

École Normale Supérieure
de Cachan

Master Parisien de Recherche
en Informatique
Année : 2004-2005

Institut Recherche en Informatique de Toulouse

Analyse Statique d'Acteurs par Interprétation Abstraite

PIERRE-LOÏC GAROCHE

14 Septembre 2005

Responsables du Stage : MARC PANTEL et XAVIER THIRIOUX



Table des matières

Introduction	1
1 État de l'Art	3
1.1 Le modèle d'acteurs	3
1.2 L'analyse d'acteurs par typage	4
1.3 Interprétation abstraite de systèmes mobiles	5
2 CAP : un Calcul d'Acteurs Primitif	6
2.1 Syntaxe	6
2.2 Sémantique	8
2.3 Exemples	10
3 Sémantique non standard	14
3.1 Interactions partielles	14
3.2 Règle Formelle	15
3.3 Extraction de la syntaxe non standard	16
3.4 Sémantique opérationnelle	18
3.4.1 Primitives	19
3.4.2 Système de transition	21
3.5 Exemples	24
3.6 Cohérence	25
3.6.1 Traduction	25
3.6.2 Fraîcheur du marquage	27
3.6.3 Correspondance	27
3.7 Expression dans le cadre générique de [Fer05b]	31
4 Interprétation abstraite	35
4.1 Principes généraux	35
4.1.1 Interprétations abstraites construites sur des correspondances de Galois	35
4.1.2 Interprétations abstraites sans correspondances de Galois	36
4.2 Interprétation abstraite de systèmes mobiles	36
4.3 Opérations sur les domaines	37

4.3.1	Domaine réduit	38
4.3.2	Produit cartésien	38
5	Sémantique abstraite	39
5.1	Sémantique collectrice	39
5.2	L'expression des comportements dynamiques	40
5.3	Domaines pour le flot de contrôle	41
5.3.1	Domaine générique	41
5.3.2	Graphe d'égalité et d'inégalité	50
5.3.3	Forme des marqueurs	57
5.3.4	Relations globales entre variables et marqueurs	61
5.3.5	Produit réduit	64
5.4	Domaines pour le dénombrement	65
5.4.1	Domaine générique	65
5.4.2	Intervalles	68
5.4.3	Contraintes numériques globales	69
5.4.4	Produit réduit	70
6	PAC-SA	72
7	Propriétés	75
7.1	Linéarité	75
7.2	File d'attente bornée	76
7.3	Comportements inaccessibles	76
7.4	Messages orphelins	77
	Conclusion	78

Introduction

Avec le développement des réseaux de communication et des programmes exploitant ces réseaux, s'est développée la modélisation de la topologie des programmes répartis sur ces réseaux. Les calculs de processus permettent de décrire une communauté d'agents concurrents coopérant par l'envoi de messages. Les agents décrits par ces calculs peuvent ainsi créer de nouveaux agents, se communiquer les informations qu'ils ont sur d'autres agents et modifier ainsi dynamiquement la topologie des interactions des agents entre eux.

Les calculs de processus sont utilisés dans des domaines bien distincts. Ils peuvent représenter des problématiques de réseaux, des protocoles cryptographiques ou encore des systèmes biologiques. Le modèle des acteurs proposé par HEWITT et AGHA est proche des préoccupations des calculs de processus.

L'analyse statique appartient au domaine de la vérification de systèmes. Parmi les techniques existantes, l'analyse statique englobe les méthodes qui vérifient un système ou un programme en analysant le source ou le terme qui décrit le programme, le système, sans exécuter ce dernier avec sa sémantique usuelle. L'interprétation abstraite introduite par COUSOT et COUSOT [CC77] est l'une de ces techniques d'analyse statique. Elle permet de sur-approximer de façon correcte et décidable les sémantiques par l'expression dans des ensembles, munis d'un pré-ordre, de propriétés de ces sémantiques.

L'objectif principal des travaux décrits dans ce mémoire était d'adapter des outils d'analyse statique de systèmes mobiles basés sur les concepts de l'interprétation abstraite au modèle des acteurs; et ce, pour observer les mêmes propriétés que celles obtenues par d'autres méthodes mais avec un outil plus générique plus précis et plus aisément adaptable aux propriétés à observer.

Dans une première partie, nous introduisons le modèle des acteurs, les analyses qui existent sur ce modèle et enfin le cadre particulier de l'interprétation abstraite de systèmes mobiles, introduit par JÉRÔME FERET, dans lequel nous travaillerons. Ensuite, nous présenterons la syntaxe et la sémantique du calcul considéré. Dans une troisième partie, nous traduirons ce langage dans un langage "non standard" sur lequel nous ferons nos analyses. Nous donnerons ensuite quelques notions d'interprétation abstraite puis détaillerons, dans une cinquième partie les domaines abstraits utilisés dans

notre analyse pour représenter les propriétés d'intérêts ainsi que la sémantique opérationnelle permettant d'utiliser ces domaines. La sixième partie sera consacrée aux propriétés observées. Elle sera suivie d'une brève description de l'analyseur réalisé pendant le stage. Nous concluons ensuite par un récapitulatif de notre apport et par les travaux futurs que nous envisageons de mener dans ce contexte.

Chapitre 1

État de l'Art

1.1 Le modèle d'acteurs

Parmi les calculs de processus, nous nous intéressons plus particulièrement au modèle des acteurs proposé par HEWITT puis développé par AGHA [AMST92, HBS73, AH87]. Ce modèle est basé sur un réseau d'entités autonomes et coopératives appelées *acteurs*, qui contiennent des données et des programmes et communiquent *via* un protocole asynchrone point-à-point. À chaque acteur est associée une file d'attente dans laquelle les messages à destination de cet acteur sont accumulés. Lorsque l'acteur veut agir, il prend dans la file un message qu'il peut traiter. Un acteur est déterminé par deux éléments, d'une part son *adresse* (nous dirons aussi son nom) et d'autre part son *comportement*. Son comportement détermine les messages qu'il peut traiter et comment il le fait. Lorsqu'un acteur traite un message, il peut réaliser les actions suivantes :

- il envoie un nombre fini de messages aux acteurs dont il a connaissance (dont il connaît l'adresse) ;
- il crée un nombre fini de nouveaux acteurs ;
- il change de comportement.

Ainsi pendant l'évolution d'un système, un acteur peut prendre en compte plus ou moins de messages. La succession des comportements dépend de l'ordre de traitement des messages, aucune hypothèse n'étant faite sur cet ordre. Nous supposons que le support de communication est sûr, nous faisons une hypothèse d'équité faible formulée comme suit : “un message, qui peut être traité une infinité de fois, sera traité”.

Dans ce modèle, le changement de comportement peut aussi bien servir à changer les connaissances d'un acteur qu'à effectuer des changements plus radicaux. Pour illustrer ceci, AGHA dit dans [Agh92] qu'un acteur représentant un compte bancaire peut très bien devenir un livreur de pizza après la prise en compte d'un message. Il ne répondra alors plus aux mêmes requêtes.

Ce modèle, connu aussi sous le nom d'objets concurrents avec un comportement non uniforme, nous donne un cadre naturel pour décrire et implémenter des systèmes distribués.

Le langage étudié ici, CAP, introduit par COLAÇO *et al.* [CPS96] et décrit dans le chapitre suivant s'inspire du modèle des acteurs d'AGHA mais est plus souple et plus permissif que ne l'est son prédécesseur puisqu'il est possible dans CAP de communiquer des comportements.

Aussi, un acteur peut, lorsqu'il traite un certain message, se suicider, c.-à.-d. prendre un comportement vide, et envoyer un message contenant son adresse et son comportement à un autre acteur. Plus tard, lorsque l'acteur destinataire du message le traitera, le premier acteur pourra être réinséré dans le système avec le même comportement ou un autre. Le comportement initial pourra aussi être associé à un ou plusieurs autres acteurs.

1.2 L'analyse d'acteurs par typage

L'analyse de calculs de processus par typage a été utilisée de nombreuses fois dans la littérature. Nous citerons par exemple les travaux de TOKORO et VASCONCELOS [VT93], DAL ZILIO [Dal99], RAVARA et VASCONCELOS [RV97], KOBAYASHI et YONEZAWA [KNY95], FOURNET [FGL⁺96], PIERCE et SANGIORGI [PS93], et au sein de l'équipe qui m'accueille.

Nous nous intéressons ici plus particulièrement à l'analyse d'acteurs. L'analyse d'un calcul de processus par typage consiste à exprimer la propriété que nous voulons vérifier par un système de type. En simplifiant, nous dirons qu'un sous-terme est bien typé s'il réalise la propriété. Un système de typage défini par induction sur la syntaxe du langage permet alors de déterminer le type global du terme analysé. Plusieurs analyses par typage ont été réalisées sur les langages d'acteurs [CPS97a, CPS97b, CPS98, CPDS99, CPS00], qui ont un comportement non uniforme. Les types utilisés sont alors des ensembles ou des multi-ensembles de contraintes sur les variables des sous-termes. L'analyse d'un terme du langage consiste donc à calculer les ensembles (ou le multi-ensemble) des contraintes puis à les résoudre pour calculer l'ensemble de propriétés associées à chaque nom du terme.

Les analyses existantes permettent de détecter la linéarité d'un terme ou la présence de certains messages orphelins. Ces notions seront détaillées dans le chapitre 7. L'inconvénient principal de cette approche est que pour chaque nouvelle propriété, il faut redéfinir un typage complexe permettant de la représenter. Il faut aussi définir un nouveau système d'inférence des ces types et surtout prouver la propriété de correction basée sur la continuité (*subject reduction*).

1.3 Interprétation abstraite de systèmes mobiles

Dans ses travaux de thèse [Fer05b], JÉRÔME FERET, qui continuait les travaux d'ARNAUD VENET [Ven98] avant lui, a défini un cadre générique permettant d'exprimer et d'analyser quelques calculs de processus de premier ordre en utilisant les techniques de l'interprétation abstraites. Dans un premier temps, le terme du calcul est traduit dans un méta-langage précisant l'historique de création de chaque processus. Puis dans une seconde étape, l'analyse par interprétation abstraite de ce méta-langage permet de prouver statiquement et automatiquement les propriétés des systèmes analysés.

Trois domaines sont définis dans [Fer05b], ils permettent respectivement d'exprimer le flot de contrôle, le dénombrement des processus et enfin la conjonction de ces deux propriétés en partitionnant l'analyse pour plus d'efficacité.

Chacun des domaines est muni d'une sémantique opérationnelle permettant de calculer dans le domaine abstrait les transitions qui peuvent être effectuées dans le méta-langage. Les fonctions nécessaires à l'expression de la sémantique opérationnelle sont définies de façon à converger vers le point fixe de l'analyse.

L'élément abstrait post plus point fixe est un ensemble de propriétés vérifiées par toutes les configurations possibles qu'un terme peut prendre lors de son évolution suite à la réception de messages. Cet ensemble de propriétés peut éventuellement être trop grand et donc n'apporter pas beaucoup d'informations sur le terme analysé. L'analyse est décrite et construite de façon très modulaire. Il est donc aisé de redéfinir les primitives ou de construire de nouveaux domaines.

Chapitre 2

CAP : un Calcul d'Acteurs Primitif

Ce chapitre est dédié à la présentation du langage CAP. Nous présenterons d'abord la syntaxe du calcul, puis les règles et les relations qui permettent de calculer l'évolution d'un terme. Enfin, nous donnerons quelques exemples illustrant les possibilités du calcul.

2.1 Syntaxe

CAP est un calcul de processus. Il permet donc de représenter des processus qui communiquent entre eux. Les processus sont appelés ici *acteurs*. Ils communiquent entre eux par l'envoi de *messages*. CAP est un calcul asynchrone ; lorsqu'un message est envoyé, l'émetteur du message passe dans un état *post*-envoi et peut poursuivre son exécution alors que son destinataire peut prendre en compte le message plus tard. C'est seulement lorsqu'il prendra en compte le message qu'il évoluera lui aussi à son tour. À un acteur donné est associé un *comportement*, c.-à.-d. un ensemble fini de noms de messages qu'il peut recevoir ainsi que l'évolution associée à chaque nom de message, c.-à.-d. l'action de l'acteur lors de la réception d'un tel message.

L'opérateur de réplication, classique dans les calculs de processus, est ici remplacé par un opérateur d'auto-application ζ qui permet d'exprimer le changement de comportement. Ainsi $\zeta(e, s)C$ permet de désigner dans le sous-terme C l'adresse (ou nom) de l'acteur ainsi que son comportement par, respectivement, les variables e (ego) et s (self).

Soit \mathcal{N} un ensemble dénombrable de noms d'acteur, \mathcal{V} un ensemble dénombrable de noms de variables et \mathcal{L}_m l'ensemble des noms de message.

Pour permettre de désigner aisément certaines parties d'un terme de CAP (aussi désigné dans la suite par le mot configuration), nous l'étiquetons en certains endroits, les points de programme. Soit \mathcal{L}_p l'ensemble des étiquettes de points de programme et \mathcal{L}_n l'ensemble des étiquettes de points

de variables. Dans la suite, nous désignerons par \mathcal{L} , l'union de ces deux ensembles : $\mathcal{L} = \mathcal{L}_p \cup \mathcal{L}_n$. Les points de programme désignent les messages, les installations de comportements sur un acteur ou les choix externes entre différents comportements d'un acteur. Les points de variables correspondent à la déclaration de variables à l'aide de l'opérateur nu (ν).

Dans la suite, le symbole \tilde{x} représente le vecteur (x_1, \dots, x_n) . L'opérateur ν agit comme un lieu. Il en est de même pour l'opérateur zêta (ζ) et pour les variables du message dans la description du comportement associé de l'acteur. Par exemple, dans la configuration $(\nu a^\alpha)C$ et dans le comportement $[m_i^{l_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1\dots n}]$, les occurrences de a dans C , des variables \tilde{x}_i dans $\zeta(e_i, s_i)C_i$ et de e_i et s_i dans C_i sont liés. α et l_i jouent ici le rôle d'étiquettes respectivement de points de variables et de points de programme.

Nous désignons par $\mathcal{FN}(C)$ l'ensemble des noms libres de la configuration C et par $\mathcal{FV}(C)$ l'ensemble de ses variables liées.

Seules les parties d'un terme qui peuvent servir à exprimer les composantes qui participent aux transitions ou les valeurs des variables sont étiquetées.

