problème (cf. sections 2.3 et 2.4).

Déterminer l'ensemble des informations de flot de données

Le semi-treillis $\mathcal{T} = (\mathcal{L}, \sqsubseteq, \sqcup, \bot)$ sur lequel se base l'analyse doit être adapté au problème. Pour le problème RD, l'ensemble qui contient toute l'information de flot de données est $\mathcal{D}_G$, l'ensemble de toutes les définitions de variables dans le graphe sous analyse (on ne peut associer plus d'information à un nœud dans le cadre de cette analyse). Comme ensemble $\mathcal{L}$ d'informations de flot de données adapté au problème, on choisit donc $\wp(\mathcal{D}_G)$, l'ensemble des parties finies de $\mathcal{D}_G$ (cf. section 2.3.3). Par exemple, si $l \in \mathcal{L}$, alors l est un sous-ensemble de $\mathcal{D}_G$.

Déterminer le reste du treillis

On souhaite ainsi associer à chaque nœud du graphe un sous-ensemble de $\mathcal{D}_G$ qui dénote l'ensemble des définitions qui atteignent ce nœud. Par définition, le pro-

blème de la propagation des définitions fait intervenir une propriété satisfaite par au moins un chemin d'exécution (cf. définition 2.22). Il s'agit donc d'une analyse où $\sqcup$ est l'union d'ensembles $\cup$, et où la solution souhaitée est formée par les plus grands ensembles qui résolvent les équations de flot de données du problème (cf. section 2.3.2). Reprenons les définitions d'une borne supérieure et d'un treillis complet (définitions 2.2 et 2.4 page 25) afin de compléter la définition du treillis adapté au problème RD :

- On sait que la borne supérieure de deux éléments du treillis est l'union de ces deux éléments. Montrons que l'inclusion d'ensembles $\subseteq$ comme ordre partiel $\sqsubseteq$ sur le treillis est en accord avec la définition de la borne supérieure et les hypothèses. Soient l_1 et l_2 deux éléments de $\mathcal{L} = \wp(\mathcal{D}_G)$. On a $(l_1 \cup l_2) \in \mathcal{L}$ et $l_1 \subseteq (l_1 \cup l_2)$ et $l_2 \subseteq (l_1 \cup l_2)$. Soit l un majorant quelconque de l_1 et l_2 : $l \in \mathcal{L}$ tel que $l_1 \subseteq l$ et $l_2 \subseteq l$, alors $(l_1 \cup l_2) \subseteq l$, par définition de l'union d'ensembles. En accord avec la définition 2.2, l'ordre partiel $\sqsubseteq$ sur ce treillis étant tel que la borne supérieure de deux éléments est l'union de ces deux éléments, l'inclusion d'ensembles $\subseteq$ convient pour être identifié à $\sqsubseteq$.
- L'élément 0, $\bot$, est tel que $\forall x \in \wp(\mathcal{D}_G), \bot \sqsubseteq x$, ce qui signifie que $\bot$ est un sous-ensemble de x pour tout $x \subseteq \mathcal{D}_G$. on prend donc $\bot = \emptyset$.

On peut donc baser la résolution du problème RD sur le semi-treillis $\mathcal{T} = (\mathcal{L}, \sqsubseteq, \sqcup, \bot) = (\wp(\mathcal{D}_G), \subseteq, \cup, \emptyset)$. Comme $\mathcal{D}_G$ est un ensemble fini, $\wp(\mathcal{D}_G)$ est aussi un ensemble fini. On sait alors que $\mathcal{T}$ possède la propriété d'être complet (cf. définition 2.4 page 25, ainsi que la remarque associée). La figure 2.5 donne une représentation graphique de ce treillis.

Exemple 2.24 La figure 2.5 représente le treillis de base pour le problème de propagation des définitions dans un CFG G : $(\wp(\mathcal{D}_G), \subseteq, \cup, \emptyset)$. Les éléments du treillis représentent les valeurs qui peuvent être associées à l'information de flot de données pour chaque nœud du graphe de flot ; ces valeurs sont prises dans $\wp(\mathcal{D}_G)$. Par commodité de notation, on pose $k = |\mathcal{D}_G|$, et on note $d_1, \ldots, d_k$ les éléments de $\mathcal{D}_G$.

À partir du semi-treillis de base $(\wp(\mathcal{D}_G), \subseteq, \cup, \emptyset)$, le semi-treillis produit cartésien pour le graphe de flot est formé sur l'ensemble $\wp(\mathcal{D}_G) \times \wp(\mathcal{D}_G) \times \ldots \times \wp(\mathcal{D}_G) = \wp(\mathcal{D}_G)^{|N|}$, muni de l'ordre partiel induit par $\subseteq$ (cf. section 2.3.4), et dont l'élément 0 est $(\emptyset, \ldots, \emptyset)$. Pour résoudre le problème RD, il faut appliquer le théorème du point fixe de Tarski (théorème 2.13) afin de trouver le plus grand point fixe de la fonction sémantique de transfert sur ce treillis produit cartésien.

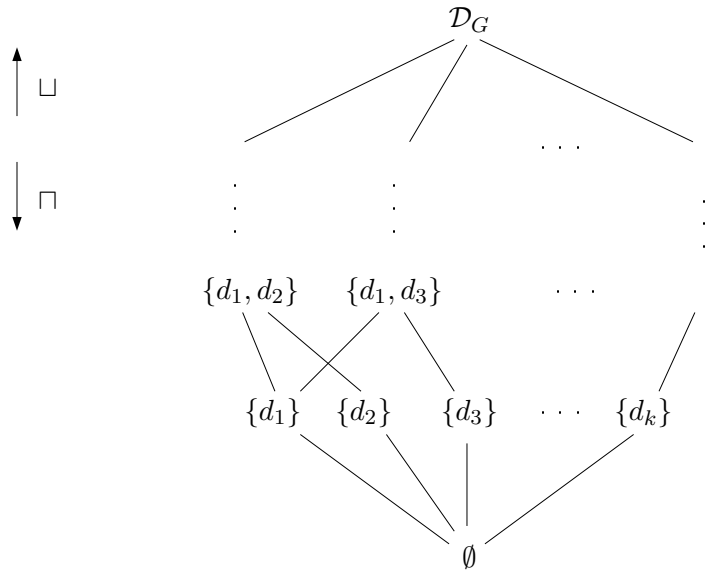


FIG. 2.5 – Treillis de base pour le problème RD dans un CFG.

2.5.3 Définition de la sémantique abstraite

Il reste à définir la fonction sémantique de transfert (cf. section 2.3.5) adaptée au problème RD. On veut définir une fonction de la forme $[\![]\!] : N \mapsto (\mathcal{L} \mapsto \mathcal{L})$, où $\mathcal{L} = \wp(\mathcal{D}_G)$ (cf. section 2.5.2), et où N est l'ensemble des nœuds du graphe de flot à analyser.

Définition 2.25 (Ensembles gen_n et $nonp_n$) *Étant donné un nœud $n \in N$, on définit gen_n et $nonp_n$ les ensembles des définitions de variables qui sont respectivement générées et non propagées par le nœud n :*

$$\begin{aligned} gen_n &= \{(n, x) \in \mathcal{D}_G \mid x \in def_n\} \\ nonp_n &= \{(n', x) \in \mathcal{D}_G \mid n' \neq n \wedge x \in def_n\} \end{aligned}$$

Remarque Notons que l'ensemble $nonp_n$ dénote l'ensemble des définitions *potentiellement* non propagées par le nœud n : Soient $n, n' \in N$, supposons $(n, x) \in \mathcal{D}_G$ et $d = (n', x) \in \mathcal{D}_G$, tels que $\forall c \in C_G[n_e, n], n' \notin c$. Dans ce cas, $d \in nonp_n$ (par définition), mais d n'est pas *effectivement* bloquée par n car d n'intervient sur aucun chemin de n_e à n.

Exemple 2.26 Soit G le graphe de la figure 1.1 page 16. Si l'on s'intéresse au nœud `6`, on a : $gen_{\texttt{6}} = \{(\texttt{6}, \texttt{somme})\}$ et $nonp_{\texttt{6}} = \{(\texttt{1}, \texttt{somme})\}$.

En modifiant légèrement le graphe G de sorte qu'au nœud `10` soit associé l'instruction `somme:=1` au lieu de `write(somme)`, alors la valeur de $gen(6)$ est inchangée ; par contre $nonp_6 = \{(1, \texttt{somme}), (10, \texttt{somme})\}$. De cette manière, $(10, \texttt{somme}) \in nonp_6$ alors que $\forall c \in C_G[\text{ENTRÉE}, 6], 10 \notin c$.

La fonction sémantique de transfert $[\![]\!]$ adaptée au problème RD décrit pour chaque nœud n du graphe de flot une fonction de transfert qui propage (ou non) les définitions atteignant le nœud n. Si l'on note l l'ensemble des définitions qui atteignent un nœud n, les fonctions de transfert pour tout nœud n sont décrites par l'équation :

$$[\![n]\!](l) = (l \backslash nonp_n) \cup gen_n \tag{1}$$

L'équation (1) signifie que les définitions qui atteignent en entrée un nœud n, et qui ne sont pas bloquées par n, sont propagées en sortie de n, avec les définitions générées par n.

Par définition des ensembles *nonp* et *gen* (définition 2.25), on a $nonp_n \subseteq \mathcal{D}_G$ et $gen_n \subseteq \mathcal{D}_G$ pour tout nœud $n \in N$. Par conséquent on a : $\forall n \in N, nonp_n \in \mathcal{L} \wedge gen_n \in \mathcal{L}$. Les conditions d'application de la proposition 2.15 page 33 sont alors réunies, on en déduit que les fonctions de transfert définies ci-dessus pour le problème RD, $[\![n]\!] : \mathcal{L} \mapsto \mathcal{L}$, sont monotones pour tout $n \in N$.

La remarque précédente nous indique que $nonp_n$ est un sur-ensemble des définitions effectivement bloquées par le nœud n. En effet, si l est l'ensemble des définitions qui atteignent le nœud n, les définitions qui sont effectivement bolquées par n sont les éléments de $l \cap nonp_n$. Cela ne remet pas en cause la correction de l'équation (1), car $(l \backslash nonp_n)$ dénote le même ensemble que $(l \backslash (l \cap nonp_n))$. L'ensemble $(l \backslash nonp_n)$ dénote donc bien l'ensemble des définitions qui atteignent le nœud n, auquel on retire les définitions qui sont effectivement bloquées par n.

L'étape suivante consiste à étendre la sémantique abstraite $[\![]\!]$ aux chemins $c = [n_1, \ldots, n_k] \in C_G[n_1, n_k]$, où $n_1, \ldots, n_k \in N$ et $k \geq 1$, avec la fonction identité $Id_{\mathcal{L}}(l) = l$, pour tout $l \in \mathcal{L}$:

$$[\![c]\!] = \begin{cases} Id_{\mathcal{L}} & \text{si } c = [] \\ [\![[n_2, \ldots, n_k]]\!] \circ [\![n_1]\!] & \text{si } c \text{ est de la forme } [n_1, \ldots, n_k], \text{ où } k \geq 1 \end{cases}$$

2.5.4 Solution JOP pour le problème RD

Étant donné un graphe de flot $G = (X, N, A, n_e, \mu)$, on se place dans le cadre théorique défini aux sections précédentes 2.5.2 et 2.5.3, à savoir le semi-treillis $\mathcal{T} =$

$(\wp(\mathcal{D}_G), \subseteq, \cup, \emptyset)$ et les fonctions sémantiques $[\![n]\!](l) = (l \backslash nonp_n) \cup gen_n$ pour tout $n \in N$.

Selon la définition 2.22, une définition atteint un nœud n s'il existe un chemin de n_e à n, sur lequel cette définition n'est pas bloquée (i.e. la variable n'est pas redéfinie). Au niveau du nœud ENTRÉE n_e, il n'y a pas de définition (l'ensemble des définitions est vide). On reprend donc la formule de la solution JOP (cf. section 2.4.2), et on définit JOP^{RD}, la solution JOP au problème de propagation des définitions, en fixant $l_0 = \emptyset$. S'il existe plusieurs chemins de n_e à n, on regroupe par l'union les définitions incidentes (conformément à la définition du treillis) :

$$\forall n \in N, JOP^{RD}_{(G,[\![])}(n) = \bigcup \{[\![c]\!](\emptyset) \mid c \in C_G[n_e, n[\}$$

Les solutions au problème RD qui peuvent être qualifiées de sûres sont les ensembles qui contiennent la solution JOP^{RD} (cf. section 2.4.4 et figure 2.4). Cela correspond bien à l'intuition du problème RD : il est correct mais imprécis de dire qu'une définition atteint un nœud lorsqu'elle ne l'atteint pas en réalité, alors qu'il est incorrect de dire qu'une définition n'atteint pas un nœud alors qu'en réalité elle l'atteint. Une solution approchée correcte MFP^{RD} doit donc être un sur-ensemble de la solution optimale JOP^{RD}.

2.5.5 Solution MFP pour le problème RD

Comme on l'a vu précédemment (cf. section 2.4.2), le calcul de la solution JOP d'une analyse de flot de donnée est indécidable en général, notamment sur des graphes qui comportent potentiellement un nombre infini de chemins. La plupart des programmes intéressants comportent des boucles (`while`, `for`...) qui dans le cas général rendent indécidable la question du nombre fini de chemin. On recherche donc une solution différemment, en employant la stratégie MFP (cf. section 2.4.3).

Dans cette formulation de la propagation des définitions, la solution MFP correspond au plus grand point fixe de la fonction sémantique de transfert dans le semi-treillis $\mathcal{T} = (\wp(\mathcal{D}_G), \subseteq, \cup, \emptyset)$ (cf. section 2.5.2). Pour rechercher la solution MFP^{RD}, on procède de manière analogue à la recherche de la solution JOP^{RD} (ci-dessus). On reprend la formule de la solution MFP, et on définit MFP^{RD}, la solution MFP au problème de propagation des définitions, en fixant $l_0 = \emptyset$ pour la même raison que précédemment.

$$\forall n \in N, MFP^{RD}_{(G,[\![])}(n) = \begin{cases} \emptyset & \text{si } n = n_e \\ \bigcup \{[\![m]\!](MFP^{RD}_{(G,[\![])}(m)) \mid (m,n) \in A\} & \text{sinon} \end{cases}$$

On a défini un cadre d'analyse de flot de données pour le problème RD, composé d'un semi-treillis complet (cf. section 2.5.2), et de fonctions de transfert monotones pour tous les nœuds du graphe de flot de contrôle (cf. section 2.5.3). Les conditions d'aplication du théorème du point fixe sont réunies, on sait donc que la solution MFP^{RD} au problème de propagation des définitions peut être formulée de manière algorithmique, en parcourant le graphe jusqu'à trouver un point fixe sur les informations de flot de données.

2.5.6 Vecteurs de bits pour le problème RD

On a défini dans les sections précédentes un cadre pour la résolution du problème RD, formellement fondé sur la théorie des treillis. Pour résoudre une instance du problème RD on a :

- un graphe de flot de contrôle G, qui constitue la donnée d'entrée du problème à résoudre ;
- un semi-treillis complet $\mathcal{T} = (\wp(\mathcal{D}_G), \subseteq, \cup, \emptyset)$ (cf. section 2.5.2). $\mathcal{D}_G$ est l'ensemble des définitions de variables dans le graphe G, c'est donc un ensemble fini et par conséquent le semi-treillis $\mathcal{T}$ est complet (cf. définition 2.4 et la remarque associée, page 25) ;
- pour chaque nœud n du graphe G une fonction de transfert monotone de la forme $\forall l \in \wp(\mathcal{D}_G), [\![n]\!](l) = (l \backslash nonp_n) \cup gen_n$ (cf. section 2.5.3).

Cette formulation du problème RD vérifiant les propriétés ci-dessus, on peut dire que cette analyse de flot de données est adaptée à une formulation à base de vecteurs de bits (cf. section 2.4.5, ainsi que la remarque ci-dessous). Par la proposition 2.21, on sait alors que les fonctions de transfert $[\![n]\!] : \mathcal{L} \mapsto \mathcal{L}$, pour tout n, sont distributives.

Les conditions d'application du théorème de coïncidence (théorème 2.18) sont alors réunies, on sait donc que la solution calculée MFP^{RD} est optimale, au sens où elle coïncide avec la solution JOP^{RD}.

Remarque (Formulation de RD avec des vecteurs de bits) On sait que pour chaque instance du problème RD, $\mathcal{D}_G$ est fini. Posons $k = |\mathcal{D}_G|$ et notons $d_1, \ldots, d_k$ les éléments de $\mathcal{D}_G$. On associe à chaque nœud n du graphe de flot un vecteur de bits $B_n = [b_1, \ldots, b_k]$ initialisés à $[0, \ldots, 0]$, et lorsque l'algorithme termine, pour tout i tel que $1 \leq i \leq k, b_i = 1$ si et seulement si la définition d_i atteint le nœud n (sinon $b_i = 0$).

Concernant la formulation du problème RD avec des vecteurs de bits, on pourra se référer à la section 2.4.5, ainsi qu'à [Muc97, NNH99, Ryd03].

Chapitre 3

Le modèle des automates communicants

Ce chapitre définit le cadre théorique de spécifications formées d'automates communicants, sur lequel est fondé notre approche pour l'analyse de dépendances et la réduction paramétrée de modèles (cf. chapitres 4 et 5).

Sommaire

3.1 Automates communicants (IOSTS)

Nous allons introduire dans cette section un formalisme de systèmes de transitions symboliques à entrées-sorties, qui permettent de spécifier le comportement de systèmes réactifs, et que nous noterons IOSTS, pour Input/Output Symbolic Transition Systems [LGP07b, GLRT06]. Ce formalisme étend les systèmes de transitions classiques en incluant la gestion de variables, de types de données, et de communications via des canaux de communication. Ce formalisme permet de modéliser des systèmes de façon compacte, et peut de plus permettre de modéliser des systèmes

ayant un nombre infini d'états. Notre méthode de réduction paramétrée a pour but d'opérer sur des spécifications décrites par des IOSTS.

3.1.1 Types de données

Les types de données sont spécifiés dans le cadre des spécifications algébriques. Chaque type de données est spécifié par un ensemble de *formules équationnelles typées*.

Syntaxe

Une *signature de types de données* est un couple $\Omega = (\Theta, \Phi)$, où Θ est un ensemble de noms de types, et chaque élément de Φ est composé d'un nom d'opération, et d'un profil $\theta_1 \dots \theta_{n-1} \rightarrow \theta_n$, où $n \geq 1$ et pour tout $i \leq n$, $\theta_i \in \Theta$.

Soit $V = \bigcup_{\theta \in \Theta} V_\theta$ un ensemble de noms de variables typées. L'ensemble des Ω-*termes* à variables dans V est dénoté par $\mathscr{T}_\Omega(V)$ et est défini par induction de la manière suivante : un Ω-terme de type θ_n est soit une variable dans V_{θ_n}, soit une expression $\phi(t_1 \dots t_{n-1})$, où $\phi : \theta_1 \dots \theta_{n-1} \rightarrow \theta_n$ est une opération dans Φ, et $t_1 \dots t_{n-1}$ sont des Ω-termes.

Une Ω-*substitution* est une function $\sigma : V \rightarrow \mathscr{T}_\Omega(V)$ qui preserve les types, i.e. si $v \in V_\theta$, alors $\sigma(v)$ est de type θ. Par la suite, nous noterons $\mathscr{T}_\Omega(V)^V$ l'ensemble de toutes les Ω-substitutions associant aux variables dans V des termes dans $\mathscr{T}_\Omega(V)$.

L'ensemble des Ω-*formules* à variables dans V est dénoté par $\mathscr{F}_\Omega(V)$ et est défini par induction de la manière suivante : une Ω-formule est soit une valeur de vérité dans l'ensemble $\{true, false\}$; une expression de la forme $t_1 = t_2$, où $t_1, t_2 \in \mathscr{T}_\Omega(V)$ sont du même type ; ou une expression formée par des Ω-formules et des opérateurs booléens dans l'ensemble $\{\neg, \vee, \wedge\}$, avec la syntaxe usuelle.

Semantique

Un Ω-*modèle* est une famille $M = \{M_\theta\}_{\theta \in \Theta}$, donnée avec un espace de fonctions défini par : $\{\phi_M : M_{\theta_1} \times \dots \times M_{\theta_n} \rightarrow M_\theta \mid (\phi : \theta_1 \dots \theta_{n-1} \rightarrow \theta_n) \in \Phi\}$. Les applications ν de V dans M, étendues aux termes dans $T_\Omega(V)$, et préservant les types, sont appelées Ω-*interprétations*. M^V est l'ensemble de toutes les Ω-interpretations de V dans M. Étant donné un modèle M et une formule f, f est dite *satisfiable* dans M, s'il existe une interprétation ν telle que $M \models_\nu f$.

3.1.2 Systèmes de Transitions Symboliques à Entrées/Sorties

La *signature* Σ d'un IOSTS est un triplet (Ω, V, C), où Ω est une signature de types de données, $V = \bigcup_{\theta \in \Theta} V_\theta$ est un ensemble de noms de variables appelées *variables attributs*, et C est un ensemble de noms de *canaux de communication*.

Un IOSTS dénote les changements d'état du système spécifié en décrivant les modifications possibles pour les valeurs associées aux variables attributs. Ces valeurs peuvent être modifiées par des interactions avec l'environnement du système, que l'on appelle *actions de communication* (cf. définition 3.1) ; ou par des opérations internes, qui sont dénotées par des substitutions de variables (cf. définition 3.2). Les actions de communication décrivent les échanges de messages par rendez-vous binaire, via des canaux de communication. L'absence de communication est dénotée par une action silencieuse, notée τ.

Définition 3.1 (Actions de Communication) *L'ensemble des* actions de communication *construit à partir de la signature d'*IOSTS $\Sigma = (\Omega, V, C)$ *est dénoté par* $Act(\Sigma)$*, défini comme suit :*

$$\begin{aligned} Act(\Sigma) &= \{\tau\} \cup Input(\Sigma) \cup Output(\Sigma) \\ Input(\Sigma) &= \{c? \mid c \in C\} \cup \{c?x \mid c \in C \wedge x \in V\} \\ Output(\Sigma) &= \{c! \mid c \in C\} \cup \{c!t \mid c \in C \wedge t \in \mathscr{T}_\Omega(V)\} \end{aligned}$$

Le système spécifié est stimulé par son environnement par l'intermédiaire des actions dans $Input(\Sigma)$. Plus particulièrement, si l'action $c?$ est effectuée, alors le système attend un signal sur le canal c ; si l'action $c?x$ est effectuée, alors le système attend une réception valuée sur le canal c, puis assigne à la variable attribut x la valeur ainsi réceptionnée.

Les réponses du système à son environnement sont dénotées par les actions dans $Output(\Sigma)$. En particulier, si l'action $c!t$ (respectivement, $c!$) est effectuée, alors le système effectue une émission valuée par t sur le canal c (respectivement, ne possédant pas argument).

Définition 3.2 (Systèmes de Transitions Symboliques : IOSTS) *Un* IOSTS *de signature* $\Sigma = (\Omega, V, C)$ *est un triplet* $(S, s_0, T)_\Sigma$ *où* S *est un ensemble de noms d'états,* $s_0 \in S$ *est l'état initial, et* $T \subseteq S \times Act(\Sigma) \times \mathscr{F}_\Omega(V) \times \mathscr{T}_\Omega(V)^V \times S$ *est une relation de transition. Une transition dans* T *est un tuple* (s, a, f, σ, s') *où les états* s *et* s' *sont appelés respectivement* état source *et* état cible*,* f *est une formule appelée*

garde *de la transition, a est une action de communication et* σ *une substitution de variables.*

Par la suite, les systèmes de transitions symboliques à entrées-sorties de la définition 3.2 seront appelés IOSTS, ou "automates".

Notation Étant donné une transition tr, on note $source_{tr}$ l'état source et $cible_{tr}$ l'état cible de tr.

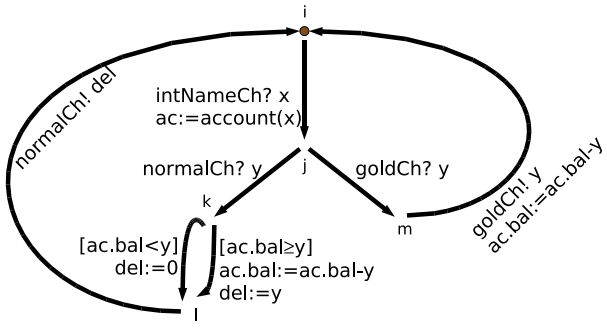


FIG. 3.1 – Exemple d'un IOSTS.

Exemple 3.3 La figure 3.1 représente un IOSTS, dont l'état initial est représenté par un point, les états sont représentés par des lettres, reliées par des flèches étiquetées qui représentent les transitions. Les étiquettes des transitions sont représentées de la façon suivante : les gardes trivialement vraies (i.e. évaluées à *vrai* pour tout paramétrage) ne sont pas représentées, et les autres gardes sont représentées entre crochets ; les affectations de variables sont représentées en utilisant la notation usuelle avec le symbole ':=', seulement pour les variables pour lesquelles la substitution associée à la transition n'est pas l'identité ; et les actions de communication sont représentées seulement si elles sont différentes de τ. Nous verrons plus précisément le rôle que peut occuper cet IOSTS dans le cadre d'une spécification, en section 3.2.

3.1.3 Semantique

La sémantique d'un IOSTS $\mathscr{A}$ dans un modèle M est donnée par la définition 3.4, sous la forme d'un système de transitions étiqueté et étendu : les états étendus contiennent des valuations de variables, et les transitions expriment la sémantique des transitions de $\mathscr{A}$.

Notations Par la suite, une substitution de variables, qui associe m à x, et est l'identité pour toutes les autres variables dans $Id_{V\setminus\{x\}}$, sera notée $(x \mapsto m)$. Pour le confort de lecture, on ne renommera pas l'extension des interprétations aux termes : plus formellement, $\nu : V \to M$ est étendu aux termes par $\nu : \mathscr{T}_\Omega(V) \to M$. Finalement, on notera Λ l'ensemble des étiquettes défini par : $\Lambda = \{\tau\} \cup \{c! \mid c \in C\} \cup \{c? \mid c \in C\} \cup \{c!m \mid c \in C \wedge m \in M\} \cup \{c?m \mid c \in C \wedge m \in M\}$.

Définition 3.4 (Sémantique d'un IOSTS) *Étant donné un* IOSTS $\mathscr{A}$ *de la forme* $(S, s_0, T)_\Sigma$*, où* $\Sigma = (\Omega, V, C)$*, et un modèle* M*, la sémantique de* $\mathscr{A}$ *dans* M *est un système de transitions étiqueté et étendu* $(\!|\mathscr{A}|\!) = ((\!|S|\!), (s_0, \nu_0), (\!|T|\!))$*, tel que :* $(\!|S|\!) \subseteq S \times M^V$*,* $\nu_0 \in M^V$ *et* $(\!|T|\!) \subseteq (\!|S|\!) \times \Lambda \times (\!|S|\!)$*. L'ensemble des états étendus* $(\!|S|\!)$ *contient les éléments de la forme* (s, ν)*, où* s *est un état de* $\mathscr{A}$*, et* ν *est une* Ω*-interprétation. L'ensemble des transitions* $(\!|T|\!)$ *est le plus petit ensemble qui contient les transitions construites à l'aide des règles suivantes :*

Règles de base :

$$\frac{(s_0, a, f, \sigma, s) \in T \quad a \in (\Lambda \cap Act(\Sigma)) \quad \nu \in M^V \quad M \models_\nu f}{((s_0, \nu), a, (s, \nu \circ \sigma))} \quad (1)$$

$$\frac{(s_0, c!t, f, \sigma, s) \in T \quad \nu \in M^V \quad M \models_\nu f}{((s_0, \nu), c!\nu(t), (s, \nu \circ \sigma))} \quad (2)$$

$$\frac{(s_0, c?m, f, \sigma, s) \in T \quad \nu \in M^V \quad M \models_\nu f \quad m \in M}{((s_0, \nu), c?m, (s_1, \nu \circ (x \mapsto m) \circ \sigma))} \quad (3)$$

Règles d'induction :

$$\frac{(e, l, (s_1, \nu))) \quad (s_1, a, f, \sigma, s_2) \in T \quad a \in (\Lambda \cap Act(\Sigma)) \quad \nu \in M^V \quad M \models_\nu f}{((s_1, \nu), a, (s_2, \nu \circ \sigma))} \quad (4)$$

$$\frac{(e, l, (s_1, \nu))) \quad (s_1, c!t, f, \sigma, s_2) \in T \quad \nu \in M^V \quad M \models_\nu f}{((s_1, \nu), c!\nu(t), (s_2, \nu \circ \sigma))} \quad (5)$$

$$\frac{(e, l, (s_1, \nu))) \quad (s_1, c?m, f, \sigma, s_2) \in T \quad \nu \in M^V \quad M \models_\nu f \quad m \in M}{((s_1, \nu), c?m, (s_2, \nu \circ (x \mapsto m) \circ \sigma))} \quad (6)$$

Dans ce qui suit, les transitions qui sont insérées dans $(\!|T|\!)$ en appliquant la règle (N) seront appelées *(N)-transitions*. Dans la définition 3.10, les (1)-, (2)- et

(3)-transitions dénotent la sémantique des transitions de $\mathscr{A}$, dont l'état source est l'état initial de $\mathscr{A}$. Plus précisément, les (1)-transitions dénotent la sémantique des transitions qui contiennent une action de communication silencieuse (τ), ou une communication par signal[1] (de la forme $c!$ ou $c?$). Les (2)-transitions, respectivement (3)-transitions, dénotent la sémantique des transitions qui contiennent une émission valuée de la forme $c!t$, respectivement une réception valuée de la forme $c?x$. Les (4)-, (5)- et (6)-transitions dénotent la sémantique des transitions dont l'état source est un état quelconque en dehors de l'état initial de $\mathscr{A}$, i.e. un état quelconque dans $S\backslash\{s_0\}$, avec la condition que cet état doit être atteignable dans l'ensemble $(\!|\mathscr{A}|\!)$ en cours de construction. On note que dans la définition 3.10, les règles (4), (5) et (6) sont respectivement analogues aux règles (1), (2) et (3).

Remarquons que la définition 3.4 est en accord avec l'idée intuitive d'une transition franchissable : pour qu'une transition puisse être franchie, tout d'abord son état source doit être atteint, et ensuite sa garde doit être satisfaite.

3.2 Spécifications formées d'IOSTS

3.2.1 Définition

Actuellement, il est courant de spécifier un système réactif par une approche modulaire, sous la forme de sous-systèmes concurrents. Dans le cadre de notre approche, une spécification formelle est considérée comme la composition parallèle asynchrone d'automates communicants (IOSTSs).

Définition 3.5 (Spécification) *Une* spécification $\mathscr{S}$ *est un ensemble d'*IOSTS, *défini par* $\mathscr{S} = \{(S_i, s_{0i}, T_i)_{\Sigma_i} \mid 0 \leq i < k \ \wedge \ \Sigma_i = (\Omega_i, V_i, C_i)\}$, *pour un* $k \in \mathbb{N}$, *et tel que pour tout* i *les ensembles de variables attribut* V_i *sont distincts deux à deux. La composition parallèle des automates d'une spécification est dénotée par une sémantique par entrelacements (cf. définition 3.10).*

Dans une spécification, si deux IOSTSs distincts communiquent sur un même canal c (i.e. $c \in C_i \cap C_j$, et $i \neq j$), alors c est un canal de communication interne, sinon c est un canal de communication avec l'environnement du système.

Exemple 3.6 La figure 3.2 montre une représentation graphique d'une spécification de distributeur de billets de banque, formée de deux IOSTS concurrents. Dans

[1] En effet, l'ensemble $(\Lambda \cap Act(\Sigma))$ contient τ, et toutes les expressions qui dénotent une émission ou une réception de signal.

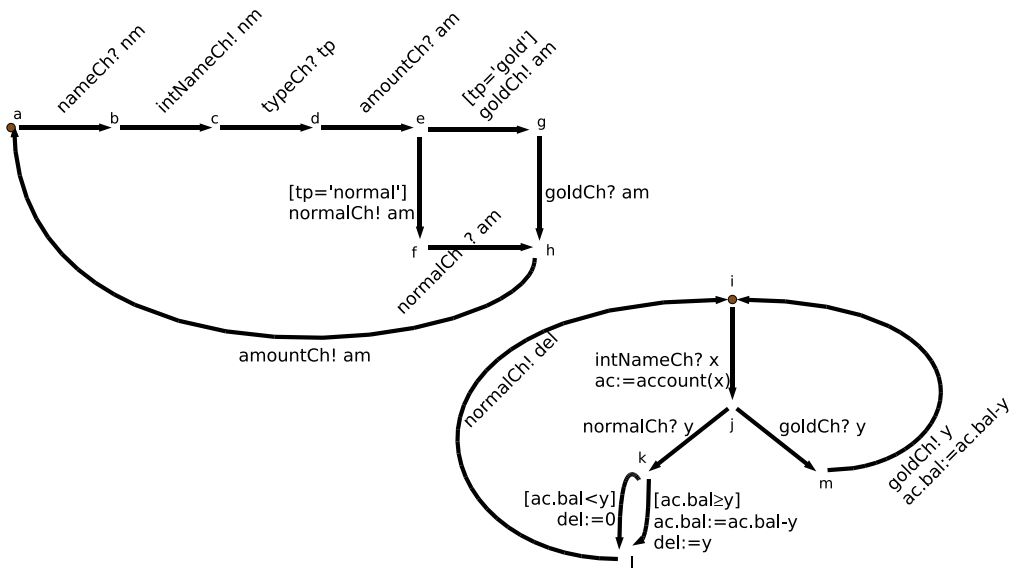


FIG. 3.2 – Exemple d'une spécification formée d'IOSTS.

la figure 3.2, les transitions d'IOSTS sont représentées par des flèches, et les états initiaux sont représentés par des points. Deux types de cartes bancaires peuvent être utilisées pour retirer des billets auprès de la machine spécifiée ici : les cartes *gold* et les cartes *normales*. Dans le cas d'une carte gold, tout découvert est autorisé, alors que dans le cas d'une carte normale, aucun découvert n'est autorisé.

Toujours dans la figure 3.2, l'automate de gauche constitue une interface avec l'environnement, et l'automate de droite effectue les opérations internes avec la banque. Dans l'automate de gauche, les données nécessaires sont acquises à travers les canaux *nameCh* (pour le nom du client), *typeCh* (pour le type de carte ayant été introduite dans la machine), et *amountCh* (le montant que le client a demandé). Ensuite, ces données sont envoyées à travers les canaux internes $intNameCh$, $normalCh$, et $goldCh$, pour que l'automate de droite puisse faire son office. Finalement, l'automate de gauche retourne à l'environnement les données appropriées, à travers le canal *amountCh* (la quantité d'argent qu'il faut délivrer au client). Dans l'automate de droite, le nom du client ainsi que le type de carte de crédit qu'il utilise sont acquis, puis le compte correspondant du client est obtenu par un appel à la fonction d'accès *account*. Ensuite, en fonction des possibilités de découvert autorisé sur le compte en question, la machine vérifie ou non si le solde du compte après retrait serait positif (ici, un compte *ac* est une structure de données qui contient au moins une variable, *bal*, qui indique le solde de *ac*). Enfin, le compte est débité

seulement si le retrait est autorisé, et le montant débité est transmis à travers le canal approprié, *normalCh* ou *goldCh* (le montant mentionné est 0 dans le cas où le retrait n'est pas autorisé).

Notation Dans ce chapitre, ainsi que les suivants, une transition de la figure 3.2 sera notée $\alpha \rightharpoonup \beta$, où α est l'état source et β est l'état cible de la transition. Il existe deux transitions dans l'automate de droite, pour lesquelles cette notation serait ambiguë. Pour ces deux transitions, la notation est raffinée de la façon suivante : $(k, \tau, ac.bal < y, (del \mapsto 0), l)$ sera notée $k \rightharpoonup_1 l$, et $(k, \tau, ac.bal \geq y, (ac.bal \mapsto ac.bal - y, del \mapsto y), l)$ sera notée $k \rightharpoonup_2 l$.