La syntaxe des termes est définie dans la Fig. 2.1.

$$\begin{array}{ll}
a & \in \mathcal{N} \\
e_i, s_i, x, e, s & \in \mathcal{V} \\
\tilde{x}_i & \in \mathcal{V}^* \\
m, m_i & \in \mathcal{L}_m \\
l, l_i & \in \mathcal{L}_p \\
\alpha, \alpha_i & \in \mathcal{L}_n \\
\\
C & ::= \emptyset \mid \nu a^\alpha C \\
& \quad \mid C \parallel C \\
& \quad \mid a \triangleright^l B \\
& \quad \mid a \triangleleft^l m(\tilde{P}) \\
& \quad \mid e \triangleright^l B \\
& \quad \mid e \triangleleft^l m(\tilde{P}) \\
\\
B & ::= s \quad \mid [m_i^{l_i}(\tilde{Var}) = \zeta(e_i, s_i)C_i^{i=1\dots n}] \\
\\
\tilde{P} & ::= a \quad \mid e \quad \mid B
\end{array}$$

FIG. 2.1 – La syntaxe de CAP

2.2 Sémantique

La sémantique opérationnelle de CAP est définie “à la Milner”, par une relation de transition (*cf.* Fig. 2.2 et Fig. 2.3) et par une relation de congruence (*cf.* Fig. 2.4)

La relation de transition permet d’exprimer la prise en compte d’un message par un acteur destinataire. C’est à dire le changement de comportement de l’acteur, les éventuels envois de message . . . Nous représentons une telle transition par le triplet composé du point de programme de l’acteur, de celui du comportement associé qui prend en charge le message et de celui du message reçu.

La relation de congruence ainsi que les règles structurales permettent, quant à elles, de telles interactions en réarrangeant la configuration pour les mettre en évidence et décrivent comment évolue le reste du terme lors d’une transition.

Une transition est possible lorsque sont présents dans le terme un acteur et un message pour cet acteur. De plus, l’acteur doit être capable de prendre en compte un tel message. Le résultat est alors obtenu, d’une part, en retirant du terme à la fois l’acteur et son ensemble de comportements ainsi que le message, et d’autre part, en ajoutant dans le terme la continuation associée au comportement activé avec les variables liées.

$$\frac{T = [m_i^{l_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1, \dots, n}] \quad \begin{cases} m = m_k, \\ \text{length}(\tilde{y}) = \text{length}(\tilde{x}_k), \\ k \in [1, \dots, n] \end{cases}}{a \triangleright^l T \parallel a \triangleleft^{l'} m(\tilde{y}) \xrightarrow{\text{comm}(l, l_k, l')} C_k[e_k \leftarrow a, s_k \leftarrow l, \tilde{x}_k \leftarrow \tilde{y}]} \text{COMM}}$$

FIG. 2.2 – Relation de transition

$$\frac{D \equiv C \quad C \longrightarrow C' \quad C' \equiv D'}{D \longrightarrow D'} \text{STRUCT} \quad \frac{C \longrightarrow C'}{C \parallel D \longrightarrow C' \parallel D} \text{PAR}$$

$$\frac{C \longrightarrow C'}{\nu x C \longrightarrow \nu x C'} \text{RES}$$

FIG. 2.3 – Règles structurelles

C	\equiv	D	$C \alpha\text{-conv. avec } D$	(α -conversion)
$C \parallel \emptyset$	\equiv	C		(inaction)
$C \parallel D$	\equiv	$D \parallel C$		(commutativité)
$(C \parallel D) \parallel E$	\equiv	$C \parallel (D \parallel E)$		(associativité)
$T \triangleright T_1$	\equiv	$T \triangleright T_2$	if $T_1 \equiv_C T_2$	(équivalence de comportements)
$(\nu a)\emptyset$	\equiv	\emptyset		(simplification)
$(\nu a)(\nu b)C$	\equiv	$(\nu b)(\nu a)C$	si $a \neq b$	(échange)
$(\nu a)C \parallel D$	\equiv	$(\nu a)(C \parallel D)$	si $a \notin \mathcal{FN}(D)$	(extrusion)
$[m_i(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1, \dots, n}]$	\equiv_C	$[m_{\pi(i)}(\tilde{x}_{\pi i}) = \zeta(e_{\pi i}, s_{\pi i})C_{\pi i}^{i=1 \dots n}]$		(permutation de comportements)
		avec π une permutation		

FIG. 2.4 – Relation de congruence

2.3 Exemples

Nous donnons ici quelques exemples pour illustrer certains comportements typiques de termes en CAP ainsi que les possibilités du calcul comme les comportements non uniformes.

Exemple 2.3.1 *Dans l'exemple décrit Fig. 2.5, nous avons un acteur, nommé a qui peut recevoir un message m . Dans la configuration initiale, nous trouvons aux points de programme 7 et 8, deux messages de ce type (d'étiquettes m et adressés à l'acteur a). Quand a reçoit un de ces deux messages, il introduit dans le système un acteur appelé b avec pour ensemble de comportements associé, l'ensemble désigné par le point de programme 3. Il émet aussi en parallèle un message n à l'intention de ce nouvel acteur avec comme argument (du message) le nom du premier acteur (i.e. a).*

Après cette transition, nous sommes dans une configuration dans laquelle est présent un unique acteur, b , l'acteur a ayant disparu pendant la première transition, ainsi que deux messages, l'un pour a et l'autre pour b . L'acteur b peut alors prendre en compte le message $n(a)$ et lancer l'acteur a avec comme ensemble de comportements son ensemble de comportements initial (i.e. 1).

Nous retrouvons donc l'état initial avec un message $a \triangleleft m(b)$ en moins. Dans ce simple exemple, les variables autres que s ne contiennent que des noms d'acteurs. Mais un acteur avec un certain comportement disparaît puis réapparaît deux transitions plus tard.

Exemple 2.3.2 *Dans le petit système décrit Fig. 2.6, nous avons deux acteurs. Le premier est appelé a et peut prendre en compte deux types de messages : m et $send$. Le second est appelé b et peut prendre en compte uniquement les messages de type beh . Il y a aussi deux messages dans la configuration initiale. L'un est étiqueté $send$ et est à l'intention de a , l'autre est étiqueté m et est envoyé à b .*

Dans la configuration initiale, il existe seulement une interaction possible. L'acteur a peut recevoir le message $send$. Le message m est orphelin (cf. section 7.4) : il est dans la configuration mais ne peut être pris en compte pour l'instant.

Après une transition entre l'acteur a et le message $send$, le message beh avec comme argument l'ensemble des comportements de l'acteur a est envoyé à l'acteur b . L'acteur b peut alors recevoir ce message. Dans la continuation associée à la réception du message beh pour b , un acteur b est inséré dans le système avec comme ensemble de comportements celui qu'avait l'acteur a au début. Le nouvel acteur b est maintenant capable de prendre en compte le message m .

Cet exemple montre comment envoyer un ensemble de comportements d'un acteur à un autre.

Exemple 2.3.3 *Le système mobile décrit Fig.2.7 est non linéaire (cf. 7.1), mais ce ne peut pas être aisément vu. L'acteur a peut prendre en compte le message m. Lorsqu'il effectue une telle transition, il se réplique et envoie son ensemble de comportements et son nom à l'acteur dont le nom était l'argument du message m. Le second acteur, appelé b, lorsqu'il reçoit alors le message, se réplique et émet dans le système un acteur avec les paramètres qu'il a reçus dans le message. Dans cet exemple, nous pouvons voir qu'après deux transitions, il y a dans la configuration, deux acteurs appelés a avec le même ensemble de comportements 1. Nous pouvons noter que le comportement associée à l'acteur a est équivalent à la réplification gardée du Π -calcul. Lorsque cet acteur reçoit un message m, il se réplique et émet le message actor à l'acteur b.*

Exemple 2.3.4 *Un buffer à une place peut être représenté en CAP comme le montre la Fig. 2.8. L'acteur a possède deux états, deux ensembles de comportements, un comportement qui correspond au buffer vide dans lequel il peut recevoir un message put, et un comportement qui correspond au buffer "plein", dans lequel il accepte un message get qui renvoie la valeur en "vidant" le buffer. Dans l'état initial, l'acteur est associé à l'ensemble de comportements correspondant au buffer vide.*

Exemple 2.3.5 *Le système décrit Fig. 2.9 est notre exemple le plus complexe. Dans le système, nous avons un serveur sur l'acteur a et un serveur de duplication sur l'acteur b. Lorsque le serveur reçoit le message reify, il envoie son nom et son ensemble de comportements au serveur de duplication et meurt. Le serveur de duplication introduit alors deux nouveaux acteurs, sur les nouveaux noms a et b (définis par les points de programme 5 et 6), auxquels il associe le comportement du serveur a précédent. Il introduit aussi un serveur auxiliaire sur l'ancien nom a. Lorsque ce serveur reçoit un message, il crée un nouvel acteur d et envoie le message à chacun des deux serveurs répliqués. Chacun des deux serveurs répliqués renvoie la réponse à l'acteur d qui transmet la première réponse et détruit la seconde avant de mourir.*

$$\begin{aligned}
Serv &= [m^5(\tilde{x}, c) = \zeta(e, s)(e \triangleright^6 s \parallel \dots \parallel c \triangleleft^7 reply(\tilde{x})), \\
&\quad reify^8(c) = \zeta(e, s)(c \triangleleft^9 state_Serv(s, e)) \\
&\quad] \\
Dup_Serv &= [state_Serv^{10}(self, ego) = \zeta(e, s), \nu a^\delta, b^\theta \\
&\quad (a \triangleright^{11} self \parallel b \triangleright^{12} self \parallel ego \triangleright^{13} Aux(a, b) \parallel e \triangleright^{14} s) \\
&\quad] \\
Aux(a, b) &= [m^{15}(\tilde{x}, c) = \zeta(e, s), \nu d^t \\
&\quad (a \triangleleft^{16} m(\tilde{x}, d) \parallel b \triangleleft^{17} m(\tilde{x}, d) \parallel d \triangleright^{18} Join(c) \parallel e \triangleright^{19} s), \\
&\quad reify^{20}(c) = \zeta(e, s) \\
&\quad (c \triangleleft^{21} state_Serv(s, e) \parallel a \triangleleft^{22} reify(c) \parallel b \triangleleft^{23} reify(c)) \\
&\quad] \\
Join(c) &= [reply^{24}(\tilde{y}) = \zeta(e, s)(c \triangleleft^{25} reply(\tilde{y}) \parallel e \triangleright^{26} [reply^{27}(\tilde{y}) = \emptyset])] \\
&\quad \nu a^\alpha, b^\beta, c^\gamma, a \triangleright^1 Serv \parallel a \triangleleft^2 reify(b) \parallel a \triangleleft^3 m(\tilde{x}, c) \parallel b \triangleright^4 Dup_Serv
\end{aligned}$$

FIG. 2.9 – Réflexivité

Chapitre 3

Sémantique non standard

Dans ses travaux sur les systèmes mobiles [Fer05b, Fer05a], JÉRÔME FERET a défini pour plusieurs calculs de processus un cadre générique dans lequel il les a exprimé pour pouvoir ensuite analyser des termes exprimés dans ces calculs. Dans un tel cadre, la méta-syntaxe d'un langage est définie par un ensemble d'interactions partielles qui peuvent être produites ou calculées par les processus, un ensemble de règles de transition qui définissent quand et comment les interactions partielles interagissent ensemble et enfin une fonction d'extraction qui permet de passer de la syntaxe d'un langage vers cette méta-syntaxe. Le but d'une telle syntaxe et de la sémantique qui lui est associée est de garder trace dans les processus et dans leurs variables de l'historique des transitions qui ont menées à leur création. Ainsi, chaque processus possède un *marqueur identité* qui caractérise cet historique. De même, chaque valeur associée à une variable dans l'environnement de chaque processus est associée à un marqueur qui précise quel historique a conduit à la création de la valeur.

Nous nous intéressons, dans ce chapitre, à une telle traduction pour le langage CAP. Nous allons donc dans une première partie, définir des primitives qui vont nous permettre de décrire un système mobile qui simule CAP. Ensuite, nous décrirons comment utiliser ces primitives et nous montreront la cohérence d'une telle traduction. Nous concluons ce chapitre en décrivant les différentes approches que nous avons eues pour l'expression de ce calcul dans le cadre du travail de FERET.

3.1 Interactions partielles

Les interactions partielles sont les composantes qui interagissent ensemble lors d'une transition. Dans CAP, les interactions partielles doivent donc représenter les acteurs, un comportement particulier ou encore l'envoi d'un message.

Nous définissons donc un ensemble d'interactions partielles \mathcal{A} et leur

arité comme suit :

$$\mathcal{A} = \{actor\} \cup \{behavior_n \mid n \in \mathbb{N}\} \cup \{message_n \mid n \in \mathbb{N}\}$$

$$Ari = \begin{cases} actor & \mapsto (2, 0) \\ behavior_n & \mapsto (2, n + 2) \\ message_n & \mapsto (n + 2, 0) \end{cases}$$

où $Ari(x)$ est le couple formé du nombre d'arguments et du nombre de variables de l'interaction partielle x .

L'interaction partielle *actor* dénote un processus représentant un acteur. Elle admet deux paramètres : son nom et son ensemble de comportements. Elle n'est paramétrée par aucune variable ni ne lie aucun variable.

L'interaction partielle *behavior_n* dénote un processus représentant le comportement particulier d'un acteur. C'est une définition, c.-à-d. un processus particulier qui reste présent dans le système (*cf.* codage du Join calcul dans [Fer05b, Chap. 5]). Elle admet un unique paramètre, le type (l'étiquette) des messages qu'elle prend en compte. Les n variables du message qu'elle peut prendre en compte sont autant de variables liées par l'interaction.

La dernière interaction partielle *message_n* représente un processus message qui est envoyé à une adresse (un acteur) particulier. Elle a donc $n + 2$ paramètres : l'adresse du destinataire, le nom du message et les n variables du message.

3.2 Règle Formelle

Nous décrivons dans cette section les règles formelles qui définissent comment interagissent les interactions partielles. Dans le cas de CAP, une unique règle permet de définir l'interaction entre un acteur, un des ses comportements et un message. Soit *com_n* une telle règle. La Figure 3.1 décrit l'action de la règle sur les interactions partielles pour la communication d'un message avec n variables à un acteur. Elle nécessite trois processus, le premier doit être associé à un interaction partielle de comportement *behavior_n*, le second à une interaction partielle d'acteur *actor*, et le troisième à une interaction partielle de message *message_n* envoyé à l'acteur.

Dans la suite, le i -ème paramètre et la j -ème variable liée de la k -ème interaction partielle, ainsi que l'identité du k -ème processus sont respectivement définis par X_i^k , Y_j^k et I^k . L'identité d'un processus permet de différencier les instances récursives du même processus. Nous définissons l'isomorphisme *behavior_set* qui associe à un couple $(p, m) \in \mathcal{L}_p \times \mathcal{M}$ où p est un point de programme de comportement, le couple (p', m) où p' désigne le point de programme où le comportement désigné par p a été défini syntaxiquement. Par exemple, dans le terme, $\nu^\alpha a, a \triangleright^1 [m^2() = \zeta(e, s)C]$, la fonction *behavior_set* appliquée au couple $(2, m)$ renverra le couple $(1, m)$.

La transition a lieu ssi :

1. le nom de l'acteur (X_1^2) et le destinataire du message (X_1^3) encodent la même valeur ;
2. le point de programme du processus de comportement (première composante du couple désigné par I_1) appartient à l'ensemble des points de programmes désigné par la variable de comportement de l'acteur (X_2^2). Une telle condition est vérifiée en utilisant la primitive *behavior_set* ;
3. enfin l'étiquette du message que peut traiter le comportement (X_1^1) et l'étiquette du message traité (X_2^3) sont égales.

L'ensemble *compatibility* représente ces contraintes.

Lors de la communication, les variables définies par l'opérateur ζ du comportement sont associées au nom et à l'ensemble de comportements de l'acteur. De la même façon, la i -ème variable liée du comportement est associée avec le $(i + 2)$ -ème paramètre du processus de type message. Ce passage de valeurs est défini par l'ensemble *v_passing*. Après la transition, les processus acteurs et messages qui sont intervenus dans la communication sont supprimés du système. Les processus désignant des comportements restent pour pouvoir être réutilisés par la suite.

$$com_n = (3, components, compatibility, v_passing)$$

où

$$\begin{aligned}
1. \text{ components} &= \begin{cases} 1 \mapsto actor, \\ 2 \mapsto behavior_n, \\ 3 \mapsto message_n \end{cases} \\
2. \text{ compatibility} &= \begin{cases} X_1^2 = X_1^3; \\ behavior_set(I^1) = X_2^2; \\ X_1^2 = X_2^3; \end{cases} \\
3. \text{ v_passing} &= \begin{cases} Y_1^1 \leftarrow X_1^2; \\ Y_2^1 \leftarrow X_2^2; \\ Y_{i+2}^1 \leftarrow X_{i+2}^3, \forall i \in \llbracket 1; n \rrbracket; \end{cases}
\end{aligned}$$

FIG. 3.1 – La règle de communication

3.3 Extraction de la syntaxe non standard

Nous définissons maintenant la fonction d'extraction qui prend un terme de CAP décrivant un système mobile dans la syntaxe standard et en extrait un terme correspondant dans la syntaxe non standard.

Les points de programme sont les envois de messages, les installations d'acteurs et les comportements d'acteurs. Nous associons à chaque point de

programme $l \in \mathcal{L}_p$, d'après la syntaxe du sous-terme associé, un ensemble d'interactions partielles ainsi qu'une interface, *i.e.* l'ensemble des variables de l'environnement du processus, ces variables sont soit contenus dans le sous-terme, soit introduites implicitement.

Une interaction partielle pi est décrite par le quadruplet $(s, (parameter_i), (bound_i), continuation)$ où $s \in \mathcal{A}$ est un nom d'interaction partielle, $(m, n) = Ari(s)$ son arité, $(parameter_i) \in \mathcal{V}^m$ sa séquence finie de variables (X_i) , qui représente ses paramètres, $(bound_i) \in \mathcal{V}^n$ sa séquence finie de variables distinctes (Y_i) , qui représente les variables liées dans les continuations, et $continuation \in \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$ sa continuation.

La continuation d'un processus est le sous-terme qui correspond à l'action effectuée par le processus lorsqu'il effectue une transition. Une continuation est un ensemble de couple (p, E_s) où p désigne un point de programme et E_s un environnement statique, c.-à-d. une fonction partielle qui associe une valeur à une variable. Cet ensemble permet de modéliser l'introduction de nouvelles variables.

Nous définissons maintenant les interactions partielles et les interfaces associées à chaque point de programme :

- l'étiquette l d'un point de programme $a \triangleright^l [m_i^{l_i}(\widetilde{Var}) = \zeta(e_i, s_i)C_i]$ est associée à l'interface $\{a, self\}$, où $self$ est une variable spéciale insérée pour représenter l'ensemble des comportement de l'acteur $([...])$, et à l'ensemble suivant d'interactions partielles :

$$\{(actor, [a, self], \emptyset, \emptyset)\}$$

Nous ajoutons les points de programme des comportements définis syntaxiquement dans $[...]$ à l'ensemble $actor_behavior(l)$. Nous ajoutons également les liens entre le point de programme l et les points de programme l_i dans la fonction $behavior_set$.

- l'étiquette l d'un point de programme $a \triangleright^l x$ est associée à l'interface $\{a, x\}$ et à l'ensemble suivant d'interactions partielles :

$$\{(actor, [a, x], \emptyset, \emptyset)\}$$

- l'étiquette l d'un point de programme $a \triangleleft^l m(\tilde{P})$ est associée à l'interface $\{a\} \cup \mathcal{FV}(\tilde{P})$ et à l'ensemble suivant d'interactions partielles :

$$\{(message_n, [a; m; \tilde{P}], \emptyset, \emptyset)\}$$

- l'étiquette l_i d'un point de programme correspondant au comportement particulier d'un acteur *i.e.* $m_i^{l_i}(\widetilde{Var}) = \zeta(e_i, s_i)C_i$ est associée à l'interface $\{e_i, s_i\} \cup \mathcal{FV}(C_i)$ et à l'ensemble suivant d'interactions partielles :

$$\{(behavior_n, [m_i], [e_i, s_i, \widetilde{Var}], \beta(C_i, \emptyset))\}$$

$$\begin{aligned}
\beta((\nu a^\alpha)C, E_s) &= \beta(C, E_s[a \mapsto \alpha]) \\
\beta(\emptyset, E_s) &= \emptyset \\
\beta(C_1 \parallel C_2, E_s) &= \beta(C_1, E_s) \cup \beta(C_2, E_s) \\
\beta(a \triangleright^l [m_i^{l_i}(\tilde{x}_i) &= \{(l, E_s [self \mapsto l])\} \\
\zeta(e_i, s_i)C_i^{i=1, \dots, n}], E_s) &= \cup \bigcup_{i=1, \dots, n} \{(l_i, E_s[])\} \\
\beta(a \triangleright^l b, E_s) &= \{(l, E_s)\} \\
\beta(a \triangleleft^l m(\tilde{P}), E_s) &= \{(l, E_s)\}
\end{aligned}$$

FIG. 3.2 – La fonction d'extraction β

où β est la fonction d'extraction de la syntaxe de CAP, définie par induction sur la syntaxe standard des continuations syntaxiques dans la figure 3.2.

L'ensemble des états initiaux d'un terme \mathcal{S} est décrit par l'ensemble des continuations dans $\wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$:

$$init_s = \beta(\mathcal{S}, \emptyset)$$

3.4 Sémantique opérationnelle

Nous décrivons ici la sémantique qui dirige le calcul d'une transition du système d'une configuration à la suivante. Une configuration est un ensemble de processus.

Un processus t est un triplet composé d'un point de programme, d'un marqueur et d'un environnement. Plus formellement, nous le définissons par :

$$t = (p, id, E) \in \mathcal{L}_p \times \mathcal{M} \times (\mathcal{V} \mapsto (\mathcal{L} \times \mathcal{M})).$$

L'interface, définie au-dessus (cf. section 3.3), est un sous-ensemble de \mathcal{V} et représente l'ensemble des variables liées dans l'environnement associé au processus.

Nous définissons maintenant quelques primitives qui vont nous permettre de calculer une transition du système, c'est à dire, la prise en charge d'un message par un acteur. Nous rappelons que la seule transition possible utilise la règle formelle com_n (cf. section 3.2).

Remarque 3.4.1 (notation) *Dans la suite, nous manipulons des séquences d'éléments et des n -uplets d'éléments. Sauf indication contraire, $(element_i)_i$ désigne la séquence des éléments $element_i$ pour tout i valable dans le contexte,*

alors que $(element_i)$ désigne la séquence composée de l'unique élément $element_i$ et enfin $element_i$ désigne l'élément seul, sans séquence.

3.4.1 Primitives

Actions exhibées Nous associons à chaque point de programme l'ensemble des interactions partielles que ce point de programme peut produire. Dans notre système, chaque point de programme peut produire une seule interaction partielle. Le processus $t = (p, id, E)$ exhibe l'interaction partielle pi ssi pi appartient à l'ensemble des interactions partielles associées au point de programme p .

Plus formellement :

Définition 3.4.2 (actions exhibées) Soit $t = (p, id, E)$ un processus et $pi = (s, (parameter_i), (bound_i), continuation)$ une interaction partielle. Nous dirons que le processus t exhibe l'interaction partielle pi et nous noterons $exhibit(t, pi)$ ssi $pi \in interaction(p)$.

Remarque 3.4.3 La construction de la fonction *interaction* assure qu'une unique interaction partielle est exhibée à chaque point de programme. Pour simplifier les notations, nous noterons dans la suite $pi = interaction(p)$ au lieu de $pi \in interaction(p)$.

Synchronisations globales Cette fonction permet de vérifier que les conditions de synchronisation de la règle entre les variables des environnements associés aux processus et leur marqueur identité sont satisfaites.

Définition 3.4.4 (synchronisations globales) Soit $(t^k)_{1 \leq k \leq 3} = (p^k, id^k, E^k)_{1 \leq k \leq 3}$ un triplet de processus, $(param_l^k)_{k,l}$ un triplet de séquences de paramètres et *compatibility* un ensemble de contraintes de synchronisations. La relation *sync* est définie comme suit :

$$sync((t^k)_k, (param_l^k)_k, compatibility) \triangleq \forall a, b \in compatibility, \sigma(a) = \sigma(b),$$

où $\sigma : \begin{cases} X_l^k & \mapsto E^k(param_l^k) \\ I^k & \mapsto (p^k, id^k) \end{cases}$

Remarque 3.4.5 La second condition de synchronisation de la règle de transition est la suivante :

$$behavior_set(I^1) = X_2^2$$

Nous définissons donc $\sigma(behavior_set(I^i)) = behavior_set(\sigma(I^i))$.

Calcul des marqueurs À chaque pas de transition, nous devons calculer un nouveau marqueur pour les processus créés. Dans notre cadre, ce marqueur peut être différent pour les différents processus insérés lors de la transition. Ce marqueur sera ensuite utilisé pour différencier les instances récursives des processus associé au même point de programme.

Nous pouvons maintenant définir le calcul des marqueurs :

Définition 3.4.6 (calcul des marqueurs) Soit $(p^i, id^i, E^i)_{1 \leq i \leq 3}$ un triplet de processus. Le marqueur $marker((p^i, id^i, E^i)_{1 \leq i \leq 3})$ est défini comme suit :

$$marker((p^i, id^i, E^i)_{1 \leq i \leq 3}) \triangleq ((p^1, p^2, p^3), id^1, id^2, id^3)$$

C'est l'arbre des étiquettes de transition qui ont conduit à la création de ce processus. Les trois fils de la racine de l'arbre représentent les historiques de la création des trois processus qui ont participé à la transition qui a créé ce processus. Le deuxième fils permet d'identifier aussi le point de programme de l'acteur qui a effectué la transition.

Suppression de processus Lorsque nous calculons une interaction, certains processus sont retirés du système. Dans le cas de CAP, lors d'une transition faisant intervenir un acteur, un comportement et un message, il nous faut supprimer les processus acteur et message, le processus de comportement restant dans le système après la transition.

Définition 3.4.7 (suppression de processus) Soit $(t^k)_{1 \leq k \leq 3}$ un triplet de processus. Soit C une configuration. La primitive *remove* est définie comme suit :

$$remove\left((t^k)_{1 \leq k \leq 3}\right) \triangleq \{(t^k)_{2 \leq k \leq 3}\}$$

Partage d'environnements Nous introduisons ici la primitive *v_passing* qui calcule le passage de paramètres, aussi appelé dans la suite passage de valeurs, associé à l'ensemble *communications* de la règle formelle utilisée pour la transition. Elle prend comme paramètres l'indice du i -ème environnement qu'elle va calculer, le n-uplet des processus concernés par la règle, les séquences de variables et de paramètres ainsi que l'ensemble *communications* des correspondances entre les variables et les paramètres des différents processus.

Définition 3.4.8 (partage d'environnement) Soit $(t^k)_{1 \leq k \leq 3} = (p^k, id^k, E^k)_{1 \leq k \leq 3}$ un triplet de processus. Soit $(bd_l)_l$ une séquence de variables, et $(param_l^k)_{k,l}$ un triplet de séquences de paramètres. Soit *communications* une fonction partielle de \mathcal{V}_f^Y dans $\mathcal{V}_f^X \cup \mathcal{V}_f^I$. Nous définissons l'environnement *vpassing* $(i, (t^k), (bd_l), (param_l^k), communications)$ par :

$$E^i[bd_j \mapsto \sigma(\text{communications}(Y_j^i))],$$

$$\text{où } \sigma = \begin{cases} X_l^k & \mapsto E^k(\text{param}_l^k) \\ I^k & \mapsto (p^k, id^k). \end{cases}$$

Démarrer les continuations Lorsqu'un processus interagit dans une règle et exhibe une interaction partielle, il est ensuite supprimé et remplacé par les processus de sa continuation. Dans chaque continuation non vide, nous avons un ou plusieurs processus statiques, *i.e.* $(p, E_s) \in \mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L})$.

Nous ajoutons alors dans la configuration un processus pour chaque processus statique de l'ensemble de continuations. Nous donnons à chaque nouveau processus le marqueur avec le nouveau *id* et mettons à jour l'environnement statique E_s des processus.