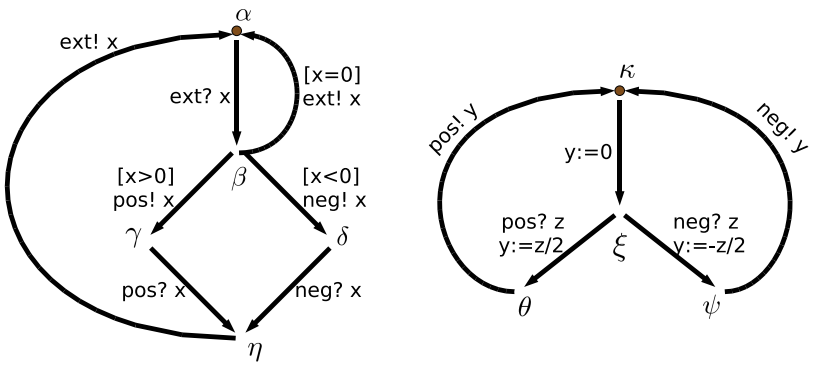


FIG. 3.3 – Un autre exemple de spécification formée d'IOSTS.

Exemple 3.7 La figure 3.3 représente une spécification, formée d'IOSTS, d'une application qui calcule le moitié de la valeur absolue de ses entrées. De même que dans la figure 3.2, les transitions sont représentées par des flèches, et les états initiaux sont représentés par des points. Basiquement, l'automate de gauche a le rôle d'une interface avec l'environnement, il communique avec celui-ci via le canal *ext*, et avec l'automate de droite à travers les canaux internes *pos* et *neg*. L'automate de droite peut être vu comme le "noyau" de l'application : il reçoit les données de l'interface, calcule le résultat escompté, puis le retourne à l'interface (qui à son tour le retournera à l'environnement).

Nous pourrons utiliser la notation ci-dessus pour nommer aussi les transitions de la figure 3.3 : en effet, les espaces de noms d'états sont disjoints entre les deux exemples de spécifications (figures 3.2 et 3.3).

3.2.2 Composition Parallèle

La composition parallèle de deux IOSTS est inhérente à notre définition d'une spécification formée d'IOSTS (définition 3.5). Une définition formelle de cette notion sera donnée par la suite. Préalablement, nous définissions une fonction Booléenne rdv, qui prend pour arguments une transition et un IOSTS, tel que $rdv(tr, \mathscr{A})$ est vrai si et seulement si il existe une transition dans $\mathscr{A}$, avec laquelle un rendez-vous avec la transition tr est possible.

Définition 3.8 (Fonction rdv) *Soit $\mathscr{S} = \{(S_i, s_{0i}, T_i)_{\Sigma_i} \mid 0 \leq i < k \ \wedge \ \Sigma_i = (\Omega_i, V_i, C_i)\}$ une spécification, la fonction Booléenne $rdv : \bigcup_{0 \leq i < k} T_i \times \mathscr{S} \to \mathbb{B}$ est définie formellement par :*

$$\begin{gathered} rdv((s_1, a_1, f_1, \sigma_1, s_1'), (S, s_0, T)_\Sigma) = a_1 \neq \tau \wedge \exists (s_2, a_2, f_2, \sigma_2, s_2') \in T, \\ (a_1 = c! \implies a_2 = c?) \ \wedge \ (a_1 = c? \implies a_2 = c!) \\ \wedge \ (a_1 = c!t \implies a_2 = c?x) \ \wedge \ (a_1 = c?x \implies a_2 = c!t) \end{gathered}$$

Exemple 3.9 Notons $\mathscr{A}$ l'automate de droite dans la figure 3.2. Alors $rdv(e \rightharpoonup f, \mathscr{A})$ est vrai car un rendez-vous est possible entre $e \rightharpoonup f$ et une transition de $\mathscr{A}$, à savoir $j \rightharpoonup k$; par contre, $rdv(a \rightharpoonup b, \mathscr{A})$ est faux car aucune transition de $\mathscr{A}$ n'effectue une réception sur le canal $nameCh$.

La notion de composition parallèle de deux IOSTS est définie formellement par la définition 3.10 : cette composition résulte en un nouvel IOSTS, qui est obtenu en synchronisant les IOSTS originaux par un méchanisme de rendez-vous lorsque c'est possible, et considérant tous les entrelaçements dans les autres cas. La synchronisation d'actions de communication résulte en une action interne du système composé (autrement dit, une action silencieuse, τ).

Définition 3.10 (Composition parallèle d'IOSTS) *Soient $\mathscr{A}_1$ et $\mathscr{A}_2$ deux* IOSTS *tels que $\forall i \in \{1, 2\}$, $\mathscr{A}_i = (S_i, s_{0_i}, T_i)_{\Sigma_i}$, et $\Sigma_i = ((\Theta_i, \Phi_i), V_i, C_i)$. La composition parallèle de $\mathscr{A}_1$ et $\mathscr{A}_2$ est un nouvel* IOSTS *$\mathscr{A} = (S, s_0, T)_\Sigma$, où $\Sigma = (\Omega, V, C)$, tel que :*

$S = S_1 \times S_2$, $s_0 = (s_{0_1}, s_{0_2})$, $\Omega = (\Theta_1 \cup \Theta_2, \Phi_1 \cup \Phi_2)$, $V = V_1 \cup V_2$, $C = C_1 \cup C_2$, et $T \subseteq S \times Act(\Sigma) \times \mathscr{F}_\Omega(V) \times \mathscr{T}_\Omega(V)^V \times S$ est le plus petit ensemble qui contient les transitions construites à l'aide des règles suivantes :

$$\frac{(s_1, a_1, f_1, \sigma_1, s'_1) \in T_1 \quad rdv(tr, \mathscr{A}_2) = faux \quad s_2 \in S}{((s_1, s_2), a_1, f_1, \sigma_1, (s'_1, s_2))} \quad (7)$$

$$\frac{(s_2, a_2, f_2, \sigma_2, s'_2) \in T_2 \quad rdv(tr, \mathscr{A}_1) = faux \quad s_1 \in S}{((s_1, s_2), a_2, f_2, \sigma_2, (s_1, s'_2))} \quad (8)$$

$$\frac{(s_1, c?, f_1, \sigma_1, s'_1) \in T_1 \quad (s_2, c!, f_2, \sigma_2, s'_2) \in T_2}{((s_1, s_2), \tau, f_1 \wedge f_2, \sigma_1 \cup \sigma_2, (s'_1, s'_2))} \quad (9)$$

$$\frac{(s_1, c!, f_1, \sigma_1, s'_1) \in T_1 \quad (s_2, c?, f_2, \sigma_2, s'_2) \in T_2}{((s_1, s_2), \tau, f_1 \wedge f_2, \sigma_1 \cup \sigma_2, (s'_1, s'_2))} \quad (10)$$

$$\frac{(s_1, c?x, f_1, \sigma_1, s'_1) \in T_1 \quad (s_2, c!t, f_2, \sigma_2, s'_2) \in T_2}{((s_1, s_2), \tau, f_1 \wedge f_2, (x \mapsto t) \circ \sigma_1 \cup \sigma_2, (s'_1, s'_2))} \quad (11)$$

$$\frac{(s_1, c!t, f_1, \sigma_1, s'_1) \in T_1 \quad (s_2, c?x, f_2, \sigma_2, s'_2) \in T_2}{((s_1, s_2), \tau, f_1 \wedge f_2, \sigma_1 \cup (x \mapsto t) \circ \sigma_2, (s'_1, s'_2))} \quad (12)$$

Dans la définition 3.10, les règles (7) et (8) dénotent l'ensemble des transitions qui peuvent être exécutées indépendamment ; ces transitions créent potentiellement des entrelacements dans la composition parallèle. La règle (7) dénote l'ensemble des transitions de $\mathscr{A}$ qui résultent du franchissement d'une transition de $\mathscr{A}_1$ en restant dans le même état dans $\mathscr{A}_2$; la règle (8) est symétrique par rapport à la règle (7). Les règles (9), (10), (11) et (12) dénotent les synchronisations possibles de $\mathscr{A}_1$ et $\mathscr{A}_2$: franchir une transition dans un de ces ensembles revient à franchir une transition dans chaque automate. Plus précisément, la règle (9) dénote l'ensemble des transitions qui résultent de la synchronisation d'une émission de signal effectuée en $\mathscr{A}_2$, avec une réception de signal effectuée en $\mathscr{A}_1$, sur le même canal ; la règle (11) est analogue, avec une émission valuée en $\mathscr{A}_2$, et une réception valuée en $\mathscr{A}_1$. Les règles (10) et (12) sont symétriques par rapport aux règles (9) et (11).

Remarquons que les expressions de la forme $\sigma_1 \cup \sigma_2$, utilisées dans la définition des règles (11) et (12), sont des expressions valides pour dénoter une fonction, puisque les domaines de définition de σ_1 et σ_2 sont disjoints.

Exemple 3.11 La figure 3.4 représente un automate qui résulte de la composition parallèle du premier de nos exemples, la spécification représentée en figure 3.2. Dans cet exemple, le système spécifié est hautement synchronisé, et par conséquent il n'y a pas énormément d'entrelacements dans la composition parallèle du système. Comme

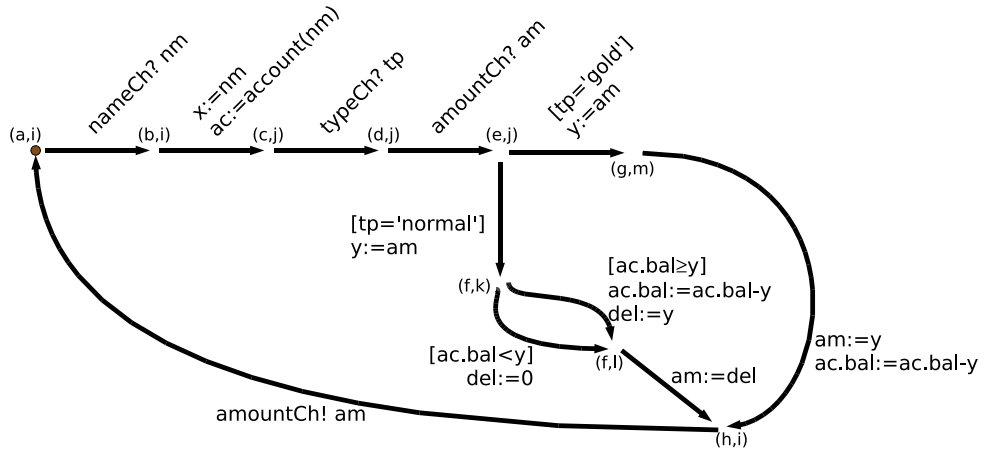


FIG. 3.4 – Composition parallèle de la spécification en figure 3.2.

autre conséquence, la plupart des transitions qui forment la composition parallèle sont de nouvelles transitions, i.e. ont été construites en utilisant les règles (9) à (12) de la définition 3.10.

Par exemple, la transition $(b,i) \rightharpoonup (c,j)$ résulte de la composition parallèle des transitions $b \rightharpoonup c$ et $i \rightharpoonup j$. Examinons plus formellement la construction de cette nouvelle transition : d'après la règle (11) (ou symétriquement (12)) de la définition 3.10, la composition parallèle des transitions $(b, intNameCh!nm, true, Id_{\{nm,tp,am\}}, c)$ et $(i, intNameCh?x, true, (ac \mapsto account(x)), j)$ résulte en une nouvelle transition $((b,i), \tau, true, \sigma, (c,j))$, où $\sigma = (x \mapsto nm, ac \mapsto account(nm))$. Conformément aux attentes, la variable x au niveau des affectations de la transition $i \rightharpoonup j$ réfère à nm (cette référence "passe" par le canal $intNameCh$).

Maintenant, intéressons-nous au cas où l'étiquette d'une transition comporte une réception valuée sur une variable ainsi qu'une affectation de cette même variable. Plus précisément, insérons une affectation de x au niveau de la transition $i \rightharpoonup j$: par exemple, remplaçons $i \rightharpoonup j$ par $i \rightharpoonup_1 j = (i, intNameCh?x, true, (x \mapsto$ "smith", $ac \mapsto account(x)), j)$. Alors, d'après la définition 3.10, la composition parallèle de $b \rightharpoonup c$ et $i \rightharpoonup j$ résulte en la transition $((b,i), \tau, true, \sigma', (c,j))$, où $\sigma' = (x \mapsto$ "smith", $ac \mapsto account(nm))$. Il est intéressant de remarquer qu'en accord avec la sémantique donnée par la définition 3.4, le terme nm qui est réceptionné dans la variable x via le canal $intNameCh$, est écrasé par l'affectation de x en $i \rightharpoonup j$ (i.e. $\sigma'(x) =$ "smith") ; alors que les utilisations de x au niveau des affectations de $i \rightharpoonup j$ sont tout de même reliées à nm, en effet, $\sigma'(ac) \neq account($"smith"$)$.

Enfin, les étiquettes de certaines transitions restent inchangées, car aucune synchronisation n'a lieu d'être, cf. transition $(a, i) \rightharpoonup (b, i)$, qui est construite en utilisant la règle (7) (ou symétriquement, (8)) de la définition 3.10.

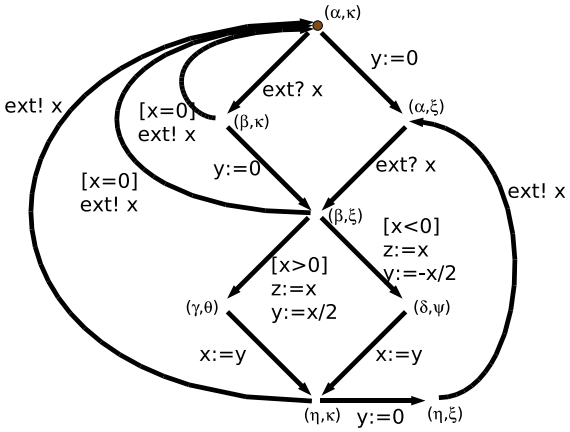


FIG. 3.5 – Composition parallèle de la spécification en figure 3.3.

Exemple 3.12 La figure 3.5 représente la composition parallèle de la spécification en figure 3.3. Cet exemple illustre bien les entrelacements créés par la composition parallèle : dans la spécification de la figure 3.3, les deux transitions $\alpha \rightharpoonup \beta$ et $\kappa \rightharpoonup \xi$ effectuent des actions indépendantes, et peuvent être exécutées à partir de l'état (α, κ) dans un ordre indifférent. Ainsi, les transitions $(\alpha, \kappa) \rightharpoonup (\beta, \kappa)$, $(\alpha, \kappa) \rightharpoonup (\alpha, \xi)$, $(\beta, \kappa) \rightharpoonup (\beta, \xi)$, et $(\alpha, \xi) \rightharpoonup (\beta, \xi)$ sont générées selon les règles (7) et (8) de la définition 3.10. Les transitions résultant de synchronisations sont générées de manière analogue à ce que l'on a vu dans l'exemple précédent 3.11.

3.3 Chemins dans les IOSTS

En supposant que $\mathscr{A} = (S, s_0, T)_\Sigma$ est un IOSTS, Les définitions suivantes introduisent les concepts de *successeur* et *prédécesseur* d'une transition dans $\mathscr{A}$, et de *chemins* et *chemins maximaux* dans $\mathscr{A}$.

Définition 3.13 (Successeur/prédécesseur d'une transition) *Soient tr_i et tr_j deux transitions dans T, tr_j est appelé* successeur *de tr_i dans $\mathscr{A}$ si $source_{tr_j} = cible_{tr_i}$; dans ce cas, on dit aussi que tr_i est un* prédécesseur *de tr_j.*

Définition 3.14 (Chemin dans un automate) *Un chemin p dans $\mathscr{A}$ est une fonction (éventuellement partielle) $\mathbb{N} \rightarrow T$, telle que :*

– *pour tout* $i \in \mathbb{N}$*, si* p *est défini en* $i+1$*, alors* $source_{p(i+1)} = cible_{p(i)}$ *;*

– *pour tout* $i \in \mathbb{N}$*, si* p *est défini en* i*, alors* p *est défini en* j *pour tout* $j \leq i$*.*

Notons que si un chemin est une fonction partielle, alors c'est un chemin fini.

Notations Étant donné une transition tr, on note $preds_{tr}$ l'ensemble des prédécesseurs, et $succs_{tr}$ l'ensemble des successeurs, de tr. Le nombre de successeurs de tr est appelé degré sortant de tr, et sera noté d_{tr} – notons que l'égalité suivante est vérifiée : $d_{tr} = |succs_{tr}|$.

Nous noterons $\langle tr_i, \ldots, tr_j \rangle$ un chemin fini p tel que k est le plus grand entier naturel où p est défini, $p(0) = tr_i$ et $p(k) = tr_j$ (ici nous disons que p est un chemin *depuis* tr_i). La *longueur* du chemin p est alors dénotée par $k+1$.

Exemple 3.15 Dans la figure 3.2, les transitions $e \rightharpoonup f$ et $e \rightharpoonup g$ sont des successeurs de la transition $d \rightharpoonup e$, et $\langle c \rightharpoonup d, d \rightharpoonup e, e \rightharpoonup g \rangle$ est un chemin fini depuis $c \rightharpoonup d$ dans l'automate de gauche.

La définition suivante rappelle la notion de chemin maximal, issue de la thérorie des graphes, et l'étend aux IOSTS. Intuitivement, un chemin maximal dans $\mathscr{A}$ est un chemin qui ne peut être prolongé par aucune transition de l'ensemble T.

Définition 3.16 (Chemin maximal) *Un chemin* p *dans* $\mathscr{A}$ *est dit* maximal *si une des propriétés suivantes est vérifiée :*

– *soit* p *est une fonction totale (i.e.* p *est un chemin infini) ;*

– *soit* p *est une fonction partielle telle que : posons* k *le plus grand entier où* p *est défini, il n'existe aucun chemin* p' *dans* $\mathscr{A}$ *tel que pour tout* $i \leq k$ $p'(i) = p(i)$*, et* p' *est défini en* $k+1$*.*

Exemple 3.17 Le chemin qui consiste en une séquence infinie de transitions dans l'expression régulière $(i \rightharpoonup j, j \rightharpoonup m, m \rightharpoonup i)^*$ est un chemin maximal dans l'automate de droite, en figure 3.2. Toujours en figure 3.2, supposons que l'on insère un nouveau nœud n, ainsi qu'une nouvelle transition $i \rightharpoonup n$ dans l'automate de droite. Alors $i \rightharpoonup n$ n'a aucun successeur dans l'automate de droite, et le chemin $\langle i \rightharpoonup j, j \rightharpoonup m, m \rightharpoonup i, i \rightharpoonup n \rangle$ est un chemin maximal depuis $i \rightharpoonup j$ dans l'automate de droite.

Chapitre 4

Analyse de dépendances dans les spécifications formées d'IOSTS

*Ce chapitre présente notre analyse des dépendances dans les spécifications formées d'*IOSTS*, conformément au cadre théorique défini aux chapitres 2 et 3, et en s'appuyant notamment sur l'état de l'art des analyses de dépendances dans le domaine de la construction de compilateurs.*

Sommaire

4.1 Introduction

Trois types de relations de dépendances seront formellement définis dans ce chapitre, et des algorithmes permettant de les calculer seront étudiés en détail : dépendances de contrôle, dépendances de données, et enfin dépendances de communication.

Les relations de dépendances de donneés et de dépendances de contrôle sont des relations binaires, amplement utilisées notamment dans le domaine de la construction de compilateurs, et de l'analyse de programmes. Ces relations de dépendances sont impliquées dans de nombreuses techniques de transformation et d'optimisation, comme la vectorisation et la parallélisation [AK02], l'ordonnancement d'instructions et l'optimisation de données de cache [Muc97], la division de nœuds, le déplacement de code et la fusion de boucles [FOW87] ; ainsi que dans des techniques d'analyse de programmes comme la réduction paramétrée de programmes [Kri03, OO84, Tip95] (cf. section 1.4).

Présentons l'organisation de ce chapitre. Les sections 4.2 et 4.3 étendent respectivement les concepts de dépendance de données et de dépendance de contrôle au cadre des spécifications formées d'IOSTS. Par la suite, nous observerons que les communications entre IOSTS induisent un autre type de dépendances, ayant une composante de contrôle et une composante de données, mais qui se propagent à travers plusieurs IOSTS. Pour cette raison, ces dépendances ne peuvent être intégrées par les relations de dépendance de contrôle et de données mentionnées ci-dessus, qui définissent les dépendances internes à un IOSTS. Ainsi, nous introduisons en section 4.4 une relation de dépendances supplémentaire – nommées dépendances de communication – pour capturer les dépendances induites par les communications et, comme nous le verrons par la suite, pour faire le lien entre les autres dépendances. Ces relations de dépendances constituent la base de notre approche pour la

réduction paramétrée de spécifications formées d'IOSTS (cf. chapitre 5).

4.2 Dépendances de données

Les dépendances de données sont des contraintes dues au flot de données entre les éléments d'un programme. Comme nous l'avons mentionné dans l'introduction du chapitre 4, les dépendances de données sont utiles dans les domaines de la construction de compilateurs et de l'analyse statique, conduisant à de nombreuses techniques d'optimisation, de transformation et d'analyse. Tout d'abord, cette section étend aux IOSTS le concept de *définitions* et *utilisations* de variables (cf. sections 1.3 et 4.2.1). Ensuite, seront rappelées les différentes idées intuitives sous-jacentes à la notion de dépendance de données (cf. section 4.2.2), ce qui nous mènera à préciser la notion que nous associerons aux dépendances de données dans le cadre de la réduction paramétrée de spécifications formées d'IOSTS (cf. section 4.2.3). Enfin, cette section sera conclue par une description détaillée de notre algorithme pour calculer les dépendances de données dans un IOSTS, qui comprend : une description intuitive en section 4.2.6, une preuve de terminaison en temps polynomial en section 4.2.7, et enfin des preuves de correction et complétude en sections 4.2.8 et 4.2.9.

4.2.1 Définitions et utilisations de variables

La définition d'une relation de dépendances de données fait traditionnellement appel à la notion de *définitions* et *utilisations* de variables. On rappelle que dans un programme, une variable x est dite *définie* par un élément e si e affecte une valeur à x ; x est dite *utilisée* par e si soit e est une affectation et x apparaît dans la partie droite de e, soit e est une condition et x apparaît dans e (cf. section 1.3 page 15, pour les définitions formelles sur les CFG).

À l'aide de la définition suivante, nous proposons une extension de ces notions aux IOSTS. Dans un IOSTS, les variables peuvent être utilisées dans les gardes des transitions et peuvent être définies ou utilisées dans les substitutions de variables, d'une manière analogue à ce qui se passe dans les programmes au niveau des conditions et affectations. Notre extension aux IOSTS doit également prendre en compte les actions de communication, qui impliquent aussi des définitions et utilisations de variables.

Définition 4.1 (Définitions/utilisations de variables) *Soit* $\mathscr{A} = (S, s_0, T)_\Sigma$ *un* IOSTS *de signature* $\Sigma = (\Omega, V, C)$. *Soit une transition* $tr = (s, a, f, \sigma, s')$ *dans* T, *et une variable* x *dans* V.

- x *est* définie *par* tr *si soit* $a = c?x$ *pour un* c *donné dans* C, *soit* $\sigma(x) \neq x$.
- x *est* utilisée *par* tr *si soit* x *est une variable de la formule* f, *soit il existe un terme* $t \in \mathscr{T}_\Omega(V)$ *tel que* x *est une variable du terme* t, *et une des propriétés suivantes est vérifiée :*
  - *soit* $\sigma(y) = t$ *pour un* y *donné dans* V,
  - *soit* $a = c!t$ *pour un* c *dans* C.

Exemple 4.2 Dans la figure 3.2 page 53, x est défini par $i \rightharpoonup j$ car x apparaît dans la partie droite d'une réception valuée, ac est défini par $i \rightharpoonup j$ car ac apparaît dans la partie gauche d'une affectation, x est utilisé par $i \rightharpoonup j$ car x apparaît dans la partie droite d'une affectation, et tp est utilisé par $e \rightharpoonup f$ car tp apparaît dans la garde de $e \rightharpoonup f$.

4.2.2 Définitions traditionnelles

Selon l'application visée, différentes définitions de dépendances de données ont été établies dans la littérature. Dans le cadre le plus général, une relation de dépendances de données est introduite comme la jonction de quatre variétés de dépendances de données [Muc97]. Soient n_i et n_j deux éléments d'un programme, tel que n_i précède n_j dans leur ordre d'exécution donné. Il y a une *dépendance de flot* entre n_i et n_j si n_i définit la valeur d'une variable, et n_j utilise cette valeur pour cette variable ; il y a une *anti-dépendance* entre n_i et n_j si n_i utilise la valeur d'une variable et n_j est l'élément qui positionne cette valeur pour cette variable ; il y a une *dépendance de sortie* entre n_i et n_j si ces deux éléments définissent la valeur d'une même variable ; et enfin, il y a une *dépendance d'entrée* entre n_i et n_j si ces deux éléments utilisent la valeur d'une même variable.

Dans le cadre d'une analyse pour réduction paramétrée, seules les dépendances de flot sont pertinentes, car elles seules suffisent à identifier les éléments qui sont potentiellement impliqués dans les calculs effectués au niveau d'un autre élément. *Dans ce travail, nous restreindrons donc la relation de dépendances de données aux dépendances de flot.*

Toute dépendance de données peut de plus être classifiée comme *portée par une boucle* ou *indépendante des boucles* [FOW87]. Une dépendance de données entre deux éléments est dite portée par une boucle si cette dépendance est due au fait que les deux éléments apparaissent au cours d'une exécution dans deux occurrences distinctes d'une boucle[1], dans le cas contraire, la dépendance est dite indépendante

[1] Par exemple, si le premier élément définit une valeur pour une variable dans une boucle, et le deuxième élément utilise cette valeur dans une itération ultérieure de cette même boucle.

des boucles.

Dans le cadre de la réduction paramétrée d'IOSTS, lorsqu'une transition tr_j qui fait partie du critère de réduction est données-dépendante d'une autre transition tr_i, tr_i doit être intégrée dans la tranche, que la dépendance de données entre tr_i et tr_j soit portée par une boucle, ou indépendante des boucles. Pour cette raison, l'algorithme décrit en section 4.2.6, pour le calcul des dépendances de données dans un IOSTS, permet de calculer les dépendances de données portées par une boucle, ainsi que les dépendances de données indépendantes d'une boucles. Notons que Nanda [Nan01] a montré qu'il s'avère utile de faire cette distinction dans le cadre de la réduction paramétrée de programmes à processus multiples, en présence de processus encapsulés dans des boucles.

4.2.3 Nouvelle définition

Dans un IOSTS, supposons qu'une variable v soit définie par une transition tr_i, et qu'il existe un chemin de tr_i vers tr_j, sur lequel v n'est pas redéfinie, alors on dit que *la définition de v par tr_i atteint tr_j* – cette notion d'atteignabilité est directement adaptée de l'atteignabilité des nœuds d'un CFG par une définition (ce problème est lié à la propagation des définitions, cf. section 2.5 page 40).

Définition 4.3 *Soit $\mathscr{A} = (S, s_0, T)_\Sigma$ un IOSTS de signature $\Sigma = (\Omega, V, C)$. Une définition $d = (tr_i, v) \in T \times V$* atteint *une transition tr_j s'il existe un chemin dans $\mathscr{A}$, de longueur au moins 2 et de la forme $p = \langle tr_i, \ldots, tr_j \rangle$, tel que :*

- *v est défini par tr_i ;*
- *et pour tout $tr \in \langle p(1), \ldots, tr_j \rangle$, v n'est pas défini par tr.*

Dans le cadre des IOSTS, une adaptation directe de la définition traditionnelle des dépendances de données serait de dire : une transition tr_j est données-dépendante d'une transition tr_i dans $\mathscr{A}$ s'il existe une variable v qui soit définie par tr_i et utilisée par tr_j, tel que la définition de v par tr_i atteigne tr_j. Cependant, un examen attentif de la sémantique des IOSTS (cf. définition 3.4 page 51) révèle que ce n'est pas suffisant. En effet, il peut arriver que v soit redéfini par tr_j par l'intermédiaire d'une réception valuée ; or, dans ce cas la définition de v par tr_i ne peut être utilisée par tr_j, à moins que v apparaisse dans la garde de tr_j.

Définition 4.4 (Dépendance de données ($tr_i \xrightarrow{dd} tr_j$)) *Soit $\mathscr{A} = (S, s_0, T)_\Sigma$ un IOSTS de signature $\Sigma = (\Omega, V, C)$. Une transition $tr_j = (s, a, f, \sigma, s') \in T$ est* données-dépendante *d'une transition $tr_i \in T$ s'il existe une définition $d = (tr_d, v) \in T \times V$ et un chemin $p = \langle tr_i, \ldots, tr_j \rangle$ dans $\mathscr{A}$ tel que :*

- *$tr_d = tr_i$ et d atteint tr_j selon la définition 4.3 ;*
- *et une des propriétés suivantes est vérifiée :*
  1. *v apparaît dans la garde f,*
  2. *ou v n'est pas défini par a et v est utilisé par tr.*

Exemple 4.5 Reprenons l'exemple de la figure 3.2 page 53 ; $m \rightharpoonup i$ est données-dépendant de $i \rightharpoonup j$ car il existe une variable (à savoir, ac) qui est définie par $i \rightharpoonup j$, est utilisée par $m \rightharpoonup i$, n'est pas définie par $m \rightharpoonup i$ par l'intermédiaire d'une réception valuée, et de plus il existe un chemin depuis $i \rightharpoonup j$ vers $m \rightharpoonup i$, sur lequel ac n'est pas redéfini – notamment, $\langle i \rightharpoonup j, j \rightharpoonup m, m \rightharpoonup i \rangle$.

À présent, insérons une utilisation de la variable y au niveau de la transition $j \rightharpoonup m$: il y a deux possibilités, soit dans la garde, soit dans les affectations. Dans le premier cas, on peut par exemple remplacer $j \rightharpoonup m$ par $j \rightharpoonup_1 m = (j, goldCh?y, y > 0, Id_{\{ac,x,y,del\}}, m)$. Alors, la définition de y par $j \rightharpoonup k$ peut être utilisée par la garde de $j \rightharpoonup_1 m$, notamment en suivant le chemin $\langle j \rightharpoonup k, k \rightharpoonup_1 l, l \rightharpoonup i, i \rightharpoonup j, j \rightharpoonup_1 m \rangle$. Dans ce cas, $j \rightharpoonup_1 m$ est donc données-dépendant de $j \rightharpoonup k$. Le second cas conduit, par exemple, à remplacer $j \rightharpoonup m$ par $j \rightharpoonup_2 m = (j, goldCh?y, true, (x \mapsto y + 1), m)$. Alors, la définition de y par $j \rightharpoonup k$ ne peut être utilisée par $j \rightharpoonup_2 m$ car la réception valuée sur y au niveau de $j \rightharpoonup_2 m$ écrase cette définition. Dans ce cas, $j \rightharpoonup_2 m$ n'est donc pas données-dépendant de $j \rightharpoonup k$.

En s'appuyant sur le problème de propagation des définitions évoqué plus haut, on voit se dessiner une méthode pour calculer la relation de la définition 4.4 : en supposant que l'on est capable de résoudre le problème de propagation des définitions dans un IOSTS, alors *la relation de dépendance de données sera déterminée en marquant toutes les transitions tr_j comme données-dépendantes d'une transition tr_i, lorsqu'il y a une utilisation au niveau de tr_j d'une définition qui atteint tr_j depuis tr_i et qui vérifie la condition 1. ou 2. de la définition 4.4*. Si l'on se réfère à la section 2.5 page 40, résoudre le problème de propagation des définitions dans un IOSTS consistera à trouver pour chaque transition tr toutes les définitions qui atteignent *potentiellement* tr lors des exécutions possibles de l'automate. Trouver de manière *exacte* les définitions qui atteignent un point donné est en effet indécidable en général ; cela reviendrait à requérir que p soit un chemin *réalisable* dans la définition 4.4.

Comme on l'a vu au chapitre 2, le problème de propagation des définitions est une application classique de la théorie des analyses de flot de données. Une analyse de flot de données est généralement définie comme l'analyse d'un CFG dans un cadre

théorique constitué d'un treillis complet et d'un espace de fonctions sémantiques de transfert monotones [KSV96, Muc97]. Définir une analyse de flot de données dans un tel cadre est avantageux pour prouver la correction et la terminaison. Nous allons montrer par la suite comment définir de façon similaire les analyses de flot de données dans le cadre des IOSTS.

4.2.4 Analyse de flot de données

Pour résoudre une analyse de flot de données sur un IOSTS, on se place dans un cadre théorique constitué d'un treillis et d'un ensemble de fonctions de transfert (de manière analogue aux sections 2.3 page 28 et 2.4 page 34). Le treillis $\mathcal{T}$ dénote un ensemble partiellement ordonné de valeurs – appelées informations de flot de données – qui sont appropriées pour l'analyse ; le but étant qu'à chaque transition de l'automate soit associée une valeur dans $\mathcal{T}$ (on rappelle que dans le cas des CFG, les informations de flot de données sont associées aux nœuds du graphe, puisque c'est là que résident les instructions, cf. section 2.3.3). L'ensemble des fonctions de transfert contient, pour chaque transition tr de l'automate, une fonction de transfert f_{tr}. La fonction f_{tr} permet de calculer les informations de flot de données à transférer aux successeurs de tr, lorsque tr est rencontré au cours de l'analyse (comme dans la section 2.3, on ne s'intéresse ici qu'aux analyses de flot de données *en avant*). *On voit que les conditions sont remplies pour que la résolution de la propagation des définitions dans un* IOSTS *puisse se faire en dérivant des algorithmes standard pour résoudre les analyses de flot de données en avant.*

On trouve dans la littérature deux familles d'algorithmes pour résoudre les analyses de flot de données : les *méthodes par élimination* [RP86, Cif93] et les *méthodes itératives*, e.g. avec liste de travail [Muc97, NNH99], par tourniquet [KU76, NNH99], ou par listing de nœuds [AU75, Ken75]. Les méthodes par élimination sont significativement plus complexes à implanter que les méthodes itératives, et sont bien adaptées pour gérer efficacement les mises à jour d'information de flot de données, dans le cadre d'un processus complexe d'optimisation [Muc97]. Parmi les méthodes itératives, la méthode avec liste de travail est la plus flexible, et permet d'éviter des analyses superflues. Les autres méthodes ne permettent pas cette économie, et analysent systématiquement des parties qui n'apportent pas de nouvelle information – pour cette raison, Tok *et al.* qualifient ces méthodes de *denses* [TGL06]. Par exemple, un algorithme basé sur le principe du tourniquet analysera systématiquement tous les points du programme à chaque itération, jusqu'à ce qu'un point fixe soit trouvé.

Notre algorithme pour calculer les dépendances de données dans un IOSTS est exposé par la suite.

4.2.5 Algorithme générique de flot de données

Comme introduction à notre algorithme, nous proposons une brève description d'un algorithme générique avec liste de travail – en fait ici un *ensemble* de travail, puisque l'algorithme ne requiert pas d'ordonnancement particulier des éléments à traiter – pour résoudre les analyses de flot de données en avant sur les IOSTS. Un ensemble de travail est à la base de cet algorithme, et représente à tout moment l'ensemble des transitions qu'il reste à visiter. Cet ensemble contient initialement l'ensemble des transitions de l'IOSTS. À chaque itération de l'algorithme, une transition tr_l est extraite de l'ensemble de travail pour être traitée. Cela signifie qu'une nouvelle information de flot de données (i.e. un élément du treillis) est calculé pour tr_l, en utilisant la fonction de transfert associée à tr_p, pour chaque prédécesseur tr_p de tr_l. Ensuite, la transition tr_l est retirée de l'ensemble de travail ; si la nouvelle information de flot de données pour tr_l n'est pas incluse dans l'information de flot de données connue au préalable pour tr_l, alors les successeurs de tr_l sont insérés dans l'ensemble de travail. L'algorithme itère jusqu'à ce qu'un point fixe soit atteint pour toutes les informations de flot de données.

4.2.6 Algorithme spécifique

L'algorithme générique de flot de données, brièvement décrit en section 4.2.5, peut être instancié pour résoudre des analyses spécifiques de flot de données en avant, en fournissant le treillis $\mathcal{T} = (\mathcal{L}, \sqsubseteq, \sqcup, \bot)$ approprié, partiellement ordonnée par une relation d'ordre $\sqsubseteq$, ainsi que l'espace de fonctions de transfert approprié, décrit par une fonction sémantique de transfert $[\![]\!] : T \to (\mathcal{L} \to \mathcal{L})$. Nous expliquons par la suite comment l'algorithme 4.1 instancie l'algorithme générique pour permettre de résoudre la propagation des définitions, et par conséquent, de calculer les dépendances de données, dans un IOSTS $\mathscr{A}$.

Propagation des définitions

Une première étape (phase (1) de l'algorithme 4.1) consiste à trouver toutes les définitions et utilisations de variables dans l'automate sous analyse $\mathscr{A}$, au sens de la définition 4.1 ; cette étape requiert un simple parcours de l'ensemble des transitions de $\mathscr{A}$. Durant ce parcours, les définitions et utilisations de variables rencontrées sont

Algorithme 4.1 : Calcul des dépendances de données.

Entrées : $\mathscr{A} = (S, s_0, T)_\Sigma$: un IOSTS, où $\Sigma = (\Omega, V, C)$
Données :
– $def[|T|], ref[|T|]$: des tableaux d'ensembles de variables
– $gen[|T|], nonp[|T|], RD[|T|]$: des tableaux d'ensembles de définitions
– $nouvelleInfo$: un ensemble de définitions
– $ensTravail$: un ensemble de transitions
Sorties : $DD[|T|]$: un tableau d'ensembles de transitions
// À la fin de l'algorithme 4.1, pour chaque $tr \in T$,
// $DD[tr]$ contient toutes les transitions, dont tr est données-dépendante.

/ (1) Initialisation */*
1 $ensTravail \leftarrow T$
2 **pour chaque** $tr = (s, a, f, \sigma, s') \in T$ **faire**
3 $\quad def[tr] \leftarrow \{v \in V \mid \exists c \in C, a = c?v\} \cup \{v \in V \mid \sigma(v) \neq v\}$
4 $\quad ref[tr] \leftarrow \{v \in V \mid \exists c \in C, \exists t \in \mathscr{T}_\Omega(V), (a = c!t) \wedge v \in vars_t\}$
$\qquad\qquad \cup \{v \in V \mid v \in vars_f\} \cup \{v \in V \mid \exists x \in V, \sigma(x) \neq x \wedge v \in vars_{\sigma(x)}\}$
5 $\quad RD[tr] \leftarrow \emptyset$
6 $\quad DD[tr] \leftarrow \emptyset$
7 **pour chaque** $tr = (s, a, f, \sigma, s') \in T$ **faire**
8 $\quad gen[tr] \leftarrow \{(tr_k, v) \in T \times V \mid tr_k = tr \wedge v \in def[tr]\}$
9 $\quad nonp[tr] \leftarrow \{(tr_k, v) \in T \times V \mid tr_k \neq tr \wedge v \in def[tr]\}$

/ (2) Propagation des définitions */*
10 **tant que** $ensTravail \neq \emptyset$ **faire**
11 $\quad tr_l \leftarrow$ `élément`$(ensTravail)$
12 $\quad nouvelleInfo \leftarrow \emptyset$
$\quad$ *// Trouver les définitions qui atteignent tr_l depuis tous ses prédécesseurs,*
$\quad$ *// et en placer l'union dans la variable $nouvelleInfo$.*
13 $\quad$ **pour chaque** $tr_p \in preds_{tr_l}$ **faire**
14 $\quad\quad nouvelleInfo \leftarrow nouvelleInfo \cup (RD[tr_p] \backslash nonp[tr_p]) \cup gen[tr_p]$
$\quad$ *// Si de nouvelles définitions atteignent tr_l, alors on met à jour $RD[tr_l]$,*
$\quad$ *// et on insère tous les successeurs de tr_l dans l'ensemble de travail.*
15 $\quad$ **si** $RD[tr_l] \subset nouvelleInfo$ **alors**
16 $\quad\quad RD[tr_l] \leftarrow nouvelleInfo$
17 $\quad\quad$ **pour chaque** $tr_s \in succs_{tr_l}$ **faire**
18 $\quad\quad\quad ensTravail \leftarrow ensTravail \cup \{tr_s\}$
19 $\quad ensTravail \leftarrow ensTravail \backslash \{tr_l\}$

suite ⊟

Algorithme 4.1 (suite) : Calcul des dépendances de données.

```
   /* (3) Calcul des dépendances de données                          */
20 pour chaque tr = (s, a, f, σ, s') ∈ T faire
21     pour chaque (tr_k, v) ∈ RD[tr] faire
           // La définition (tr_k, v) atteint tr. Alors tr est données-dépendant
           // de tr_k si soit v apparaît dans f, soit v n'est pas définie par a,
           // et dans le même temps v est utilisé en tr.
22         si (v ∈ vars_f) ∨ ((∄c ∈ C, a = c?v) ∧ (v ∈ ref[tr])) alors
23             DD[tr] ← DD[tr] ∪ {tr_k}
24 retourner DD
```

stockées dans les tableaux d'ensembles de variables def et ref, qui sont indexés par l'ensemble des transitions de $\mathscr{A}$ (cette façon de procéder est inspirée de la section 1.3 page 15). Notons def_{tr} et ref_{tr} les ensembles de variables respectivement définies et utilisées par une transition tr, au sens de la définition 4.1. Alors les éléments de tableaux $def[tr]$ et $ref[tr]$ dénotent respectivement les ensembles def_{tr} et ref_{tr}. L'initialisation de def et ref aux lignes 3 et 4 fait appel à des expressions de la forme $vars_t$ et $vars_f$, pour un terme $t \in \mathscr{T}_\Omega(V)$ et une formule $f \in \mathscr{F}_\Omega(V)$; ces expressions dénotent les ensembles de variables qui apparaissent dans t et dans f.

Dans ce qui suit, nous allons définir le cadre théorique de la propagation des définitions dans un IOSTS $\mathscr{A}$, de manière analogue à la section 2.5 page 40. Une définition de variable sera représentée par un couple (tr, v), signifiant que la variable v est définie par la transition tr. L'ensemble $\mathcal{D}_\mathscr{A}$ des définitions de variables dans $\mathscr{A}$ est l'ensemble des informations de flot de données pour la propagation des définitions, et le treillis $\mathcal{T}$ est basé sur l'ensemble des parties finies de $\mathcal{D}_\mathscr{A}$ (cf. figure 4.2).

$$\mathcal{D}_\mathscr{A} = \{(tr, v) \in T \times V \mid v \in def_{tr}\}$$

Exemple 4.6 La figure 4.2 représente le treillis de base pour le problème de propagation des définitions dans un IOSTS $\mathscr{A}$: $(\wp(\mathcal{D}_\mathscr{A}), \subseteq, \cup, \emptyset)$. Les éléments du treillis représentent les valeurs qui peuvent être associées à l'information de flot de données pour chaque transition de l'automate ; ces valeurs sont prises dans $\wp(\mathcal{D}_\mathscr{A})$. Par commodité de notation, on pose $k = |\mathcal{D}_\mathscr{A}|$, et on note $d_1, \ldots, d_k$ les éléments de $\mathcal{D}_\mathscr{A}$.

Par construction, $\mathcal{T}$ est partiellement ordonné par $\subseteq$, l'inclusion d'ensembles. On a $\mathcal{T} = (\wp(\mathcal{D}_\mathscr{A}), \subseteq, \cup, \emptyset)$. La fonction sémantique de transfert est telle que pour

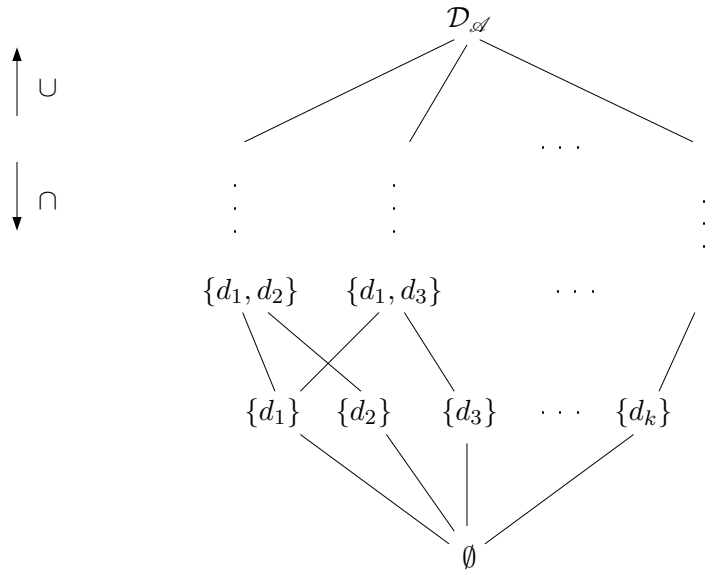


FIG. 4.2 – Treillis de base pour le problème RD dans un IOSTS.

chaque transition tr, $[\![tr]\!] : \wp(\mathcal{D}_{\mathscr{A}}) \to \wp(\mathcal{D}_{\mathscr{A}})$ est une fonction de transfert dédiée à tr. Cela signifie que si l est un ensemble de définitions qui atteint tr, alors $[\![tr]\!](l)$ est l'ensemble des définitions qui atteignent les successeurs de tr. La réalisation de ces fonctions de transfert est permise par le calcul des ensembles de définitions gen_{tr} et $nonp_{tr}$, pour chaque transition tr, toujours en phase (1). L'ensemble gen_{tr} contient les définitions de variables générées par tr, et $nonp_{tr}$ contient l'ensemble des définitions dont la propagation sera bloquée par tr. On définit ces ensembles de manière analogue à la section 2.5.3 page 43 :

$$\begin{aligned} gen_{tr} &= \{(tr, v) \in \mathcal{D}_{\mathscr{A}} \mid v \in def_{tr}\} \\ nonp_{tr} &= \{(tr', v) \in \mathcal{D}_{\mathscr{A}} \mid tr' \neq tr \wedge v \in def_{tr}\} \end{aligned}$$

Ces ensembles sont calculés lors de la phase (1) et stockés dans les tableaux gen et $nonp$: de fait, $gen[tr]$ et $nonp[tr]$ dénotent respectivement gen_{tr} et $nonp_{tr}$. Alors, pour chaque transition tr dans $\mathscr{A}$, $[\![tr]\!]$ est défini par $\forall l \in \wp(\mathcal{D}_{\mathscr{A}}), [\![tr]\!](l) = (l \backslash nonp_{tr}) \cup gen_{tr}$.

À tout moment, $RD[tr]$ contient l'ensemble des définitions dont on sait qu'elle atteignent tr. Ainsi, à chaque étape du déroulement de l'algorithme 4.1, le tableau RD représente l'information couramment connue sur la propagation des définitions. Cette information est mise à jour en phase (2) à chaque fois qu'une transition tr_l

est traitée, à l'aide des fonctions $[\![tr_p]\!]$ pour chaque prédécesseur tr_p de tr_l. Plus précisément, si de nouvelles définitions atteignant tr_l ont été trouvées, l'information connue concernant tr_l (i.e. $RD[tr_l]$) est mise à jour, et tous les successeurs de tr_l sont insérés dans $ensTravail$. L'algorithme 4.1 itère jusqu'à ce qu'un point fixe soit trouvé pour tous les ensembles $RD[tr]$.

Dépendances de données

Lorsque l'information de propagation des définitions est connue, le calcul de la relation de dépendances de données est effectué en phase (3) de l'algorithme, en accord avec la définition 4.4 : une transition tr_j est données-dépendante d'une transition tr_i, s'il existe une définition qui atteigne tr_j depuis tr_i, qui soit utilisée par la garde de tr_j, ou qui soit utilisée par tr_j alors qu'elle n'est pas bloquée par une réception valuée en tr_j. La relation de dépendances de données est stockée dans le tableau DD, qui contient des ensembles de transitions. DD est indexé par l'ensemble des transitions de $\mathscr{A}$, tel que pour toute transition tr dans $\mathscr{A}$, $DD[tr]$ soit l'ensemble des transitions de $\mathscr{A}$ dont tr est données-dépendant.

4.2.7 Analyse de complexité et preuve de terminaison

Nous montrons dans cette section que l'algorithme 4.1 termine, et a une complexité polynomiale.

Lemme 4.7 *La phase (1) de l'algorithme 4.1 termine, avec une complexité en* $O(|T|.|\mathcal{D}_{\mathscr{A}}|)$.

Démonstration. La phase (1) de l'algorithme peut être effectuée à l'aide d'un parcours simple de l'ensemble de transitions T : chaque transition tr est analysée syntaxiquement pour initialiser les ensembles $def[tr]$ et $ref[tr]$, en temps proportionnel au nombre de définitions dans $\mathcal{D}_{\mathscr{A}}$. Remarquons que contrairement aux apparences, dans notre implantation de cet algorithme, les ensembles gen_{tr} et $nonp_{tr}$, pour chaque transition tr, ne sont pas explicitement stockés dans des tableaux ; la raison en sera expliquée dans la preuve du lemme 4.8. Par conséquent, la complexité de la phase (1) de l'algorithme est en $O(|T|.|\mathcal{D}_{\mathscr{A}}|)$. ◇

Lemme 4.8 *La phase (2) de l'algorithme 4.1 termine, avec une complexité en* $O(\sum_{tr \in T} d_{tr}.|T|.|\mathcal{D}_{\mathscr{A}}|.lg(|\mathcal{D}_{\mathscr{A}}|))$.

Démonstration. L'information des définitions générées et non propagées par chaque transition (i.e. les tableaux *gen* et *nonp*) est utilisée seulement à la ligne 14, pour le calcul des définitions qui atteignent tr_l depuis tr_p, tr_p étant un prédécesseur de la transition sous analyse, tr_l. L'objectif de la boucle englobante, à la ligne 13, est de calculer $\bigcup_{tr_p \in preds_{tr_l}}((RD[tr_p] \backslash nonp[tr_p]) \cup gen[tr_p])$, la nouvelle information de flot de données[2] pour tr_l. En se référant à la section 4.2.6, on sait que cela revient à dire que la boucle à la ligne 13 calcule $\bigcup_{tr_p \in preds_{tr_l}} [\![tr_p]\!](RD[tr_p])$. Dans une implantation de cette boucle, calculer séparément les ensembles gen_{tr} et $nonp_{tr}$ puis les stocker dans des tableaux n'est pas nécessaire, puisque $(RD[tr_p] \backslash nonp[tr_p])$ peut être déduit de $RD[tr_p]$ et $def[tr_p]$; et $gen[tr_p]$ peut être obtenu directement par une transcription simple de $def[tr_p]$.

Chaque itération de la boucle principale `tant que` (à la ligne 10) peut contribuer à de futures itérations seulement si la condition à la ligne 15 est satisfaite. Formellement, si la nouvelle information de flot de données $nouvelleInfo$, calculée par la boucle à la ligne 13 pour tr_l, n'est pas comprise dans l'information que l'on connaissait déjà, concernant la propagation des définitions, pour tr_l (i.e. $RD[tr_l]$). Si c'est le cas, chaque successeur de la transition sous analyse est inséré dans l'ensemble de travail ; cela conduit, dans le pire cas, à un total de $\sum_{tr \in T} d_{tr}$ insertions dans l'ensemble de travail – on rappelle que d_{tr} est le degré sortant de tr (cf. section 3.3).

Comme on l'a vu en section 4.2.6, les fonctions de transfert de notre analyse de flot de données sont définies, pour chaque $tr \in T$, par $\forall l \in \wp(\mathcal{D}_{\mathscr{A}}), [\![tr]\!](l) = (l \backslash nonp_{tr}) \cup gen_{tr}$. On a $l, nonp_{tr}, gen_{tr} \in \wp(\mathcal{D}_{\mathscr{A}})$; en se référant à la proposition 2.15 page 33, ces fonctions sont monotones.

Revenons à notre algorithme : comme on l'a vu plus haut, une itération de la boucle principale (ligne 10) contribue à $\sum_{tr \in T} d_{tr}$ nouvelles itérations au pire, si la nouvelle information calculée à la ligne 14 inclut strictement l'information précédente. Comme les fonctions $[\![tr_p]\!]$ sont monotones, et comme le treillis $\mathcal{T}$ construit sur $\wp(\mathcal{D}_{\mathscr{A}})$ est fini, ce cas peut se produire seulement un nombre fini de fois. Ce nombre est, dans le pire cas, la longueur de la plus longue chaîne dans $\mathcal{T}$. La plus longue chaîne dans $\mathcal{T}$ "mène" du plus petit élément $\emptyset$ au plus grand $\mathcal{D}_{\mathscr{A}}$; la longueur de cette chaîne est $|\mathcal{D}_{\mathscr{A}}| + 1$. À partir de là, on sait que la boucle principale termine après $O(|\mathcal{D}_{\mathscr{A}}|.\sum_{tr \in T} d_{tr})$ itérations. Lors de chaque itération de la boucle principale, la boucle à la ligne 13 a un coût en $O(|T|.lg(|\mathcal{D}_{\mathscr{A}}|))$: le facteur $lg(|\mathcal{D}_{\mathscr{A}}|)$ reflète le coût des opérations sur des ensembles, à la ligne 14, qui contiennent au plus tous les éléments de $\mathcal{D}_{\mathscr{A}}$. Enfin, la complexité totale de la phase (2) est

[2] Dans le cas présent, le nouvel ensemble des définitions qui atteignent tr_l.

$O(\sum_{tr \in T} d_{tr}.|T|.|\mathcal{D}_{\mathscr{A}}|.lg(|\mathcal{D}_{\mathscr{A}}|))$. ◇

Lemme 4.9 *La phase (3) de l'algorithme 4.1 termine, avec une complexité en* $O(|T|.|\mathcal{D}_{\mathscr{A}}|)$.

Démonstration. En phase (3), l'algorithme 4.1 calcule la relation de dépendances de données, en comparant les informations de propagation des définitions, obtenues en phase (2), avec les utilisations de variables, conformément ce qui à été décrit en section 4.2.6. Dans le pire des cas, toutes les définitions atteignent chaque transition, et alors la complexité de la phase (3) est en $O(|T|.|\mathcal{D}_{\mathscr{A}}|)$ (de même que la phase (1), cf. lemme 4.7). ◇

Théorème 4.10 (Complexité de l'algorithme 4.1) *L'algorithme 4.1 termine en temps* $O(\sum_{tr \in T} d_{tr}.|T|.|\mathcal{D}_{\mathscr{A}}|.lg(|\mathcal{D}_{\mathscr{A}}|))$.

Démonstration. Le résultat découle des lemmes 4.7, 4.8 et 4.9, en observant que la complexité des phases (1) et (3) est dominée par la complexité de la phase (2). □

4.2.8 Preuve de correction de l'algorithme

Cette section permet d'assurer que l'algorithme 4.1 est correct vis-à-vis de la définition 4.4. On rappelle tout d'abord quelques éléments de la section 4.2.6 : la résolution du problème de propagation des définitions dans un IOSTS $\mathscr{A} = (S, s_0, T)_\Sigma$ – et donc l'algorithme 4.1 – se place dans un cadre formé du treillis $\mathcal{T} = (\wp(\mathcal{D}_{\mathscr{A}}), \subseteq, \cup, \emptyset)$, et de la fonction sémantique de transfert $[\![]\!] : T \to \wp(\mathcal{D}_{\mathscr{A}}) \to \wp(\mathcal{D}_{\mathscr{A}})$ telle que $\forall tr \in T, \forall l \in \wp(\mathcal{D}_{\mathscr{A}}), [\![tr]\!](l) = (l \backslash nonp_{tr}) \cup gen_{tr}$.

La proposition 2.15 page 33 montre que les fonctions $[\![tr]\!]$ sont monotones. De manière analogue à la section 2.5.3 page 43, on étend ces fonctions aux chemins de $\mathscr{A}$:

$$[\![ch]\!] = \begin{cases} Id_{\wp(\mathcal{D}_{\mathscr{A}})} & \text{si } ch = \langle\rangle \\ [\![\langle tr_2, \ldots, tr_k \rangle]\!] \circ [\![tr_1]\!] & \text{si } ch \text{ est de la forme } \langle tr_1, \ldots, tr_k \rangle, \text{ où } k \geq 1 \end{cases}$$

La définition des solutions JOP et MFP dans ce contexte découle naturellement de l'état de l'art (cf. sections 2.5.4 et 2.5.5, page 44) :

$$\forall tr \in T, JOP^{RD}_{(\mathscr{A}, [\![]\!])}(tr) =$$
$$\bigcup \{ [\![ch]\!](\emptyset) \mid ch = \langle tr_i, \ldots, tr_p \rangle, \text{ où } source_{tr_i} = s_0 \text{ et } tr_p \in preds_{tr} \}$$

$$\forall tr \in T, MFP^{RD}_{(\mathscr{A},[\![]\!])}(tr) =$$

$$\begin{cases} \emptyset & \text{si } source_{tr} = s_0 \\ \bigcup\{[\![tr_p]\!](MFP^{RD}_{(\mathscr{A},[\![]\!])}(tr_p)) \mid tr_p \in T \wedge tr_p \in preds_{tr})\} & \text{sinon} \end{cases}$$

Lemme 4.11 *Pour toute transition tr, les valeurs prises par l'ensemble $RD[tr]$ à la fin de chaque itération de la phase (2) de l'algorithme 4.1 forment une chaîne ascendante dans le treillis $\mathcal{T}$.*

Démonstration. Soit tr une transition, notons $RD^{(i)}[tr]$ la valeur de l'ensemble $RD[tr]$ à la fin de l'itération i de la phase (2). Considérons une itération I quelconque de la phase (2). Si la ligne 16 n'est pas exécutée lors de l'itération I, RD n'est pas modifié, et l'on a $RD^{(I)}[tr] = RD^{(I-1)}[tr]$. Si la ligne 16 est exécutée lors de I, alors le test à la ligne 15 est nécessairement vérifié lors de I, i.e. $RD^{(I-1)}[tr] \subset nouvelleInfo$. Dans ces conditions, l'exécution de la ligne 16 implique que $RD^{(I-1)}[tr] \subset RD^{(I)}[tr]$. Quel que soit $I > 0$, on a donc $RD^{(I-1)}[tr] \subseteq RD^{(I)}[tr]$. ◇

Lemme 4.12 *Si une définition d atteint une transition tr, alors à la fin de la phase (2) de l'algorithme 4.1, on a $d \in RD[tr]$.*

Démonstration. Au cours de cette démonstration, pour toute transition tr, $RD[tr]$ fera référence à la valeur de $RD[tr]$ à la fin de l'algorithme 4.1 (ces valeurs existent puisque l'algorithme termine, cf. théorème 4.10), et l'on notera $E^{(i)}$ l'ensemble E à la fin de la i-ème itération de la phase (2). Soient $d = (tr_d, v) \in T \times V$ une définition, et $tr \in T$ une transition. Supposons que d atteigne tr ; selon la définition 4.3, il existe un chemin de longueur au moins 2, qui mène de tr_d à tr. On se propose de montrer le résultat par récurrence sur la longueur n de ce chemin noté $\langle tr_d, \ldots, tr \rangle$.

- Cas de base : $n = 2$, i.e. tr_d est un prédécesseur de tr. À l'issue de la phase (1), on a $d \in gen[tr]$ (par construction). Comme $ensTravail$ est initialisé à T en phase (1), et comme l'algorithme 4.1 termine (cf. théorème 4.10), il existe une itération I de la boucle à la ligne 10, où la transition tr est extraite de l'ensemble de travail ; tr_l est alors identifié à tr, à la ligne 11. Comme $tr_d \in preds_{tr}$ et $d \in gen[tr_d]$, la boucle à la ligne 13 a pour effet que $d \in nouvelleInfo^{(I)}$. Si le test à la ligne 15 est vérifié, alors la ligne 16 est exécutée, et l'on a $d \in RD^{(I)}[tr]$. Si le test à la ligne 15 n'est pas vérifié, alors la ligne 16 n'est pas exécutée, et l'on a $RD^{(I-1)}[tr] = RD^{(I)}[tr]$. Or dans ce cas, le fait que les $RD^{(i)}[tr]$ forment une chaîne (cf. lemme 4.11) implique que $RD^{(I-1)} \supseteq nouvelleInfo$. De nouveau, on a donc $d \in$

$RD^{(I)}[tr]$. Or, d'après le lemme 4.11, pour tout $J \geq I$, on a $RD^{(J)}[tr] \supseteq RD^{(I)}[tr]$. On en conclut que l'on a bien $d \in RD[tr]$.

- Cas général : $n > 2$. Par hypothèse, d atteint tr ; selon la définition 4.3, cela implique qu'il existe un prédécesseur tr' de tr tel que d atteint tr', et tel que $d \notin nonp_{tr'}$. D'autre part, on sait que si d atteint un prédécesseur de tr, c'est via un chemin de longueur strictement inférieure à n. Par hypothèse de récurrence, on a donc $d \in RD[tr']$. Pour que $RD[tr']$ soit non-vide, la ligne 16 a nécessairement été exécutée pour tr', autrement dit, il existe une itération J de la phase (2), où tr' est extraite de l'ensemble de travail, où tr_l est identifié à tr', et telle que $d \in RD^{(J)}[tr']$. Lors d'une itération de la phase (2), l'exécution de la ligne 16 implique l'exécution de la boucle à la ligne 17 ; tous les successeurs de tr' – et notamment tr – sont donc insérés dans l'ensemble de travail lors de l'itération J. Par le théorème 4.10, l'algorithme 4.1 termine, et il existe donc une itération K de la phase (2), ultérieure à l'itération J, où tr est extraite de l'ensemble de travail. Le lemme 4.11 implique que $d \in RD^{(K)}[tr']$; comme de plus $tr' \in preds_{tr}$, la boucle à la ligne 13 a pour effet que $d \in nouvelleInfo^{(K)}$. La suite de la preuve est identique à la fin de la preuve pour le cas de base (en remplaçant I par K) : que le test à ligne 15 soit vérifié ou non, on a $d \in RD[tr]$. ◇

Théorème 4.13 (Correction de l'algorithme 4.1) *Soient tr_i, tr_j deux transitions de l'automate $\mathscr{A}$. Si $tr_i \xrightarrow{dd} tr_j$, alors à la fin de l'algorithme 4.1, on a $tr_i \in DD[tr_j]$.*

Démonstration. Posons $tr_j = (s, a, f, \sigma, s')$, et supposons qu'à la fin de l'algorithme 4.1, on ait $tr_i \notin DD[tr_j]$. Par construction de la boucle en phase (3) à la ligne 20, pour que l'on ait $tr_i \notin DD[tr_j]$, il faut que la ligne 23 ne soit pas exécutée avec tr_j identifié à tr, et tr_i identifié à tr_k. Pour cela, il y a deux possibilités : soit il n'existe aucune variable v telle que $(tr_i, v) \in RD[tr_j]$, soit il existe une telle variable, mais le test à la ligne 22 échoue.

Dans le premier cas, d'après le lemme 4.12, cela signifie qu'aucune définition de tr_i n'atteint tr_j. La définition d'une dépendance de données (définition 4.4 page 65) n'est alors pas respectée ; on n'a donc pas $tr_i \xrightarrow{dd} tr_j$. Dans le second cas, examinons le test à la ligne 22. Pour rendre ce test faux, il faut que v n'apparaisse pas dans la garde de tr_j, et que v soit défini par l'action de communication de tr_j, ou v ne soit pas utilisé par tr_j. À nouveau, la définition 4.4 n'est pas respectée, et l'on n'a donc pas $tr_i \xrightarrow{dd} tr_j$. □

4.2.9 Preuve de complétude de l'algorithme

Dans cette section, nous démontrons que l'algorithme 4.1 est complet vis-à-vis de la définition 4.4.

Lemme 4.14 *Pour toute transition $tr \in T$, $MFP^{RD}_{(\mathscr{A},[\![])}(tr)$ est l'ensemble des définitions qui atteignent tr.*

Démonstration. Le théorème de coïncidence (2.18 page 37), et la proposition 2.21 page 39, étendus aux IOSTS, montrent que la solution MFP décrite en section 4.2.8 coïncide avec la solution JOP, i.e. la solution recherchée. ◇

Lemme 4.15 *Si à la fin de la phase (2) de l'algorithme 4.1, on a $d \in RD[tr]$, alors la définition d atteint la transition tr.*

Démonstration. En se basant sur le résultat du lemme 4.14, le lemme 4.15 revient à montrer que pour tout tr dans T, à la fin de l'algorithme 4.1 on a $RD[tr] \subseteq MFP^{RD}_{(\mathscr{A},[\![])}(tr)$. On le montre par induction sur le nombre n d'itérations de la phase (2) que $RD[tr] \subseteq RD'[tr]$. On notera $E^{(n)}$ l'ensemble E à l'issue de la n-ème itération de la phase (2).

- Cas $n = 0$. Au début de la phase (2), pour tout $tr \in T$, on a $RD^{(0)}[tr] = \emptyset$ (par construction de la boucle à la ligne 2). Le cas de base est donc trivialement vérifié.
- Étape d'induction, $n > 0$. Lors de la n-ème itération de la phase (2), i.e. de la boucle à la ligne 10, parmi tous les ensembles $RD^{(n)}$, seul l'ensemble $RD^{(n)}[tr_l]$ associé à la transition tr_l en cours d'analyse est potentiellement modifié. Par hypothèse d'induction, pour toute transition $tr \in T \backslash \{tr_l\}$, on a $RD^{(n-1)}[tr] \subseteq MFP^{RD}_{(\mathscr{A},[\![])}(tr)$. Pour ces transitions, comme on a $RD^{(n)}[tr] = RD^{(n-1)}[tr]$, l'induction est donc vérifiée à l'itération n.

Maintenant, considérons la transition tr_l. Si l'état source de tr_l est s_0, tr_l n'a aucun prédécesseur, et par construction la boucle à la ligne 13 n'est alors pas exécutée. Dans ce cas $RD^{(n)}$ reste donc à $\emptyset$, et l'induction est donc vérifiée à l'itération n. Si tr_l a au moins un prédécesseur, à l'issue de la boucle à la ligne 13, $nouvelleInfo^{(n)}$ contient l'union des ensembles $(RD^{(n-1)}[tr_p] \backslash nonp[tr_p]) \cup gen[tr_p]$, pour tous les prédécesseurs tr_p de la transition tr_l ; autrement dit, $nouvelleInfo^{(n)}$ contient $\bigcup [\![tr_p]\!](RD^{(n-1)}[tr_p])$. En section 4.2.8, on a vu que $MFP^{RD}_{(\mathscr{A},[\![])}(tr) = \bigcup \{[\![tr_p]\!](MFP^{RD}_{(\mathscr{A},[\![])}(tr_p))\}$. Or, par hypothèse d'induction, pour chaque tr_p on a $RD^{(n-1)}[tr_p] \subseteq MFP^{RD}_{(\mathscr{A},[\![])}(tr_p)$. La monotonicité des fonctions $[\![tr_p]\!]$ (par la proposition 2.15 page 33) implique donc que $nouvelleInfo^{(n)} \subseteq MFP^{RD}_{(\mathscr{A},[\![])}(tr)$ $(*)$. Dans

le reste de cette n-ème itération de la phase (2), soit $RD[tr_l]$ reste inchangé, auquel cas l'induction est vérifiée à l'itération n, de manière analogue au cas des transitions de $T \backslash \{tr_l\}$; soit $RD^{(n)}[tr_l]$ prend la valeur de $nouvelleInfo^{(n)}$ à la ligne 16, et dans ce cas, $(*)$ implique que l'induction reste vérifiée à l'itération n. $\diamond$

Théorème 4.16 (Complétude de l'algorithme 4.1) *Soit tr_i et tr_j deux transitions de l'automate $\mathscr{A}$. Si à la fin de l'algorithme 4.1, on a $tr_i \in DD[tr_j]$, alors $tr_i \xrightarrow{dd} tr_j$ est vérifié.*

Démonstration. On remarque tout d'abord que seule la ligne 23 peut provoquer l'insertion d'un élément dans l'ensemble DD associé à une transition. Supposons que tr_j ne soit pas données-dépendant de tr_i. Selon la définition 4.4 page 65, cela revient exactement à dire qu'une des conditions suivantes est vérifiée :

- soit aucune définition de tr_i n'atteint tr_j, auquel cas d'après le lemme 4.15 $RD[tr_j]$ ne contient en phase (3) aucune définition de la forme (tr_i, v). Par conséquent, la ligne 23 ne peut être exécutée avec tr_j identifié à tr et tr_i identifié à tr_k, ce qui implique qu'à la fin de l'algorithme 4.1, on a $tr_i \notin DD[tr_j]$.
- soit il existe une définition (tr_i, v) qui atteint tr_j, et la variable v est telle que v n'apparaît pas dans la garde de tr_j, et soit v est définie par l'action de communication de tr_j, soit v n'est pas utilisée par tr_j. Or, cet ensemble de cas correspond exactement à la négation du test à la ligne 22. Dans aucun de ces cas le test ne sera donc vérifié, et la ligne 23 ne sera donc pas exécutée avec tr_j identifié à tr et tr_i identifié à tr_k. De même que dans le cas précédent, on a donc $tr_i \notin DD[tr_j]$ à la fin de l'algorithme 4.1. $\square$

4.3 Dépendances de contrôle

Intuitivement, un élément n d'un programme est contrôle-dépendant d'un autre élément m, si lorsqu'une exécution du programme atteint m, il est possible de faire un choix qui détermine si n sera exécuté ou non dans la suite de l'exécution. Autrement dit, n n'est pas contrôle-dépendant de m si lorsque l'on retire m du programme, n est exécuté exactement dans les mêmes conditions. Cette section est organisée de la façon suivante : tout d'abord, nous nous intéresserons à l'état de l'art sur les dépendances de contrôle (c'est-à-dire, les dépendances dans des programmes impératifs, cf. sections 4.3.1 et 4.3.2), puis nous exhiberons les obstacles à l'application de ces définitions dans le cadre des IOSTS. Enfin, nous proposerons une solution pour pallier à ces difficultés, sous la forme d'une définition (cf. section 4.3.3) et d'un algorithme détaillé (cf. section 4.3.4). Nous prouverons respectivement en sections 4.3.5,

4.3.6 et 4.3.7, la terminaison en temps polynomial, la correction et la complétude de cet algorithme.

4.3.1 Définition traditionnelle

Les dépendances de contrôle sont traditionnellement définies en termes d'une *relation de post-dominance* dans un *graphe de flot de contrôle*, CFG en abrégé. Comme nous l'avons vu en section 1.3 page 15, un CFG est un graphe qui représente un programme séquentiel et impératif, et possède notamment la propriété de comporter un unique nœud de sortie. Intuitivement, un nœud n_i est post-dominé par un nœud n_j dans un CFG si dans toute exécution du programme, l'exécution de n_j est toujours précédée par l'exécution de n_i. Une façon courante de définir cette relation est la suivante : *dans un* CFG, *un nœud* n_i *est* post-dominé *par un nœud* n_j *si tous les chemins depuis* n_i *jusqu'au nœud de sortie contiennent* n_j. Toutes les méthodes que nous avons rencontré dans la littérature, pour calculer une relation de post-dominance, sont basées sur une définition semblable, nécessitant que la structure sous analyse possède un unique nœud de sortie.

Exemple 4.17 Dans le CFG représenté en figure 1.1 page 16, le nœud `4` est post-dominé par le nœud `10`, mais pas par le nœud `6`.

Le calcul de la relation de post-dominance dans un CFG peut être formulé comme une analyse de flot de données *en arrière*, où l'information est propagée depuis le nœud de sortie du CFG, dans le sens inverse du flot de contrôle (cf. section 2.3 page 28). Par contre, dans ce cas l'information n'est pas calculée en fonction des données, mais plutôt de la structure du flot de contrôle ; c'est pourquoi on parle plutôt d'*analyse de flot de contrôle* [Muc97]. La mise en œuvre peut aussi se faire en appliquant des algorithmes pour le calcul de la relation de dominance[3] [LT79, Har85, CHK01] dans le CFG *inverse* (c'est-à-dire, dans lequel on a inversé tous les arcs, et interverti les nœuds n_e et n_s). Après avoir calculé la relation de post-dominance, la relation de dépendance de contrôle peut être calculée en accord avec la définition traditionnelle d'une dépendance de contrôle [Muc97, AK02] (cf. définition 4.18).

Définition 4.18 (Dépendance de contrôle traditionnelle [Muc97]) *Un élément* n_j *du programme analysé est dit* contrôle-dépendant *d'un autre élément* n_i *si il existe un chemin non trivial c, depuis* n_i *jusqu'à* n_j*, tel que chaque élément* $n_k \neq n_i$ *dans c soit post-dominé par* n_j*, et* n_i *ne soit pas post-dominé par* n_j*.*

[3] Duale de la relation de post-dominance : dans un CFG, un nœud n_i est *dominé* par un nœud n_j si tous les chemins depuis le nœud d'entrée jusqu'à n_i contiennent n_j.

Exemple 4.19 Dans le CFG de la figure 1.1 page 16, les nœuds `6`, `7` et `8` sont contrôle-dépendants du nœud `5` ; par contre, le nœud `10` n'est pas contrôle-dépendant du nœud `5`.