Définition 3.4.9 (mise à jour de l'environnement) Soit *id* un marqueur dans \mathcal{M} . Soit E_d un environnement sur V_d (*i.e.* $E_d \in v_d \rightarrow \mathcal{L} \times \mathcal{M}$). Soit E_s un environnement statique sur V_s (*i.e.* $E_s \in V_s \rightarrow \mathcal{L}$). Nous définissons l'environnement $update(id, E_d, E_s)$ sur $V_d \cup V_s$ comme suit :

$$update(id, E_d, E_s)(x) \triangleq \begin{cases} (E_s(x), id) & \text{si } x \in V_s \\ E_d(x) & \text{si } x \in V_d \setminus V_s \end{cases}$$

Définition 3.4.10 (démarrage des continuations) Soit $Ct \in \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$ une continuation. Soit *id* $\in \mathcal{M}$ un marqueur de processus. Soit $E \in \mathcal{V} \rightarrow \mathcal{L} \times \mathcal{M}$ un environnement. Nous définissons l'ensemble de processus $launch(ct, id, E)$ comme suit :

$$launch(Ct, id, E) \triangleq \left\{ (t, id, \overline{E}) \mid \begin{array}{l} I(t) \text{ l'interface du processus } t, \\ (t, E_s) \in Ct \end{array} \right\}$$

avec $\overline{E} = (update(id, E, E_s))|_{I(t)}$

3.4.2 Système de transition

Nous utilisons les primitives que nous avons définies au-dessus pour décrire à la fois l'état initial et la sémantique des étapes de calcul des transitions. Les configurations initiales sont obtenues en démarrant un ensemble de continuations de $init_s$ avec un marqueur vide et un environnement vide. Ainsi l'ensemble \mathcal{C}_0 des configurations initiales est défini par :

$$\mathcal{C}_0 = \{launch(continuation, \epsilon, \emptyset) \mid continuation \in init_s\}.$$

Les étapes de calcul sont décrites par une relation de réduction (*cf.* Fig 3.4). Il faut, avant tout, trouver l'interaction correcte. Cela signifie que nous devons trouver dans la configuration courante, les processus qui exhibent les interactions partielles appropriées afin de pouvoir effectuer la transition associée à la règle formelle com_n . Nous vérifions alors que leur interface satisfait les contraintes de synchronisation. Nous pouvons alors calculer l'interaction :

- nous supprimons du système les processus acteur et message qui interagissent
- nous déterminons les continuations des processus qui ont interagit et calculons les données dynamiques pour chacune des ces continuations :
 - ★ nous calculons le marqueur ;
 - ★ nous prenons en compte le passage de valeurs ;
 - ★ nous créons les nouvelles variables et leur associons la valeur adéquate ;
 - ★ enfin, nous restreignons l'environnement à l'ensemble des variables de l'interface associé au point de programme.

Soit un terme \mathcal{S} de CAP. Ce terme est décrit dans la syntaxe non standard par l'ensemble des continuations dans $\wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$:

$$init_s = \beta(\mathcal{S}, \emptyset)$$

FIG. 3.3 – États initiaux

Soit C une configuration.

Soit $\mathcal{R} = (n, components, compatibility, v_passing)$ une règle de réduction ($\mathcal{R} = com_n$).

Nous supposons qu'il existe à la fois un triplet $(t^k)_{1 \leq k \leq 3} = (p^k, id^k, E^k)_{1 \leq k \leq 3} \in C^3$ de processus distincts et un triplet $(pi^k)_{1 \leq k \leq 3} = (s^k, (parameter_l)^k, (bd_l)^k, constraints^k, continuation^k)_{1 \leq k \leq 3}$ d'interactions partielles, tels que :

1. $\forall k \in \llbracket 1; 3 \rrbracket, exhibits(t^k, pi^k)$;
2. $\forall k \in \llbracket 1; 3 \rrbracket, components(k) = s^k$;
3. $sync((t^k)_k, ((parameter_l)^k)_k, compatibility)$ ne soit pas l'élément \perp .

Alors

$$C \xrightarrow{(com_n, (\alpha^1, \alpha^2, \alpha^3))} (C \setminus removed_threads) \cup news_threads$$

avec :

- $removed_threads = remove((t^k)_{1 \leq k \leq 3})$;
- $news_threads = \bigcup_{1 \leq k \leq 3} launch(Ct^k, \bar{id}, \bar{E}^k)$,
où $\forall k \in \llbracket 1; 3 \rrbracket$:
 - ★ $Ct_k \in continuation^k$;
 - ★ $\bar{id} = marker((p^{k'}, id^{k'}, E^{k'})_{1 \leq k' \leq 3})$;
 - ★ $\bar{E}^k = vpassing(k, (t^{k'})_{1 \leq k' \leq 3}, ((bd_l)^k)_k, ((parameter_l)^k)_k, communications)$.
- $\forall k \in \llbracket 1; 3 \rrbracket, \alpha^k = (t^k, pi^k, E^k)$.

FIG. 3.4 – Règle de transition

3.5 Exemples

Le passage de comportement Nous décrivons ici comment utiliser la syntaxe et la sémantique abstraite pour calculer l'évolution d'un terme traduit de CAP. Pour un soucis de lisibilité, les variables associées aux étiquettes des messages ne sont pas présente dans les environnements des processus. Nous admettrons dans la suite que leur correspondance est vérifiée par convention, puisqu'elle est possible de façon statique.

$$\begin{array}{l} \nu a^\alpha, b^\beta, \quad a \triangleright^1 [m^2() = \zeta(e, s)(a \triangleright^3 s), \text{send}^4(x) = \zeta(e, s)(x \triangleleft^5 \text{beh}(s))] \\ \parallel \quad a \triangleleft^6 \text{send}(b) \\ \parallel \quad b \triangleright^7 [\text{beh}^8(x) = \zeta(e, s)(e \triangleright^9 x)] \\ \parallel \quad b \triangleleft^{10} m() \end{array}$$

Nous avons l'état initial¹ :

$$\begin{array}{l} (1, \epsilon, \left[\begin{array}{l} a \mapsto \alpha, \epsilon \\ \text{self} \mapsto 1, \epsilon \end{array} \right]) \quad (2, \epsilon, [a \mapsto \alpha, \epsilon]) \quad (4, \epsilon, []) \\ \\ (6, \epsilon, \left[\begin{array}{l} a \mapsto \alpha, \epsilon \\ b \mapsto \beta, \epsilon \end{array} \right]) \quad (7, \epsilon, \left[\begin{array}{l} b \mapsto \beta, \epsilon \\ \text{self} \mapsto 7, \epsilon \end{array} \right]) \quad (8, \epsilon, []) \\ \\ (10, \epsilon, [b \mapsto \beta, \epsilon]) \end{array}$$

qui devient après la transition 1, 4, 6

$$\begin{array}{l} (2, \epsilon, [a \mapsto \alpha, \epsilon]) \quad (4, \epsilon, []) \quad (5, id_1, \left[\begin{array}{l} e \mapsto \alpha, \epsilon \\ s \mapsto 1, \epsilon \\ x \mapsto \beta, \epsilon \end{array} \right]) \\ \\ (7, \epsilon, \left[\begin{array}{l} b \mapsto \beta, \epsilon \\ \text{self} \mapsto 7, \epsilon \end{array} \right]) \quad (8, \epsilon, []) \quad (10, \epsilon, [b \mapsto \beta, \epsilon]) \\ \text{avec } id_1 = (1, 4, 6), \epsilon, 1, \epsilon \end{array}$$

¹Nous pouvons noter ici l'absence de processus correspondant aux points de programme 3, 5 et 9 qui correspondent à des sous-termes. Il ne sont pas présent dans le système au début.

puis après la transition 7, 8, 5

$$(2, \epsilon, [a \mapsto \alpha, \epsilon]) \quad (4, \epsilon, []) \quad (7, \epsilon, \left[\begin{array}{l} b \mapsto \beta, \epsilon \\ self \mapsto 7, \epsilon \end{array} \right])$$

$$(8, \epsilon, []) \quad (9, id_2, \left[\begin{array}{l} e \mapsto \beta, \epsilon \\ x \mapsto 1, \epsilon \end{array} \right]) \quad (10, \epsilon, [b \mapsto \beta, \epsilon])$$

avec $id_2 = (7, 8, 5), \epsilon, 7, ((1, 2, 6), \epsilon, 1, \epsilon)$

enfin, nous terminons avec la transition 9, 2, 10

$$(2, \epsilon, [a \mapsto \alpha, \epsilon]) \quad (4, \epsilon, []) \quad (7, \epsilon, \left[\begin{array}{l} b \mapsto \beta, \epsilon \\ self \mapsto 7, \epsilon \end{array} \right])$$

$$(8, \epsilon, []) \quad (3, id_3, \left[\begin{array}{l} a \mapsto \alpha, \epsilon \\ s \mapsto 1, \epsilon \end{array} \right])$$

avec $id_3 = ((9, 2, 10), ((7, 8, 5), \epsilon, 7, ((1, 2, 6), \epsilon, 1, \epsilon)), ((7, 8, 5), \epsilon, 7, ((1, 2, 6), \epsilon, 1, \epsilon)), \epsilon)$

3.6 Cohérence

3.6.1 Traduction

Nous voulons maintenant prouver la correspondance entre la sémantique de CAP et son expression dans le méta-langage comme défini ci-dessus. Nous définissons pour cela une fonction de traduction Π qui associe à une configuration non standard bien formée C la configuration correspondante dans CAP.

Lemme 3.6.1 (bonne formation d'un terme non standard) *Soit C un terme non standard. C est donc un ensemble de triplet de la forme (p, id, E) où $p \in \mathcal{L}_p$ désigne un point de programme, $id \in \mathcal{M}$ un marqueur et $E \in \wp(\mathcal{V} \rightarrow (\mathcal{L} \times \mathcal{M}))$. Soit une fonction partielle interaction qui associe à un point de programme une interaction partielle. Soit une fonction partielle actor_behavior qui associe à un point de programme un ensemble de points de programme. Soit un ensemble dynamic_behavior de points de programme. Le terme C est bien formé ssi il vérifie les conditions suivantes :*

- $\forall (p, id, E)$, interaction(p) = (name, var, param, cont) est défini et $\forall v \in var$, $E(v)$ est défini ;
- $\forall (p, id, E)$, tq interaction(p) soit de nom actor, alors $|var| = 2$ et soit s la second variable de var. Il y a dans le système exactement un pro-

- cessus* $(p_i, \text{snd}(E(s), E_i))$ pour chaque $p_i \in \text{actor_behavior}(\text{fst}(E(s)))$.
 Le nom de l'interaction partielle exhibée par chaque p_i est behavior_n .
- $\forall (p, \text{id}, E)$, tq $\text{interaction}(p)$ soit de nom behavior_n , alors $|\text{var}| = 1$.
 - pour chaque variable associée à un point de programme de comportement, les processus associés aux comportements décrits par ce point de programme doivent être présent dans le système avec comme marqueur le même marqueur que celui de la variable.

Preuve 3.6.2 Par induction sur les termes qui peuvent être créés par des transitions à partir de l'élément initial $\beta(\mathcal{S}, \emptyset)$. \square

Pour faciliter la traduction et permettre de différencier les instances récursives de la même déclaration de variable, nous définissons la fonction auxiliaire f qui associe à chaque couple $(p, m) \in \mathcal{L} \times \mathcal{M}$ le nom p^m si p correspondant à un terme de la forme νa^p et le terme $[m_i^{l_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1, \dots, n}]$ si p correspondant à un terme de la forme $a \triangleright^p [m_i^{l_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1, \dots, n}]$. Cette fonction permet ainsi, d'une part, de marquer dans les termes la différence entre les instances récursives du même opérateur ν , et d'autre part, de remplacer les acteurs ayant un comportement dynamique par des acteurs ayant un ensemble de comportements défini syntaxiquement. La règle d' α -conversion de la relation de congruence de CAP permet ensuite de réarranger le terme.

$$\text{Soit } \{c_i \mid i \in \llbracket 1; k \rrbracket\} = \left\{ f(E(x)) \left| \begin{array}{l} (P, \text{id}, E) \in C, \\ E(x) \text{ défini,} \\ \text{et } \text{fst}(E(x)) \notin \text{dynamic_behavior} \end{array} \right. \right\}$$

est l'ensemble des noms d'acteurs utilisés dans le terme. La fonction f nous permet d'éviter l'ambiguïté entre deux instances récursives du même lieu d'adresse.

La fonction Π alors est définie par :

$$\Pi(C) = (\nu_{c_1}) \dots (\nu_{c_k})(M_1 \parallel \dots \parallel M_p \parallel A_1 \parallel \dots \parallel A_q)$$

Nous définissons C comme l'union disjointes de M , A et B : $C = M \cup A \cup B$ où M est l'ensemble des processus associés à des interactions partielles message_n , A est l'ensemble des processus associés à des interactions partielles actor et enfin B est l'ensemble des processus associés à des interactions partielles behavior_n .

Nous construisons $\{M_i\}$ et $\{A_j\}$ comme suit :

- $i \in \llbracket 1; \text{Card}(M) \rrbracket$
- M_i est le message $a \triangleleft^l(\widetilde{Var})$ correspondant à la traduction du processus $(l, \text{id}, E) \in M$. Pour ce faire, nous substituons a et chaque variable libre de \widetilde{Var} par l'image de sa valeur, dans l'environnement E , par la fonction f .

- $j \in \left[\left[1; \text{Card} \left(\left\{ \left(f(E(a)), id \right) \mid \begin{array}{l} a \text{ est le premier paramètre } pi, \\ pi \in \text{interaction}(p), \\ (p, id, E) \in A \end{array} \right\} \right) \right] \right]$
- A_j est l'acteur $a \triangleright^l [m_i^{l_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1, \dots, n}]$ composé de tous les comportements représentés par les processus $(l_i, id, E) \in B$ qui lui sont associé; c'est à dire, des processus de comportement dont le point de programme est associé, par la fonction *behavior_set*, à la valeur de la seconde variable du processus acteur et dont le marqueur est identique au marqueur de cette même variable. Plus formellement, la seconde variable du processus acteur est associée au couple (l', id') , $id = id'$ et tous les l_i sont associés à l par *behavior_set*.

Nous substituons, dans A_j , toutes les occurrences libres de a et chaque variable libre par son image, si elle existe, par la fonction qui, à x , associe $f(E(x))$. Comme le terme C est supposé bien formé, aucun processus représentant un comportement ne peut être présent dans le système sans un acteur auquel il est associé. De la même façon, lorsqu'un acteur est présent, tous les comportements de son ensemble de comportements doivent être présent dans le système.

3.6.2 Fraîcheur du marquage

La fraîcheur du marquage est assurée par la construction de la règle formelle, des interactions partielles et de la fonction d'extraction. La preuve peut être trouvée dans [Fer05b, Chap. 4].

Nous définissons la fonction $\psi : \mathcal{M} \rightarrow \mathcal{L}^*$ qui permet de simplifier les marqueurs :

$$\psi \triangleq \begin{cases} \mathcal{M} & \rightarrow \mathcal{L}^* \\ ((p_1, p_2, p_3), id_1, id_2, id_3) & \mapsto (p_1) \cdot \psi(id_1) \\ \epsilon & \mapsto \epsilon \end{cases}$$

Proposition 3.6.3 *La fonction ψ conserve la non-ambiguïté du marquage*

Preuve 3.6.4 *La syntaxe non standard est telle que seul les continuations des interactions partielles associées à des comportements sont non vide. De plus, lors des transitions, les processus de ce type sont toujours placés en première position dans le triplet des point de programme participant à la transition. La preuve du théorème de correction du marquage est donc encore valable si l'on prend systématiquement $(p_1, id', E') \in \text{consumed}(i', k') \subseteq C_{i'-1}$.* □

3.6.3 Correspondance

Pour prouver la correspondance entre la sémantique standard et non standard, nous devons pouvoir comparer deux termes de CAP qui se comportent de façon similaire.

Nous définissons pour cela la règle de réduction \rightarrow qui remplace les comportements dynamiques par des comportements statiques. Plus formellement, la règle \rightarrow réduit un terme t quelconque de CAP en un terme t' dans lequel chaque acteur présent dans le système, c.-à-d. accessible en passant uniquement à travers des opérateurs ν et \parallel et ayant un comportement dynamique, c.-à-d. de la forme $a \triangleright^l s$ est remplacé par le sous-terme $[m_i^{l_i}(\widetilde{Var}) = \zeta(e_i, s_i)C_i^{i=1, \dots, n}]$ qui correspond à la valeur de la variable s .

Une telle règle termine. Nous désignons dans la suite la réduction $\cong = (\equiv \cup \rightarrow)$. Une telle relation est confluente. Nous dirons que deux configurations A et B sont équivalentes ssi $A \cong B$. Les sémantiques standards et non standards sont en bisimulation forte comme l'exprime le théorème suivant :

Théorème 3.6.5 *Nous avons $\mathcal{S} \equiv \Pi(\mathcal{C}_0)$, et pour toute configuration non standard C et tout mot $u \in (\mathcal{L}^3)^*$ tels que $\mathcal{C}_0 \xrightarrow{u}^* C$:*

1. $\forall \lambda \in \mathcal{L}^3, C \xrightarrow{\lambda} C' \implies \Pi(C) \xrightarrow{\lambda} \Pi(C')$;
2. $\forall \lambda \in \mathcal{L}^3, \Pi(C) \xrightarrow{\lambda} P \implies \exists D, \left\{ \begin{array}{l} C \xrightarrow{\lambda} D \\ \Pi(D) \cong P \end{array} \right.$

Preuve 3.6.6 *Soit C une configuration non standard et un mot $u \in (\mathcal{L}^3)^*$ tels que $\mathcal{C}_0 \xrightarrow{u}^* C$,*

1. *soit C' une configuration non standard telle que $C \xrightarrow{\lambda} C'$. Supposons que C contient uniquement les trois processus qui participent à la transition λ . C est alors nécessairement de la forme*

$$(p_1, id_1, E_1) \quad (p_2, id_2, E_2) \quad (p_3, id_3, E_3)$$

où les interactions partielles associées aux points de programme sont de la forme :

- pour p_1 : $(behavior_n, [label], [e_i, s_i, \tilde{y}_k], \beta(C_i, \emptyset))$
- pour p_2 : $(actor, [ego, self], [], \emptyset)$
- pour p_3 : $(message_n, [dest, label, \tilde{x}_l], \emptyset)$

avec $|\tilde{y}_k| = |\tilde{x}_l| = n$. Nous avons alors les relations $E_2(ego) = E_3(dest)$, $E_1(label) = E_3(label)$ et $behavior_set((p_1, id_1)) = E_2(self)$. Le terme C' obtenu après la transition $\lambda = (p_1, p_2, p_3)$ est :

$$(p_1, id_1, E_1) \cup \left\{ \begin{array}{l} (p_j, new_id, E_j) \left| \begin{array}{l} (p_j, Es_j) \in \beta(C_i) \\ E_j = Es_j \left[\begin{array}{l} e_i \mapsto E_2(ego) \\ s_i \mapsto E_2(self) \\ \forall k \in \llbracket 1; n \rrbracket, y_k \mapsto E_3(x_k) \end{array} \right] \end{array} \right. \end{array} \right\}$$

où $new_id = id_1.p_1$.

La configuration standard $\Pi(C')$ est le terme clos constitué de l'ensemble des messages et des acteurs avec leur comportement associé tel que défini par le terme C' . Étudions les cas de C_i par induction sur la syntaxe de CAP :

- si $C_i = a \triangleright^{p_a} [m_l^{p_{m_l}}(\tilde{z}_l) = \zeta(e_l, s_l)C_l]$, alors $\beta(C_i, \emptyset) = (p_a, [self \mapsto p_a]) \cup \bigcup (p_{m_l}, [])$.

La fonction *exhibits* associe à chaque point de programme p_{m_l} , une interaction partielle de la forme :

$$(\text{behavior}_n, [m_l], [e_l, s_l, \tilde{z}_l], \beta(C_l, \emptyset))$$

Le terme résultat C' est donc, une fois le passage de valeurs effectué :

$$(p_1, id_1, E_1),$$

$$\left(p_a, \text{new_id}, E_a \left[\begin{array}{l} e_i \mapsto E_2(\text{ego}) \\ s_i \mapsto E_2(\text{self}) \\ \forall k \in \llbracket 1; n \rrbracket, y_k \mapsto E_3(x_k) \\ x \mapsto E_1(x) \end{array} \middle| \begin{array}{l} \text{tels que } E_a(x) \\ \text{soit défini ssi} \\ x \in \text{interface}(p_a) \end{array} \right] \right),$$

$$\left(p_l, \text{new_id}, E_l \left[\begin{array}{l} e_i \mapsto E_2(\text{ego}) \\ s_i \mapsto E_2(\text{self}) \\ \forall k \in \llbracket 1; n \rrbracket, y_k \mapsto E_3(x_k) \\ x \mapsto E_1(x) \end{array} \middle| \begin{array}{l} \text{tels que } E_l(x) \\ \text{soit défini ssi} \\ x \in \text{interface}(p_l) \end{array} \right] \right)_l$$

La traduction par la fonction Π de ce terme est

$$\nu \tilde{c}, a \triangleright^{p_a} [m_l^{p_l}(\tilde{z}_l) = \zeta(e_l, s_l)C_l]$$

où $\tilde{c} = \{x \mid E_a(x) \text{ ou } E_l(x) \text{ soit défini et désigne un point de programme correspondant à un } \nu\}$.

- si $C_i = a \triangleright^{p_a} b$, alors nécessairement l'ensemble des comportements désignés par la variable b a déjà été défini et est présent dans le système. Le terme résultat est donc similaire au cas précédent.
- si $C_i = a \triangleleft^{p_a} m(\tilde{z})$, alors $\beta(C_i, \emptyset) = (p_a, [])$. La fonction *exhibits* qui associe à chaque point de programme une interaction partielle, lui associe :

$$(\text{message}_n, [a, m, \tilde{z}], [], \emptyset)$$

L'interface de p_a est $\{a\} \cup \mathcal{FN}(\tilde{z})$. L'unique processus présent dans le terme résultat C' est donc, une fois le passage de valeurs effectué :

$$\left(p_a, \text{new_id}, E \left[\begin{array}{l} e_i \mapsto E_1(\text{ego}) \\ s_i \mapsto E_1(\text{self}) \\ \forall k \in \llbracket 1; n \rrbracket, y_k \mapsto x_k \end{array} \middle| \begin{array}{l} \text{tels que } E(x) \\ \text{soit défini ssi} \\ x \in \text{interface}(p_a) \end{array} \right] \right)$$

La traduction par la fonction Π de ce terme est

$$\nu \tilde{c}, a \triangleleft^{p_a} m(\tilde{z})$$

où $\tilde{c} = \{x \mid E(x) \text{ soit défini et désigne un point de programme correspondant à un } \nu\}$.

- si $C_i = \nu a^\alpha, C_a$, alors $\beta(C_i, \emptyset) = \beta(C_a, [a \mapsto \alpha])$. Par hypothèse d'induction, le terme obtenu en mettant à jour l'environnement statique des points de programme de $\beta(C_a, \emptyset)$ est C_a . La variable a peut éventuellement être libre dans le terme C_a . Le terme obtenu en mettant l'environnement des processus à jour par la relation $a \mapsto \alpha, \text{new_id}$ se traduit par Π en $\nu a^\alpha C_a$;
- si $C_i = C_1 \parallel C_2$ alors $\beta(C_i, \emptyset) = \beta(C_1, \emptyset) \cup \beta(C_2, \emptyset)$. Par induction, nous avons $\beta(C_1, \emptyset)$ qui se traduit en C_1 par Π et $\beta(C_2, \emptyset)$ qui se traduit en C_2 par Π . La traduction du terme $\beta(C_i, \emptyset)$ avec l'environnement mis à jour par le passage de valeurs, est donc le terme clos de CAP composé des acteurs et de messages de C_1 ainsi que des acteurs et des messages de C_2 . Nous avons bien C_i ;
- enfin, si $C_i = \emptyset$ alors $\beta(C_i, \emptyset) = \emptyset$. L'implication est triviale.

Nous avons montré que l'implication est vraie si le terme C comporte uniquement les trois processus nécessaires à la transition. Les règles de compatibilité ainsi que celles définies par la relation de congruence nous permettent d'effectuer un pas de calcul, lorsque l'acteur contient plus d'un comportement, et lorsque les processus requis sont sous une restriction de variables ou encore en parallèle avec d'autres processus, et donc de prouver l'implication dans le cas général pour les configurations bien formées.

Nous avons bien l'implication $\forall \lambda \in \mathcal{L}^3, C \xrightarrow{\lambda} C' \implies \Pi(C) \xrightarrow{\lambda} \Pi(C')$;

2. Soit P une configuration standard telle que $\Pi(C) \xrightarrow{\lambda} P$. Soit $(p_1, p_2, p_3) = \lambda$ l'étiquette de la transition. $\Pi(C)$ contient donc au moins l'acteur étiqueté par le point de programme p_2 associé au comportement étiqueté par p_1 , ainsi que le message étiqueté par le point de programme p_3 . Par définition de la fonction de traduction Π , le terme non standard C doit donc contenir au moins les processus (p_1, id_1, E_1) , (p_2, id_2, E_2) et (p_3, id_3, E_3) avec les contraintes nécessaires sur les marqueurs et les environnements des processus pour permettre la transition λ . Il existe donc une configuration non standard D image de C par la transition λ . Soit $\Pi(D)$ son image dans la syntaxe standard de CAP par la fonction de traduction Π . En réutilisant la propriété précédente, et en utilisant le fait que le terme C' est bien formé, nous obtenons $\Pi(D) \cong P$ par α -conversion, extrusion et l'utilisation de la réduction \rightarrow .
3. Montrons maintenant que $\mathcal{S} \equiv \Pi(C_0)$. Par induction, nous montrons aisément que $\forall C_i \text{ clos}, \beta(C_i, \emptyset)$ est clos.

De plus $\Pi(\beta(C_i, \emptyset)) \equiv C_i$ clos si C_i ne contient pas d'acteur avec un comportement dynamique, c.-à.-d. de la forme $a \triangleright b$; sinon, nous avons l'équivalence $\Pi(\beta(C_i, \emptyset)) \cong C_i$ clos. Les variables de comportements ne pouvant pas être liées par des lieurs de la forme ν mais seule-

ment par un opérateur ζ , la configuration initiale ne contient donc pas d'acteur ayant un comportement dynamique puisque le terme est clos. Soit $C_0 = C_0$ la configuration initiale. Comme $C_0 = \beta(\mathcal{S}, \emptyset)$ par définition, nous avons donc $\Pi(C_0) \equiv \mathcal{S}$.

Nous avons donc $\Pi(C) \xrightarrow{\lambda} P \implies \exists D, \left\{ \begin{array}{l} C \xrightarrow{\lambda} D \\ \Pi(D) \cong P \end{array} \right.$.

□

3.7 Expression dans le cadre générique de [Fer05b]

L'expression du langage CAP dans la méta-syntaxe et en particulier de son paradigme d'ordre supérieur ne permettait pas d'utiliser le cadre de la méta-syntaxe et de la sémantique associée de [Fer05b] tout en conservant une équivalence structurelle forte. Nous présentons dans cette sous-section nos différentes étapes de modélisation qui nous ont permis d'arriver à la description précédente.

Notre première modélisation était une expression *ad-hoc* du langage CAP dans le cadre général adapté pour avoir une forme d'équivalence structurelle forte. Ainsi chaque processus acteur est associé dans ce cadre à un ensemble de processus de comportement; lorsqu'une transition a lieu, les processus de comportements non activés sont supprimés du système. Les processus de comportement sont liés à l'acteur qu'ils décrivent par une variable spécifique *act* de leur environnement. Une des difficultés d'une telle modélisation était l'insertion dans le système des différentes branches de comportements appropriées lors de l'ajout d'un acteur ayant un comportement dynamique (c.-à-d. définit par une variable). Nous présentons ici deux difficultés de cette première modélisation :

Dans le cadre général, chaque interaction partielle est associée à un type parmi l'ensemble {réplication, calcul, migration}. Ces types permettent de générer les nouveaux marqueurs de nom systématiquement. Seules les transitions comportant une interaction partielle de type réplication génèrent des processus avec des nouveaux marqueurs. Le processus associé à cette interaction partielle réplication n'est pas supprimé du système.

Un premier souci se présente : la réplication du processus acteur dépend du processus comportement activé. Ainsi un acteur n'a pas un type général mais peut se répliquer ou non, suivant si la continuation de tel ou tel comportement comporte le sous-terme $e \triangleright^l s$ avec e et s les variables liées par l'opérateur ζ du comportement.