La définition 4.18 est bien en accord avec le concept intuitif de dépendance de contrôle, puisqu'elle permet d'identifier les éléments d'un programme qui peuvent influencer l'exécution d'autres éléments. Dans les IOSTS, les éléments analogues aux éléments d'un programme sont situés sur les transitions (e.g. gardes, affectations, cf. définition 3.2 page 49). Par conséquent, le concept de dépendance de contrôle pourrait à première vue être étendu aux IOSTS en redéfinissant la relation de post-dominance entre les transitions d'un IOSTS, plutôt qu'entre les nœuds d'un CFG. Cependant, cette définition suppose que la structure analysée possède un unique nœud de sortie, ce qui n'est pas le cas des automates ou structures modernes de programmes (et notamment des IOSTS), qui peuvent avoir plusieurs nœuds de sortie ou même aucun nœud de sortie. Le problème des nœuds de sortie multiples peut être traité en insérant dans la structure un nœud de sortie additionnel, ainsi que des arcs depuis chaque nœud de sortie original vers le nouveau nœud de sortie [Muc97, RAB+05].

La principale vocation des IOSTS est de spécifier et étudier des systèmes réactifs, lesquels ont pour caractéristique de s'exécuter indéfiniment, et par conséquent n'ont habituellement pas d'état(s) de sortie spécifique. Pour ce type de systèmes, il n'est pas évident de déterminer automatiquement où insérer un état de sortie additionnel, tout en préservant les dépendances de contrôle du système avant modification. Nous avons en effet observé qu'insérer un état de sortie arbitraire, ainsi que des transitions vers cet état, de manière analogue à la technique décrite plus haut sur les CFG, induit la plupart du temps des dépendances de contrôle additionnelles. Ces dépendances artificielles provoquent une sur-approximation, indésirable lorsque l'on souhaite calculer précisément la relation de dépendance de contrôle.

4.3.2 Définition récente

Les travaux récents de Ranganath *et al.* [RAB+05] établissent de nouvelles définitions pour calculer les dépendances de contrôle dans un CFG, sans imposer aucune restriction sur l'existence de nœuds de sortie dans le programme. La principale idée menant à ce résultat est que, en ce qui concerne les dépendances de contrôle, atteindre de nouveau un nœud dans un programme réactif est analogue à atteindre un nœud de sortie dans un programme classique. Dans le papier [RAB+05], la définition d'un nœud n_j comme étant contrôle-dépendant d'un nœud n_i étudie les occurrences

de n_j dans les chemins maximaux depuis n_i, plutôt que dans les chemins depuis n_i jusqu'à un nœud de sortie supposé unique (comme dans les définitions précédentes). Plus précisément, un nœud n_j est dit contrôle-dépendant d'un nœud n_i si n_i a au moins deux successeurs n'_i et n''_i tels que tous les chemins maximaux depuis n'_i contiennent n_j, et il existe un chemin depuis n''_i qui ne contient pas n_j.

4.3.3 Nouvelle définition

Les IOSTS et les structures analysées par la méthode de [RAB+05] ont en commun de spécifier des systèmes réactifs, et donc de ne pas imposer de restriction sur le nombre d'états de sortie. Une condition Booléenne dans les CFG de Ranganath *et al.* correspond forcément à un branchement du flot de contrôle ; par contre, dans les IOSTS les conditions Booléennes sont situées au niveau des transitions après le branchement (e.g. transitions $e \rightharpoonup f$ et $e \rightharpoonup g$, en figure 3.2 page 53). Cette différence nécessite de donner une définition des dépendances de contrôle, qui prend en compte les spécificités des IOSTS. On se base sur l'observation suivante : pour qu'une transition soit exécutable, il faut que son état source soit atteignable. De façon informelle, on dira qu'une transition tr_j est contrôle-dépendante d'une transition tr_i s'il y a un branchement au niveau de tr_i qui a un impact direct sur la possibilité d'exécuter tr_j : plus précisément, s'il y a un branchement à partir de l'état source de tr_i, tel que tous les chemins maximaux depuis tr_i contiennent l'état source de tr_j, et s'il existe un chemin maximal depuis une autre branche, qui ne contient pas l'état source de tr_j. De plus, pour que tr_i puisse vraiment avoir un impact sur la possibilité d'exécuter tr_j, il faut que la garde de tr_i ne soit pas trivialement évaluée à *vrai* pour tout paramétrage – on dira que la garde soit *non-triviale*. Dans ce qui suit, on se propose de formaliser cette définition, puis de la valider en s'appuyant sur des exemples. On étudiera ainsi le cas des boucles potentiellement infinies, puis tous les cas possibles de branchements dans un IOSTS.

Définition 4.20 (Dépendance de contrôle ($tr_i \xrightarrow{cd} tr_j$)) *Soit $\mathscr{A} = (S, s_0, T)_\Sigma$ un* IOSTS *de signature $\Sigma = (\Omega, V, C)$. Une transition $tr_j \in T$ est* contrôle-dépendante *d'une transition $tr_i \in T$ si les propriétés suivantes sont vérifiées :*

- *tr_i a une garde non-triviale ;*
- *$source_{tr_j}$ apparaît sur tous les chemins maximaux depuis tr_i dans $\mathscr{A}$;*
- *et il existe une transition tr_k telle que :*
  - *$source_{tr_k} = source_{tr_i}$*
  - *et il existe un chemin maximal depuis tr_k dans $\mathscr{A}$, sur lequel $source_{tr_j}$ n'apparaît pas.*

Dans un IOSTS, l'exécution infinie d'une boucle peut empêcher l'exécution de certaines transitions[4] ; selon l'idée intuitive d'une dépendance de contrôle décrite en introduction de la section 4.3, ce cas de figure induit une dépendance de contrôle entre ces transitions. *On sait que le problème de savoir si une boucle termine est indécidable en général ; de plus, on sait que pour qu'une analyse statique soit correcte, il faut qu'elle prenne en compte toutes les exécutions possibles du système. Nous partirons donc du principe que chaque boucle dans un* IOSTS *est potentiellement infinie* : dans le cas contraire, la relation de dépendance de contrôle omettrait les dépendances induites par les boucles infinies, ce qui contredirait le principe de correction des analyses statiques. En outre, le fait que la relation de dépendance de contrôle, impliquée dans une méthode de réduction paramétrée, prenne en compte la non-terminaison potentielle des boucles, est essentiel pour que les tranches calculées préservent les propriétés dynamiques de la spécification originale, vis-à-vis du critère (e.g. dans le cas où l'application visée est la vérification de propriétés).

Exemple 4.21 Dans la figure 3.2 page 53, on considère le branchement au niveau de e, l'état source de $e \rightharpoonup f$ et $e \rightharpoonup g$: de façon évidente, tous les chemins maximaux à partir de $e \rightharpoonup g$ passent par l'état g, qui est source de $g \rightharpoonup h$. De plus il existe un chemin maximal depuis $e \rightharpoonup f$ sur lequel l'état g n'est jamais atteint – notamment, le chemin maximal constitué de la séquence infinie de transitions dans $(e \rightharpoonup f, f \rightharpoonup h, h \rightharpoonup a, a \rightharpoonup b, b \rightharpoonup c, c \rightharpoonup d, d \rightharpoonup e)^*$. On est dans le cas d'une boucle potentiellement infinie à partir de $e \rightharpoonup f$, qui empêche l'exécution de $g \rightharpoonup h$; on souhaite donc que $g \rightharpoonup h$ soit contrôle-dépendant du branchement au niveau de l'état e. Si l'on retire la garde de $e \rightharpoonup g$, il devient possible d'exécuter $g \rightharpoonup h$ lorsque tp s'évalue à '*normal*' ; dans ce cas, les propriétés dynamiques de $g \rightharpoonup h$ ne sont pas conservées. Par contre, si l'on retire la transition $e \rightharpoonup f$, les possibilités pour exécuter $g \rightharpoonup h$ sont inchangées. Notre définition informelle et l'intuition concordent donc sur le fait que $g \rightharpoonup h$ est contrôle-dépendant de $e \rightharpoonup g$ et non de $e \rightharpoonup f$.

Exemple 4.22 Considérons à nouveau le branchement au niveau de e dans la figure 3.2. On constate que tous les chemins maximaux à partir de $e \rightharpoonup f$ et $e \rightharpoonup g$ passent par l'état h, donc selon notre définition informelle d'une dépendance de contrôle, $h \rightharpoonup a$ n'est contrôle-dépendant ni de $e \rightharpoonup f$ ni de $e \rightharpoonup g$. À nouveau, ce résultat est conforme à l'intuition, puisque le retrait des gardes de $e \rightharpoonup f$ et $e \rightharpoonup g$ ne change en rien les conditions d'exécution de $h \rightharpoonup a$.

[4] Par exemple, l'exécution d'une transition qui n'est pas dans une boucle l, mais est un successeur d'une transition dans l, sera empêchée lorsque l s'exécute de manière infinie.

Jusqu'à présent dans la discussion, nous avons envisagé seulement le cas où les branchements dans la spécification sous analyse sont déterministes et couvrent toutes les possibilités – on les appellera branchements déterministes *complets*. Or dans les spécifications formées d'IOSTS, il y a trois autres cas à étudier. Un branchement peut être déterministe sans prendre en compte toutes les possibilités ; dans ce cas, on se ramène aisément au cas complet, sans modifier la sémantique de la spécification, en ajoutant au branchement une transition vers un état puits, dont la garde est la négation de la disjonction des gardes des autres transitions du branchement.

Exemple 4.23 Imaginons que dans la spécification de la figure 3.2, un troisième type de carte de crédit existe, disons '*master*'. Cette possibilité n'est alors pas prise en compte par le branchement au niveau de l'état e. Sur cet exemple, on se ramène au cas complet en ajoutant un état puits, ainsi qu'une transition de e vers cet état, dont la garde est $\neg(tp = \text{'}normal\text{'} \vee tp = \text{'}gold\text{'})$.

Un branchement peut aussi être non déterministe, avec seulement des gardes triviales ; ces branchements sont habituellement représentés avec des transitions sans garde (cf. figure 3.2, transitions $j \rightharpoonup k$ et $j \rightharpoonup m$). Dans ce cas, d'après la définition 4.20, les transitions du branchement n'induisent pas de dépendance de contrôle. Ce résultat est conforme à l'intuition, puisque ces transitions ne modifient pas les conditions d'exécution d'autres transitions.

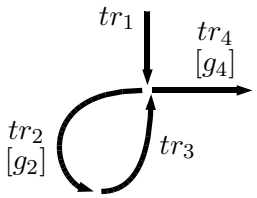


FIG. 4.3 – Exemple de branchement non déterministe

Exemple 4.24 La figure 4.3 représente un IOSTS de manière abstraite. Supposons que dans cet automate, les gardes g_2 et g_4 soient trivialement vraies ; on est dans le cas d'un branchement non déterministe avec gardes triviales. Sur cet exemple, les conditions d'atteignabilité de l'état source de la transition tr_3 sont indépendantes des transitions tr_2 et tr_4.

Enfin, un branchement peut être non déterministe avec des gardes non triviales. On montre à l'aide de l'exemple suivant que la définition 4.20 n'est pas suffisante

dans ce cas. Le cas de ces branchements ne sera pas explicitement traité par notre méthode. Comme on le mentionnera par la suite, il est toutefois aisé, à partir de la définition 4.20 et de l'algorithme 4.4 correspondant, d'écrire une définition de dépendance de contrôle et un algorithme pour prendre en compte ce cas. Si de plus le branchement n'est pas complet, on se ramène au cas complet à l'aide d'un état puits, de la même façon que pour le cas déterministe non complet.

Exemple 4.25 Soit x une variable de l'automate de la figure 4.3, de type entier. Supposons que $[g_2]$ soit $[x \leq 0]$, et que $[g_4]$ soit $[x > 0]$; on est alors dans le cas d'un branchement déterministe complet. Comme on l'a vu précédemment, tr_3 dépend alors de tr_2 et non de tr_4, car les conditions d'atteignabilité de l'état source de tr_3 dépendent de tr_2 et sont indépendantes de tr_4 – on note que si l'on supprime la transition tr_4, l'état source de tr_3 est atteignable exactement sous les mêmes conditions sur les variables de l'automate. Par contre, si $[g_2]$ est $[x < 5]$, le branchement devient non déterministe. Dans ce cas, si l'on restreint le domaine de x à $]-\infty, 0]$ ou à $[5, +\infty[$, on est ramené au cas d'un branchement déterministe, et dans ce cas, tr_3 est contrôle-dépendant de tr_2 et non de tr_4. Si l'on restreint le domaine de x à $]0, 5[$, on est ramené au cas d'un branchement non déterministe avec gardes triviales, et comme on l'a vu précédemment, dans ce cas tr_3 n'est contrôle-dépendant ni de tr_2 ni de tr_4. Pour décider du résultat dans ce cas, on remarque que la présence de tr_4 fait que l'état source de tr_3 peut ne pas être atteint lorsque la garde de tr_2 est vraie ; si l'on supprime tr_4, cela ne devient plus possible, ce qui signifie que tr_4 a une influence sur les conditions d'exécution de tr_3. Une définition pour les dépendances de contrôle dans le cas non-déterministe avec gardes non triviales doit donc conclure que tr_3 est contrôle-dépendant de tr_2 et de tr_4.

En s'appuyant sur l'exemple précédent, pour prendre en compte les branchements non-déterministes avec gardes non triviales, on suggère de modifier légèrement la définition 4.20 de la manière suivante : si les propriétés énumérées dans cette définition sont vérifiées, alors tr_j est contrôle-dépendante, non seulement de tr_i, mais aussi des transitions qui ont le même état source que tr_i.

4.3.4 Description de l'algorithme

L'algorithme qui suit (algorithme 4.4) est à l'origine inspiré du travail de Ranganath *et al.* sur l'analyse symbolique des chemins maximaux dans un graphe de flot de contrôle [RAB+05]. Cet algorithme permet de calculer les informations appropriées

concernant les chemins maximaux dans un IOSTS, pour en déduire les dépendances de contrôle dans un IOSTS, conformément à la définition 4.20.

L'algorithme 4.4, pour calculer les dépendances de contrôle, est basé sur une analyse symbolique du flot de contrôle. Comme nous le verrons par la suite, cette analyse de flot de contrôle présente des similarités avec une analyse de flot de données (cf. section 4.2, et chapitre 2) : les valeurs symboliques peuvent être ordonnées partiellement dans un treillis (cf. figure 4.5), et la recherche d'une solution correspond à la recherche d'un point fixe dans ce treillis. Cependant, les valeurs symboliques, au lieu d'être calculées en fonction des données (i.e. des variables), sont calculées en fonction de la structure de contrôle. En outre, les ensembles de valeurs symboliques, au lieu d'être associés aux transitions d'un IOSTS, sont associés à des couples de transitions, et comme nous le verrons, il y a plusieurs façons de propager les valeurs symboliques – elles ne sont notamment pas systématiquement propagées d'une transition à ses successeurs. Pour ces raisons, on ne formulera pas le problème en se basant sur des fonctions de transfert.

Analyse symbolique du flot de contrôle

L'algorithme 4.4 effectue une analyse symbolique de flot de contrôle sur un automate $\mathscr{A}$, pour collecter de l'information sur le flot de contrôle, et la stocker dans des ensembles de valeurs symboliques. *Le but de l'algorithme 4.4 est de chercher itérativement un point fixe pour les ensembles de valeurs symboliques, puis, une fois le point fixe atteint, d'utiliser cette information pour calculer les dépendances de contrôle dans* $\mathscr{A}$.

Les valeurs symboliques sont appelées p_{tr_i,tr_j}, et dénotent tous les chemins maximaux depuis tr_i dans $\mathscr{A}$, dans lesquels tr_i est immédiatement suivi de tr_j. Dans l'algorithme 4.4, la variable *conds* représente l'ensemble $conds_{\mathscr{A}}$, qui dénote l'ensemble des transitions de $\mathscr{A}$ qui ont une garde non-triviale (i.e. non trivialement évaluée à *vrai* pour tout paramétrage). Dans la suite de cette section, les éléments de *conds* seront appelés transitions conditionnelles. Un ensemble P_{s,tr_c} est associé à chaque couple (s, tr_c) dans $S \times conds_{\mathscr{A}}$. Plus précisément, P_{s,tr_c} dénote l'ensemble des chemins maximaux depuis la transition tr_c dans $\mathscr{A}$ qui contiennent l'état s.

Pour compléter l'analogie avec les analyses de flot de données, ébauchée dans l'introduction de la section 4.3.4, le treillis associé à un couple (s, tr_c) est formé par toutes les valeurs que peut prendre l'ensemble P_{s,tr_c}, de l'ensemble $\emptyset$ à l'ensemble de toutes les valeurs symboliques ; ce treillis est partiellement ordonné par la relation $\subseteq$ (cf. figure 4.5). Par contre, pour réaliser l'analyse de flot de contrôle, on ne définit pas

Algorithme 4.4 : Calcul des dépendances de contrôle.

Entrées : $\mathscr{A} = (S, s_0, T)_\Sigma$: un IOSTS
Données :
- *conds* : un ensemble de transitions
- $P[|S|, |T|]$: une matrice d'ensembles de valeurs symboliques, telle que $P[s, tr]$ représente $P_{s,tr}$
- *ensTravail* : un ensemble de transitions

Sorties : $CD[|T|]$: un tableau d'ensembles de transitions
// À la fin de l'algorithme 4.4, pour chaque $tr \in T$,
// $CD[tr]$ contient toutes les transitions dont tr est contrôle-dépendante.

/ (1) Initialisation */*
1 $conds \leftarrow \emptyset$
2 **pour chaque** $tr = (s, a, f, \sigma, s') \in T$ **faire**
3 $CD[tr] \leftarrow \emptyset$
4 **si** $f \neq vrai$ **alors**
5 $conds \leftarrow conds \cup \{tr\}$
6 **pour chaque** $s \in S$ **faire**
7 **pour chaque** $tr_c \in conds$ **faire**
8 $P[s, tr_c] \leftarrow \emptyset$
9 **pour chaque** $tr_c \in conds$ **faire**
10 **pour chaque** $tr_s \in succs_{tr_c}$ **faire**
11 $P[source_{tr_c}, tr_c] \leftarrow P[source_{tr_c}, tr_c] \cup \{p_{tr_c,tr_s}\}$
12 $P[cible_{tr_c}, tr_c] \leftarrow P[cible_{tr_c}, tr_c] \cup \{p_{tr_c,tr_s}\}$
13 $ensTravail \leftarrow conds$

/ (2) Analyse symbolique de flot de contrôle */*
14 **tant que** $ensTravail \neq \emptyset$ **faire**
15 $tr_l \leftarrow$ `élément`$(ensTravail)$

/ (2.1) Cas où tr_l est à la jonction de plusieurs chemins */*
16 **si** $|\{tr \in T \mid cible_{tr} = cible_{tr_l}\}| > 1$ **alors**
17 **pour chaque** $s \in S$ *tel que* $\exists tr \in conds, source_{tr} = s$ **faire**
18 **si** $\forall tr \in \{tr \in conds \mid source_{tr} = s\}, |P[cible_{tr_l}, tr]| = |succs_{tr}|$ **alors**

// Comme tous les chemins maximaux depuis l'état s
// contiennent $cible_{tr_l}$, tous les chemins maximaux
// qui contiennent s contiennent aussi $cible_{tr_l}$.
19 **pour chaque** $tr_c \in conds \backslash \{tr_l\}$ **faire**
20 **si** $P[s, tr_c] \backslash P[cible_{tr_l}, tr_c] \neq \emptyset$ **alors**
21 $P[cible_{tr_l}, tr_c] \leftarrow P[cible_{tr_l}, tr_c] \cup P[s, tr_c]$
22 $ensTravail \leftarrow ensTravail \cup succs_{tr_l}$

suite ⊟

Algorithme 4.4 (suite) : Calcul des dépendances de contrôle.

```
    /* (2.2) Cas où tr_l a un seul successeur                                  */
23  si |succs_{tr_l}| = 1 ∧ tr_l ∉ succs_{tr_l} alors
24      tr_s ← élément(succs_{tr_l})
        // Comme tr_s est l'unique successeur de tr_l, tous les chemins
        // maximaux qui contiennent cible_{tr_l} contiennent aussi cible_{tr_s}.
25      pour chaque tr_c ∈ conds faire
26          si P[cible_{tr_l}, tr_c]\P[cible_{tr_s}, tr_c] ≠ ∅ alors
27              P[cible_{tr_s}, tr_c] ← P[cible_{tr_s}, tr_c] ∪ P[cible_{tr_l}, tr_c]
28              ensTravail ← ensTravail ∪ {tr_s}
29  ensTravail ← ensTravail\{tr_l}
    // Fin de la boucle tant que, ligne 14.
    /* (3) Calcul des dépendances de contrôle                                  */
30  pour chaque s ∈ S faire
31      pour chaque tr_c ∈ conds faire
32          si |P[s, tr_c]| = |succs_{tr_c}|
            ∧ ∃tr'_c ∈ conds, source_{tr'_c} = source_{tr_c} ∧ |P[s, tr'_c]| < |succs_{tr'_c}| alors
                // Tous les chemins maximaux depuis tr_c contiennent s et il existe
                // un chemin maximal à partir de source_{tr_c} qui ne contient pas s.
                // Les transitions ayant s pour état source sont donc
                // contrôle-dépendantes de tr_c.
33              pour chaque tr ∈ T tel que source_{tr} = s faire
34                  CD[tr] ← CD[tr] ∪ {tr_c}
35  retourner CD
```

de fonction de transfert entre les couples de $S \times conds_{\mathscr{A}}$ comme on l'aurait fait dans le cadre d'une analyse de flot de données (cf. section 2.3.5 page 32) ; l'algorithme procède d'une autre manière, décrite ci-dessous.

Lors de la phase (1), l'ensemble des transitions conditionnelles est calculé, et utilisé pour initialiser l'ensemble de travail ; les éléments des ensembles P qui sont trivialement connus sont insérés. Lors de la phase (2), les valeurs symboliques sont propagées à travers l'automate analysé, et insérées dans les ensembles P_{s_j,tr_i} lorsque nécessaire : typiquement, p_{tr_i,tr_k} est inséré dans P_{s_j,tr_i} si et seulement si tous les chemins maximaux depuis tr_i, dans lesquels tr_i est immédiatement suivi par tr_k, contiennent l'état s_j. Deux cas sont distingués lors du traitement d'une transition tr_l de l'ensemble de travail. En phase (2.1) est traité le cas où tr_l se trouve à la jonction de plusieurs chemins dans $\mathscr{A}$, l'idée étant que s'il existe un état s au niveau duquel il y a un branchement, à partir duquel tous les chemins maximaux contiennent l'état

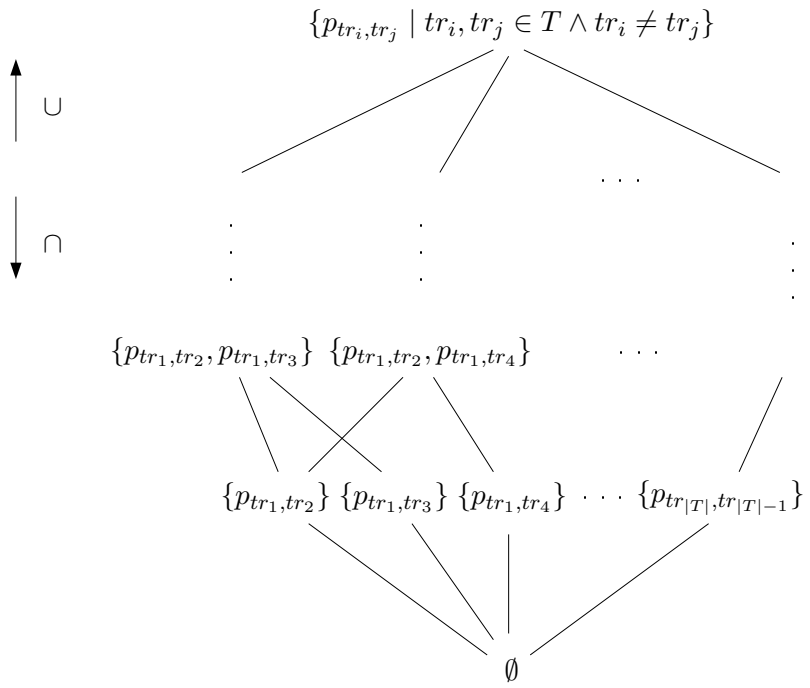


FIG. 4.5 – Treillis de base pour l'analyse symbolique de flot de contrôle dans un IOSTS.

cible de tr_l, alors tous les chemins maximaux depuis une transition conditionnelle tr_c quelconque, qui contiennent s, contiennent aussi $cible_{tr_l}$. La phase (2.2) a pour but de traiter le cas où tr_l possède un seul successeur tr_s dans $\mathscr{A}$, l'idée étant que si un chemin maximal depuis une transition tr_c contient l'état $cible_{tr_l}$, alors dans ce cas ce chemin contient aussi $cible_{tr_s}$.

Dépendances de contrôle

Lorsqu'un point fixe est trouvé pour tous les ensembles de valeurs symboliques P_{s,tr_c}, la phase (2) de l'algorithme termine et la relation de dépendance de contrôle peut être trouvée en analysant les ensembles P_{s,tr_c}, lors de la phase (3). Plus précisément, si la cardinalité de l'ensemble P_{s,tr_c} est strictement supérieure à zéro, alors cela signifie qu'il existe un successeur de tr_c depuis lequel tous les chemins maximaux contiennent s ; en outre, si la cardinalité de l'ensemble P_{s,tr_c} est strictement inférieure au nombre de successeurs de tr_c, alors cela nous indique qu'il existe un chemin maximal depuis un successeur de tr_c, qui ne contient pas s. Ces propriétés sont suffisantes pour déterminer la relation de dépendance de contrôle : étant donné un état s quelconque, pour chaque branchement dans l'automate à par-

tir d'un état s_b, s'il existe une transition conditionnelle tr'_c dont l'état source est s_b, et telle que $|P_{s,tr'_c}| < |succs_{tr'_c}|$, alors, en accord avec la définition 4.20, chaque transition dont l'état source est s est contrôle-dépendante de toutes les transitions tr_c dont l'état source est s_b, et telles que $|P_{s,tr_c}| = |succs_{tr_c}|$. Le résultat final de l'algorithme 4.4 est constitué par CD, un tableau d'ensembles de transitions indexé par l'ensemble des transitions de $\mathscr{A}$, qui à la fin de l'algorithme contient la relation de dépendance de contrôle ainsi calculée. Plus précisément, CD est tel que pour tout tr dans $\mathscr{A}$, $CD[tr]$ est l'ensemble des transitions de $\mathscr{A}$ dont tr est contrôle-dépendant.

Remarque (Branchements non déterministes)

L'algorithme 4.4 calcule les dépendances de contrôle conformément à la définition 4.20, ce qui implique, d'après la section 4.3.3, que le cas des branchements non déterministes avec gardes non triviales n'est pas pris en compte. Comme on l'a suggéré à la fin de ladite section, une modification légère de la définition permet de prendre ce cas en compte ; l'algorithme 4.4 pourra alors être modifié de manière analogue, en particulier à la ligne 34, en insérant dans $CD[tr]$ non plus seulement tr_c, mais toutes les transitions qui ont le même état source que tr_c.

4.3.5 Analyse de complexité et preuve de terminaison

Dans cette section est évaluée la complexité de l'algorithme 4.4 : on montre que cet algorithme termine, et a une complexité polynomiale.

Lemme 4.26 *La phase (1) de l'algorithme 4.4 termine, avec une complexité en* $O(|T|.|conds_{\mathscr{A}}| + \sum_{tr_c \in conds_{\mathscr{A}}} d_{tr_c})$.

Démonstration. Lors de la phase (1), la boucle à la ligne 2 parcourt l'ensemble des transitions (notamment pour calculer l'ensemble *conds*), en temps $O(|T|)$. La boucle à la ligne 6 initialise les ensembles $P[s, tr_c]$, pour chaque état s dans S, et chaque transition tr_c dans *conds*, donc en temps $O(|S|.|conds_{\mathscr{A}}|)$. Enfin, la boucle à la ligne 9 traite chaque successeur de chaque transition conditionnelle, ajoutant à chaque itération un élément dans chacun des ensembles $P[source_{tr_c}, tr_c]$ et $P[cible_{tr_c}, tr_c]$. Par conséquent, chaque itération de la boucle à la ligne 9 a un coût constant en temps, et cette boucle a un coût total en $O(\sum_{tr_c \in conds_{\mathscr{A}}} d_{tr_c})$. Le lemme est obtenu en considérant que $O(|S|) = O(|T|)$, par construction des IOSTS. ◇

Lemme 4.27 *La phase (2) de l'algorithme 4.4 termine, avec une complexité en* $O(\sum_{tr_c \in conds_{\mathscr{A}}} d_{tr_c}.|T|.|conds_{\mathscr{A}}|^2.lg(|T|))$.

Démonstration. Lors de la phase (2), la condition à vérifier pour que la boucle principale (à la ligne 14) termine, est que tous les ensembles $P[s, tr_c]$ se stabilisent : en effet, dans ce cas aucune des conditions aux lignes 20 et 26 ne pourra être vraie, et donc aucune transition ne pourra être ajoutée dans l'ensemble de travail, garantissant la terminaison. Par construction, chaque ensemble P_{s,tr_c} contient au maximum d_{tr_c} éléments (cf. section 4.3.4). À chaque itération de la boucle à la ligne 14, soit tous les ensembles P_{s,tr_c} restent inchangés, soit il y a au moins un de ces ensembles dont la taille est augmentée (à la ligne 21 ou 27). Dans ce dernier cas seulement, une transition peut être insérée dans l'ensemble de travail, ce qui contribuera à une itération supplémentaire de la boucle principale `tant que` (ligne 14). Par conséquent, étant donnés un état $s \in S$ et une transition $tr_c \in conds_{\mathscr{A}}$, l'ensemble P_{s,tr_c} se stabilise après au plus d_{tr_c} itérations. Partant, étant donné un état s dans S, tous les ensembles P_{s,tr_c} se stabilisent après au plus $O(\sum_{tr_c \in conds_{\mathscr{A}}} d_{tr_c})$ itérations de la boucle principale. Enfin, pour tous $s \in S$ et $tr_c \in conds_{\mathscr{A}}$, les ensembles P_{s,tr_c} se stabilisent après $O(|S|.\sum_{tr_c \in conds_{\mathscr{A}}} d_{tr_c})$ itérations. On en déduit que le nombre maximal d'itérations de la boucle principale est en $O(|T|.\sum_{tr_c \in conds_{\mathscr{A}}} d_{tr_c})$.

Lors de chaque itération de la boucle principale, la boucle à la ligne 25 traite au plus toutes les transitions dans $conds_{\mathscr{A}}$, et la boucle à la ligne 17 traite au plus toutes les transitions dans $conds_{\mathscr{A}}$, pour chaque état qui est source d'au moins une transition conditionnelle – le nombre de ces états est en $O(|conds_{\mathscr{A}}|)$. Par ailleurs, au cours de chaque itération de la boucle à la ligne 17, le coût du test à la ligne 18 est dominé par le traitement de la boucle à la ligne 19, et le test à la ligne 16 s'effectue en temps constant par l'emploi d'une structure de données adaptée, qui associe à chaque état l'ensemble des transitions qui ont cet état pour cible. De ce fait, le nombre d'itérations de la phase (2.1) est en $O(|conds_{\mathscr{A}}|^2)$, et le nombre d'itérations de la phase (2.2) est en $O(|conds_{\mathscr{A}}|)$. Par conséquent, la complexité globale de la phase (2) est en $O(\sum_{tr_c \in conds_{\mathscr{A}}} d_{tr_c}.|T|.|conds_{\mathscr{A}}|^2.lg(|T|))$. Le facteur $lg(|T|)$ dénote le coût des tests et opérations sur les ensembles $P[tr, tr_c]$ (aux lignes 20, 21, 26 et 27), chacun de ces ensembles contenant au plus $|T|$ éléments. ◇

Lemme 4.28 *La phase (3) de l'algorithme 4.4 termine, avec une complexité en* $O(|T|^2)$.

Démonstration. Lors de la phase (3), l'algorithme calcule la relation de dépendance de contrôle, en parcourant l'ensemble $conds_{\mathscr{A}}$ pour chaque état dans S (cf. section 4.3.4). On emploie des structures de données adaptées, qui associent à chaque état l'ensemble des transitions qui ont cet état pour source, et l'ensemble des tran-

sitions conditionnelles qui ont cet état pour source. Ces structures permettent, lorsqu'on exécute la boucle à la ligne 31, d'effectuer le test d'existence à la ligne 32 une seule fois pour chaque état source d'un branchement, en ne cherchant que parmi les transitions qui ont cet état pour source ; et de même, d'effectuer la boucle à la ligne 33 en ne parcourant que les transitions dont l'état source est s. Dans ces conditions, exécuter la boucle à la ligne 31 correspond à traiter au plus une fois chaque transition de $conds_{\mathscr{A}}$ pour le test à la ligne 32, et au plus une fois chaque transition de T pour la boucle à la ligne 33. On en déduit que la boucle à la ligne 31 a un coût en $O(|conds_{\mathscr{A}}| + |T|)$; cela revient à $O(|T|)$ car $conds_{\mathscr{A}} \subseteq T$ par construction. La boucle à la ligne 31 étant exécutée pour chaque s dans S lors de la phase (3), le coût total de la phase (3) est donc en $O(|S|.|T|)$, ce qui revient à $O(|T|^2)$, par construction des IOSTS. ◇

Théorème 4.29 (Complexité de l'algorithme 4.4) *L'algorithme 4.4 termine en temps* $O(\sum_{tr_c \in conds_{\mathscr{A}}} d_{tr_c}.|conds_{\mathscr{A}}|^2.|T|.lg(|T|))$.

Démonstration. Le résultat découle des lemmes 4.26, 4.27 et 4.28, en observant que la complexité des phases (1) et (3) est dominée par la complexité de la phase (2). □

4.3.6 Preuve de correction de l'algorithme

Cette section permet d'assurer que l'algorithme 4.4 est correct vis-à-vis de la définition 4.20.

Lemme 4.30 *Soient tr_i une transition conditionnelle, s_j un état, et tr_k une transition de l'automate $\mathscr{A}$. Si tous les chemins maximaux depuis tr_i dans $\mathscr{A}$, dans lesquels tr_i est immédiatement suivi de tr_k, contiennent s_j, alors on a $p_{tr_i,tr_k} \in P[s_j, tr_i]$ à la fin de la phase (2) de l'algorithme 4.4.*

Démonstration. Supposons que tous les chemins maximaux depuis tr_i dans $\mathscr{A}$, dans lesquels tr_i est immédiatement suivi de tr_k, contiennent s_j. On se propose de démontrer le lemme 4.30 par induction sur les chemins qui mènent de tr_i à s_j. On distingue trois cas disjoints, à partir desquels on peut construire tous les cas possibles. Dans ces trois cas, on montre qu'au cours du déroulement de l'algorithme 4.4, il y a une insertion de p_{tr_i,tr_k} dans $P[s_j, tr_i]$; le résultat découle ensuite du fait qu'aucun élément ne peut être retiré d'un ensemble P lors des phases (1) et (2).

- Cas 1 : s_j est soit $source_{tr_i}$, soit $cible_{tr_i}$. Comme tr_i est une transition conditionnelle, et tr_k un successeur de tr_i, lors de la phase (1) de l'algorithme 4.4, il y a

une itération de la boucle à la ligne 9 où tr_c est identifié à tr_i et tr_s à tr_k. Lors de cette itération p_{tr_i,tr_k} est donc inséré dans $P[s_j, tr_i]$.