Une première approche serait de proposer trois règles formelles : une première modélisant la réplication de l'acteur, une seconde la réplication de l'acteur avec changement de comportement et une troisième pour supprimer l'acteur. Un point de programme correspondant à une installation d'acteur

pourrait donc exhiber deux interactions partielles, une pour répliquer l'acteur et la seconde pour le supprimer. Un point de programme correspondant à un comportement particulier pourrait exhiber trois interactions partielles, la première pour répliquer le comportement, la seconde pour modifier tous les comportements de l'acteur et la troisième pour les supprimer tous. Les contraintes associées à ces interactions partielles permettent de vérifier que le sous-terme ne contient pas d'acteur installé sur la même adresse que l'acteur qui participe à la transition pour la troisième, ou alors qu'il en contient un avec le même comportement pour la première, et enfin qu'il en contient un avec un comportement différent pour la seconde. Cette approche permettrait de générer moins de marqueurs mais n'est pas satisfaisante puisqu'elle peut conduire à une ambiguïté. Par exemple dans le terme suivant :

$$\nu a^\alpha, b^\beta, (\\ a \triangleright^1 [m^2(x) = \zeta(e, s)(x \triangleright^3 [n^4(y) = \zeta(e', s')(y \triangleright^5 s)] \parallel x \triangleleft^6 n(e))] \parallel \\ a \triangleleft^7 m(b) \parallel a \triangleleft^8 m(b) \\)$$

Déroulons un peu cet exemple pour souligner l'ambiguïté. La configuration initiale non standard d'un tel terme est :

$$(1, \epsilon, \left[\begin{array}{l} a \mapsto \alpha, \epsilon \\ self \mapsto 1, \epsilon \end{array} \right]) \quad (2, \epsilon, [act \mapsto 1, \epsilon]) \\ (7, \epsilon, \left[\begin{array}{l} a \mapsto \alpha, \epsilon \\ b \mapsto \beta, \epsilon \end{array} \right]) \quad (8, \epsilon, \left[\begin{array}{l} a \mapsto \alpha, \epsilon \\ b \mapsto \beta, \epsilon \end{array} \right])$$

Nous effectuons la transition (1, 2, 7) qui consiste en la destruction de l'acteur 1, ϵ . Aucun processus n'a donc le type répliation et nous ne calculons pas de nouveaux marqueurs. Les processus insérés dans le système porteront donc le marqueur du processus qui les a créés.

$$(3, \epsilon, \left[\begin{array}{l} x \mapsto \beta, \epsilon \\ self \mapsto 3, \epsilon \end{array} \right]) \quad (4, \epsilon, \left[\begin{array}{l} act \mapsto 3, \epsilon \\ s \mapsto 1, \epsilon \end{array} \right]) \\ (6, \epsilon, \left[\begin{array}{l} x \mapsto \beta, \epsilon \\ e \mapsto \alpha, \epsilon \end{array} \right]) \quad (8, \epsilon, \left[\begin{array}{l} a \mapsto \alpha, \epsilon \\ b \mapsto \beta, \epsilon \end{array} \right])$$

La transition (3, 4, 6) est possible et nous sommes dans le même cas que précédemment : pas de calcul de nouveaux marqueurs.

$$(5, \epsilon, \left[\begin{array}{l} y \mapsto \alpha, \epsilon \\ s \mapsto 1, \epsilon \end{array} \right]) \quad (2, \epsilon, [act \mapsto 5, \epsilon]) \\ (8, \epsilon, \left[\begin{array}{l} a \mapsto \alpha, \epsilon \\ b \mapsto \beta, \epsilon \end{array} \right])$$

Enfin nous effectuons la transition (5, 2, 8) : encore une fois, c'est un calcul, aucun nouveau marqueur n'est calculé.

$$(3, \epsilon, \left[\begin{array}{l} x \mapsto \beta, \epsilon \\ self \mapsto 3, \epsilon \end{array} \right]) \quad (4, \epsilon, \left[\begin{array}{l} act \mapsto 3, \epsilon \\ s \mapsto 1, \epsilon \end{array} \right])$$

$$(6, \epsilon, \left[\begin{array}{l} x \mapsto \beta, \epsilon \\ e \mapsto \alpha, \epsilon \end{array} \right])$$

Il n'y a alors aucune façon de différencier, tant par les marqueurs de processus que par les valeurs et les marqueurs de leur environnement, les instances récursives de l'acteur au point de programme 3 de la seconde et quatrième configurations.

Une seconde approche serait de proposer une interaction partielle de type répliation pour tous les processus acteurs. Suivant la forme du sous-terme associé à chaque continuation des comportements, nous préciserions alors si l'acteur répliqué doit être supprimé lorsque le comportement est activé. Les comportements de la forme $\zeta(e, s)(a \triangleright s)$ avec $E(e) = E(a)$ forceraient donc à une suppression de l'acteur dans un premier temps puis à une réinsertion ensuite avec un nouveau marqueur associé. Cette approche fonctionnerait mais obligerait à modifier le cadre générique original tout en étant inesthétique.

Nous avons donc d'abord choisi une approche similaire à celle décrite au début du chapitre qui consiste à calculer de nouveaux marqueurs à *chaque* transition et à s'affranchir des types pour les interactions partielles. Malheureusement, un tel système permet encore des ambiguïtés. Il ne permet pas de différencier les instances différentes d'un même point de programme comportement inséré dans le système à la même transition. Un exemple simple utilise la propriété de non linéarité 7.1. Soit un terme de la forme

$$a \triangleright^1 [m^2() = \zeta(e, s)(e \triangleright^3 s \parallel e \triangleright^4 s)]$$

Lorsque l'acteur a prendra en compte un message $m()$ qui lui sera adressé, il devra calculer un nouveau marqueur id et insérer dans le système :

1. un processus de la forme $(3, id, E_3)$ pour représenter l'acteur $e \triangleright^3 s$;
2. un processus de la forme $(4, id, E_4)$ pour représenter l'acteur $e \triangleright^4 s$;
3. un processus de la forme $(2, id, E_{2_1})$ pour représenter le comportement de l'acteur $e \triangleright^3 s$;
4. un processus de la forme $(2, id, E_{2_2})$ pour représenter le comportement de l'acteur $e \triangleright^4 s$.

Nous pourrions donc différencier les processus $(2, id, E_{2_1})$ et $(2, id, E_{2_2})$ par leur environnement puisque $E_{2_1}[act] = (3, id)$ et $E_{2_2}[act] = (4, id)$ mais nous ne satisfaisons plus à la condition qui stipule que dans tout système, le couple (p, id) identifie le processus (p, id, E) .

Pour résoudre cette ambiguïté, nous avons choisi de modifier légèrement le marqueur des processus de comportement pour qu'il précise l'acteur auquel le comportement est lié. Le système obtenu est donc correct, comme nous l'avons montré dans la section 3.6.2 et plus simple puisque nous avons une unique règle formelle et une seule interaction partielle par point de programme.

Une autre caractéristique du cadre générique diffère : initialement, les règles formelles comportent un ensemble de *broadcast* qui permet lors d'une transition de mettre à jour l'environnement de processus qui n'ont pas participé à la transition. Cela permet, par exemple, de représenter dans le calcul des AMBIENTS l'ouverture d'une boîte où tous les éléments de la boîte voient leur localité modifiée. Aucun principe du langage CAP n'a besoin d'un tel mécanisme et nous l'avons supprimé pour permettre une meilleure lisibilité.

Par contre, un mécanisme similaire, non présent dans le système original, que nous avons introduit est un ensemble *global_remove* dans la règle formelle qui permettrait de modéliser la suppression de processus qui n'ont pas interagi mais dont l'environnement réalise une certaine contrainte. Bien qu'un tel mécanisme pourrait s'encoder dans la syntaxe du méta-langage est séparant dans deux partitions d'un ensemble les processus actifs des processus morts, il compliquerait l'abstraction et serait difficile à interpréter.

Une modélisation, telle que nous venons de la décrire, permet bien de représenter le langage CAP sans le passage de comportement en paramètre. En effet, nous pouvons modéliser un acteur qui a un comportement cyclique, comme le buffer à une place (*cf.* page 12), et faire référence dans un acteur à ses précédents ensembles de comportements, mais nous ne pouvons pas récupérer son comportement et l'envoyer par un message à un second acteur. Pour ce faire, il faudrait calculer la fermeture transitive d'un comportement et y faire référence d'une certaine façon lors de l'utilisation d'un comportement contenu dans une variable. Nous pourrions par exemple envoyer l'ensemble des valeurs des variables, qu'il faut définir pour utiliser correctement le comportement contenu dans la variable émise, à la suite des variables du message. Mais cela nous obligerait à avoir des messages avec un nombre d'arguments changeant.

Pour modéliser un tel mécanisme, nous avons choisi de représenter les comportements d'un acteur sous la forme de définitions, à la manière des *définitions* du Join calcul. C'est la représentation décrite dans le début de cette section. Les comportements d'un acteur ne font donc pas référence à cet acteur et sont, une fois définis, présents de façon irrévocable dans le système. Une telle modélisation n'est pas structurellement identique au langage CAP mais permet de définir les instances récursives d'un même comportement ayant des variables *internes* différentes.

Chapitre 4

Interprétation abstraite

4.1 Principes généraux

L'interprétation abstraite est une théorie de l'approximation discrète de sémantiques. Un principe fondamental de cette théorie est que toute sémantique peut être exprimée comme point fixe d'opérateurs monotones sur une structure partiellement ordonnée.

La spécification d'une sémantique peut être définie par un ensemble S muni d'un élément \perp base de l'itération et d'un endomorphisme f de S complet pour l'union (c.-à-d. tel que l'élément plus petit point fixe de l'opérateur f plus grand ou égal à \perp existe et appartient à S). Par contre, cet élément n'est, dans le cas général, pas toujours calculable de manière finie.

Nous définissons donc dans un autre ensemble S' , une approximation calculable de cette sémantique, une spécification sémantique abstraite. Il existe plusieurs approches pour construire une telle sémantique.

Nous désignons dans la suite par domaine, un ensemble muni d'un pré-ordre, d'un élément \perp ainsi que d'opérateurs comme l'union ou l'intersection qui sont utilisés pour construire S' et calculer les points fixes de la sémantique abstraite.

4.1.1 Interprétations abstraites construites sur des correspondances de Galois

La première approche et la plus connue est celle basée sur les correspondances de GALOIS [CC77, CC79] (ou plutôt les correspondances de GALOIS semi-duales). Une correspondance de GALOIS entre les sémantiques définies sur les domaines S et S' est donc constituée d'un couple de fonctions (α, γ) définies telles que :

Définition 4.1.1 (correspondance de Galois) *Soit (S, \subseteq^S) et $(S', \subseteq^{S'})$ deux ensembles partiellement ordonnés. Nous définissons une correspondance de GALOIS entre ces ensembles par un couple de fonctions (α, γ) vérifiant :*

- $\alpha : S \rightarrow S'$
- $\gamma : S' \rightarrow S$
- $\forall x \in S, x' \in S', \alpha(x) \sqsubseteq^{S'} x' \Leftrightarrow x \sqsubseteq^S \gamma(x')$.

Ainsi les fonctions α et γ sont monotones et nous pouvons en déduire que $x \sqsubseteq^S \gamma(\alpha(x))$ ce qui nous permet d'affirmer que l'approximation est correcte.

Étant donné un opérateur f dans le domaine concret (par opposition à abstrait), un opérateur abstrait f' associé est défini comme *correct* s'il vérifie la propriété suivante :

Proposition 4.1.2 (opérateur correct)

$$\forall x' \in S', (\alpha \circ f \circ \gamma)(x) \sqsubseteq^S f'(x')$$

Ce type d'abstraction n'est pas toujours aisé à mettre en place puisqu'il requiert des domaines abstraits ayant des structures particulières.

4.1.2 Interprétations abstraites sans correspondances de Galois

Ainsi dans [CC92], COUSOT et COUSOT ont expliqué comment affaiblir les hypothèses pour utiliser des abstractions construites sur uniquement l'une ou l'autre des fonctions. Nous nous intéressons ici à des abstractions basées sur la fonction de concrétisation.

La fonction $\gamma : S' \rightarrow S$ doit donc être monotone et vérifier la propriété suivante :

Proposition 4.1.3 (fonction de concrétisation) *Soit x' une abstraction de x , alors*

$$x \sqsubseteq \gamma(x')$$

Les opérateurs abstraits f' correspondant dans l'abstrait aux opérateurs concrets f sont donc définis corrects s'ils vérifient :

$$\forall x' \in S', (f \circ \gamma)(x') \sqsubseteq^{S'} (\gamma \circ f')(x')$$

4.2 Interprétation abstraite de systèmes mobiles

Nous cherchons ici à approximer la sémantique de systèmes mobiles décrits dans la méta-syntaxe, à la manière de [Fer05b, Ven98]. Soit \mathcal{C} l'ensemble des configurations non standards qui satisfont la condition de non-ambiguïté des marqueurs (soit deux processus (p_1, id_1, E_1) et (p_2, id_2) alors si $p_1 = p_2$ et $id_1 = id_2$ alors $E_1 = E_2$).

Nous cherchons donc à approximer la sémantique collectrice d'une configuration \mathcal{C}_0 définie comme le plus petit point fixe du morphisme complet pour l'union \mathbb{F} défini par :

$$\mathbb{F}(X) = (\{\epsilon\} \times \mathcal{C}_0) \cup \left\{ (u.\lambda, C') \mid \exists C \in \mathcal{S}, (u, C) \in X \text{ et } C \xrightarrow{\lambda} C' \right\}$$

Définition 4.2.1 Une abstraction est un n -uplet $\mathcal{A} = (\mathcal{C}^\#, \sqsubseteq^\#, \cup^\#, \perp^\#, \gamma^\#, C_0^\#, \rightsquigarrow^\#, \nabla)$ qui vérifie les propriétés suivantes :

1. $(\mathcal{C}^\#, \sqsubseteq^\#)$ est un pré-ordre ;
2. $\cup^\#$ est tel que $\forall a, b \in \mathcal{C}^\#, a^\# \sqsubseteq^\# a \cup^\# b$ et $b^\# \sqsubseteq^\# a \cup^\# b$;
3. $\perp^\# \in \mathcal{C}^\#$ est tel que $\forall a \in \mathcal{C}^\#, \perp^\# \sqsubseteq^\# a$;
4. $\gamma : \mathcal{C}^\# \rightarrow \wp(\Sigma^* \times \mathcal{C})$ est une fonction monotone ;
5. $C_0^\# \in \mathcal{C}^\#$ est tel que $\{\epsilon\} \times \mathcal{C}_0 \subseteq \gamma(C_0^\#)$
6. $\rightsquigarrow^\# \in \wp(\mathcal{C}^\# \times \Sigma \times \mathcal{C}^\#)$ est une relation de transition abstraite telle que $\forall C^\# \in \mathcal{C}^\#, \forall (u, C) \in \gamma(C^\#), \forall \lambda \in \Sigma, \forall C' \in \mathcal{C}$,

$$C \xrightarrow{\lambda} C' \implies \exists C'^\# \in \mathcal{C}^\#, (C^\# \rightsquigarrow^\lambda C'^\#) \text{ et } (u.\lambda, C') \in \gamma(C'^\#)$$

7. ∇ doit vérifier la même condition que l'union, mais doit, de plus, être défini tel que $\forall (C_n^\#)_{n \in \mathbb{N}} \in (\mathcal{C}^\#)^{\mathbb{N}}$, la séquence $(C_n^\nabla)_{n \in \mathbb{N}}$ définie par :

$$\begin{cases} C_0^\nabla & = & C_0^\# \\ C_{n+1}^\nabla & = & C_n^\nabla \nabla C_{n+1}^\# \end{cases}$$

doit être ultimement stationnaire.

Il nous reste enfin à définir le pendant dans l'abstrait de la fonction \mathbb{F} . Soit $\mathbb{F}^\#$ une telle fonction. Elle est définie par

$$\mathbb{F}^\#(C^\#) = \bigcup^\# \left(\left\{ C'^\# \mid \exists \lambda \in \Sigma, C^\# \rightsquigarrow^\lambda C'^\# \right\} \cup \{C_0^\#; C^\#\} \right)$$

Comme la fonction \mathbb{F} était définie monotone et complète pour l'union, la fonction $\mathbb{F}^\#$ satisfait les critères de la définition 4.2.1

4.3 Opérations sur les domaines

Nous décrivons ici comment raffiner ou combiner des abstractions pour représenter des propriétés plus précises.

4.3.1 Domaine réduit

Un produit réduit d'un domaine est une abstraction dans laquelle la fonction de transition est raffinée en utilisant un opérateur ρ qui vérifie la propriété de correction pour les opérateurs du domaine :

$$\forall a \in \mathcal{C}^\#, \gamma(a) \subseteq \gamma(\rho(a))$$

Proposition 4.3.1 (domaine réduit) *Soit une abstraction $\mathcal{A} = (\mathcal{C}^\#, \sqsubseteq^\#, \cup^\#, \perp^\#, \gamma^\#, C_0^\#, \rightsquigarrow, \nabla)$ et ρ un opérateur de réduction. Le n -uplet $(\mathcal{C}^\#, \sqsubseteq^\#, \cup^\#, \perp^\#, \gamma^\#, C_0^\#, \rightsquigarrow_p, \nabla)$ où \rightsquigarrow_p est défini par :*

$$a \rightsquigarrow_p c \text{ ssi } \exists b \in \mathcal{C}^\# \text{ tel que } \rho(a) \rightsquigarrow b \text{ et } c = \rho(b)$$

4.3.2 Produit cartésien

Le produit cartésien de deux domaines \mathcal{A} et \mathcal{B} construits respectivement sur les ensembles $C_1^\#$ et $C_2^\#$ est un domaine \mathcal{D} dont les éléments sont des couples $(a, b) \in \mathcal{C}^\# = \mathcal{C}^\# \times \mathcal{C}^\#$. Les fonctions d'inclusion, d'union et d'élargissement ainsi que le plus petit élément sont définis composante par composante. La fonction γ_D est définie comme l'intersection des images des éléments abstraits par leur fonction de concrétisation :

$$\gamma_D : \begin{cases} \mathcal{C}^\# \rightarrow \wp(\Sigma^* \times \mathcal{C}) \\ (a, b) \mapsto \gamma_A(a) \cap \gamma_B(b) \end{cases}$$

L'élément initial est défini comme le couple des éléments initiaux des abstractions \mathcal{A} et \mathcal{B} . Enfin, la fonction de transition est définie comme l'image de chacune des composantes par sa fonction de transition.

Chapitre 5

Sémantique abstraite

Nous présenterons dans ce chapitre l'analyse par interprétation abstraite du langage CAP exprimé dans la méta-syntaxe définie au chapitre 3. Nous décrirons tout d'abord quelle sémantique nous voulons approximer. Puis nous détaillerons les problèmes posés par CAP. Ensuite, dans les deux dernières parties, nous décrirons deux domaines définis dans [Fer05b] pour effectuer notre analyse. Nous redonnons dans les deux dernières parties beaucoup de primitives et de définitions issues des chapitres [Fer05b, chap. 8.1, 8.3 & 9] dans lesquels les preuves non explicitées ici pourront être trouvées.

5.1 Sémantique collectrice

La sémantique collectrice décrit l'ensemble des configurations que peut prendre un système à partir d'une configuration \mathcal{C}_0 . Soit \mathbf{Conf} un tel ensemble :

$$\mathbf{Conf} = \varnothing \left(\left\{ (p, id, E) \left| \begin{array}{l} p \text{ est un point de programme de la configuration } \mathcal{C}_0 \\ E \text{ est un environnement dans } (\mathcal{V} \rightarrow (\mathcal{L} \times \mathcal{M})) \\ \text{valide pour } \mathcal{C}_0 \end{array} \right. \right\} \right)$$

Définition 5.1.1 *La sémantique collectrice d'une configuration \mathcal{C}_0 est définie par l'ensemble \mathfrak{C} :*

$$\mathfrak{C} = \{(u, C) \in (\Sigma_{(\mathcal{L}_p \times \mathcal{L}_p \times \mathcal{L}_p)}^* \times \mathbf{Conf}) \mid \mathcal{C}_0 \xrightarrow{u} C\}$$

Nous voulons donc calculer le plus petit point fixe de la relation de transition pour pouvoir observer certaines propriétés. Comme nous l'avons défini précédemment, nous devons définir un domaine abstrait dans lequel nous observerons une abstraction des propriétés du système et dans lequel nous calculerons un point fixe de la relation de transition. Par construction, l'élément abstrait plus petit point fixe du domaine abstrait pour la fonction de transition sera une sur-approximation correcte quant aux propriétés abstraites exprimées par le domaine abstrait.

nouveaux ensembles de comportement, mais seulement de récupérer le comportement d'un acteur pour l'envoyer à un autre.

Une approche simple pour résoudre ce problème consisterait à calculer les transitions avec, pour chaque comportement dynamique, *i.e.* un sous-terme de la forme $a \triangleright x$, l'ensemble des ensembles de comportements définis syntaxiquement. Une telle approche serait correcte mais bien trop imprécise, puisqu'elle conduirait à une trop grande sur-approximation des propriétés du système.

Nous avons choisi d'utiliser un domaine abstrait constitué du produit de deux domaines. Le premier permet d'approximer le flot de contrôle de façon précise tandis que le second nous sert à observer des propriétés numériques sur le système. Le calcul de la fonction de transition se fait alors en deux étapes. Dans la première, nous vérifions que les conditions de synchronisation sont respectées. Le premier domaine nous permet alors de restreindre pour chaque comportement dynamique l'ensemble des ensembles de comportement qui peut lui être associé. La seconde étape utilise alors cette information pour calculer le passage des paramètres, lancer les continuations et supprimer les processus concernés.

Dans les sections suivantes, nous définissons le domaine permettant d'approximer le flot de contrôle puis celui permettant de représenter certaines propriétés numériques sur le système.

5.3 Domaines pour le flot de contrôle

Nous décrivons dans cette section un domaine générique [Fer02] permettant de décrire le flot de contrôle du système. Ensuite, nous instancierons ce domaine pour refléter certaines propriétés. Le domaine du flot de contrôle sera décrit dans la dernière partie dans laquelle nous définirons le produit réduit des domaines précédemment définis.

5.3.1 Domaine générique

Notre analyse est paramétrée par un domaine *Atom* qui permet de représenter certaines propriétés. Un élément abstrait est une fonction qui, à chaque point de programme, associe un élément de *Atom* représentant l'ensemble des propriétés du point de programme. Lors du calcul d'une transition dans le domaine abstrait, nous mettons en relation trois de ces *Atom* pour former une molécule. Nous définirons, dans une première partie, les primitives permettant de manipuler l'*Atom* associé à un point de programme. Puis, dans une seconde, celles que nous utiliserons pour les *Molecule*. Enfin dans la troisième et la quatrième parties, nous montrerons comment utiliser ces éléments pour calculer l'itéré d'une transition dans l'abstrait.

Abstraction de processus

Soit une famille $(Atom_V^\#, \gamma_V, \sqsubseteq_V, \sqcup_V, \perp_V)_{V \subseteq \mathcal{V}}$ de domaines abstraits représentant des propriétés. Le domaine $Atom_V^\#$ est utilisé pour abstraire les propriétés sur le marqueur et sur les variables du processus dont l'interface est l'ensemble de variables $V \subseteq \mathcal{V}$.

La relation \sqsubseteq_V est un pré-ordre sur l'ensemble des propriétés représentées par le domaine.

Soit $Env_V^\#$ l'ensemble $\mathcal{M} \times (V \rightarrow (\mathcal{L} \times \mathcal{M}))$ des couples constitués d'un marqueur et d'un environnement sur V . Chaque élément abstrait $a \in Atom_V^\#$ est associé à $\wp(Env_V^\#)$ par la fonction monotone de concrétisation γ_V .

L'opération \sqcup_V affaiblit les propriétés de l'ensemble d'éléments auquel il est affecté. Nous avons donc pour chaque ensemble fini $A \subseteq \wp(Atom_V^\#)$ et pour chaque élément $a \in A$, $a \sqsubseteq_V (\sqcup_V A)$.

L'élément \perp_V est le plus petit élément de $Atom_V^\#$.

Soit l'environnement abstrait $\mathcal{C}_{env}^\#$ défini comme le produit des $Atom$ associés à chaque point de programme.

$$\mathcal{C}_{env}^\# = \prod_{p \in \mathcal{L}_p} (Atom_{I(p)}^\#)$$

où $I(p)$ désigne l'interface du point de programme p .

La fonction de concrétisation γ_{env} associe à chaque élément abstrait de $(f_p)_{p \in \mathcal{L}_p} \in \mathcal{C}_{env}^\#$ l'ensemble des couples $(u, C) \in \Sigma^* \times \mathcal{C}$ des configurations atteignables après les transitions formant le mot u et tel que chaque processus (q, id, E) de C satisfasse la condition $(id, E) \in \gamma_{I(q)} f_q$.

Les opérateurs $(\sqsubseteq_{env}, \sqcup_{env}, \perp_{env})$ sont construits aisément en utilisant le produit des opérateurs $(\sqsubseteq_V, \sqcup_V, \perp_V)_{V \subseteq \mathcal{V}}$.

Nous introduisons quatre primitives qui vont nous permettre de manipuler les éléments de $Atom_V$:

- *environnement initial* : l'élément abstrait $\epsilon_0^\#$ décrit l'abstraction de la paire constitué du mot vide (il ne s'est déroulé encore aucune transition) et de l'environnement vide : (ϵ, \emptyset) . $\epsilon_0^\#$ satisfait donc la relation :

$$\{(\epsilon, \emptyset)\} \subseteq \gamma_{\emptyset}(\epsilon_0^\#);$$

- *restriction* : lors du calcul des transitions, il va nous falloir représenter dans le domaine abstrait le pendant de l'affectation de variable dans le concret. La primitive $\nu^\#$ réalise cette fonction. Ainsi $\nu^\#(v, l, A)$ représente l'élément abstrait A dans lequel nous avons inséré la contrainte : "la variable v prend la valeur l et le marqueur du processus".

Pour satisfaire les critères de correction, la primitive $\nu^\#$ doit réaliser la condition :

$$\forall A \in Atom_{V \cup \{x\}}^\#,$$

$$\left\{ (id, E) \in Env_{V \cup \{x\}}^{\#} \left| \begin{array}{l} (id, E|_V) \in \gamma_V(A) \\ E(x) = (l, id) \\ \forall y \in V, E(y) \neq (l, id) \end{array} \right. \right\} \subseteq \gamma_{V \cup \{x\}}(\nu^{\#}(x, l, A));$$

- *projection* : lors de la manipulation des atomes, il nous faudra parfois restreindre l'ensemble des variables de l'atome pour, par exemple, ne conserver que les variables de l'interface du point de programme auquel est associé l'atome. C'est le rôle de la primitive $GC^{\#}$. Ainsi $GC^{\#}(X, A)$ où $A \in Atom_V^{\#}$ est un élément de $Atom_{V \cap X}^{\#}$ qui satisfait la relation :

$$\{(id, E|_{V \cap X}) \in Env_X^{\#} \mid (id, E) \in \gamma_V(A)\} \subseteq \gamma_{V \cap X}(GC^{\#}(X, A))$$

Chaque domaine abstrait permettant de représenter des propriétés sur les processus devra donc définir la fonction de concrétisation γ_V , les opérateurs \sqsubseteq_V, \sqcup_V et \perp_V mais aussi ces quatre primitives.

Abstraction d'agrégats de processus

Pour mettre en relation plusieurs processus lors des communications, nous définissons le domaine suivant paramétré par une famille de domaines abstraits $(Molecule_{(V_i)_i}^{\#})$. Ce domaine est utilisé pour mettre en relation des processus entre eux mais aucune itération n'est calculé directement dans ce domaine, le domaine $Molecule_{(V_i)_i}^{\#}$ n'est donc pas équipé d'une structure d'ordre partiel. Cette famille est indexée par les n-uplets $(V_i)_{1 \leq i \leq n} \in \wp(\mathcal{V})^n$. Chaque élément de $Molecule_{(V_i)_{1 \leq i \leq n}}$ est associé par la fonction de concrétisation $\gamma_{(V_i)_{1 \leq i \leq n}}$ à l'ensemble des éléments de $\wp(\prod_{1 \leq i \leq n} Env_{V_i}^{\#})$. Ils nous faut définir des primitives permettant de lier les familles $(Atom_V^{\#})$ et $(Molecule_{(V_i)_i}^{\#})$.

- *injection* : l'injection d'un atome dans une molécule permet de transformer un élément abstrait d' $Atom_V^{\#}$ représentant les propriétés d'un processus dans un élément de $Molecule_{(V)}$ constitué d'un 1-uplet de processus. Soit $V \subseteq \mathcal{V}$ une interface et A un élément abstrait dans $Atom_V^{\#}$. La primitive abstraite $INJ^{\#}$ satisfait la relation :

$$\gamma_V(A) \subseteq \gamma_{(V)}(INJ^{\#}(A))$$

- *concaténation* : le produit de molécules permet de mettre en relation plusieurs familles d'atomes. La primitive $\bullet^{\#}$ représente l'opérateur de concaténation. Elle doit satisfaire la relation :

$$\left\{ (e_i)_{i \in [1; m+n]} \left| \begin{array}{l} (e_i)_{1 \leq i \leq m} \in \gamma_{(U_i)}(A) \\ (e_{i+m})_{1 \leq i \leq n} \in \gamma_{(V_i)}(B) \end{array} \right. \right\} \subseteq \gamma_{(W_i)}(A \bullet^{\#} B)$$

où W_i est défini comme la concaténation des interfaces U_i et V_i , *i.e.* $\forall j, 1 \leq j \leq |(U_i)_i|$, $W_j = U_j$ et $\forall j, |(U_i)_i| + 1 \leq j \leq |(U_i)_i| + |(V_i)_i|$, $W_j = V_{j - |(U_i)_i|}$.