- Cas 2 : il y a un seul chemin de tr_i à s_j qui, excepté tr_i, ne contient pas de transition conditionnelle. Cela implique que chaque transition sur ce chemin possède un unique successeur. Comme tr_i est conditionnelle, tr_i est inséré dans l'ensemble de travail lors de la phase (1), et à l'issue de cette phase, $P[cible_{tr_i}, tr_i]$ contient p_{tr_i,tr_k}. Comme l'algorithme 4.4 termine (cf. théorème 4.29), il y a une itération de la phase (2) où tr_i est extrait de l'ensemble de travail, i.e. tr_l est identifié à tr_i. Comme tr_i a un seul successeur, la phase (2.2) est effectuée, avec pour conséquence que p_{tr_i,tr_k} est inséré dans $P[cible_{tr_k}, tr_i]$, et tr_k est inséré dans l'ensemble de travail. Chaque transition sur le chemin considéré ayant un seul successeur, ce raisonnement s'applique récursivement à partir de tr_k pour aboutir au fait que p_{tr_i,tr_k} est inséré dans $P[s_j, tr_i]$.
- Cas 3 : plusieurs chemins depuis tr_i se rejoignent en s_j, et chacun de ces chemins contient une transition conditionnelle en plus de tr_i, i.e. il y a un embranchement formé des n transitions conditionnelles $tr_c^1, \ldots, tr_c^n$, et pour tout m, chaque chemin de tr_c^m à s_j ne contient pas d'autre transition conditionnelle que tr_c^m. En appliquant le raisonnement du cas 2, on sait qu'à la fin de la phase (2), $P[source_{tr_c^m}, tr_i]$ contient p_{tr_i,tr_k}, et que pour tout m, $P[s_j, tr_c^m]$ contient $p_{tr_c^m,tr_s^m}$, où tr_s^m est l'unique successeur de tr_c^m. Considérons l'itération I où le dernier des ensembles $P[s_j, tr_c^m]$ est modifié : comme on l'a vu dans le cas 2, cette modification a lieu lors de la phase (2.2). De plus, par construction lors de l'itération I, tr_l est identifié à une transition dont l'état cible est s_j. Or, par hypothèse plusieurs transitions ont s_j comme état cible, ce qui implique que lors de cette itération, le test à la ligne 16 est vérifié, et ainsi la phase (2.1) est aussi exécutée. Par hypothèse, les transitions tr_c^m vérifient la condition du test à la ligne 18. Par conséquent, lorsque s est identifié à $source_{tr_c^m}$, à la ligne 17, la branche `alors` est exécutée, et tr_c finit par être identifié à tr_i lors de l'itération I, à l'intérieur de la boucle à la ligne 25. Cela conduit à l'insertion de p_{tr_i,tr_k} dans $P[s_j, tr_i]$, à la ligne 21.

Le raisonnement du cas 3 s'applique récursivement dans les cas où les chemins de tr_c^m à s_j peuvent contenir des transitions conditionnelles. De même, dans le cas où les chemins depuis les transitions tr_c^m se rejoignent en un état $s \neq s_j$, on peut appliquer récursivement les raisonnements 2 et 3 sur les chemins de s à s_j, car les successeurs de la transition dont la cible est s sont insérés dans l'ensemble de travail lors du traitement de cette transition en phase (2.2) – cf. ligne 22. ◇

Lemme 4.31 *Soient tr_i une transition conditionnelle, s_j un état, et tr_k une tran-*

sition de l'automate $\mathscr{A}$. Si l'on a $p_{tr_i,tr_k} \in P[s_j, tr_i]$ à la fin de la phase (2) de l'algorithme 4.4, alors tous les chemins maximaux depuis tr_i dans $\mathscr{A}$, dans lesquels tr_i est immédiatement suivi de tr_k, contiennent s_j.

Démonstration. On se propose de démontrer le lemme 4.31 par récurrence sur le nombre n d'itérations de la phase (2) ; on notera $P^{(i)}$ l'ensemble P à la fin de la i-ème itération de la phase (2).

- Cas de base : $n = 0$. Lors de la phase (1), pour chaque transition conditionnelle tr_c et chaque couple (tr_c, tr_s) où tr_s est un successeur de tr_c, p_{tr_c,tr_s} est inséré dans $P[source_{tr_c}, tr_c]$ et $P[cible_{tr_c}, tr_c]$, aux lignes 11 et 12. Or, les chemins maximaux depuis tr_c, dans lesquels tr_c est immédiatement suivi de tr_s, contiennent bien $source_{tr_c}$ et $cible_{tr_c}$. Ces insertions dans les ensembles P étant les seules qui interviennent lors de la phase (1), le résultat est établi au début de la phase (2).
- Récurrence : $n > 0$. Supposons que pour tout $\lambda < n$, si l'on a $p_{tr_i,tr_k} \in P^{(\lambda)}[s_j, tr_i]$, alors tous les chemins maximaux depuis tr_i, dans lesquels tr_i est immédiatement suivi de tr_k, contiennent s_j. Par construction, lors d'une itération de la phase (2), un ensemble P peut être modifié seulement par l'une des lignes 21 et 27. Si la ligne 21 est exécutée lors de la n-ème itération, alors $P^{(n)}[cible_{tr_m}, tr_c]$ contient les éléments de $P^{(n-1)}[cible_{tr_m}, tr_c]$ et ceux de $P^{(n-1)}[cible_{tr_l}, tr_c]$. Or, dans ce cas, le test à la ligne 18 est nécessairement vérifié lors de la n-ème itération, ce qui implique $|P^{(n-1)}[cible_{tr_l}, tr]| = |succs_{tr}|$, pour toute transition tr dont l'état source est s. Par hypothèse de récurrence, cela signifie que tous les chemins maximaux depuis l'état s contiennent $cible_{tr_l}$. Par conséquent, tous les chemins maximaux dans $\mathscr{A}$ qui contiennent s contiennent aussi $cible_{tr_l}$. Le résultat est donc maintenu à l'itération n dans ce cas. Si la ligne 27 est exécutée lors de la n-ème itération, alors $P^{(n)}[cible_{tr_s}, tr_c]$ contient les éléments de $P^{(n-1)}[cible_{tr_s}, tr_c]$ et ceux de $P^{(n-1)}[cible_{tr_l}, tr_c]$. Or, dans ce cas, le test à la ligne 23 est nécessairement vérifié lors de la n-ème itération, ce qui implique que tr_s est le seul successeur de tr_l, et ainsi, que tous les chemins maximaux dans $\mathscr{A}$ qui contiennent $cible_{tr_l}$ contiennent aussi $cible_{tr_s}$. Par application de l'hypothèse de récurrence, le résultat est maintenu à l'itération n dans ce cas. On en conclut que dans tous les cas, le résultat est maintenu à l'itération n. ◇

Théorème 4.32 (Correction de l'algorithme 4.4) *Soient tr_i, tr_j deux transitions de l'automate $\mathscr{A}$. Si $tr_i \xrightarrow{cd} tr_j$, alors à la fin de l'algorithme 4.4, on a $tr_i \in CD[tr_j]$.*

Démonstration. On remarque que la boucle à la ligne 30 parcourt l'ensemble des

états ; on considère alors l'itération I de cette boucle, où s est identifié à $source_{tr_j}$. Supposons que $tr_i \xrightarrow{cd} tr_j$ est vérifié. Selon la définition 4.20, tr_i est une transition conditionnelle ($*^1$), $source_{tr_j}$ apparaît sur tous les chemins maximaux depuis tr_i dans $\mathscr{A}$ ($*^2$), et il existe une transition tr_k telle que $source_{tr_k} = source_{tr_i}$ et un chemin maximal depuis tr_k dans $\mathscr{A}$, qui ne contient pas $source_{tr_j}$ ($*^3$). Par hypothèse ($*^1$) on a $tr_i \in conds_{\mathscr{A}}$, et par conséquent tr_c finit par être identifié à tr_i lors de l'itération I, à l'intérieur de la boucle à la ligne 31. D'après le lemme 4.30, ($*^2$) implique qu'à la fin de la phase (2), on a $|P[source_{tr_j}, tr_i]| = |succs_{tr_i}|$; d'après le lemme 4.31, ($*^3$) implique qu'à la fin de la phase (2), on a $|P[source_{tr_j}, tr_k]| < |succs_{tr_k}|$. Par conséquent, lors de l'itération I, le test à la ligne 32 est vérifié, ce qui implique que la boucle à la ligne 33 est exécutée. Au cours de l'exécution de cette boucle, tr_i est inséré dans $CD[tr]$, pour toutes les transitions tr dans T, dont l'état source est $source_{tr_j}$. Comme aucun élément ne peut être retiré de l'ensemble $CD[tr_j]$ jusqu'à la terminaison, on a bien $tr_i \in CD[tr_j]$ à la fin de l'algorithme 4.4. □

4.3.7 Preuve de complétude de l'algorithme

Dans cette section, nous démontrons que l'algorithme 4.4 est complet vis-à-vis de la définition 4.20.

Théorème 4.33 (Complétude de l'algorithme 4.4) *Soit tr_i et tr_j deux transitions de l'automate $\mathscr{A}$. Si à la fin de l'algorithme 4.4, on a $tr_i \in CD[tr_j]$, alors $tr_i \xrightarrow{cd} tr_j$ est vérifié.*

Démonstration. Supposons qu'à la fin de l'algorithme 4.4, on a $tr_i \in CD[tr_j]$. Par construction de l'algorithme, cela implique qu'il y a une itération de la boucle à la ligne 33, au cours de laquelle la ligne 34 a été exécutée avec $tr = tr_j$, $s = source_{tr_j}$ et $tr_c = tr_i$. Pour que cette ligne puisse être exécutée, il faut que le test à la ligne 32 soit vérifié, ce qui signifie que $|P[source_{tr_j}, tr_i]| = |succs_{tr_i}| \wedge \exists tr'_c \in conds, source_{tr'_c} = source_{tr_i} \wedge |P[source_{tr_j}, tr'_c]| < |succs_{tr'_c}|$ est vérifié. Or, d'après le lemme 4.31, $|P[source_{tr_j}, tr_i]| = |succs_{tr_i}|$ implique que tous les chemins maximaux depuis tr_i contiennent $source_{tr_j}$, et d'après le lemme 4.30, $|P[source_{tr_j}, tr'_c]| < |succs_{tr'_c}|$ implique qu'il existe un chemin maximal depuis tr'_c qui ne contient pas $source_{tr_j}$. Étant donné que $source_{tr'_c} = source_{tr_i}$, selon la définition 4.20 on a bien $tr_i \xrightarrow{cd} tr_j$. □

4.4 Dépendances de communication

Un IOSTS communique avec son environnement au moyen d'actions de communication (cf. section 3.1.2). Comme nous l'avons vu en section 4.2, une variable peut être définie à travers une réception valuée, qui consiste à recevoir une valeur via un canal, puis l'affecter à une variable. Cette valeur dépend des variables utilisées dans l'(les) émission(s) valuée(s) correspondante(s). Ici, on remarque que les actions de communication induisent des dépendances de données inter-automates (i.e. qui se propagent à travers plusieurs automates). D'autre part, les actions de communication induisent aussi des dépendances de contrôle inter-automates, car une émission ne peut être exécutée que si une action de réception qui correspond est exécutée, et *vice-versa* (conformément à la sémantique des IOSTS, cf. définition 3.4 page 51). Ces dépendances ne sont pas prises en compte par les définitions et les algorithmes des sections 4.2 page 63 et 4.3 page 78 pour les dépendances de contrôle et de données, puisque ces définitions considèrent seulement les flots de contrôle et de données internes à un IOSTS. Dans ce qui suit, on verra que notre définition d'une *dépendance de communication* englobe les différentes dépendances induites par les actions de communication. Notre algorithme pour calculer les dépendances de communication dans les spécifications formées d'IOSTS sera présenté en section 4.4.3, et nous prouverons que cet algorithme termine en temps polynomial en section 4.4.4. Enfin, la correction et la complétude de notre algorithme sera démontrée en sections 4.4.5 et 4.4.6.

4.4.1 Définition

Intuitivement, il y a une dépendance de communication entre deux transitions dans deux IOSTS distincts s'il existe un canal qui permet potentiellement un transfert de flot de contrôle ou de flot de données entre ces deux transitions.

Définition 4.34 (Dépendance de communication ($tr_i \overset{com}{\longleftrightarrow} tr_j$)) *Soit $\mathscr{S}$ une spécification qui contienne au moins deux* IOSTS *; soient $(S_i, s_{0i}, T_i)_{\Sigma_i}$ et $(S_j, s_{0j}, T_j)_{\Sigma_j}$ deux* IOSTS *distincts dans $\mathscr{S}$, où $\Sigma_i = (\Omega_i, V_i, C_i)$, et $\Sigma_j = (\Omega_j, V_j, C_j)$. Deux transitions $tr_i = (s_i, a_i, f_i, \sigma_i, s'_i) \in T_i$ et $tr_j = (s_j, a_j, f_j, \sigma_j, s'_j) \in T_j$ sont* communication-dépendantes *l'une de l'autre, s'il existe un canal $c \in C_i \cap C_j$ tel qu'une des propriétés (1) ou (2) soit vérifiée :*

$$(1)\begin{cases} \bullet\ a_j = c?x \text{ pour un } x \text{ donné dans } V_j; \\ \bullet\ \text{et } a_i = c!t \text{ pour un } t \text{ donné dans } \mathscr{T}_\Omega(V_i). \end{cases} \qquad (2)\begin{cases} \bullet\ a_j = c? \\ \bullet\ \text{et } a_i = c! \end{cases}$$

La relation de dépendance de communication $\overset{com}{\longleftrightarrow}$ dénote toutes les dépendances induites par les communications. Dans les cas (1) et (2), il y a deux dépendances de contrôle inter-automates $tr_i \overset{com}{\longrightarrow} tr_j$ et $tr_j \overset{com}{\longrightarrow} tr_i$: par définition du mécanisme de communication par rendez-vous, tr_i et tr_j contrôlent mutuellement leurs exécutions.

Il y a de plus une dépendance de données inter-automates $tr_i \overset{com}{\longrightarrow} tr_j$ dans le cas (1), puisqu'une définition de variable au niveau de tr_j utilise des valeurs de variables de tr_i. Nous appelons dépendances de données inter-automates ce type de dépendance car il est en rapport avec les données, i.e. les valeurs des variables, mais on remarque que cette notion n'est pas analogue à la notion de dépendance de données définie en section 4.2. Nous verrons par la suite que les dépendances de données inter-automates $tr_i \overset{com}{\longrightarrow} tr_j$, combinées avec les informations de relations de dépendances de données dans les automates correspondants, seront utiles pour retrouver le lien entre tr_j et les transitions tr_d, qui sont telles que des variables utilisées dans l'action de communication de tr_i puissent faire référence à une définition par tr_d (en d'autres termes, telles que tr_i soit données-dépendant de tr_d, au sens de la définition 4.4). Ces dépendances indirectes seront traitées par notre algorithme de réduction paramétrée (cf. section 5.3 page 110), et pour cette raison il n'est pas nécessaire de faire la distinction, dans la définition 4.34, entre les dépendances de contrôle et les dépendances de données inter-automates qui composent la relation de dépendances de communications $\overset{com}{\longrightarrow}$ ainsi définie.

Exemple 4.35 Dans la figure 3.2 page 53, $j \rightharpoonup k$ et $e \rightharpoonup f$ sont mutuellement communication-dépendants : il existe un canal (à savoir, $normalCh$), sur lequel $e \rightharpoonup f$ effectue une action d'émission valuée et $j \rightharpoonup k$ effectue une réception valuée. De plus, la dépendance de données inter-automates $(e \rightharpoonup f) \overset{com}{\longrightarrow} (j \rightharpoonup k)$ induit une dépendance de données indirecte de $d \rightharpoonup e$ sur $j \rightharpoonup k$, puisque la définition de la variable am par $d \rightharpoonup e$ peut-être utilisée par $e \rightharpoonup f$ (i.e. $(d \rightharpoonup e) \overset{dd}{\longrightarrow} (e \rightharpoonup f)$), avant d'être transmise à $j \rightharpoonup k$ par communication.

4.4.2 Travaux connexes

La plupart des travaux ayant un rapport avec les dépendances de communications, telles que nous les avons définies en section 4.4.1, peuvent être trouvés dans le domaine de l'analyse statique de programmes à processus multiples.

Dans l'article [Sar97], Sarkar définit une méthode d'analyse de flot de données dans des graphes parallèles de programmes (appelés PPG, pour *Parallel Program Graph*), qui communiquent par évènements. Les analyses de flot de données sont rendues plus précises par la prise en compte des contraintes de synchronisation imposées par un mécanisme de communication basé sur les primitives `wait` et `post`. Dans les PPG, une instruction `wait` pour un évènement e doit attendre tous ses prédécesseurs de synchronisation (i.e. les instructions `post` pour e) pour terminer son exécution. Cela ne fonctionne pas dans le cadre des IOSTS, où une réception sur un canal c, pour terminer son exécution, attend seulement qu'une des émissions sur c soit exécutée (cf. définition 3.10 page 55).

Millett *et al.* définissent dans l'article [MT00] une méthode pour la réduction paramétrée de programmes exprimés dans le langage Promela (langage d'entrée du model checker SPIN). Dans [MT00], les canaux sont traités comme des variables, et ainsi les auteurs ne définissent pas de relation spécifique de dépendance de communication ; cependant, leur façon de traiter les variables partagées est similaire à notre façon de traiter les dépendances de communications. Le lecteur est invité à se référer à la section 5.6 pour une plus ample discussion à propos des travaux connexes.

4.4.3 Description de notre algorithme

L'algorithme 4.6 permet de calculer la relation de dépendances de communications dans une spécification $\mathscr{S}$ formée d'IOSTS, conformément à la définition 4.34.

Initialisation

La phase (1) de l'algorithme 4.6 calcule deux ensembles de transitions, qui sont stockés dans les tableaux em et rec, indexés par l'ensemble C des canaux dans $\mathscr{S}$. Le tableau em représente, pour chaque canal c, l'ensemble des transitions qui effectuent une émission sur c, et le tableau rec représente, pour chaque canal c, l'ensemble des transitions qui effectuent une réception sur c.

Dépendances de communication

La phase (2) traite chaque transition, marquant les transitions tr_e et tr_r comme mutuellement communication-dépendantes, dès lors que tr_r effectue une réception sur un canal et tr_e effectue une émission correspondante sur le même canal – plus précisément, le marquage est réalisé en insérant tr_e dans $ComD[tr_r]$ et tr_r dans

Algorithme 4.6 : Calcul des dépendances de communication.

Entrées : $\mathscr{S} = \{(S_i, s_{0i}, T_i)_{\Sigma_i} \mid 0 \leq i < k \ \wedge \ \Sigma_i = (\Omega_i, V_i, C_i)\}$: une spécification composée de k IOSTS (pour $k \in \mathbb{N}^*$ donné)
Données :
– $em[|\bigcup_{0\leq i<k} C_i|]$: un tableau d'ensembles de transitions
– $rec[|\bigcup_{0\leq i<k} C_i|]$: un tableau d'ensembles de transitions
– C : un ensemble de canaux
Sorties : $Com[|\bigcup_{0\leq i<k} T_i|]$: un tableau d'ensembles de transitions
// À la fin de l'algorithme 4.6, pour chaque $tr \in T$, $ComD[tr]$ contient
// toutes les transitions dont tr est communication-dépendante.

/ (1) Initialisation */*
$C \leftarrow \bigcup_{0\leq i<k} C_i$
$T \leftarrow \bigcup_{0\leq i<k} T_i$
pour chaque $c \in C$ **faire**
    $em[c] \leftarrow \emptyset$
    $rec[c] \leftarrow \emptyset$
pour chaque $tr = (s, a, f, \sigma, s') \in T$ **faire**
    $ComD[tr] \leftarrow \emptyset$
    Soit i tel que $tr \in T_i$
    si *a est de la forme $c!$ ou $c!x$, pour $c \in C_i$ et $x \in V_i$* **alors**
        $em[c] \leftarrow em[c] \cup tr$
    si *a est de la forme $c?$ ou $c?x$, pour $c \in C_i$ et $x \in V_i$* **alors**
        $rec[c] \leftarrow rec[c] \cup tr$

/ (2) Calcul des dépendances de communication */*
pour chaque $c \in C$ **faire**
    pour chaque $tr_e \in em[c]$ **faire**
        Soit i tel que $tr_e \in T_i$
        pour chaque $tr_r \in rec[c]$ **faire**
            Soit j tel que $tr_r \in T_j$
            si $i \neq j$ **alors**
                // $tr_e \xrightarrow{com} tr_r$ dénote les deux dépendances inter-automates
                // entre tr_e et tr_r : de données, et de contrôle.
                $ComD[tr_r] \leftarrow ComD[tr_r] \cup \{tr_e\}$
                // $tr_r \xrightarrow{com} tr_e$ dénote une dépendance de contrôle
                // inter-automates.
                $ComD[tr_e] \leftarrow ComD[tr_e] \cup \{tr_r\}$
retourner $ComD$

$ComD[tr_e]$, où $ComD[tr_r]$ (respectivement, $ComD[tr_e]$) dénote l'ensemble des transitions dont tr_r (respectivement, tr_e) est communication-dépendant. Conformément à la définition 4.34, l'algorithme 4.6 calcule les dépendances de communication de manière conservative, mais comme nous le verrons au chapitre 5, ce choix peut trouver une justification dans le cadre de la réduction paramétrée ; en contrepartie, l'algorithme est efficace (cf. section 4.4.4).

4.4.4 Analyse de complexité et preuve de terminaison

Cette section propose une démonstration de la terminaison en temps polynomial de l'algorithme 4.6.

Lemme 4.36 *La phase (1) de l'algorithme 4.6 termine, avec une complexité en* $O(|C| + |T|)$.

Démonstration. Les boucles aux lignes 3 et 6 sont complétées en parcourant respectivement l'ensemble des canaux C et l'ensemble des transitions T dans la spécification $\mathscr{S}$, et chaque itération de ces boucles a un coût en temps constant ; la phase (1) a donc une complexité en $O(|C| + |T|)$. ◇

Lemme 4.37 *La phase (2) de l'algorithme 4.6 termine, avec une complexité en* $O(|C|.|T|)$.

Démonstration. Le résultat de terminaison de la phase (2) est immédiat, par le fait que les ensembles de canaux et de transitions sont finis. Par la définition 3.2, les transitions des IOSTS portent au plus une action de communication. Par conséquent, lorsque la boucle `pour chaque` à la ligne 6 est terminée, une transition ne peut être à la fois dans em et dans rec, et le nombre total d'éléments dans "$em \cup rec$" est au plus le nombre total de transitions dans T. La boucle à la ligne 14 a donc une complexité en $O(|T|)$. La phase (2) consiste à exécuter la boucle à la ligne 14 pour chaque élément de C, donc la phase (2) termine en $O(|C|.|T|)$. ◇

Théorème 4.38 (Complexité de l'algorithme 4.6) *L'algorithme 4.6 termine en temps* $O(|C|.|T|)$.

Démonstration. Le résultat découle des lemmes 4.36, et 4.37, en observant que la complexité de la phase (1) est dominée par la complexité de la phase (2). □

4.4.5 Preuve de correction de l'algorithme

Cette section permet d'assurer que l'algorithme 4.6 est correct vis-à-vis de la définition 4.34.

Théorème 4.39 (Correction de l'algorithme 4.6) *Soient tr_i et tr_j deux transitions de la spécification $\mathscr{S}$. Si $tr_i \overset{com}{\longleftrightarrow} tr_j$, alors à la fin de l'algorithme 4.6, on a $tr_i \in ComD[tr_j]$ et $tr_j \in ComD[tr_i]$.*

Démonstration. Lors de la phase (1), la boucle à la ligne 6 traite chaque transition de la spécification, et donc en particulier tr_i et tr_j. Supposons que l'on a $tr_i \overset{com}{\longrightarrow} tr_j$. Selon la définition 4.34, il y a deux cas de figure : soit l'action de tr_j est de la forme $c?x$ et l'action de tr_i est de la forme $c!t$, soit l'action de tr_j est de la forme $c?$ et l'action de tr_i est de la forme $c!$ (pour $c \in C$ donné). Par hypothèse, lors de l'itération où tr_i est traité, le test à la ligne 11 est donc vérifié, et lors de l'itération où tr_j est traité, le test à la ligne 9 est aussi vérifié. Cela implique qu'à la fin de la phase (1), on a $tr_j \in em[c]$ et $tr_i \in rec[c]$. En phase (2), la boucle à la ligne 13 traite exactement tous les couples (tr_e, tr_r), où $tr_e \in em[c]$ et $tr_r \in rec[c]$, et ce pour chaque canal c. Par conséquent, il existe une itération I où tr_e est identifié à tr_j, et tr_r est identifié à tr_i ; or, par hypothèse on a $tr_i \overset{com}{\longrightarrow} tr_j$, ce qui implique que les automates dont tr_i et tr_j sont extraits sont distincts (cf. définition 4.34). Lors de l'itération I, le test à la ligne 18 est donc vérifié, ce qui entraîne l'exécution des lignes 19 et 20. Aucun élément ne pouvant être retiré de l'ensemble $ComD$ par la suite, on a $tr_i \in ComD[tr_j]$ et $tr_j \in ComD[tr_i]$ à la fin de l'algorithme 4.6. □

4.4.6 Preuve de complétude de l'algorithme

Dans cette section, nous démontrons que l'algorithme 4.6 est complet vis-à-vis de la définition 4.34.

Théorème 4.40 (Complétude de l'algorithme 4.6) *Soient tr_i et tr_j deux transitions de la spécification $\mathscr{S}$. Si à la fin de l'algorithme 4.6, on a $tr_i \in ComD[tr_j]$ et $tr_j \in ComD[tr_i]$, alors $tr_i \overset{com}{\longleftrightarrow} tr_j$ est vérifié.*

Démonstration. Supposons qu'à la fin de l'algorithme 4.6, on a $tr_i \in ComD[tr_j]$ et $tr_j \in ComD[tr_i]$. On remarque que pour qu'une transition soit insérée dans un des ensembles $ComD$, il faut qu'une des lignes 19 et 20 soit exécutée, et par construction, l'exécution d'une de ces lignes implique l'exécution de l'autre. Par hypothèse, les lignes 19 et 20 sont donc exécutées avant la terminaison de l'algorithme 4.6, avec

tr_e identifié à tr_j et tr_r identifié à tr_i (ou symétriquement, tr_e identifié à tr_i et tr_r identifié à tr_j). Le fait que ces lignes soient exécutées implique que le test à la ligne 18 est vérifié, i.e. les automates dont tr_i et tr_j sont extraits sont distincts. De plus, pour exécuter ces lignes, il faut entrer dans les boucles aux lignes 14 et 16, ce qui implique qu'il existe un canal c tel que $tr_j \in em[c]$ et $tr_i \in rec[c]$ (symétriquement, $tr_i \in em[c]$ et $tr_j \in rec[c]$). Pour qu'une transition soit dans l'ensemble $em[c]$, il faut nécessairement que la ligne 10 ait été exécutée en phase (1), ce qui implique que le test à la ligne 9 est vérifié. On en déduit que l'action de tr_j est de la forme $c!$ ou $c!t$. En considérant la ligne 12 et le test à la ligne 11, on trouve de manière analogue que l'action de tr_i est de la forme $c?$ ou $c?x$. Symétriquement, l'action de tr_i est de la forme $c!$ ou $c!t$, et l'action de tr_j est de la forme $c?$ ou $c?x$. En supposant que le typage des canaux dans $\mathscr{S}$ est cohérent, on ne prend pas en compte les cas où l'on a des réceptions/émissions valuées et des réceptions/émissions de signaux sur un même nom de canal ; tous les cas restants valident la définition 4.34, donc on a $tr_i \overset{com}{\longleftrightarrow} tr_j$. □

Chapitre 5

Réduction paramétrée de spécifications formées d'IOSTS

Ce chapitre expose notre méthode de réduction paramétrée de spécifications formées d'automates communicants, en se basant sur les analyses de dépendances et le cadre théorique décrits aux chapitres précédents.

Sommaire

5.1 Introduction

Comme nous l'avons vu en section 1.4 page 17, le concept originel de réduction paramétrée de programmes (réduction paramétrée *à la* Weiser) peut être synthétisé comme la recherche d'un sous-ensemble du programme (appelé *tranche*) par rapport à un autre sous-ensemble du programme (appelé *critère*). Nous souhaitons établir un algorithme pour la réduction paramétrée de spécifications, qui soit conforme à ce concept initial. C'est pourquoi les critères de réduction paramétrée utilisés par notre algorithme, et les tranches qu'il produit, sont tous deux des "sous-ensembles" de la spécification : les critères sont des ensembles de transitions, et les tranches sont des spécifications, extraites de la spécification originale. Notre approche pour la réduction paramétrée de spécifications formelles à base d'automates communicants est inspirée de l'état de l'art concernant la réduction paramétrée de programmes basé sur des relations de dépendances (cf. section 1.4, et chapitre 4). L'approche en question consiste à caractériser l'ensemble des transitions qui ont une influence potentielle sur un critère donné, au sens des relations de dépendances définies au chapitre 4. Ces nouvelles relations de dépendances peuvent être considérées comme des extensions des travaux de la littérature sur les relations de dépendances dans les programmes ; elles vont permettre la construction d'un graphe de dépendances, à partir duquel des tranches peuvent être extraites de façon efficace.

Ce chapitre est organisé comme suit. La section 5.2 décrit notre conception de l'évaluation de la précision d'une approche de réduction paramétrée basée sur des relations de dépendance. Comme nous le verrons par la suite, lorsque les trois relations de dépendance sont connues, la construction d'un graphe de dépendances est aisée La section 5.3 présente une description détaillée de notre méthode pour la

réduction paramétrée de spécifications par rapport à un critère. On verra aussi dans cette section que la construction d'un graphe de dépendances est aisée lorsque les trois relations de dépendance sont connues, ce qui permet de décrire notre méthode pour la réduction paramétrée de spécifications par rapport à un critère comme la résolution d'un problème d'atteignabilité – paramétré par le critère – dans le graphe de dépendances. La section 5.4 présente la mise en œuvre de ces travaux, que nous avons réalisé sous la forme d'un outil logiciel nommé CARVER. La section 5.5 est constituée de nos réflexions préliminaires concernant l'application de ces travaux à différents domaines, comme la preuve de propriétés et la gestion des évolutions. Enfin, en section 5.6 notre approche sera brièvement comparée aux travaux les plus proches que nous avons trouvé dans la littérature.

5.2 De la réduction paramétrée correcte, précise et optimale

Cette section introduit les notions qui permettent d'évaluer la précision d'une approche de réduction paramétrée basée sur des relations de dépendances, et notamment la notion de dépendance transitive.

5.2.1 Dépendance indirecte, dépendance transitive

On dit qu'une transition tr_j est *directement dépendante* d'une transition tr_i – on note $tr_i \xrightarrow{d} tr_j$ – si tr_j est soit contrôle-dépendant, soit données-dépendant, soit communication-dépendant de tr_i (i.e. la relation $\xrightarrow{d}$ est l'union des relations $\xrightarrow{cd}$, $\xrightarrow{dd}$, et $\xrightarrow{com}$).

Intuitivement, tr_j est *indirectement dépendant* de tr_i s'il existe une séquence de dépendances dans $\xrightarrow{d}$, qui mène de tr_i à tr_j. Toujours intuitivement, tr_j est *transitivement dépendant* de tr_i si tr_j est indirectement dépendant de tr_i, avec la condition supplémentaire qu'il existe un chemin, dans la spécification, qui corresponde à la séquence de dépendances qui mène de tr_i à tr_j. On pourrait estimer que le nom de *dépendance précise* serait mieux approprié pour ce concept. Cependant, notre définition peut être entendue comme une extension aux spécifications formées d'IOSTS des travaux de Krinke sur les dépendances transitives dans les programmes concurrents [Kri98].

Définition 5.1 (Dépendance transitive, indirecte) *Une transition tr_j est* transitivement dépendante *d'une transition tr_i dans une spécification $\mathscr{S}$, s'il existe une*

séquence $[tr_1, \ldots, tr_k]$ *où* $tr_1 = tr_i$ *et* $tr_k = tr_j$*, tel que pour tout* $1 \leq m < k$*, une des propriétés suivantes soit vérifiée :*

1. *soit* $tr_m \xrightarrow{com} tr_{m+1}$ *et il existe un rendez-vous entre* tr_m *et* tr_{m+1} *dans la composition parallèle de* $\mathscr{S}$ *;*
2. *soit* $tr_m \xrightarrow{cd,dd} tr_{m+1}$ *et il existe un chemin* $\langle tr_m, \ldots, tr_{m+1} \rangle$ *dans la composition parallèle de* $\mathscr{S}$*.*

D'autre part, si $tr_m \xrightarrow{d} tr_{m+1}$ *on dit que* tr_j *est* indirectement dépendant *de* tr_i*.*

5.2.2 Correction, optimalité, précision

La première contrainte qu'un algorithme de réduction paramétrée doit satisfaire, est d'être *correct*. Par là, on entend que les transitions qui ont effectivement une influence[1] sur le critère de réduction donné doivent être dans la tranche résultante. Par conséquent, une spécification est naturellement une réduction correcte d'elle-même, quel que soit le critère de réduction[2], et la présence dans la tranche résultante de transitions qui n'ont pas effectivement d'influence sur le critère ne remet pas en cause la correction de l'approche (par contre, elle en réduit la précision, comme on le verra par la suite).

Intuitivement, un algorithme de réduction paramétrée est *optimal* si, étant donné un critère de réduction, il produit une tranche qui contient seulement les transitions qui ont effectivement une influence sur le critère. Une tranche optimale est donc la plus petite tranche, en termes de nombre de transitions : il n'existe pas de solution meilleure, puisque retirer une transition rendrait la tranche incorrecte, et ajouter une transition en affaiblirait la précision.

Cependant, le résultat d'indécidabilité suivant montre que l'on ne peut pas écrire un algorithme de réduction paramétrée optimale – en fait, ce résultat concerne la plupart des analyses statiques : le problème de savoir si une condition peut être satisfaite parmi les exécutions possibles du système est indécidable en général (c'est une conséquence de [Göd31]). Weiser [Wei81] a montré, comme conséquence, que la réduction paramétrée optimale est indécidable. Même dans une arithmétique décidable (e.g. l'arithmétique de Presburger [Pre30]), l'indécidabilité du problème de terminaison empêche toujours l'optimalité des algorithmes de réduction paramétrée. Dans les analyses de flot de données (cf. chapitre 2), une façon courante de

[1] C'est-à-dire qu'il existe une exécution de la spécification, dans laquelle une dépendance est effective, au sens des définitions du chapitre 4.

[2] Cela correspond au plus conservatif des algorithmes de réduction paramétrée, analogue à la fonction identité.

contourner ce problème, est de ce placer dans une abstraction non-déterministe du système sous analyse, et définir les chemins réalisables et l'optimalité d'une solution dans cette abstraction, i.e. les branchements sont interprétés comme des choix non-déterministes. Notre définition d'une dépendance transitive (définition 5.1) sous-tend une telle abstraction, en ce sens que l'on parle de chemins réalisables indépendamment du fait que les gardes correspondantes soient satisfiables ou non. On a pu remarquer au chapitre 4 qu'en tant qu'analyses statiques, les définitions et algorithmes pour les relations de dépendance dans une spécification sous-tendent aussi l'abstraction non-déterministe.

On considère qu'une méthode pour la réduction paramétrée basée sur des relations de dépendances est précise *si elle prend en compte toutes les dépendances transitives dans la spécification, et seulement ces dépendances.* De façon informelle, la notion de méthode précise de réduction paramétrée introduit un niveau d'exigence intermédiaire, en terme de précision, entre les notions de méthode correcte et de méthode optimale.

5.2.3 Propriétés des relations de dépendances

À présent, on se propose d'étudier les relations de dépendances définies au chapitre 4, dans l'optique des notions définies aux sections précédentes 5.2.1 et 5.2.2 : on identifie trois propriétés importantes.

1. Par définition, les dépendances de données et les dépendances de contrôle sont toujours transitives. En effet, les définitions 4.4 et 4.20 impliquent que lorsqu'il y a une dépendance de contrôle ou une dépendance de données entre deux transitions, il existe un chemin entre ces transitions dans l'automate, et donc aussi un chemin dans la composition parallèle de la spécification. À partir de là, toute chaîne de dépendances de données ou de contrôle correspond à un chemin dans la composition parallèle de la spécification (cela nous mène au résultat) ;
2. Soient tr_i, tr_j et tr_k tels que $tr_i \xrightarrow{com} tr_j \xrightarrow{com} tr_k$. En se référant à la définition de la relation $\xrightarrow{com}$ (définition 4.34), et puisque les transitions d'IOSTS transition comportent une unique action de communication (cf. définition 3.2), si tr_i comporte une réception, alors tr_j comporte une émission, et tr_k une réception. Alors, il ne peut y avoir de dépendance de communication entre tr_i et tr_k (par définition). Réciproquement, si tr_i comporte une émission, alors tr_k comporte aussi une émission ; de nouveau, il ne peut y avoir de dépendance de communication entre tr_i et tr_k. Il en découle que selon la définition 5.1,

une séquence de dépendances de communication ne correspond pas à une dépendance transitive ;

3. Soit $\mathscr{A}_1$ et $\mathscr{A}_2$ deux IOSTS ; soient tr_{i_1}, tr_{j_1} deux transitions dans $\mathscr{A}_1$, et tr_{i_2}, tr_{j_2} deux transitions dans $\mathscr{A}_2$. Si $tr_{i_1} \xrightarrow{cd,dd} tr_{j_1}$, $tr_{j_1} \xrightarrow{com} tr_{i_2}$ et $tr_{i_2} \xrightarrow{cd,dd} tr_{j_2}$ sont vérifiés, alors en se basant sur la propriété 1, et en accord avec la sémantique d'entrelacements d'une spécification (cf. définition 3.10) : s'il existe une exécution dans laquelle le rendez-vous s'opère, alors il existe un chemin $\langle tr_{i_1}, \ldots, tr_c, \ldots, tr_{j_2} \rangle$ dans la composition parallèle de $\mathscr{A}_1$ et $\mathscr{A}_2$, où tr_c est la transition résultant de la composition parallèle de tr_{j_1} et tr_{i_2}. Par contre, si le rendez-vous ne peut s'opérer, alors la précision de la tranche résultante est amoindrie : des transitions peuvent être insérées dans la tranche en suivant des dépendances non-transitives.

Notation On note $\xrightarrow{d} *$ la fermeture transitive de $\xrightarrow{d}$, et $\xrightarrow{d} \star$ le sous-ensemble de $\xrightarrow{d} *$, tel que $tr_i \xrightarrow{d} \star\ tr_j$ est vérifié si et seulement si il existe une séquence de dépendances dans $\xrightarrow{d}$, menant de tr_i à tr_j, dans laquelle il n'y a jamais consécutivement deux éléments de $\xrightarrow{com}$. Plus formellement, $tr_i \xrightarrow{d} \star\ tr_j$ est vérifié si et seulement si il existe une séquence de dépendances dans $\xrightarrow{d}$, menant de tr_i à tr_j, qui constitue un mot accepté par l'automate représenté par la figure 5.1 (dans cette figure, les doubles cercles représentent les états acceptants de l'automate).

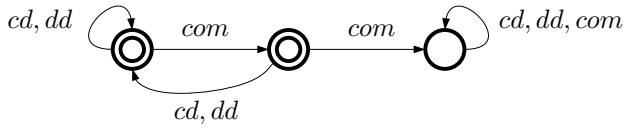


FIG. 5.1 – Automate reconnaissant les séquences de dépendances qui déterminent la relation $\xrightarrow{d} \star$.

5.2.4 Discussion sur les dépendances de communication

Dans le cadre d'une instance de réduction paramétrée de spécification, où une transition tr_n fait partie du critère, la propriété 2 indique que dans toute séquence $tr_1 \xrightarrow{com} \ldots \xrightarrow{com} tr_{n-1} \xrightarrow{com} tr_n$, on ne doit considérer que la dépendance $tr_{n-1} \xrightarrow{com} tr_n$, sous peine d'obtenir un résultat imprécis (au sens de la notion de précision définie en section 5.2.2).

Exemple 5.2 La figure 5.2 montre de façon abstraite une spécification formée de

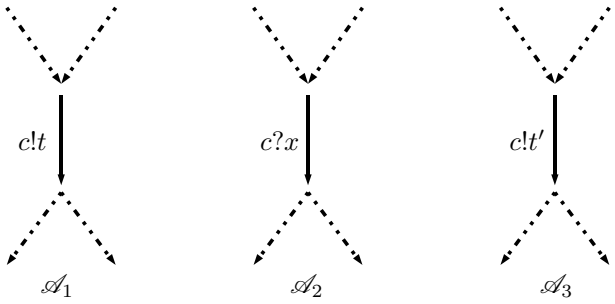


FIG. 5.2 – Les dépendances de communication ne sont pas transitives

trois IOSTS $\mathscr{A}_1$, $\mathscr{A}_2$ et $\mathscr{A}_3$, qui peuvent tous trois communiquer sur le canal c. Si l'on note tr_i la transition de $\mathscr{A}$ qui est affichée dans la figure 5.2, on a (entre autres) la séquence de dépendances $tr_1 \xrightarrow{com} tr_2 \xrightarrow{com} tr_3$. Dans le cadre d'une instance de réduction paramétrée, si tr_3 fait partie du critère de réduction, alors la propriété 2 nous dit qu'on ne doit considérer que la dépendance $tr_2 \xrightarrow{com} tr_3$. En effet, on voit bien sur cet exemple que tr_3 ne dépend pas de tr_1, et donc qu'ajouter tr_1 dans la tranche fait perdre de la précision.

La propriété 3 indique une potentielle altération de la précision : les dépendances dans $\xrightarrow{d}\star$ ne sont pas transitives dans le cas où aucun rendez-vous ne peut avoir lieu, dans l'abstraction non-déterministe de la spécification, entre deux transitions qui sont mis en relation par une dépendance de communication. Cependant, l'information des rendez-vous qui peuvent avoir lieu est contenue dans la composition parallèle de la spécification. Cette remarque peut suggérer deux alternatives à notre méthode de réduction paramétrée qui sera exposée en section 5.3.

La première alternative consisterait à effectuer la réduction paramétrée sur la composition parallèle de la spécification, en tenant compte seulement des dépendances de données et de contrôle. Cette solution n'est pas convenable pour la compréhension, le débogage, ou même la simulation de spécifications : pour ces applications, il est en effet préférable que les tranches soient sous la forme d'ensembles d'IOSTS concurrents, et de conserver les informations concernant les actions de communication. De plus, la composition parallèle étant de taille exponentielle dans le pire cas, il n'apparaît pas judicieux de se baser sur cette représentation pour une approche visant à la réduction de modèles.

La seconde alternative suggérée consisterait à utiliser des informations de la composition parallèle pour raffiner notre définition des dépendances de communication. Cette solution semble intéressante de prime abord, mais seule une étude

empirique dira si le gain en précision peut compenser le coût en complexité induit par la manipulation de cette structure (on peut effectivement envisager un sur-coût exponentiel).

5.2.5 Considérations sur la précision

Toutefois, nous garderons dans ce travail la définition 4.34 pour les dépendances de communication, et ce choix est justifié dans ce qui suit.

Premièrement, tous les rendez-vous possibles sont pris en compte par notre relation de dépendance de communication, ainsi la correction des tranches résultantes n'est pas compromise.

Deuxièmement, la définition 4.34 permet d'établir un algorithme simple et très efficace pour calculer les dépendances de communication[3].

Enfin, troisièmement, la propriété 3 montre aussi que, dans le cas où pour tous les couples de transitions qui sont mis en relation par une dépendance de communication, le rendez-vous correspondant peut être opéré dans l'abstraction non-déterministe de la spécification, alors les dépendances dans $\xrightarrow{d}\star$ sont transitives. Cela signifie que lorsque $tr_i \xrightarrow{d}\star tr_j$ est vérifié, tr_j est transitivement dépendant de tr_i. Selon notre expérience, il est raisonnable de faire cette hypothèse : dans les spécifications concrètes, la plupart des couples de transitions qui sont mutuellement communication-dépendants, au sens de la définition 4.34, ont été spécifiés pour pouvoir être exécutés en parallèle. Dans ce cas, un rendez-vous peut être opéré dans l'abstraction non-déterministe de la spécification. Sous cette hypothèse, les dépendances dans $\xrightarrow{d}\star$ sont donc transitives.

5.3 Méthode de réduction paramétrée

Cette section comporte une description détaillée de notre approche pour la réduction de spécifications formées d'automates communicants, a savoir des IOSTS. Dans un premier temps, la section 5.3.1 introduit les définitions fondamentales de graphe de dépendances et de tranche d'une spécification. Ensuite, notre algorithme pour calculer des tranches, conformément à ces définitions, est présenté en sections 5.3.2 et 5.3.3. La terminaison en temps polynomial de cet algorithme sera démontrée en section 5.3.4. Enfin, nous démontrons que l'algorithme se comporte exactement

[3] Les algorithmes de Krinke et Nanda pour calculer les dépendances transitives ont une complexité exponentielle au pire, bien que leurs résultats expérimentaux montrent qu'il est possible en pratique de faire tourner ces algorithmes en temps raisonnable [Kri03, Nan01].

de la manière décrite en section 5.3.3, au moyen d'une preuve de correction en section 5.3.5, et d'une preuve de complétude en section 5.3.6.

5.3.1 Définitions

Une fois que l'on a calculé les trois relations de dépendances $\xrightarrow{dd}$, $\xrightarrow{cd}$ et $\xrightarrow{com}$, respectivement définies aux sections 4.2, 4.3, et 4.4, on peut construire le graphe de dépendances de la spécification, en se basant sur la définition suivante.

Définition 5.3 (Graphe de dépendances) *Soit $\mathscr{S}$ une spécification, définie par $\mathscr{S} = \{(S_i, s_{0i}, T_i)_{\Sigma_i} \mid 0 \leq i < k\}$, étant donné $k \in \mathbb{N}^*$ quelconque. Un graphe $\mathcal{G}_{\mathscr{S}} = (N_{\mathscr{S}}, A_{\mathscr{S}})$ est un graphe de dépendances pour $\mathscr{S}$ si et seulement si :*

- *Il existe une bijection entre les ensembles $N_{\mathscr{S}}$ et $\bigcup_{0\leq i<k} T_i$;*
- *et pour chaque couple de transitions (tr_i, tr_j) tel que $tr_i \xrightarrow{d} tr_j$ soit vérifié, si nous appelons n_i et n_j les nœuds représentant respectivement tr_i et tr_j, alors il existe un arc de n_i à n_j dans $A_{\mathscr{S}}$.*

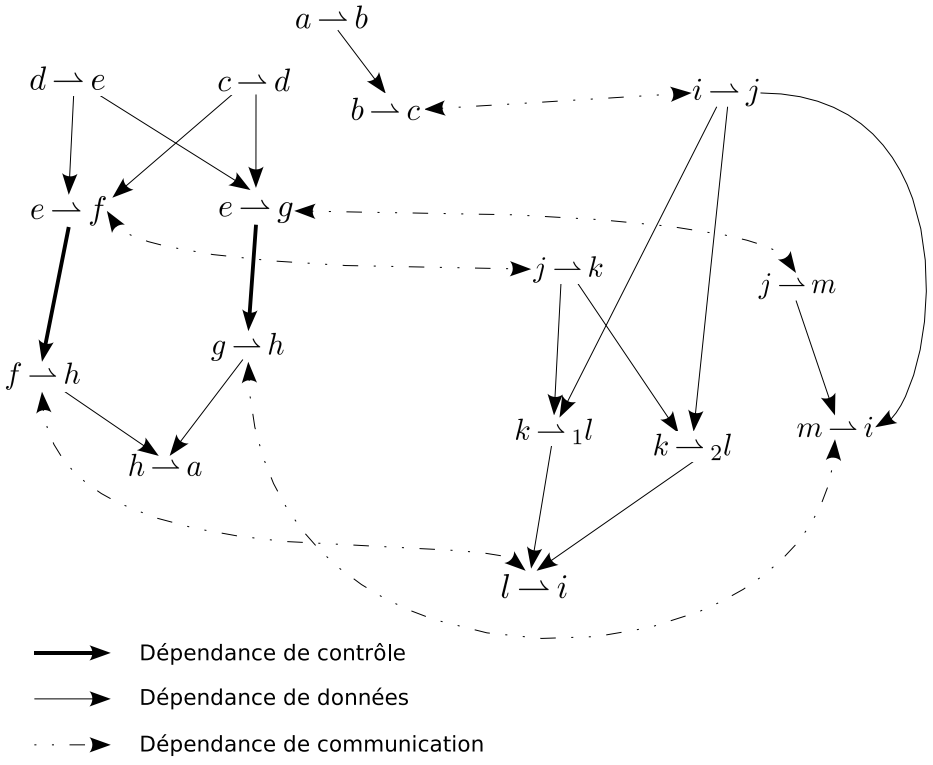


FIG. 5.3 – Graphe de dépendances de la spécification en figure 3.2

Exemple 5.4 La figure 5.3 montre le graphe de dépendances de la spécification représentée en figure 3.2 page 53. Les dépendances de données, de contrôle et de communication ont été calculées sur cette spécification respectivement à l'aide des algorithmes 4.1, 4.4 et 4.6 (pages 69, 86 et 98).

Si l'on se réfère à l'intuition principale de la réduction paramétrée (cf. section 1.4 page 17), à première vue on peut penser que l'information nécessaire et suffisante pour extraire des tranches de spécifications est la relation de dépendance de données. Cependant, comme nous l'avons vu au chapitre 4, les dépendances de contrôle et les dépendances de communication doivent aussi être prises en compte, si l'on souhaite que les tranches préservent les propriétés dynamiques de la spécification originale, et pour prendre en compte les flots de données et de contrôle induits par les communications inter-automates dans le système.

Une *tranche de spécification* (ou plus simplement, tranche) est calculée par rapport à une spécification et un *critère de réduction*. Une tranche est une nouvelle spécification, et le critère de réduction est un ensemble de transitions choisies dans la spécification originale. De façon informelle, une tranche d'une spécification $\mathscr{S}$ contient toutes les transitions dont le critère est directement ou indirectement dépendant, au sens des définitions du chapitre 4 et de la section 5.2. Une tranche de $\mathscr{S}$ par rapport à un critère de réduction $Crit$ est construite en parcourant le graphe de dépendances de $\mathscr{S}$ pour trouver les ensembles de transitions T'_i, depuis lesquelles les transitions de $Crit$ sont atteignables via $\stackrel{d}{\longrightarrow}\star$. En effet, l'atteignabilité via $\stackrel{d}{\longrightarrow}\star$ est liée à l'existence d'un chemin dans le graphe de dépendances. La tranche résultante $Tranche_{Crit}(\mathscr{S})$ est alors la spécification formée des IOSTS construits à partir des ensembles T'_i.

Définition 5.5 (Tranche de spécification) *Soit $\mathscr{S}$ une spécification, définie par $\mathscr{S} = \{(S_i, s_{0i}, T_i)_{\Sigma_i} \mid 0 \leq i < k \ \wedge \ \Sigma_i = (\Omega_i, V_i, C_i)\}$, étant donné $k \in \mathbb{N}^*$ quelconque. Soit $T = \bigcup_{0 \leq i < k} T_i$ l'ensemble global de transitions, et $Crit \subseteq T$ un critère de réduction.*

Pour chaque $0 \leq i < k$, soient $T'_i = \{tr \in T_i \mid \exists tr' \in Crit,\ tr \stackrel{d}{\longrightarrow}\star tr'\}$, et $S'_i = \bigcup_{tr \in T'_i} \{source_{tr}, cible_{tr}\}$.

Pour chaque $0 \leq i < k$, soient V'_i et C'_i respectivement l'ensemble des variables et l'ensemble des canaux qui apparaissent dans T'_i.

Alors, la spécification $Tranche_{Crit}(\mathscr{S})$ est une tranche de $\mathscr{S}$ par rapport à $Crit$:

$$Tranche_{Crit}(\mathscr{S}) = \{(S'_i, s_{0i}, T'_i)_{\Sigma'_i} \mid 0 \leq i < k \wedge T'_i \neq \emptyset \wedge \Sigma'_i = (\Omega_i, V'_i, C'_i)\}$$

Remarques Par définition, une tranche peut être de taille moins importante que la spécification originale (c'est même souhaitable en général), en termes de transitions, mais aussi en termes d'automates : si l'on pose $k' = |Slice_{Crit}(\mathscr{S})|$, alors on a en général $k' \leq k$. S'il existe dans la spécification un IOSTS qui ne contient aucune transition ayant une influence potentielle sur le critère, alors on a $k' < k$.

La définition 5.5 établit de façon formelle ce que l'on appelle une tranche *arrière* de spécification : les tranches résultantes doivent représenter ce qui, dans la spécification, influence potentiellement le critère. Au contraire, les tranches *avant* d'une spécification doivent représenter les parties de la spécification qui sont potentiellement influencées par le critère (cf. [Kri03], et section 1.4). Cela correspond à une légère modification de la définition 5.5 : pour que la tranche résultante soit une tranche avant, il suffit de remplacer dans la définition 5.5 la définition des ensembles T'_i par $T'_i = \{tr \in T_i \mid \exists tr' \in Crit,\ tr' \xrightarrow{d} \star\, tr\}$. Notons que dans ce cas, il faudra modifier en conséquence l'algorithme correspondant (i.e. l'algorithme 5.4).

Comme mentionné plus haut, l'atteignabilité via $\xrightarrow{d} \star$ est liée à l'existence d'un chemin dans le graphe de dépendances. Plus précisément, une tranche d'une spécification, par rapport à un critère, est l'ensemble des transitions telles qu'il existe un chemin dans le graphe de dépendances, depuis leur nœud correspondant jusqu'à un nœud qui correspond à une transition du critère, et qui ne contient pas plusieurs dépendances de communication consécutives.

5.3.2 Algorithme de réduction paramétrée : idée générale

Un algorithme efficace pour extraire des tranches de spécifications, en accord avec toutes les définitions précédentes, commence par la construction du graphe de dépendances (cf. definition 5.3). L'idée générale est d'extraire des tranches d'une spécification en identifiant les nœuds atteignables dans le graphe de dépendances, depuis les nœuds qui représentent les transitions du critère, en suivant les arcs dans le sens inverse. La tranche souhaitée est alors formée des IOSTS construits à partir des transitions qui correspondent aux nœuds ainsi identifiés.

5.3.3 Description de notre algorithme

En se basant sur cette idée générale, l'algorithme 5.4 extrait automatiquement une tranche $\mathscr{S}'$ d'une spécification $\mathscr{S}$, étant donné un critère de réduction $Crit$. Dans un souci d'efficacité, le graphe de dépendances n'est pas explicitement construit comme structure indépendante ; à la place, les relations de dépendances sont stockées sous la forme des tableaux CD, DD et $ComD$, indexés par les transitions de

Algorithme 5.4 : Réduction paramétrée de spécifications formées d'IOSTS.

Entrées :
– $\mathscr{S} = \{\mathscr{A}_i = (S_i, s_{0i}, T_i)_{\Sigma_i} \mid 0 \leq i < k \ \wedge \ \Sigma_i = (\Omega_i, V_i, C_i)\}$:
une spécification formée de k IOSTS (pour un $k \in \mathbb{N}$)
– $Crit$: un ensemble de transitions
Données :
– $CD[|\bigcup_{0\leq i<k} T_i|]$, $DD[|\bigcup_{0\leq i<k} T_i|]$, $ComD[|\bigcup_{0\leq i<k} T_i|]$: des tableaux d'ensembles de transitions
– $ensTravail$: un ensemble de transitions
– $fait[|\bigcup_{0\leq i<k} T_i|]$, $viaCom[|\bigcup_{0\leq i<k} T_i|]$: des tableaux de Booléens
Sorties : $\mathscr{S}' = \{(S'_i, s_{0i}, T'_i)_{\Sigma'_i} \mid 0 \leq i < k' \ \wedge \ \Sigma'_i = (\Omega_i, V'_i, C'_i)\}$:
une spécification formée de k' IOSTS, où $k' \leq k$
// À la fin de l'algorithme 5.4, $\mathscr{S}'$ est une tranche de $\mathscr{S}$ par rapport
// au critère $Crit$.

/ (1) Calcul des dépendances */*
$CD \leftarrow \emptyset$; $DD \leftarrow \emptyset$; $ComD \leftarrow \emptyset$

// Dépendances de données et de contrôle
pour chaque $i \in \{0 \dots k-1\}$ **faire**
  $CD \leftarrow CD \cup$ (CalculDépendancesContrôle($\mathscr{A}_i$))
  $DD \leftarrow DD \cup$ (CalculDépendancesDonnées($\mathscr{A}_i$))

// Dépendances de communication
si $k > 1$ **alors**
  $ComD \leftarrow$ (CalculDépendancesCommunication($\mathscr{S}$))

/ (2) Initialisation */*
$ensTravail \leftarrow \emptyset$
pour chaque $i \in \{0 \dots k-1\}$ **faire**
  $S'_i \leftarrow \emptyset$; $T'_i \leftarrow \emptyset$; $V'_i \leftarrow \emptyset$; $C'_i \leftarrow \emptyset$
  pour chaque $tr \in T_i$ **faire**
    $viaCom[tr] \leftarrow faux$
    $fait[tr] \leftarrow faux$
pour chaque $tr \in Crit$ **faire**
  Soit i tel que $tr \in T_i$
  $T'_i \leftarrow T'_i \cup \{tr\}$
  $ensTravail \leftarrow ensTravail \cup \{tr\}$
  $fait[tr] \leftarrow vrai$

suite ⊟

Algorithme 5.4 (suite) : Réduction paramétrée de spécifications formées d'IOSTS.

/ (3) Recherche des transitions qui induisent une dépendance sur le critère */*

tant que $ensTravail \neq \emptyset$ **faire**
  $tr_l \leftarrow$ `élément`$(ensTravail)$

  // On traite les transitions qui sont nouvellement atteintes.
  // Dans le cas d'une dépendance $tr \xrightarrow{com} tr_l$*, où* tr_l *est atteinte*
  // au préalable seulement via une dépendance de communication,
  // on ne traite pas tr *car la relation* $\xrightarrow{com}$ *n'est pas transitive.*
  pour chaque tr *tel que* $\neg(fait[tr]) \wedge ((tr \in CD[tr_l] \cup DD[tr_l]) \vee (tr \in ComD[tr_l] \wedge \neg viaCom[tr] \wedge \neg viaCom[tr_l]))$ **faire**

    // Ajout de tr dans la tranche et l'ensemble de travail
    Soit i tel que $tr \in T_i$
    si $\neg(viaCom[tr])$ **alors** $T'_i \leftarrow T'_i \cup \{tr\}$
    $ensTravail \leftarrow ensTravail \cup \{tr\}$

    // Mise à jour des Booléens
    si $tr \in ComD[tr_l]$ **alors**
      $viaCom[tr] \leftarrow vrai$
    sinon
      $viaCom[tr] \leftarrow faux$
      $fait[tr] \leftarrow vrai$
  $ensTravail \leftarrow ensTravail \backslash \{tr_l\}$

/ (4) Finaliser le calcul de la tranche par rapport au critère */*
pour chaque $i \in \{0 \dots k-1\}$ **faire**
  pour chaque $tr = (s, a, f, \sigma, s') \in T'_i$ **faire**
    $S'_i \leftarrow S'_i \cup \{source_{tr}, cible_{tr}\}$
    $V'_i \leftarrow V'_i \cup def[tr] \cup ref[tr]$
    si *a s'identifie avec* $c!$*,* $c!x$*,* $c?$*, ou* $c?x$ **alors** $C'_i \leftarrow C'_i \cup \{c\}$
$\mathscr{S}' \leftarrow \{(S'_i, s_{0i}, T'_i)_{\Sigma'_i} \mid T'_i \neq \emptyset\}$
$k' \leftarrow |\mathscr{S}'|$
retourner $\mathscr{S}'$

la spécification, et tels que pour chaque transition tr de la spécification, $CD[tr]$, $DD[tr]$ et $ComD[tr]$ dénotent l'ensemble des transitions dont tr est respectivement contrôle-, données- et communication-dépendant.

Calcul des dépendances

Lors de la phase (1), la boucle à la ligne 2 calcule les relations de dépendances de données et de contrôle pour chaque IOSTS dans $\mathscr{S}$, en faisant appel respectivement aux algorithmes 4.1 et 4.4 ; l'information résultante de dépendances de données et de contrôle est placée dans les tableaux CD et DD. Il y a ici un léger abus de langage : strictement parlant, les expressions aux lignes 3 et 4 ne devraient pas employer l'union d'ensembles, puisque CD et DD sont des tableaux. Dans notre implantation de l'algorithme 5.4, la ligne 3 est remplacée par les deux instructions suivantes : tout d'abord (CalculDépendancesContrôle($\mathscr{A}_i$)), puis une boucle où, pour chaque $tr \in T_i$, $CD[tr]$ est affecté de l'ensemble correspondant. La ligne 4 est remplaçée de manière analogue. Enfin, si la spécification contient au moins deux IOSTS, alors la relation de dépendances de communication est calculée en faisant appel à l'algorithme 4.6, puis stockée dans $ComD$.

Initialisation

En phase (2) de l'algorithme 5.4, la spécification réduite $\mathscr{S}'$ est initialisée avec les transitions contenues dans $Crit$, car le critère est utilisé comme base de la spécification réduite finale. L'information nécessaire concernant l'atteignabilité dans le graphe de dépendances depuis les transitions de $Crit$ sera trouvée notamment à l'aide d'un ensemble de travail. L'ensemble $ensTravail$ est donc initialisé avec l'ensemble des transitions contenues dans $Crit$. À chaque transition tr dans la spécification, on associe un Booléen $viaCom[tr]$, qui sera vrai si et seulement si la transition tr est atteinte depuis le critère seulement via une dépendance de communication, ainsi qu'un Booléen $fait[tr]$ qui sera vrai si et seulement si la transition tr ne devra plus être insérée dans l'ensemble de travail (concrètement, cela correspond au cas où tr est atteinte via une dépendance de contrôle ou de données).

Recherche des transitions de la tranche

Une fois que la phase (1) est terminée, $CD[tr_l] \cup DD[tr_l] \cup ComD[tr_l]$ dénote pour chaque tr_l l'ensemble des transitions tr, dont tr_l est dépendant, i.e. $\{tr \mid tr \xrightarrow{d} tr_l\}$. Cette propriété est utilisée en phase (3), pour déduire des ensembles CD, DD

et $ComD$ les informations recquises concernant l'atteignabilité dans le graphe de dépendances. Lors de chaque itération de la boucle à la ligne 18, une transition est extraite de l'ensemble de travail, et chacun de ses prédécesseurs dans le graphe de dépendances est inséré dans l'ensemble de travail pour analyse lors des itérations ultérieures, tel que les chemins contenant deux dépendances de communications consécutives ne soient pas pris en compte. Cela est permis par le test à la ligne 20, qui assure que l'algorithme considère la relation $\xrightarrow{d} \star$, et non $\xrightarrow{d} *$. Plus précisément, lorsqu'une transition tr est atteinte depuis le critère :

- via une dépendance de données ou de contrôle pour la première fois, le Booléen $fait[tr]$ est positionné à $vrai$, de manière à s'assurer qu'on ne traite pas à nouveau cette transition lors d'itérations ultérieures, le Booléen $viaCom[tr]$ est positionné à $faux$ afin qu'on prenne en compte les dépendances de communication à partir de tr, et si auparavant tr n'a pas été atteint via une dépendance de communication, on l'insère dans la tranche ;
- via une dépendance de communication pour la première fois, on insère tr dans la tranche si la transition par laquelle on a atteint tr n'a pas été atteinte seulement via une dépendance de communication, et le Booléen $viaCom[tr]$ est positionné à $vrai$ pour qu'on ne prenne pas en compte les dépendances de communications à partir de tr.

Ainsi, chaque transition tr atteignable dans le graphe de dépendances depuis le critère est insérée exactement une fois dans l'ensemble T'_i approprié. La manipulation des Booléens $fait[tr]$ et $viaCom[tr]$ assure que les transitions atteignables depuis seulement des dépendances de contrôle ou de données sont examinées exactement une fois, et que les dépendances atteignables depuis au moins une dépendance de communication sont examinées au moins une fois et au plus deux fois par l'algorithme.

Finalisation

Enfin, la construction de la tranche $\mathscr{S}'$ est achevée en phase (4) : la boucle à la ligne 30 détermine les données restantes qui composent $\mathscr{S}'$, en fonction des transitions qui ont été insérées dans les ensembles T'_i (on suppose que les informations des ensembles def et ref, calculées par l'algorithme 4.1, sont toujours disponibles). À la ligne 35, tous les automates non-vides sont insérés dans $\mathscr{S}'$, puis la construction de $\mathscr{S}'$ est complétée en déterminant k', la taille définitive de la tranche, en termes de nombre d'automates.

Remarque (Atteignabilité)

Les phases (1), (2) et (3) de l'algorithme 5.4 étant complétées, les ensembles de transitions T'_i qui composeront la tranche ont été construits avec succès. Alors, nous attirons l'attention sur le fait qu'il peut arriver que des transitions qui étaient atteignables dans la spécification originale, ne soient plus atteignables dans la tranche construite sur les ensembles de transitions T'_i. Selon l'application visée, cela peut constituer un problème (par exemple, si l'application visée est l'exécution symbolique, avec AGATHA [BFG+03a]). Une solution consisterait à effectuer une procédure de réduction supplémentaire pour chaque automate, lors de la phase (4). Basiquement, cette procédure de réduction consisterait à modifier de manière appropriée les états source et cible de transitions, et éventuellement l'état initial des IOSTS de la spécification (cf. exemple 5.6). Un algorithme pour réaliser cette fonctionnalité pourrait être inspiré de la τ-réduction des systèmes de transitions étiquetés[4], ou l'ϵ-réduction des automates finis non-déterministes avec ϵ-transitions ; mais nous pensons qu'un algorithme *ad-hoc* devra être établi pour chaque application de la méthode (cf. section 5.5).

Exemple 5.6 Reprenons l'exemple de la figure 3.2 page 53, et insérons une utilisation de la variable nm au niveau de la transition $c \rightharpoonup d$, i.e. $c \rightharpoonup d$ est remplacé par $c \rightharpoonup_1 d = (c, typeCh?tp, true, (nm \mapsto \text{“customer ”} + nm), d)$. Étant donné le critère de réduction $\{c \rightharpoonup_1 d\}$, l'algorithme 5.4 détermine que seules les transitions $a \rightharpoonup b$ et $c \rightharpoonup_1 d$ influencent potentiellement le critère. L'atteignabilité dans la tranche peut alors être rétablie soit en ajoutant une transition sans label de b vers c (solution inspirée de la τ-réduction), soit en créant un nouvel état, pour constituer à la fois l'état cible de $a \rightharpoonup b$ et l'état source de $c \rightharpoonup_1 d$ (solution inspirée de l'ϵ-réduction). Ce dernier cas peut alors nécessiter d'impacter les modifications au niveau de l'état initial de l'IOSTS considéré : par exemple, si un des états b ou c était l'état initial de l'IOSTS, alors le nouvel état ajouté devrait être le nouvel état initial de l'IOSTS.

Remarque (Réutilisabilité)

Le graphe de dépendance est une structure de données hautement réutilisable, et pour cette raison, une approche de réduction paramétrée basée sur des relations de dépendances est très efficace pour calculer de multiples tranches d'une même spécification. Effectivement, lorsque le graphe de dépendances a été construit pour une spécification donnée, chaque tranche de la spécification peut être extraite en

[4] Labelled Transition Systems (LTS).

temps quasiment linéaire (à savoir, en $O(|T|.lg(|T|))$, cf. section 5.3.4). Pour réaliser cela sur une spécification donnée, on peut effectuer la phase (1) exactement une fois, puis les phases (2), (3) et (4) consécutivement, pour chaque critère.

5.3.4 Analyse de complexité et preuve de terminaison

Dans cette section, nous allons analyser la complexité de l'algorithme 5.4, et démontrer dans le même temps sa terminaison.

Phases (1) et (2) de l'algorithme

Proposition 5.7 *La phase (1) de l'algorithme 5.4 termine, avec une complexité en* $O(\sum_{tr \in T} d_{tr}.|T|^3.lg(|T|))$.

Démonstration. Lors de la phase (1), la boucle à la ligne 2 calcule les relations de dépendance de contrôle et de données, pour chaque IOSTS $\mathscr{A}_i$ dans la spécification en entrée $\mathscr{S}$. D'après le théorème 4.29 la complexité de la ligne 3 est en $O(\sum_{tr_c \in conds_{\mathscr{A}_i}} d_{tr_c}.|conds(\mathscr{A})|^2.|T_i|.lg(|T_i|))$. En outre, la complexité de la ligne 4 est en $O(\sum_{tr \in T_i} d_{tr}.|T_i|.|\mathcal{D}_{\mathscr{A}_i}|.lg(|\mathcal{D}_{\mathscr{A}_i}|))$, par application du théorème 4.10. On rappelle que $conds_{\mathscr{A}_i}$ est l'ensemble des transitions conditionnelles de $\mathscr{A}_i$ (cf. section 4.3.4), et que $\mathcal{D}_{\mathscr{A}_i}$ est l'ensemble des définitions de variables dans l'automate $\mathscr{A}_i$ (cf. section 4.2.6). Par définition, on a donc $conds(\mathscr{A}_i) \subseteq T_i$, et de plus, il est raisonnable de dire qu'en pratique $\mathcal{D}_{\mathscr{A}_i}$ est proportionnel à T_i. Par conséquent, la complexité d'une itération de la boucle à la ligne 2, pour calculer les dépendances internes à $\mathscr{A}_i$, est dominée par $O(Expr_i)$, où $Expr_i = \sum_{tr \in T_i} d_{tr}.|T_i|^3.lg(|T_i|)$. La complexité totale de la boucle à la ligne 2 est dominée par la somme, pour tout i, de l'expression $Expr_i$. Or, $O(\sum_i |T_i|^3.lg(|T_i|))$ étant dominé par $O(|T|^3.lg(|T|))$, la complexité de la boucle à la ligne 2 est dominée par $O(\sum_{tr \in T} d_{tr}.|T|^3.lg(|T|))$.

D'après le théorème 4.38, la complexité de la ligne 6 est $O(|C|.|T|)$. Par définition des transitions d'un IOSTS, $|C| \leq |T|$ (cf. définition 3.2). La complexité de la ligne 6 est donc en $O(|T|^2)$, ce qui est dominé par la complexité de la boucle à la ligne 2, comme on l'a vu ci-dessus. En conclusion, la complexité de la phase (1) est dominée par $O(\sum_{tr \in T} d_{tr}.|T|^3.lg(|T|))$. △

Proposition 5.8 *La phase (2) de l'algorithme 5.4 termine, avec une complexité en* $O(|T|)$.

Démonstration. En phase (2), la boucle à la ligne 8 parcourt l'ensemble des transitions de la spécification, exactement une fois et la boucle à la ligne 13 parcourt

l'ensemble des transitions du critère. Le traitement effectué lors de chaque itération de ces boucles a un coût constant. Cette phase termine donc en $O(|T|)$. △

Phase (3) de l'algorithme

Démontrer le lemme 5.12 nous permettra de borner le nombre d'itérations de la boucle à la ligne 18, afin de démontrer la proposition 5.13. Pour ce faire, nous démontrons trois résultats préliminaires.

Lemme 5.9 *Toute transition $tr \in Crit$ est insérée exactement une fois dans l'ensemble de travail de l'algorithme 5.4, et cette insertion a lieu en phase (2).*

Démonstration. Pour qu'une transition tr soit insérée dans l'ensemble de travail, il faut exécuter la ligne 16 ou la ligne 23. La ligne 16 est contenue dans la boucle à la ligne 13. Par construction, cette boucle est exécutée exactement une fois pour chaque transition du critère, et seulement pour ces transitions.

Lors de la boucle à la ligne 13, $fait[tr]$ est positionné à $vrai$ (à la ligne 17) pour tout $tr \in Crit$. Après la phase (2), chaque transition du critère a donc été insérée exactement une fois dans l'ensemble de travail. La ligne 16 est exécutée seulement si la condition de la boucle à la ligne 20, i.e. $\neg(fait[tr]) \wedge ((tr \in CD[tr_l] \cup DD[tr_l]) \vee (tr \in ComD[tr_l] \wedge \neg viaCom[tr] \wedge \neg viaCom[tr_l]))$, est vérifiée. En phase (3), $fait[tr]$ est vrai pour les transitions du critère, et n'est jamais remis à $faux$. Ces transitions ne vérifient donc jamais la condition de la boucle à la ligne 20, et donc la ligne 23 n'est jamais exécutée pour celles-ci. On a donc montré que les transitions du critère sont insérées dans l'ensemble de travail exactement une fois au total. ◇

Lemme 5.10 *Toute transition tr de $\mathscr{S}$ qui n'est pas dans le critère, pour laquelle il existe une itération de la boucle à la ligne 20 où $fait[tr]$ est à $faux$, et tr_l est tel que $tr \in CD[tr_l] \cup DD[tr_l]$ est vérifié, est insérée exactement une fois dans l'ensemble de travail lors de la phase (3) de l'algorithme 5.4.*

Démonstration. On remarque qu'une transition qui ne fait pas partie du critère ne peut être insérée dans l'ensemble de travail avant la phase (3). Par hypothèse, la condition d'entrée dans la boucle 20 est vérifiée, et tr_l est tel que $tr \in CD[tr_l] \cup DD[tr_l]$. Dans ce cas, le nombre d'insertions de tr est incrémenté à la ligne 23 (c'est-à-dire, porté à 1), et la branche `sinon` à la ligne 26 est exécutée, ce qui conduit au positionnement de $fait[tr]$ à $vrai$. Comme une remise à $faux$ de $fait[tr]$ est impossible dans l'algorithme, la condition de la boucle à la ligne 20 ne sera jamais

vérifiée de nouveau pour tr, et par conséquent, tr ne sera plus inséré dans l'ensemble de travail à nouveau. ◇

Lemme 5.11 *Toute transition tr de $\mathscr{S}$ qui n'est pas dans le critère, pour laquelle il existe une itération de la boucle à la ligne 20 où la condition d'entrée est vérifiée, est insérée au moins une fois et au plus deux fois dans l'ensemble de travail lors de la phase (3) de l'algorithme 5.4.*

Démonstration. On considère la première itération I de la boucle à la ligne 20, telle que la condition d'entrée est vraie pour tr (cette itération existe par hypothèse). Il y a deux cas possibles :

- Il existe tr_l tel que $tr \in CD[tr_l] \cup DD[tr_l]$. Dans ce cas, le résultat découle du lemme 5.10.
- Il existe tr_l tel que $tr \in Com[tr_l]$ et $\neg viaCom[tr]$ et $\neg viaCom[tr_l]$. Par hypothèse, I est la première itération où ces conditions sont vérifiées, donc comme on l'a remarqué dans la démonstration du lemme 5.10, le nombre d'insertions de tr avant I est nul. Dans ce cas, le nombre d'insertions de tr est incrémenté à la ligne 23 lors de I, et la branche `alors` à la ligne 25 est exécutée, ce qui conduit au positionnement de $viaCom[tr]$ à $vrai$. Considérons alors la prochaine itération de la boucle à la ligne 20, telle que la condition d'entrée est vraie pour tr (il se peut que cette itération n'existe pas, auquel cas le résultat est établi). La seule possibilité pour que $viaCom[tr]$ ait été remis à $faux$ entre-temps, est que la ligne 27 ait été exécutée ; or cela n'est pas possible puisque cela nécessite qu'il y ait eu une itération de la boucle à la ligne 20. Comme $viaCom[tr]$ est $vrai$, on a nécessairement $tr \in CD[tr_l] \cup DD[tr_l]$. On est alors ramené au cas précédent, où le nombre d'insertions est incrémenté exactement une fois au total. ◇

Lemme 5.12 *Pour toute transition tr dans la spécification $\mathscr{S}$, tr est insérée au plus deux fois dans l'ensemble de travail de l'algorithme 5.4.*

Démonstration. Si $tr \in Crit$, le résultat est immédiat par le lemme 5.9. Maintenant, supposons que $tr \notin Crit$. Remarquons qu'une transition peut être insérée dans l'ensemble de travail seulement aux lignes 16 et 23. Au début de la phase (3), tr n'a été insérée aucune fois dans l'ensemble de travail, car la ligne 16 n'a pas été exécutée pour celle-ci, par construction. S'il n'existe aucune itération de la boucle à la ligne 20, telle que la condition d'entrée soit vérifiée pour tr, alors la ligne 16 ne pourra être exécutée pour tr, et tr ne sera par conséquent jamais inséré dans l'ensemble de travail. Dans le cas contraire, le résultat est immédiat par application

du lemme 5.11. En résumé, les transitions du critère sont insérées exactement une fois au total. Les autres transitions sont insérées au total zéro, une ou deux fois dans l'ensemble de travail. ◇

Proposition 5.13 *La phase (3) de l'algorithme 5.4 termine, avec une complexité en* $O(|T|)$.

Démonstration. En examinant l'algorithme 5.4, on voit que chaque itération de la boucle à la ligne 20 induit l'insertion d'un élément dans l'ensemble de travail (à la ligne 23). Le lemme 5.12 montre donc qu'au cours d'une exécution de l'algorithme 5.4, le nombre total d'itérations de la boucle à la ligne 20 est borné par $2|T|$. À chacune de ces itérations, le traitement est effectué en temps constant, et le test à la ligne 20, étant donné une transition dans $CD[tr_l]$, $DD[tr_l]$ ou $Com[tr_l]$, consiste à effectuer des tests simples sur des Booléens accédés en temps constant ; la complexité de la boucle à la ligne 18 est donc en $O(|T|)$. △

Phase (4) de l'algorithme

Proposition 5.14 *La phase (4) de l'algorithme 5.4 termine, avec une complexité en* $O(|T|.lg(|T|))$.

Démonstration. La boucle à la ligne 30 traite chaque transition de la spécification, exactement une fois ; le traitement d'une transition peut comporter des opérations sur des ensembles de taille maximale en $O(|T_i|)$. La complexité de la boucle à la ligne 30 est donc dominée par $O(\sum_i |T_i|.lg(|T_i|))$. Enfin, la complexité de la phase (4) est dominée par $O(|T|.lg(|T|))$. △

Complexité globale

Théorème 5.15 (Complexité de l'algorithme 5.4) *L'algorithme 5.4 termine en temps* $O(\sum_{tr \in T} d_{tr}.|T|^3.lg(|T|))$.

Démonstration. Le résultat découle des propositions 5.7, 5.8, 5.13 et 5.14, étant donné que la complexité de la phase (1) domine la complexité des phases (2), (3) et (4). □

On remarque que ce résultat de complexité a été obtenu par sur-approximations successives[5] pour simplifier l'expression et mettre en valeur son caractère polynomial. Si l'on se réfère à la remarque à propos de réutilisabilité, en section 5.3.3,

[5] Par exemple, l'algorithme le plus coûteux (l'algorithme 4.4), est appelé sur chaque $\mathscr{A}_i$, et non sur la spécification entière en une fois. Or, le coût réel de

lorsqu'une première tranche a été extraite, les informations de dépendances peuvent être réutilisées pour extraire chacune des tranches suivantes en temps $O(|T|.lg(|T|))$.

5.3.5 Preuve de correction de l'algorithme

Prouver la correction de l'algorithme 5.4 par rapport à la description de la section 5.3.3 consiste à démontrer que la tranche construite par l'algorithme contient toutes les transitions atteignables depuis le critère via $\xrightarrow{d} \star$, conformément à la définition 5.5. On montrera même que ces transitions sont insérées une seule fois dans la tranche, cf. théorème 5.17.

Lemme 5.16 *Au commencement d'une itération de la boucle à la ligne 20, le Booléen $viaCom[tr]$ est vrai seulement si tr a été atteint par l'algorithme, seulement via une dépendance de communication, au cours des itérations précédentes de la boucle.*

Démonstration. On se propose de démontrer le lemme 5.16 par contraposée. Remarquons tout d'abord que pour toute transition tr dans la spécification, $viaCom[tr]$ est initialisé à $faux$ lors de la phase (2), à la ligne 11. Ensuite, seules les lignes 25 et 27 peuvent affecter la valeur de $viaCom[tr]$. Par construction, les lignes 25 et 27 peuvent être exécutées seulement si la ligne 23 (où tr est inséré dans l'ensemble de travail) est exécutée. Supposons que tr ne soit pas atteinte seulement via une dépendance de communication ; les trois cas suivants envisagent toutes les possibilités.

- $tr \in Crit$. Le lemme 5.9 montre qu'aucune des lignes 25 et 27 n'est exécutée pour une transition $tr \in Crit$; $viaCom[tr]$ reste donc toujours à $faux$.
- $tr \notin Crit$, et tr est atteint seulement via des dépendances de données et de contrôle. En d'autres termes, il existe au moins une itération de la boucle 20 où la condition d'entrée est vérifiée, et plus précisément, lors de cette itération la partie $tr \in CD[tr_l] \cup DD[tr_l]$ est vérifiée alors que la partie $tr \in Com[tr_l] \wedge \neg viaCom[tr] \wedge \neg viaCom[tr_l]$ n'est vérifiée lors d'aucune itération. Le lemme 5.10 montre que cette itération est unique ; d'après les déductions ci-dessus, la branche `sinon` est exécutée à la ligne 26, et $viaCom[tr]$ reste donc toujours à $faux$.
- $tr \notin Crit$, et tr est atteint via au moins une dépendance de données ou de contrôle, et au moins une dépendance de communication. En d'autres termes, il existe au moins deux itérations de la boucle 20 où la condition d'entrée est vérifiée : I_1 où $tr \in CD[tr_l] \cup DD[tr_l]$ est vérifié, et I_2 où $tr \in Com[tr_l] \wedge \neg viaCom[tr]$

$\sum_i \sum_{tr_c \in conds(\mathscr{A}_i)} d_{tr_c}.|T_i|.|conds_{\mathscr{A}_i}|^2.lg(|T_i|)$ est habituellement nettement moindre que le coût de l'approximation $\sum_{tr \in T} d_{tr}.|T|^3.lg(|T|)$; sur l'exemple de la figure 3.2, la première expression vaut 385, alors que la dernière vaut 311296.

$\wedge\neg viaCom[tr_l]$ est vérifié. Si I_1 est antérieure I_2, alors on démontre, en appliquant le lemme 5.10 de manière analogue au cas précédent, que $viaCom[tr]$ reste toujours à $faux$. Si I_2 est antérieure à I_1 : lors de I_2 la branche `alors` à la ligne 25 est exécutée, et $viaCom[tr]$ est positionné à $vrai$. Lors de I_1, la branche `sinon` à la ligne 26 est exécutée, et $viaCom[tr]$ est positionné à $faux$. On applique à nouveau le lemme 5.10 pour montrer qu'il n'y a pas d'itération ultérieure où la condition d'entrée est vérifiée, et que $viaCom[tr]$ reste à $faux$. ◇

Théorème 5.17 (Correction de l'algorithme 5.4) *Si une transition tr est atteignable via $\xrightarrow{d} \star$ depuis une transition du critère, alors tr est inséré exactement une fois dans la tranche construite par l'algorithme 5.4.*

Démonstration. Soit tr une transition atteignable depuis le critère via $\xrightarrow{d} \star$. En d'autres termes, soit tr est dans le critère, soit il existe une séquence de dépendances contenant au plus une dépendance de communication consécutive $tr \xrightarrow{d} tr_1 \xrightarrow{d} \ldots \xrightarrow{d} tr_n$, où tr_i sont des transitions de la spécification, $n \geq 1$, et $tr_n \in Crit$. On se propose de démontrer le théorème 5.17 par récurrence sur la longueur de la séquence de dépendances. Tout d'abord, on remarque que dans l'algorithme 5.4, seules les lignes 15 et 22 permettent d'insérer une transition dans la tranche.

- Cas de base : $tr \in Crit$. Le lemme 5.9 dit que tr est inséré une seule fois dans l'ensemble de travail, à la ligne 16, et aussi que la ligne 23 n'est jamais exécutée pour tr. Or, par construction la ligne 15 est exécutée exactement le même nombre de fois que la ligne 16, et lorsque la ligne 23 n'est pas exécutée, il en est de même pour la ligne 22. La transition tr est donc insérée exactement une fois dans la tranche.
- Autres cas : $n \geq 1$. On a $tr \notin Crit$, et de ce fait la boucle à la ligne 13 n'a pas été exécutée pour tr, donc au début de la phase (3), on a $viaCom[tr] = faux$ et $fait[tr] = faux$. Supposons par récurrence que $\forall i \in [1..n], tr_i$ soit inséré exactement une fois dans la tranche. Pour que cela soit possible, il faut exécuter la ligne 22, et donc, par construction, la ligne 23. On a donc $\forall i \in [1..n]$, tr_i est inséré au moins une fois dans l'ensemble de travail. D'après le théorème 5.15, l'algorithme 5.4 termine ; il y a donc une itération de la boucle à la ligne 18, dans laquelle tr_l est identifié à tr_1 ; considérons cette itération.

Pour la dépendance $tr \xrightarrow{d} tr_1$, il y a trois possibilités, que l'on regroupe en deux cas. Si l'on a $tr \xrightarrow{dd} tr_1$ ou $tr \xrightarrow{cd} tr_1$, d'après les théorèmes de correction 4.13 et 4.32, on a $tr \in DD[tr_1] \cup CD[tr_1]$. Comme $fait[tr]$ est $faux$, la condition à la ligne 20 est vraie ; de plus, comme $viaCom[tr]$ est $faux$, la condition à la ligne 22 est vraie. La transition tr est donc insérée dans la tranche à la ligne 22. Or, la non-exécution de la ligne 23 impliquant la non-exécution de la ligne 22 (par construction),

le lemme 5.10 implique que l'insertion de tr dans la tranche ne sera pas renouvelée lors des itérations ultérieures.

Envisageons le dernier cas : si l'on a $tr \xrightarrow{com} tr_1$, alors par hypothèse, soit on a $tr_1 \in Crit$, soit on a $tr_1 \xrightarrow{dd} tr_2$ ou $tr_1 \xrightarrow{cd} tr_2$. Par hypothèse de récurrence, tr_1 est inséré dans la tranche, et donc atteint par l'algorithme ; la phrase précédente implique donc que tr_1 n'est pas atteint seulement via une dépendance de communication. Le lemme 5.16 montre donc que $viaCom[tr_1]$ est $faux$. Par ailleurs, d'après le théorème de correction 4.39, on a $tr \in ComD[tr_1]$. Comme on a $viaCom[tr] = faux$ et $fait[tr] = faux$, les conditions aux lignes 20, 22 et 25 sont vérifiées, et par conséquent tr est inséré dans la tranche, et $viaCom[tr]$ est positionné à vrai. Partant, supposons que tr soit inséré de nouveau dans la tranche lors d'itération(s) ultérieure(s) ; considérons la première de ces itérations. Il faut que la condition à la ligne 20 soit vérifiée. On a alors soit $tr \in CD[tr_l] \cup DD[tr_l]$, soit $tr \in ComD[tr_l] \wedge \neg viaCom[tr] \wedge \neg viaCom[tr_l]$. Comme $viaCom[tr]$ est $vrai$, on a nécessairement $tr \in CD[tr_l] \cup DD[tr_l]$; or dans ces conditions le test à la ligne 22 n'est pas vérifié, donc tr n'est pas inséré dans la tranche lors de cette itération. □

5.3.6 Preuve de complétude de l'algorithme

Prouver la complétude de l'algorithme 5.4 par rapport à la description de la section 5.3.3 consiste à démontrer que la tranche ne contient aucune des transitions qui ne sont pas atteignables depuis le critère via $\xrightarrow{d} \star$, conformément à la définition 5.5 (cf. théorème 5.19).

Lemme 5.18 *Au commencement d'une itération de la boucle à la ligne 20, si tr a été atteint par l'algorithme, seulement via une dépendance de communication, au cours des itérations précédentes de la boucle, alors $viaCom[tr]$ est vrai.*

Démonstration. Plaçons nous au début d'une itération quelconque de la boucle à la ligne 20, et supposons que $viaCom[tr]$ soit positionné à $faux$. On remarque qu'il y a seulement deux possibilités pour positionner $viaCom[tr]$ à $faux$. Si cette valeur provient de la ligne 11, cela signifie que tr n'a pas été atteint précédemment ; si cette valeur provient de la ligne 27, alors lors de la précédente itération qui a modifié $viaCom[tr]$, le test à la ligne 24 s'est évalué à $faux$, i.e. $tr \notin ComD[tr_l]$, donc nécessairement la partie $tr \in CD[tr_l] \cup DD[tr_l]$ du test à la ligne 20 s'est évaluée à $vrai$. Par application des théorèmes de complétude 4.16 et 4.33, l'algorithme a atteint tr via $tr \xrightarrow{cd} tr_l$, ou via $tr \xrightarrow{dd} tr_l$. ◇

Théorème 5.19 (Complétude de l'algorithme 5.4) *Les transitions non atteignables via* $\xrightarrow{d} \star$ *depuis une transition du critère, ne sont jamais insérées dans la tranche construite par l'algorithme 5.4.*

Démonstration. Tout d'abord, remarquons que par construction, l'insertion d'une transition dans la tranche, à la ligne 16, implique l'insertion dans l'ensemble de travail, à la ligne 23. Montrons que l'insertion d'une transition tr dans l'ensemble de travail implique l'atteignabilité recherchée pour tr. Posons l'hypothèse de récurrence suivante : au début de la n-ième itération de la boucle à la ligne 18, l'ensemble de travail contient seulement des transitions atteignables depuis le critère via une séquence de dépendances dans $\xrightarrow{d} \star$, de longueur $\leq n-1$.

- Cas $n = 1$. Le résultat découle d'une observation de la boucle à la ligne 13 : après la phase (2), l'ensemble de travail contient toutes les transitions du critère, et seulement celles-ci.
- Cas $n > 1$. Si tr est inséré dans l'ensemble de travail, alors la condition de la boucle à la ligne 20 est vérifiée, et donc il y a dans l'ensemble de travail une transition tr_l telle que $tr \in DD[tr_l] \cup CD[tr_l] \cup ComD[tr_l]$. En appliquant les théorèmes de complétude 4.16, 4.33, et 4.40, on a soit $tr \xrightarrow{dd} tr_l$, soit $tr \xrightarrow{cd} tr_l$, soit $tr \xrightarrow{com} tr_l$. Pour que le dernier cas soit retenu, il faut que $viaCom[tr_l]$ soit à $faux$, ce qui implique d'après le lemme 5.18 que tr_l n'a pas été atteint seulement via une dépendance de communication ; de plus, par hypothèse, tr_l est atteignable par une séquence de longueur $\leq n-1$. Ainsi, tr est atteignable depuis le critère par une séquence dans $\xrightarrow{d} \star$, de longueur $\leq n$, et au début de l'itération suivante $(n+1)$ de la boucle à la ligne 18, l'hypothèse de récurrence est donc toujours vérifiée. □

5.3.7 Exemples d'application de l'algorithme

Premier exemple

La figure 5.5 représente la tranche de la spécification en figure 3.2 page 53, par rapport au critère de réduction $\{m \rightharpoonup i\}$, qui est produite par l'algorithme 5.4. Les transitions grisées dans la figure 5.5 ne sont pas incluses dans la tranche, et les transitions en rouge font partie du critère (et sont incluses dans la tranche). Pour obtenir ce résultat, l'algorithme de réduction paramétrée calcule tout d'abord toutes les dépendances de données, de contrôle et de communication dans la spécification. Ces dépendances peuvent être représentées sous la forme d'un graphe de dépendances (cf. la figure 5.3, qui représente le graphe de dépendances pour cette spécification). Enfin, partant des transitions du critère, l'algorithme collecte toutes les transitions

atteignables dans le graphe de dépendances, en suivant dans le sens contraire[6] les arcs du graphe. Les transitions ainsi collectées forment la tranche souhaitée.

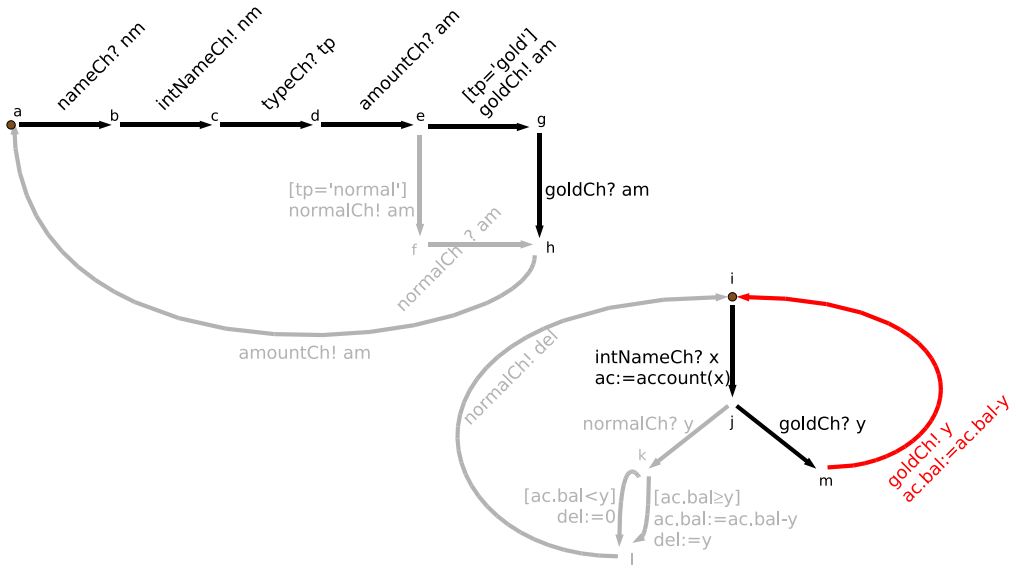


FIG. 5.5 – Tranche de la spécification en figure 3.2, par rapport au critère $\{m \rightharpoonup i\}$.

Dans la spécification de la figure 3.2, $m \rightharpoonup i$ est données-dépendant de $i \rightharpoonup j$, comme on l'a vu en section 4.2.3 ; $m \rightharpoonup i$ est aussi communication-dépendant de $g \rightharpoonup h$, pour cause de rendez-vous potentiel entre ces deux transitions, sur le canal *goldCh* ; $g \rightharpoonup h$ est à son tour contrôle-dépendant de $e \rightharpoonup g$, comme on l'a vu en section 4.3.3. Le processus continue jusqu'à ce que plus aucune transition ne puisse être ajoutée dans la tranche en construction : de manière analogue, et comme on peut le voir dans la figure 5.3, $b \rightharpoonup c$ et $a \rightharpoonup b$ sont atteignables depuis $i \rightharpoonup j$, $c \rightharpoonup d$ et $d \rightharpoonup e$ sont atteignables depuis $e \rightharpoonup g$, et $j \rightharpoonup m$ est atteignable depuis $m \rightharpoonup i$ dans le graphe de dépendance, en suivant dans le sens contraire les arcs du graphe. Pour résumer, l'expression suivante représente un sous-ensemble du graphe de dépendances, suffisant pour calculer la tranche souhaitée dans cet exemple :

$$(m \rightharpoonup i) \begin{cases} \xleftarrow{dd} (i \rightharpoonup j) & \xleftarrow{com} (b \rightharpoonup c) & \xleftarrow{dd} (a \rightharpoonup b) \\ \xleftarrow{com} (g \rightharpoonup h) & \xleftarrow{cd} (e \rightharpoonup g) \begin{cases} \xleftarrow{dd} (c \rightharpoonup d) \\ \xleftarrow{dd} (d \rightharpoonup e) \end{cases} & \\ \xleftarrow{dd} (j \rightharpoonup m) & & \end{cases}$$

[6] Les arcs sont suivis dans le sens contraire puisque l'algorithme 5.4 calcule une tranche *arrière* de la spécification donnée en paramètre (cf. section 5.3.1). Pour calculer une tranche avant, il s'agit de suivre les arcs du graphe, toujours à partir des transitions du critère.

D'autres dépendances existent dans le graphe de dépendances (cf. figure 5.3), mais soit elles ne sont pas atteignables depuis le critère, soit elles n'ajoutent pas d'information. Par exemple : $f \rightharpoonup h$ n'étant pas atteignable depuis $m \rightharpoonup i$ dans le graphe de dépendances, la dépendance de contrôle $(e \rightharpoonup f) \xrightarrow{cd} (f \rightharpoonup h)$ n'est pas prise en compte ; d'autre part, malgré que $j \rightharpoonup m$ soit dans la tranche, la dépendance de communication $(j \rightharpoonup m) \xrightarrow{com} (e \rightharpoonup g)$ n'est pas prise en compte si, comme dans le parcours décrit ci-dessus, l'algorithme a déjà atteint au préalable la dépendance $(e \rightharpoonup g) \xrightarrow{cd} (g \rightharpoonup h)$, et donc inséré $e \rightharpoonup g$ dans la tranche.

En résumé, l'algorithme de réduction paramétrée 5.4 a identifié, sur l'exemple de la figure 5.5, les parties de la spécification qui n'ont aucune influence potentielle sur le critère de réduction ; ces parties n'étant par conséquent pas incluses dans la tranche finale. Plus précisément, le critère $\{m \rightharpoonup i\}$ dénote un retrait effectif avec une carte *gold*, et l'on remarque dans la figure 5.5 que toutes les parties de la spécification qui traitent seulement des retraits avec une carte de type *normal* ne sont pas retenus dans la tranche finale.

Second exemple

De manière analogue au premier exemple ci-dessus, étudions une application de l'algorithme 5.4 sur la spécification de la figure 3.3 page 54. La figure 5.6 représente la tranche de cette spécification, par rapport au critère de réduction $\{\psi \rightharpoonup \kappa\}$, qui est extraite par l'algorithme 5.4. De même que précédemment, les transitions grisées sont celles qui ne sont pas retenues par l'algorithme pour constituer la tranche finale, et les transitions en rouge forment le critère.

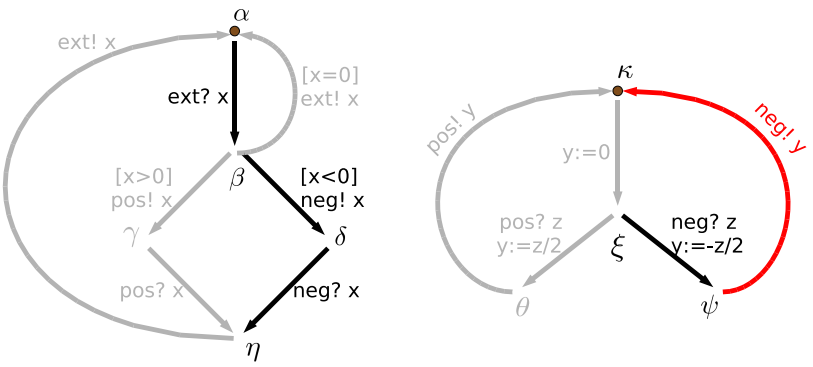


FIG. 5.6 – Tranche de la spécification en figure 3.3, par rapport au critère $\{\psi \rightharpoonup \kappa\}$.

Dans la spécification de la figure 5.6, $\psi \rightharpoonup \kappa$ est données-dépendant de $\xi \rightharpoonup \psi$, et cela est dû à la définition et à l'utilisation de la variable y par ces transitions ;

$\psi \rightharpoonup \kappa$ est aussi communication-dépendant de $\delta \rightharpoonup \eta$, pour cause de rendez-vous potentiel entre ces transitions, sur le canal neg ; $\delta \rightharpoonup \eta$ est à son tour contrôle-dépendant de $\beta \rightharpoonup \delta$, notamment car il existe un chemin maximal à partir de l'état β, sur lequel l'état δ n'apparaît pas ; et $\beta \rightharpoonup \delta$ est à son tour données-dépendant de $\alpha \rightharpoonup \beta$, ce qui est dû à la définition et aux utilisations de la variable x, par ces transitions.

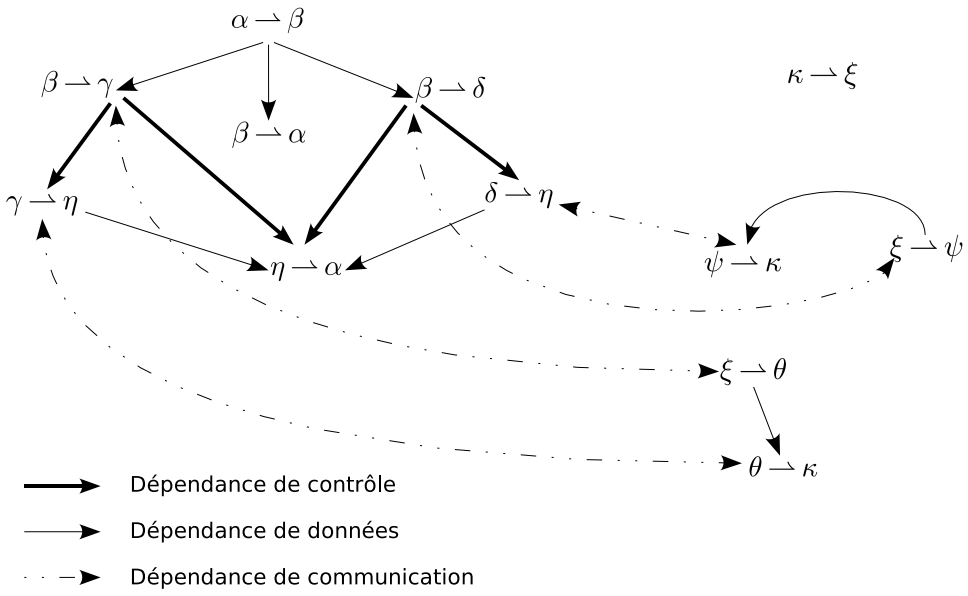


FIG. 5.7 – Graphe de dépendances de la spécification en figure 3.3

La figure 5.7 montre une représentation du graphe de dépendances de la spécification en figure 3.3. De même que pour la figure 5.3, les dépendances qui forment ce graphe sont calculées à l'aide des algorithmes du chapitre 4. Comme on peut le voir sur la figure 5.7, les transitions qui n'ont pas été mentionnées au paragraphe précédent ne sont pas atteignables dans le graphe de dépendances depuis $\psi \rightharpoonup \kappa$, en suivant les arcs du graphe dans le sens contraire. Par exemple, comme $\gamma \rightharpoonup \eta$ n'est pas atteignable depuis $\psi \rightharpoonup \kappa$, la dépendance de contrôle $(\beta \rightharpoonup \gamma) \xrightarrow{cd} (\gamma \rightharpoonup \eta)$ n'est pas prise en compte.

Pour résumer, l'expression suivante représente un sous-ensemble suffisant du graphe de dépendance, pour le calcul de cette tranche particulière :

$$(\psi \rightharpoonup \kappa) \begin{cases} \xleftarrow{dd} (\xi \rightharpoonup \psi) \\ \xleftarrow{com} (\delta \rightharpoonup \eta) \end{cases} \xleftarrow{cd} (\beta \rightharpoonup \delta) \quad \xleftarrow{dd} (\alpha \rightharpoonup \beta)$$

À nouveau, notre algorithme de réduction paramétrée a détecté des parties de la spécification qui n'ont aucun impact potentiel sur le critère, ce qui permet de mettre en lumière des fonctionnalités orthogonales dans la spécification. Les parties de la spécification qui traitent uniquement des nombres positifs ou nuls ne sont en effet pas retenues pour constituer la tranche finale. On peut aussi remarquer que la transition $\kappa \rightharpoonup \xi$, qui définit une valeur qui n'est jamais utilisée, n'est pas incluse dans la tranche.

5.4 Mise en œuvre

Les algorithmes exposés aux chapitres 4 et 5 ont été implantés dans un outil pour la réduction paramétrée de spécifications formées d'IOSTS, nommé CARVER, qui sera décrit dans cette section.

5.4.1 CARVER, un outil pour la réduction paramétrée de spécifications

Nous montrons dans cette section, que CARVER est intégré avec succès dans l'environnement de l'outil AGATHA. On rappelle que l'outil AGATHA, développé au CEA LIST dans le cadre du projet du même nom [Lap02, Pie03, Rap04, RGLG03, BFG+03a, LBV+04], permet d'effectuer un dépliage comportemental de spécifications formées d'automates communicants, en s'appuyant sur des techniques d'exécution symbolique, dans le but de générer automatiquement des cas de test, et de valider des spécification formelles.

L'interface utilisateur d'AGATHA

Les spécifications sont exprimées dans un langage intermédiaire, appelé xLIA[7] ; la figure 5.8 donne un aperçu de ce langage textuel, qui permet notamment d'exprimer intégralement les spécifications formées d'IOSTS. Ainsi, l'association de CARVER avec AGATHA forme un environnement de validation et de génération de tests, avec réduction paramétrée de modèles, pour les spécifications formées d'automates communicants et exprimées dans la restriction aux spécifications formées d'IOSTS du langage xLIA.

La figure 5.8 montre l'interface utilisateur de l'outil AGATHA. On voit en haut de la fenêtre principale des boutons qui permettent de lancer l'application noyau

[7] *extensible Langage Intermédiaire* AGATHA.

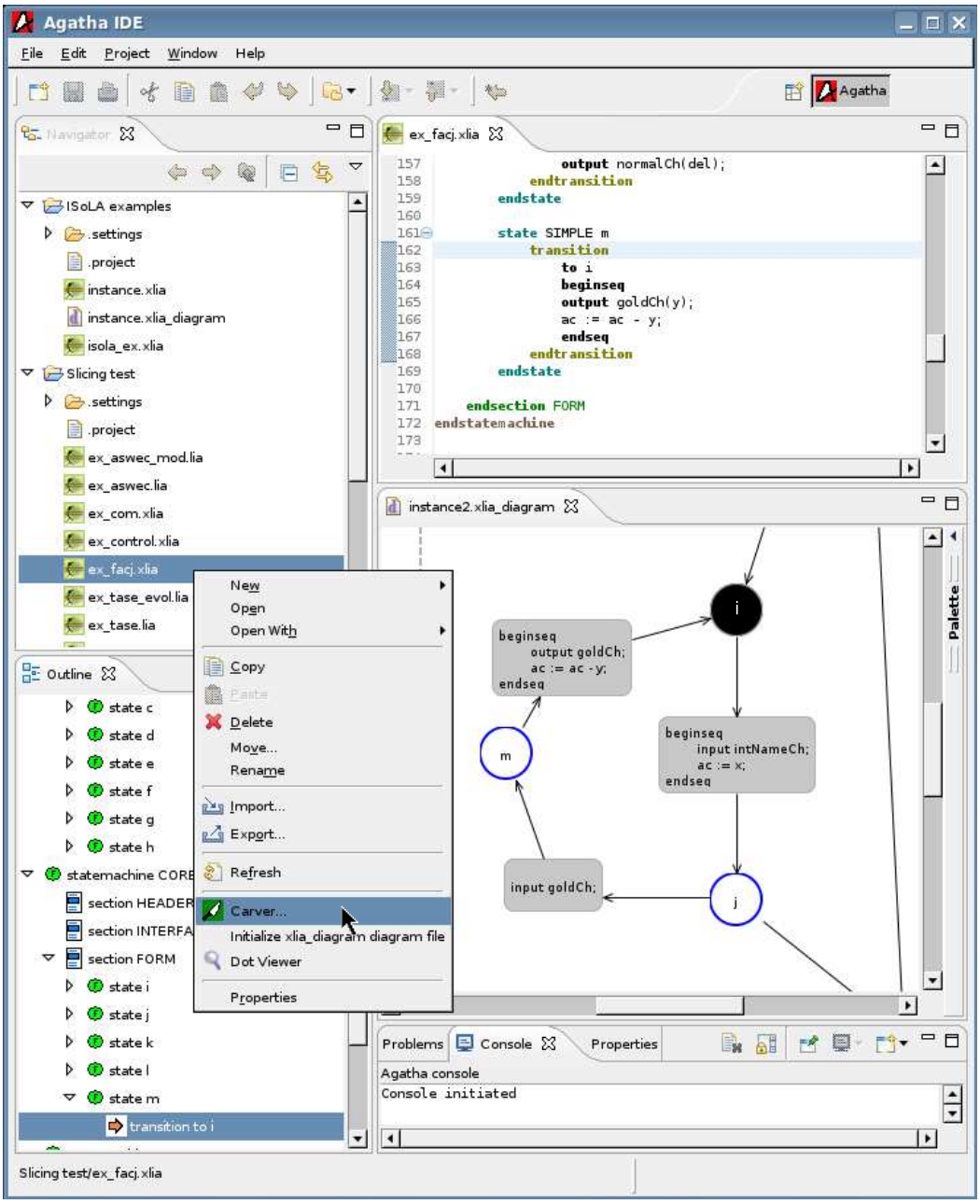


FIG. 5.8 – Intégration de CARVER dans l'interface utilisateur d'AGATHA.

d'AGATHA et de configurer des paramètres ; sur la droite de la fenêtre, un éditeur textuel de xLIA avec coloration syntaxique, et en dessous un éditeur graphique de xLIA ; en haut à gauche, un explorateur de projets ; en bas à gauche une vue de l'arbre de syntaxe abstraite du fichier xLIA actuellement ouvert ; et enfin, en bas à droite une console qui permet d'afficher les traces des exécutions et analyses effectuées par AGATHA et CARVER.

Utilisation de CARVER

Dans l'interface utilisateur de l'outil AGATHA, l'intégration de CARVER est réalisée sous la forme d'un *plug-in*, qui peut être invoqué en activant l'item correspondant dans le menu contextuel associé aux fichiers qui contiennent des spécifications xLIA. Pour faire apparaître ce menu, l'utilisateur peut effectuer un clic droit sur un fichier, dont l'extension est "`.xlia`", dans l'explorateur de projets. La figure 5.8 montre une situation où le fichier sélectionné pour être analysé par CARVER contient une version xLIA de la spécification en figure 3.2 (ce fichier est nommé `ex_facj.xlia`).

Dans la structure de données interne de CARVER, les labels des transitions sont abstraits par des listes de variables, qui indiquent les variables définies ou utilisées, pour chaque élément de chaque transition (i.e. la garde, l'action, et les affectations). Lorsque le menu contextuel associé à CARVER est activé, un analyseur grammatical (ou *parser*) dédié à cette phase d'abstraction est appelé, et un nom unique est donné à chaque transition de la spécification. Ensuite, la fenêtre de la figure 5.9 est affichée, permettant à l'utilisateur de configurer les paramètres qu'il souhaite voir appliquer lors du calcul de la tranche souhaitée. Dans cette fenêtre, une liste des transitions abstraites de la spécification est affichée dans un tableau, et l'utilisateur est invité à sélectionner un ensemble non-vide de transitions, pour constituer le critère de réduction. Lorsqu'il a choisi un critère de réduction, l'utilisateur peut choisir de calculer une tranche *avant*, *arrière*, ou *complète*, de la spécification donnée, par rapport au critère choisi. On rappelle qu'une tranche avant est formée des transitions qui sont potentiellement influencées par le critère, et qu'une tranche arrière est formée des transitions qui influencent potentiellement le critère ; on appelle alors tranche complète l'union des tranches arrière et avant, par rapport au même critère. Une tranche complète a pour but de montrer de façon exhaustive tout ce qui peut avoir un rapport avec le critère.

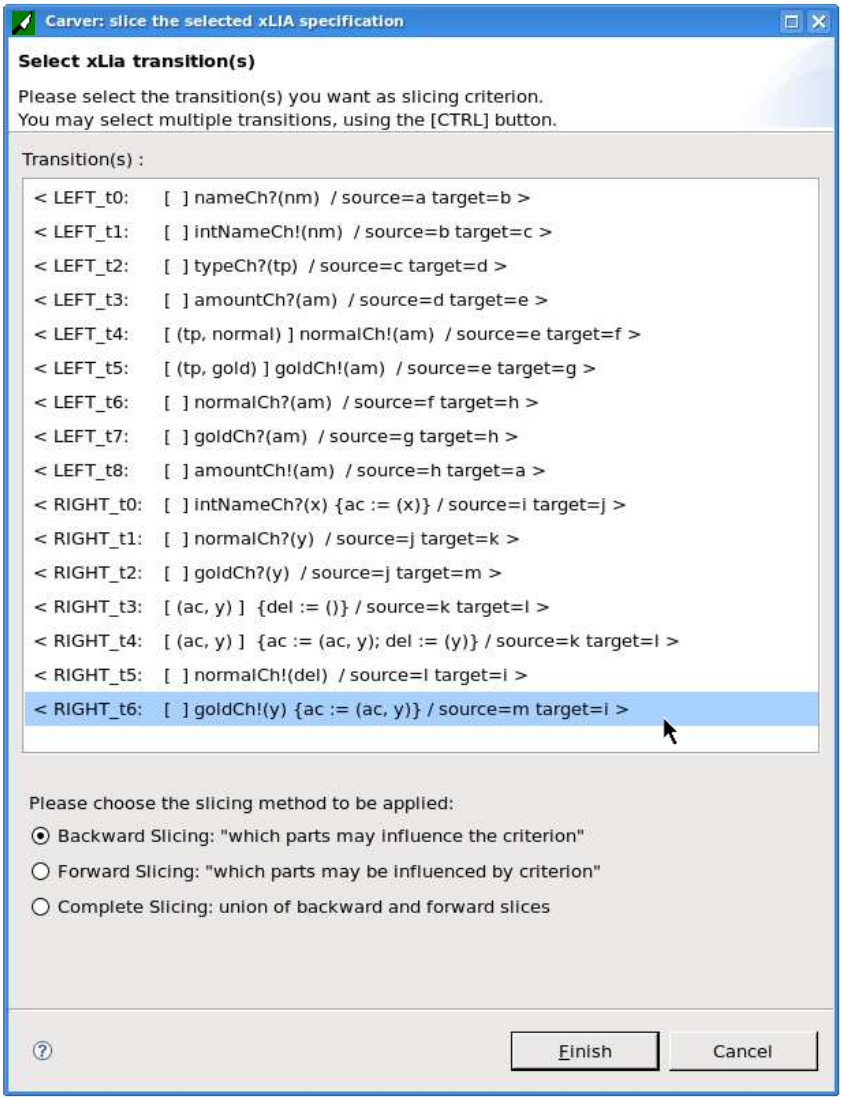
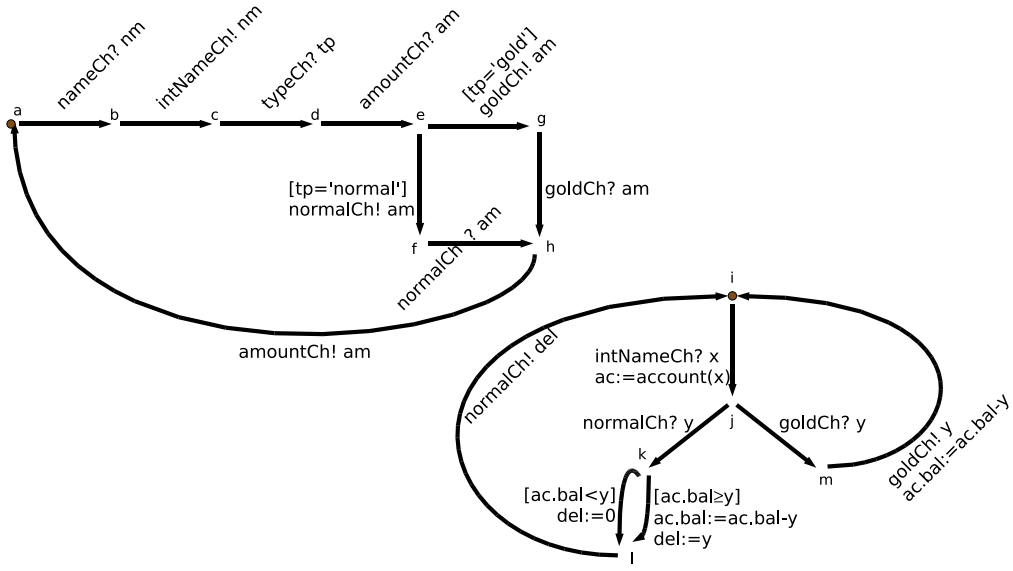


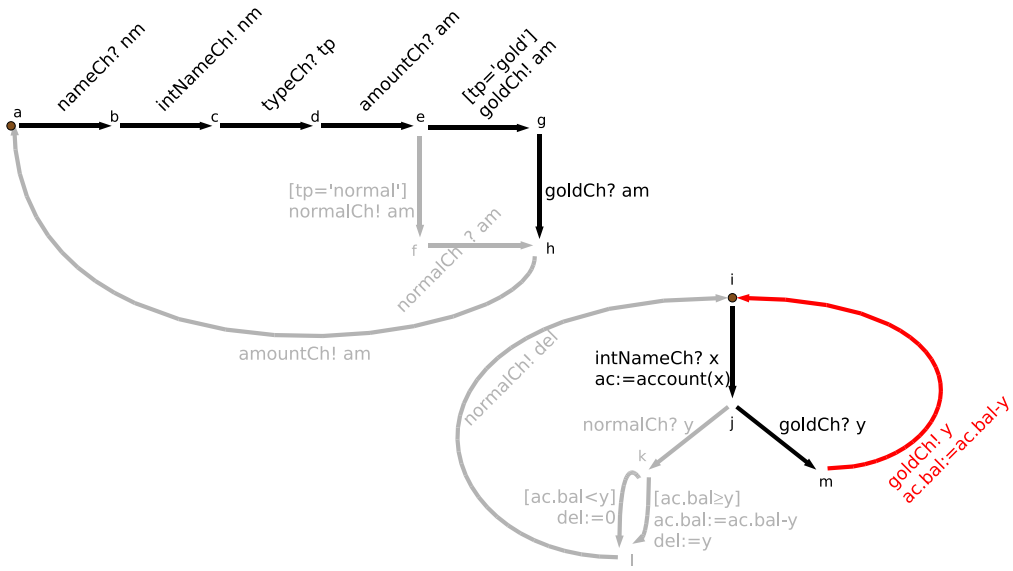
FIG. 5.9 – Sélection d'un critère de réduction dans CARVER.

```
========================================================================

LIA specification successfully parsed for slicing: ex_facj.xlia

New dependences successfully computed...

New parameter(s) found...
Backward slice successfully computed!

INPUT DATA:
Specification:

IOSTS LEFT ; initState=a
{
< LEFT_t0:  [  ] nameCh?(nm)  / source=a target=b >
< LEFT_t1:  [  ] intNameCh!(nm)  / source=b target=c >
< LEFT_t2:  [  ] typeCh?(tp)  / source=c target=d >
< LEFT_t3:  [  ] amountCh?(am)  / source=d target=e >
< LEFT_t4:  [ (tp, normal) ] normalCh!(am)  / source=e target=f >
< LEFT_t5:  [ (tp, gold) ] goldCh!(am)  / source=e target=g >
< LEFT_t6:  [  ] normalCh?(am)  / source=f target=h >
< LEFT_t7:  [  ] goldCh?(am)  / source=g target=h >
< LEFT_t8:  [  ] amountCh!(am)  / source=h target=a >
}


IOSTS RIGHT ; initState=i
{
< RIGHT_t0:  [  ] intNameCh?(x) {ac := (x)} / source=i target=j >
< RIGHT_t1:  [  ] normalCh?(y)  / source=j target=k >
< RIGHT_t2:  [  ] goldCh?(y)  / source=j target=m >
< RIGHT_t3:  [ (ac, y) ]  {del := ()} / source=k target=l >
< RIGHT_t4:  [ (ac, y) ]  {ac := (ac, y); del := (y)} / source=k target=l >
< RIGHT_t5:  [  ] normalCh!(del)  / source=l target=i >
< RIGHT_t6:  [  ] goldCh!(y) {ac := (ac, y)} / source=m target=i >
}


Slicing criterion:

< RIGHT_t6:  [  ] goldCh!(y) {ac := (ac, y)} / source=m target=i >


RESULTING SLICE:

IOSTS LEFT ; initState=a
{
< LEFT_t0:  [  ] nameCh?(nm)  / source=a target=b >
< LEFT_t1:  [  ] intNameCh!(nm)  / source=b target=c >
< LEFT_t2:  [  ] typeCh?(tp)  / source=c target=d >
< LEFT_t3:  [  ] amountCh?(am)  / source=d target=e >
< LEFT_t5:  [ (tp, gold) ] goldCh!(am)  / source=e target=g >
< LEFT_t7:  [  ] goldCh?(am)  / source=g target=h >
}


IOSTS RIGHT ; initState=i
{
< RIGHT_t0:  [  ] intNameCh?(x) {ac := (x)} / source=i target=j >
< RIGHT_t2:  [  ] goldCh?(y)  / source=j target=m >
< RIGHT_t6:  [  ] goldCh!(y) {ac := (ac, y)} / source=m target=i >
}

========================================================================
```

FIG. 5.10 – Sortie dans la console de CARVER, étant donnés la spécification en figure 3.2 et le critère $\{m \rightharpoonup i\}$.



5.11 (a)



5.11 (b)

FIG. 5.11 – Représentation graphique de l'entrée et de la sortie de CARVER, étant donnés la spécification en figure 3.2 et le critère $\{m \rightharpoonup i\}$.

Résultats

La figure 5.10 montre la sortie de CARVER, pour le critère $\{m \rightharpoonup i\}$ et la spécification en figure 3.2 (cette sortie est affichée dans la console de l'outil). Comme attendu, la sortie décrit la même tranche que la figure 5.5 : on peut s'en assurer à l'aide de la figure 5.11, qui rappelle en 5.11 (a) la spécification de la figure 5.5, et en 5.11 (b) la tranche de cette spécification par rapport au critère $\{m \rightharpoonup i\}$, qui est extraite par notre algorithme, et représentée en figure 5.5. Toujours dans la figure 5.10, les premières lignes de la sortie montrent que le fichier xLIA a été analysé par l'analyseur grammatical dédié, que les relations de dépendances dans la spécification ont été calculées, et enfin que la tranche souhaitée a été calculée. Si ultérieurement, l'utilisateur demande une nouvelle tranche de la même spécification, alors CARVER réutilise les résultats de la phase d'analyse grammaticale, et de la phase de calcul des dépendances, conformément à la remarque à propos de réutilisabilité, en section 5.3.3. L'algorithme de réduction paramétrée est alors appliqué, par rapport au nouveau critère de réduction donné – si de plus le critère de réduction est inchangé par rapport au calcul précédent, l'outil le détecte, n'effectue aucun calcul et affiche directement la tranche précédente.

5.4.2 Bilan de l'intégration

L'outil de réduction paramétrée CARVER, introduit en section 5.4.1, a été développé à l'origine dans le but d'appliquer et de démontrer l'efficacité de notre méthode de réduction paramétrée de spécifications formées d'automates communicants (cf. chapitres 4 et 5, [LGP07b]). Le noyau de l'application implante les algorithmes décrits aux chapitres sus-cités, pour extraire automatiquement des tranches des spécifications données en paramètre. Ce noyau peut être intégré dans l'environnement d'outils existants, dans le but de réduire la complexité de leurs analyses. Comme on l'a vu, AGATHA compte parmi les outils appropriés pour une telle intégration, mais on peut aussi penser à un model-checker, ou tout autre outil impliquant des analyses complexes de spécifications formées d'automates communicants (dans un formalisme compatible avec les IOSTS).

En extrapolant à partir de l'expérience d'intégration avec AGATHA rapportée dans cette section, on peut dire que la connection de CARVER avec un outil T peut être effectuée en deux étapes simples : tout d'abord, définir un analyseur grammatical pour traduire le langage d'entrée de T dans la structure de données interne de CARVER, et ensuite, définir un plug-in pour CARVER dans l'interface utilisateur de T, afin que l'utilisateur puisse définir les paramètres de la réduction, et afin

d'afficher les tranches en sortie. De manière générale, une telle intégration peut se faire en laissant le noyau de CARVER inchangé ; cependant, certaines modifications peuvent s'avérer nécessaires dans le cas où le langage de T fait appel à des concepts ne pouvant pas être traduits dans le formalisme des IOSTS.

L'interface utilisateur de CARVER, décrite par les figures 5.8, 5.9 et 5.10, a été créée pour démontrer que l'application noyau est opérationnelle. En cela, l'objectif est rempli puisque les tranches sont automatiquement extraites de spécifications, et ensuite affichées ; cependant, cette interface reste rudimentaire et nécessiterait des améliorations. Il serait notamment souhaitable que l'utilisateur puisse sélectionner ses critères de réduction sur une représentation graphique de la spécification. De plus, les tranches en sortie devront être converties à nouveau dans le langage XLIA, et enregistrées dans des fichiers `.xlia`, qui à leur tour pourront être ouverts dans l'éditeur textuel et graphique d'AGATHA.

Enfin, grâce à cette intégration nous avons appliqué notre méthode de réduction paramétrée sur un jeu étendu d'exemples académiques (comme celui représenté en figure 5.11), qui inclut tous les cas de figure[8] pouvant être rencontrés pour une spécification formée d'IOSTS. Si l'on considère de plus la complexité polynomiale de la méthode (cf. section 5.3.4, théorème 5.15), cette expérience d'intégration a pour débouché naturel l'application de CARVER à des spécifications industrielles.

5.5 Applications

Cette section dresse une liste rapide des applications envisagées pour continuer le travail présenté dans ce manuscrit.

5.5.1 Preuve de propriétés

La première application est la preuve de propriété : la technique retenue est basée sur le fait que les propriétés à prouver peuvent être traduites en automates (par exemple toute formule de logique temporelle LTL est représentable en "automate de Büchi"). La solution préconisée est d'intégrer la propriété dans le modèle sous la forme d'un automate supplémentaire qui contraint le modèle initial et de choisir comme critère de réduction paramétrée toutes les transitions de cet automate. Ainsi, on obtient un modèle réduit qui contient tous les éléments du modèle initial qui sont "utiles" à la preuve de la propriété considérée.

[8] On se réfère notamment à l'étude de cas de la section 4.3.3, concernant les différents types de branchements dans un IOSTS.

5.5.2 Modèles paramétrés

Une autre application concerne les modèles paramétrés pour lesquels on veut déterminer certains paramètres inconnus en fonction de propriétés à vérifier (application faite au CEA LIST en bioinformatique pour les réseaux de régulation génétiques [MGCL07]), la technique préconisée est basée sur une idée similaire à celle utilisée dans le cadre de la preuve de propriétés : on intègre la propriété à vérifier sous la forme d'un automate supplémentaire qui contraint le modèle initial, et cela permet de garder dans la tranche résultante toutes les parties utiles pour déterminer les paramètres inconnus. Le critère de réduction paramétrée est dans ce cas le même que pour la preuve, c'est l'ensemble des transitions de l'automate qui représente la propriété.

5.5.3 Test de conformité

Une troisième application est le test de conformité. Pour cela, la technique est classiquement basée sur des objectifs de tests qui "brident" le modèle lors du procédé de génération de tests. L'objectif de test peut être vu comme une propriété à tester, ainsi la technique préconisée peut-elle être similaire aux deux applications précédentes : la propriété à tester est intégrée au modèle sous la forme d'un automate supplémentaire qui la représente. À nouveau, les transitions de cet automate constituent le critère de réduction paramétrée.

5.5.4 Gestion des évolutions

Enfin une quatrième application s'inscrit dans le cadre de la gestion des évolutions d'un modèle, i.e. des évolutions du système correspondant (par exemple dans une famille de produits). Dans ce cadre, si la partie qui évolue est représentable sous la forme d'un sous-ensemble du système, cela permet de déterminer un sous-ensemble minimal d'automates du modèle incluant toutes les évolutions possibles a priori, les autres automates du modèle étant considérés comme "stables", c'est-à-dire non sujets à évolutions. La technique préconisée est de calculer quelle est la partie utile (pour les phases de vérification/validation du produit) du sous-ensemble stable du système lors de l'intégration d'évolutions à venir. On applique pour cela le procédé de réduction paramétrée à la partie stable du modèle, en prenant comme critère l'ensemble des transitions de cette partie stable qui peuvent communiquer avec la partie sujette à évolution. Par exemple, à la suite d'une évolution, on peut tester la partie stable de façon plus efficace en effectuant la génération de tests directement

sur la tranche résultante – il n'est en effet pas nécessaire de tester à nouveau les éléments qui ne sont pas dans la tranche. Cette technique possède donc un potentiel de réduction des temps de calcul et des nombres de tests à effectuer après chaque évolution.

5.6 Travaux connexes

Dans cette section, nous allons passer brièvement en revue les travaux, à notre connaissance les plus proches, qui ont été publiés dans le domaine. Cette section est découpée en trois thèmes dans lesquels ce manuscrit est fortement impliqué : la réduction paramétrée de spécifications formées d'automates communicants, les questions de précision concernant les relations dépendances, et enfin les spécifications formées d'automates communicants. Le lecteur est invité à se référer aux sections précédentes pour les travaux connexes sur les dépendances de données [Muc97, FOW87, Nan01, KSV96, TGL06] (en section 4.2), les dépendances de contrôle [RAB+05, AK02, Muc97] (en section 4.3), et les dépendances de communication [Sar97, MT00] (en section 4.4).

5.6.1 Réduction paramétrée de spécifications formées d'automates communicants

Parmi les travaux dont nous connaissons l'existence, et qui ont été publiés avant [LGP07b], seuls les travaux de Bozga *et al.* peuvent correspondre précisément à cet intitulé. Dans [BFG03b], les auteurs présentent une méthode de réduction paramétrée pour des spécifications formées d'automates communicants qui communiquent par files FIFO ; toutefois, cette méthode est focalisée sur la génération automatique de cas de tests, et n'est pas basée sur des relations de dépendances comme la notre – nous avons discuté les avantages d'une telle approche en sections 1.4.2 et 5.3. Enfin, dans [BFG03b], les critères de réduction comprennent des ensembles de signaux et des données externes à la spécification (*objectifs de test*, "*feeds*"). À l'opposé, notre définition d'une tranche de spécification fait intervenir seulement des éléments propres à la spécification ; la méthode correspondante a donc pour vocation d'être plus générale et plus proche des concepts traditionnels de la réduction paramétrée de programmes (cf. section 1.4 page 17).

5.6.2 De la précision des relations de dépendances

À propos de la précision relations de dépendance, Krinke [Kri98] est à l'origine du concept de dépendance transitive dans des programmes concurrents exprimés dans un langage avec le mécanisme `fork/join`. Dans ce travail, la transitivité des dépendances est définie en se basant sur la notion de *threaded witness*[9], ce qui garantit la précision de la méthode.

Ces travaux préliminaires ont été étendus dans [Kri03, Nan01], notamment pour prendre en compte les appels de procédure. En section 5.2.1, nous avons étendu le concept de dépendance transitive aux spécifications formées d'IOSTS (cf. définition 5.1), bien que nous n'ayons pas mis en œuvre les mêmes moyens que Krinke et Nanda[10] pour vérifier la transitivité (cf.section 5.2). Dans les spécifications formées d'IOSTS, il n'y a pas de variables partagées, ni d'appels de procédure ; nous avons ainsi montré que la seule source d'intransitivité peut être trouvée dans les actions de communication.

Enfin, nous soutenons l'idée de Millett *et al.* dans [MT00], que même des tranches imprécises (au sens de la transitivité des dépendances) peuvent avoir de l'utilité, en particulier dans le cas où les algorithmes précis comme ceux de Krinke et Nanda seraient trop lents pour répondre en temps requis par l'utilisateur. Cependant, les considérations sur la précision en section 5.2.5 peuvent conduire à penser qu'il y a un plus grand potentiel de calculer des dépendances imprécises avec la gestion des variables partagées de [MT00], qu'avec notre gestion des actions de communication.

5.6.3 Spécifications formées d'automates communicants

Parmi les travaux connexes sur les spécifications formées d'automates communicants, en général, Gaston *et al.* proposent dans [GLRT06] une méthode pour définir des objectifs de test et générer automatiquement des cas de test pour des spécifications formées d'IOSTS, en se basant sur des techniques d'exécution symbolique. Rapin *et al.* proposent dans [RGLG03] des techniques d'exécution symbolique dans le but d'exhiber tous les comportements de spécifications formées d'IOLTS, sachant que le formalisme des IOSTS étend celui des IOLTS, pour gérer les types de données (cf. section 3.1).

[9] Intuitivement, il s'agit d'une séquence qui témoigne de l'existence d'une exécution dans un système à processus multiples.

[10] Les algorithmes de Krinke et Nanda pour calculer des dépendances transitives ont une complexité exponentielle au pire, mais leurs implémentations rendent ces techniques praticables sur des programmes de taille non négligeable [Kri03, Nan01].

Chapitre 6

Conclusion

6.1 Synthèse

La liste suivante résume les principales contributions de la thèse :

1. Un cadre théorique pour les analyses statiques de spécifications formées d'automates communicants ;
2. La définition formelle des relations de dépendances qui existent dans les spécifications formées d'automates communicants définies par la contribution 1 ;
3. La définition formelle d'une tranche de spécification formée d'automates communicants, par rapport à un critère ;
4. La mise au point d'algorithmes efficaces pour calculer les relations de dépendances et les tranches définies par les contributions 2 et 3, ainsi que les démonstrations de la complexité polynomiale, de la correction et complétude par rapport aux définitions correspondantes, de ces algorithmes ;
5. La mise en œuvre des algorithmes de la contribution 4 dans l'outil CARVER, pour la réduction paramétrée de spécifications formées d'automates communicants.

Les travaux de recherche rapportés dans cette thèse nous conduisent à proposer une méthode de réduction paramétrée pour des spécifications formées d'automates communicants (et plus précisément, formées d'IOSTS), ainsi qu'un outil qui implante cette méthode. En plus des principales contributions citées ci-dessus, nous avons apporté des éléments d'évaluation de la précision d'une telle approche de réduction paramétrée (cf. section 5.2) ; nous avons alors identifié les conditions nécessaires pour que notre approche soit précise, selon cette notion d'évaluation. Cette étude a mis en évidence une possibilité pour que la précision de notre approche soit réduite.

Toutefois, cette possibilité découle des choix que nous avons fait pour la définition de la relation de dépendance de communication, qui nous ont permis de mettre au point un algorithme de complexité réduite, et nous avons observé que cette possibilité ne caractérise habituellement pas les spécifications qui ont vocation à être analysées avec notre méthode. D'autre part, les autres composantes de notre méthode, les dépendances de données et de contrôle, sont précises dans tous les cas, toujours selon la notion introduite en section 5.2.

Enfin, l'outil CARVER, développé dans le cadre de cette thèse, a été intégré avec succès dans l'environnement de l'outil AGATHA du CEA LIST, comme première étape d'un processus d'évaluation de l'impact de la réduction paramétrée pour la validation de spécifications formelles.

6.2 Perspectives

Les résultats expérimentaux obtenus avec l'outil CARVER ouvrent des perspectives intéressantes dans le cadre des méthodes formelles pour la validation et la vérification de spécifications formés d'automates communicants. Ces résultats mettent en valeur l'intérêt que cet outil présente dans la mise en œuvre des applications évoquées en section 5.5 (e.g. vérification de propriétés, génération de tests). De façon plus générale, cet outil présentera un intérêt particulier pour toute analyse complexe et de préférence ciblée : on pourra alors extraire – éventuellement automatiquement – un critère, pour appliquer la réduction paramétrée en amont de l'analyse, et ainsi bénéficier de gains potentiellement importants en termes à la fois de temps de calcul, d'espace mémoire et de compréhension. En plus des applications évoquées, les pistes pour continuer ce travail sont relativement nombreuses ; nous en citerons trois.

La méthode de réduction paramétrée présentée dans ce manuscrit peut être raffinée en une méthode où les transitions ne seraient plus considérées comme des éléments atomiques, ce qui permettrait de réduire ces dernières afin de calculer des tranches plus petites et plus précises. Pour ce faire, les relations de dépendances devront être adaptées pour mettre en relations des parties de transitions (e.g. garde, action de communication), et non plus des transitions. Ce raffinement sera utile pour des applications dans lesquelles les critères seront préférablement constitués de parties de transitions.

Une autre idée serait d'améliorer la précision de la relation de dépendance de communication, afin de satisfaire dans tous les cas la notion de précision introduite en section 5.2. Cette amélioration pourrait se faire par la mise au point d'un al-

gorithme "*May Happen in Parallel*" [1] (cf. [NA98]) pour les spécifications formées d'automates communicants : l'idée serait que si deux transitions communiquent sur le même canal, mais ne peuvent pas être exécutées de façon concurrente, alors il ne devrait y avoir de dépendance de communication entre ces deux transitions. Comme nous l'avons suggéré en section 5.2.4, la précision de la relation de dépendance de communication pourrait aussi être améliorée à l'aide d'informations sur la composition parallèle. Dans les deux cas, la méthode résultante devrait être comparée empiriquement avec la méthode d'origine (présentée dans cette thèse), car il n'est *a priori* pas évident que le gain en précision justifie le surcoût de complexité induit.

Enfin, selon la définition d'une spécification formée d'IOSTS, les ensembles de variables sont disjoints deux à deux. Les variables partagées étant proscrites, les IOSTS communiquent donc seulement via les canaux de communication. Si l'on retire cette hypothèse, il faudra adapter la méthode de réduction paramétrée pour prendre en compte une autre sorte de dépendances de données inter-automates, similaire aux dépendances d'interférence de Krinke [Kri98]. Il est alors intéressant de remarquer que Müller-Olm et Seidl ont montré dans [MOS01] que la réduction paramétrée dans ce contexte est un problème PSPACE-complet ; d'ailleurs, toutes les méthodes dont nous avons connaissance, qui traitent ce problème de façon précise, ont une complexité exponentielle dans le pire cas, bien qu'il ait été montré que ces méthodes sont applicables en pratique [Kri03, Nan01].

[1] On suggère la traduction "Peuvent être exécutés de façon concurrente".

Table des figures

Bibliographie

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers : principles, techniques and tools*. Addison Wesley, 1986.

[AU75] Alfred V. Aho and Jeffrey D. Ullman. Node listings for reducible flow graphs. In *ACM Symposium on Theory of Computing (STOC)*, pages 177–185, 1975.

[BFG+03a] Céline Bigot, Alain Faivre, Jean-Pierre Gallois, Arnault Lapitre, David Lugato, Jean-Yves Pierron, and Nicolas Rapin. Automatic test generation with AGATHA. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 591–596, 2003.

[BFG03b] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. Using static analysis to improve automatic test generation. *Software Tools for Technology Transfer (STTT)*, 4(2):142–152, 2003.

[BH93] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Automated and Algorithmic Debugging (AADEBUG)*, pages 206–222, 1993.

[BW05] Ingo Brückner and Heike Wehrheim. Slicing an integrated formal method for verification. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 360–374, 2005.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[CHK01] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. Technical report, Rice University, 2001. `http://www.cs.rice.edu/~keith/EMBED/dom.pdf`.

[Cif93] Cristina Cifuentes. A structuring algorithm for decompilation. In *Conferencia Latinoamericana de Informatica (CLEI)*, pages 267–276, 1993.

[CK98] Jean-François Collard and Jens Knoop. A comparative study of reaching definitions analyses. Technical Report 1998/22, PRiSM, Université de Versailles, 1998.

[CR94] Juei Chang and Debra J. Richardson. Static and dynamic specification slicing. In *Fourth Irvine Software Symposium*, 1994.

[DHH+06] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Prasad Ranganath, Robby, and Todd Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 73–89, 2006.

[Fou93] Jean-Pierre Fournier. *Fiabilité du Logiciel.* Hermes, 1993.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[GLRT06] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *IFIP International Conference on Testing of Communicating Systems (TestCom)*, pages 1–18, 2006.

[Göd31] Kurt Gödel. Über formal unentscheidbare sätze der *Principia Mathematica* und verwandter systeme I. *Monatshefte für mathematik und physik*, 38:173–198, 1931. English translation: *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*, translated by Bernard Meltzer. Dover, 1992.

[GR02] Vinod Ganapathy and S. Ramesh. Slicing synchronous reactive programs. *Electronic Notes in Theoretical Computer Science*, 65(5), 2002.

[GSL96] Susanne Graf, Bernhard Steffen, and Gerald Luttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computing (FACJ)*, 8(5):607–616, 1996.

[Har85] Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *ACM Symposium on Theory of Computing (STOC)*, pages 185–194, 1985.

[HCD+99] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Static Analysis Symposium (SAS)*, pages 1–18, 1999.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* North-Holland, New York, 1977.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Notices*, 23(7):35–46, June 1988.

[HW97] Mats Per Erik Heimdahl and Michael W. Whalen. Reduction and slicing of hierarchical state machines. In *European Software Engineering Conference (ESEC) / Foundations of Software Engineering (SIGSOFT FSE)*, pages 450–467, 1997.

[Ken75] Ken Kennedy. Node listings applied to data flow analysis. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 10–21, 1975.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 268–299, 1973.

[KL88] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[Kri98] Jens Krinke. Static slicing of threaded programs. In *Program Analysis For Software Tools and Engineering (PASTE)*, pages 35–42, 1998.

[Kri03] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs.* PhD thesis, Universitat Passau, Germany, april 2003.

[KSV96] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):268–299, 1996.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[Lap02] Arnault Lapitre. *Procédure de réduction pour les systèmes à base d'automates communicants : formalisation et mise en œuvre.* PhD thesis, Université Paris XI Orsay, France, december 2002.

[LBV+04] David Lugato, Céline Bigot, Yannick Valot, Jean-Pierre Gallois, Sébastien Gérard, and François Terrier. Validation and automatic test generation on UML models: the AGATHA approach. *Software Tools for Technology Transfer (STTT)*, 5(2–3):124–139, march 2004.

[LG06] Sébastien Labbé and Jean-Pierre Gallois. Towards slicing communicating extended automata. In *International Symposium on Formal Methods (FM), Doctoral Symposium*, 2006. `http://fm06.mcmaster.ca/01SebastienLabbe.pdf`.

[LG07] Sébastien Labbé and Jean-Pierre Gallois. Slicing communicating automata specifications: Polynomial algorithms for model reduction. *Formal Aspects of Computing (FACJ)*, 2007. Under review (submitted Jan'07).

[LGP07a] Sébastien Labbé, Jean-Pierre Gallois, and Marc Pouzet. Efficient reduction of communicating automata specifications using slicing techniques. *Journal of Software (JSW)*, 2007. Under review (submitted Sept'07).

[LGP07b] Sébastien Labbé, Jean-Pierre Gallois, and Marc Pouzet. Slicing communicating automata specifications for efficient model reduction. In *Australian Software Engineering Conference (ASWEC)*, pages 191–200, 2007.

[LL06] Sébastien Labbé and Arnault Lapitre. CARVER: a slicing tool for communicating automata specifications. In *Post-proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, 2006. Under review (submitted Jan'07).

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[MGCL07] Daniel Mateus, Jean-Pierre Gallois, Jean-Paul Comet, and Pascale Le Gall. Symbolic modeling of genetic regulatory networks. *Journal of Bioinformatics and Computational Biology*, 5(2b):627–640, 2007.

[MOS01] Markus Müller-Olm and Helmut Seidl. On optimal slicing of parallel programs. In *ACM Symposium on Theory of Computing (STOC)*, pages 647–656, 2001.

[MT00] Lynette I. Millett and Tim Teitelbaum. Issues in slicing promela and its applications to model checking, protocol understanding, and simulation. *Software Tools for Technology Transfer (STTT)*, 2(4):343–349, 2000.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[NA98] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. *SIGSOFT Software Engineering Notes*, 23(6):24–34, 1998.

[Nan01] Mangala Gowri Nanda. *Slicing Concurrent Java Programs: Issues and Solutions*. PhD thesis, Indian Institute of Technology, 2001.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[NR00] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. *SIGSOFT Software Engineering Notes*, 25(5):180–190, 2000.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Software Development Environments (SDE)*, pages 177–184, 1984.

[Pho01] Phoebus – La revue de la sûreté de fonctionnement, No.17. *Conception sûre, Procédés sûrs – La démarche d'un constructeur automobile*. Société Editions Préventique, juillet 2001.

[Pie03] Jean-Yves Pierron. *Définition de critères de sélection de tests fonctionnels pour la validation des systèmes électroniques embarqués*. PhD thesis, Université Evry Val d'Essonne, France, april 2003.

[Pre30] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu Matematyków Krajów Słowiańskich*, pages 92–101, Warszawa, 1930. Annotated English version: Ryan Stansifer. Presburger's article on integer arithmetic: remarks

and translation. Technical Report TR84–639, Cornell University Computer Science Department, 1984.

[RAB+05] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, and John Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *European Symposium on Programming (ESOP)*, pages 77–93, 2005.

[Rap04] Nicolas Rapin. *Validation de spécifications à base d'automates par des techniques de dépliage et d'exécution symbolique*. PhD thesis, Université Evry Val d'Essonne, France, july 2004.

[RGLG03] Nicolas Rapin, Christophe Gaston, Arnault Lapitre, and Jean-Pierre Gallois. Behavioural unfolding of formal specifications based on communicating extended automata. In *Automated Technology for Verification and Analysis (ATVA)*, 2003.

[RHTR06] Christopher A. Rouff, Michael G. Hinchey, Walter F. Truszkowski, and James L. Rash. Experiences applying formal approaches in the development of swarm-based space exploration systems. *Software Tools for Technology Transfer (STTT)*, 8(6):587–603, 2006.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Compututing Surveys*, 18(3):277–316, 1986.

[Ryd03] Barbara G. Ryder. Programming languages and compilers II – 198:516, lecture notes on dataflow analysis, spring 2003. `http://www.cs.rutgers.edu/~ryder/516/sp03/lectures/`.

[Sar97] Vivek Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *Languages and Compilers for Parallel Computing (LCPC)*, pages 94–113, 1997.

[Tar55] Alfred Tarski. A fixed point theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[TGL06] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Compiler Construction (CC)*, pages 17–31, 2006.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[WDQ03] Ji Wang, Wei Dong, and Zhi-Chang Qi. Slicing hierarchical automata for model checking UML statecharts. *Lecture Notes in Computer Science*, 2495:435–446, 2003.

[Wei81] Mark Weiser. Program slicing. In *International Conference on Software Engineering (ICSE)*, pages 439–449, 1981.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[WL03] Alan Wassyng and Mark Lawford. Lessons learned from a successful implementation of formal methods in an industrial project. In *International Symposium on Formal Methods (FM)*, pages 133–153, 2003.

[XQZ+05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[ZCU96] Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima. Static slicing of concurrent object-oriented programs. In *Computer Software and Applications Conference (COMPSAC)*, pages 312–320, 1996.

[ZRI+00] Marc Zimmerman, Mario Rodriguez, Benjamin Ingram, Masafumi Katahira, Maxime de Villepin, and Nancy Leveson. Making formal methods practical. In *Digital Avionics Systems Conference (DASC)*, pages 1.B.2.1–1.B.2.8, 2000.

Index

G

I

J

L

M

N