- *projection* : la projection d'une molécule sur une de ses composantes permet d'obtenir un atome d'une molécule. Soit A , la molécule définie sur la famille d'interface $(V_i)_i$. $PROJ^\#(k, A)$ correspond donc au k -ème atome de la molécule. Nous avons $PROJ^\#(k, A) \in Atom_{V_k}$ et qui satisfait :

$$\{(id_k, E_k) \mid \exists (id_i, E_i)_i \in \gamma_{(V_i)_i}(A)\} \subseteq \gamma_{V_k}(PROJ^\#(k, A))$$

- *extension* : l'opérateur d'extension $NEW^\#_\top$ permet d'insérer des nouvelles variables fraîches dans une molécule ou encore d'*oublier* les contraintes ou les propriétés sur un ensemble de variables. Soit une molécule $A \in molecule^\#_{(V_i)}$. Nous construisons les ensembles d'interfaces U_i et W_i par $U_i = V_i \setminus \{(x, i) \mid x \in X\}$ et $W_i = V_i \cup \{(x, i) \mid x \in X\}$. L'opérateur $NEW^\#_\top$ doit réaliser la relation :

$$\left\{ (id_i, E_i) \in \Pi(Env_{W_i}^\#) \mid \begin{array}{l} \exists (id_i, E'_i) \in \gamma_{(V_i)}(A), \\ \forall i \in \llbracket 1; n \rrbracket, \forall x \in U_i, \\ E'_i(x) = E_i(x) \end{array} \right\} \subseteq \gamma_{(W_i)}(NEW^\#_\top(X, A))$$

- *synchronisation* : Soit S un ensemble de contraintes de la forme $(x_k, k) \diamond (x_l, l)$ où $x_i \in V_i \cup \{I\}$, $\diamond \in \{=, \neq\}$ et $k, l \in [1; 3]$. La primitive $SYNC^\#(S, (p_i)_i, A)$ permet d'insérer dans la molécule A les contraintes de S . La synchronisation de la molécule par l'ensemble de contraintes est définie par :

$$\left\{ (id_i, E_i) \in \gamma_{(V_i)}(A) \mid \begin{array}{l} \forall (a \diamond b) \in S, \\ \rho(a) \diamond \rho(b) \end{array} \right\} \subseteq \gamma_{(V_i)}(SYNC^\#(S, (p_i), A)),$$

$$\text{où } \rho(x, i) = \begin{cases} E_i(x) & \text{si } x \in V_i \\ (p_i, id) & \text{si } x = I \end{cases}$$

- *calcul de marqueur* : La primitive $FETCH^\#$ permet de simuler dans l'abstrait le calcul des nouveaux marqueurs. Chaque marqueur de processus est remplacé par le nouveau marqueur. $FETCH^\#$ est défini par :

$$\left\{ (\bar{id}, E_i) \mid \begin{array}{l} \forall i \in \llbracket 1; n \rrbracket, (id_i, E_i) \in \gamma_{(V_i)}(A), \\ id_i \neq \bar{id} \\ \forall x \in V_i, y \in \mathcal{L}, (y, \bar{id}) \neq E_i(x) \end{array} \right\} \subseteq \gamma_{(V_i)}(FETCH^\#((p_i), A))$$

où $\bar{id} = \psi(N((p_i)_{1 \leq i \leq 3}, id_1, id_2, id_3))$ avec ψ l'abstraction du marqueur définie dans la section 3.6.2.

Primitives abstraites

Action exhibée Lors de la traduction du terme de CAP vers la syntaxe non standard, nous avons associé à chaque point de programme une interaction partielle. Nous définissons qu'une interaction partielle pi est exhibée au point de programme p et nous notons $pi \in exhibits(p)$ ssi $pi = interaction(p)$

Molécule réactive Afin de représenter dans l'abstrait les contraintes que doivent réaliser l'ensemble des processus qui participent à une transition, nous définissons la primitive $reagents^\#$ qui construit une *Molécule*, c.-à-d. un ensemble de processus représentés par des *Atoms*, à partir des atomes représentant les processus de la transition. Nous injectons ensuite dans le système les contraintes exprimées par la règle de compatibilité de la règle formelle qui dirige la transition. Nous rappelons que la transition peut avoir lieu ssi ces contraintes sont réalisées par les interactions partielles associées aux points de programme des processus qui participent à la transition. La primitive $reagents^\#$ prend donc en arguments le triplet des points de programme constituant l'étiquette de la transition, les paramètres des interactions partielles associées, l'ensemble de compatibilité de la règle formelle et l'élément abstrait qui, à chaque point de programme, associe un *Atom*. Elle calcule alors l'élément du domaine *Molécule* dans lequel les contraintes sont insérées. Si ces contraintes ne peuvent pas être satisfaites, nous obtenons l'élément \perp de l'ensemble du domaine *Molécule*.

Définition 5.3.1 (molécule réactive) Soit (p_k) un triplet de points de programme, $(param_{k,l})_{k,l}$ un triplet de séquences de paramètres, *compatibility* un ensemble de contraintes et $C^\# \in \mathcal{C}_{env}$. Nous définissons $reagents^\#((p_k), (param_{k,l})_{k,l}, compatibility, C^\#) \in Molecule^\#_{(I(p_k))_{1 \leq k \leq 3}}$ par

$$SYNC^\#(cons, (p_k), mol)$$

où

- $mol \triangleq INJ^\#(C^\#(p_1)) \bullet^\# INJ^\#(C^\#(p_2)) \bullet^\# INJ^\#(C^\#(p_3))$;
- $cons \triangleq \{\sigma(X) = \sigma(Y) \mid (X, Y) \in compatibility\}$,

$$avec \sigma : \begin{cases} I_k & \mapsto (I, k) \\ X_k^l & \mapsto (param_{k,l}, k) \end{cases} .$$

Calcul de marqueur et passage des valeurs Si la primitive précédente ne donne pas l'élément \perp du domaine *Molécule*, la transition est possible. Il faut donc faire le passage des valeurs et calculer les nouveaux marqueurs, c'est le rôle de la primitive *marker_value*. Mais le calcul des nouveaux marqueurs peut dépendre de ces nouvelles valeurs, comme le calcul du passage de valeurs doit utiliser les nouveaux marqueurs pour les variables qui viennent d'être créées (par exemple avec un opérateur ν dans la continuation d'un acteur). Nous procédons donc en trois étapes. Nous insérons des variables temporaires pour chaque variable liée qui va être modifiée par le passage de valeur. Nous calculons ensuite les nouveaux marqueurs. Puis nous remplaçons les anciennes valeurs des variables liées par celles contenues dans les variables temporaires.

Définition 5.3.2 Soit (p_k) un triplet de points de programme, $(param_{k,l})_{k,l}$ un triplet de séquences de paramètres, $(bd_{k,l})_{k,l}$ un triplet de séquences de

variables liées, \mathcal{V}_{bd} l'ensemble des paires (k, l) telles que $bd_{k,l}$ soit défini. Soit $v_passing$ une fonction partielle définie sur $\mathcal{V}_f^Y \rightarrow \mathcal{V}_f^X \cup \mathcal{V}_f^I$ et $(Z_l)_{l \in \mathbb{N}}$ une famille de variables distinctes et fraîches. Enfin, soit $molecule^\#$ un élément du domaine *Molecule*. Nous insérons tout d'abord les variables temporaires :

$$C_1 \triangleq NEW_{\top}^\#(\{(Z_l, k) \mid (k, l) \in \mathcal{V}_{bd}\}, molecule^\#)$$

Puis nous synchronisons la molécule avec le passage de valeur :

$$C_2 \triangleq SYNC^\#(cons_1 \cup cons_2, (p_k), C_1),$$

où

$$- cons_1 = \{(Z_l, k) = (param_{k',l}, k') \mid \exists k', l, l' \in \mathbb{N}, v_passing(Y_k^l) = X_{k'}^{l'}\},$$

$$- cons_2 = \{(Z_l, k) = (I, k') \mid \exists k', l \in \mathbb{N}, v_passing(Y_k^l) = I_{k'}\},$$

Puis nous calculons les nouveaux marqueurs :

$$C_3 \triangleq FETCH^\#((p_k), C_2)$$

Nous supprimons de la molécule ainsi obtenue toute information sur un variable dont la valeur a été modifiée, par le passage de valeurs.

$$C_4 \triangleq NEW_{\top}^\#(\{(bd_{k,l}, k) \mid (k, l) \in \mathcal{V}_{bd}\}, C_3)$$

La molécule $marker_value((p_k)_k, molecule, (bd_{k,l})_{k,l}, (parameters_{k,l})_{k,l}, v_passing)$ est définie par :

$$SYNC^\#(\{(Z_l, k) = (bd_{k,l}, l) \mid (k, l) \in I\}, (p_k), C_4)$$

Démarrage des continuations Enfin il nous faut calculer l'ensemble des processus qui doivent être ajoutés au système. Nous définissons d'abord une primitive $update^\#$ qui simule dans l'abstrait la mise à jour de l'environnement d'un processus par un environnement statique¹. La primitive $update^\#$ est définie comme suit :

Définition 5.3.3 (mise à jour de l'environnement) Soit E_s un environnement statique. Soit $p \in \mathcal{L}_p$ un point de programme. Nous pouvons munir l'ensemble des variables \mathcal{V} d'un ordre total $\leq_{\mathcal{V}}$ pour définir $Dom(E_s) = \{x_i \mid 1 \leq i \leq n\}$ avec $x_1 \leq_{\mathcal{V}} \dots \leq_{\mathcal{V}} x_n$. Soit $A \in Atom_{\mathcal{V}}^\#$ une abstraction d'un processus. Nous définissons l'atome $update^\#(p, E_s, A) \in Atom_{\mathcal{V} \cup Dom(E_s)}^\#$ par :

$$update^\#(p, E_s, C) \triangleq C_n$$

où

¹Nous rappelons qu'un environnement statique est une relation dans $\mathcal{V} \times \mathcal{L}_p$ qui à une variable associe une valeur sans marqueur. La mise à jour d'un processus par un environnement statique consiste à insérer les variables en utilisant comme marqueur le marqueur identité du processus.

- $C_0 \triangleq GC^\#(V \setminus Dom(E_s), C)$,
- $C_{k+1} \triangleq \nu^\#(x_{k+1}, E_s(x_{k+1}), C_k), \forall k \in \llbracket 0; n \rrbracket$.

Nous pouvons maintenant simuler dans l'abstrait le démarrage des continuations :

Définition 5.3.4 (démarrage des continuations) *Soit $(V_i)_{1 \leq i \leq 3}$ un triplet d'interfaces, soit $mol \in Molecule^\#_{(V_i)_{1 \leq i \leq 3}}$ une abstraction d'un triplet de processus et $ct_k \in \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))^3$ un triplet de continuations. L'élément abstrait $launch^\#((p_k, ct_k), mol) \in \mathcal{C}_{env}$ est défini par :*

$$\left[p' \mapsto \bigsqcup_{I(p')} \left\{ GC^\#(I(p'), update^\#(p', E_s, A)) \mid \begin{array}{l} \exists k', A = PROJ^\#(k', mol) \text{ et} \\ (p', E_s)ct_{k'} \end{array} \right\} \right]$$

Sémantique opérationnelle

Nous utilisons maintenant les primitives définies dans la sous-section précédente pour décrire la sémantique opérationnelle abstraite. C'est à dire la façon dont nous allons simuler dans le domaine abstrait une transition dans le concret.

L'état initial, c'est à dire l'élément abstrait du domaine correspondant au terme CAP que nous cherchons à analyser, est obtenu en exécutant les continuations du terme initial. Plus formellement, soit $C_0 \in \mathcal{C}_{env}$ l'abstraction des états initiaux définie par :

$$C_0 = launch^\#((init_s), INJ^\#(\epsilon_0))$$

où $init_s$ est l'ensemble des couples (p, E_s) où p est un point de programme correspondant à un processus présent dans la configuration initiale et E_s son environnement statique associé.

Le calcul d'une étape de transition se fait en quatre étapes :

- il faut d'abord trouver le triplet des processus qui vont interagir ;
- il faut ensuite vérifier les conditions de synchronisation de la règle ;
- nous calculons ensuite les continuations possibles ;
- il faut alors faire le passage de valeurs pour les variables contenues dans le message reçu. De la même façon, nous lions les variables définies sous l'opérateur ζ avec les variables correspondant au nom et au comportement réel de l'acteur.

Dans la sémantique abstraite pour approximer le flot de contrôle, nous ne prenons pas en compte la suppression des processus. L'objectif d'un tel domaine étant d'approximer les valeurs des variables des processus ainsi que leur marqueur, la présence dans une configuration de tel ou tel processus n'est pas représentable dans ce domaine. La suppression de ces processus n'apporte donc aucune information et provoque des calculs inutiles.

Conjecture 5.3.5 $(\mathcal{C}_{env}^\#, \sqsubseteq_{env}^\#, \cup_{env}^\#, \perp_{env}^\#, \gamma_{env}, C_0^{env}, \rightsquigarrow_{env})$ est une abstraction.

Soit $C^\# \in \mathcal{C}_{env}^\#$ une configuration abstraite,
soit $(3, components, compatibility, v_passing)$ une règle de réduction.
soit $(p_k)_{1 \leq k \leq 3} \in \mathcal{L}_p^3$ un triplet d'étiquettes de points de programme et
 $(pi_k)_{1 \leq k \leq 3} = (s_k, (parameter), (bd), constraints, static_continuation_k)$ un
triplet d'interactions partielles.
nous définissons :

$$mol \triangleq reagents^\#((p_k), (parameter_{k,l}), (constraints_k), C^\#).$$

si

1. $\forall k \in \llbracket 1; n \rrbracket, pi_k \in interaction(p_k)$;
2. $mol \neq \perp_{(I(p_k))_k}$

alors :

$$C \xrightarrow{(p_k)_k} \# \sqcup \{C; mol; new_threads\}$$

où :

- $mol' = marker_value(type(s_1), (p_k)_k, mol, (bd_{k,l})_{k,l}, (parameter_{k,l})_{k,l}, v_passing)$
- $new_threads = launch^\#((p_k, continuations_k)_k, mol')$.

FIG. 5.1 – Sémantique opérationnelle abstraite pour l'analyse de flot de contrôle.

5.3.2 Graphe d'égalité et d'inégalité

Nous définissons maintenant un domaine qui permet de représenter les égalités et les inégalités entre variables. Un élément du domaine T_X est un graphe dont les sommets sont des partitions de X . La présence de deux éléments dans la même partition indique leur égalité tandis que la présence d'un arc entre deux sommets marque l'inégalité entre les variables des partitions associées aux sommets.

Nous définissons deux relations sur le graphe $G = (P, E) \in T_X$:

- $a =_G b \iff \exists C \in P, \{a, b\} \subseteq C$,
- $a \neq_G b \iff \exists C_1, C_2 \in P, a \in C_1, b \in C_2, (C_1, C_2) \in E$.

L'opérateur $[_]_G \in X \rightarrow P$ permet d'obtenir pour chaque élément $x \in X$ la classe d'équivalence dont il fait partie : $[x]_G = \{y \mid x =_G y\}$.

Le domaine T_X est partiellement ordonné par la relation \leq_X^T définie par :

$$\forall G_1, G_2 \in T_X, G_1 \leq_X^T G_2 \iff \begin{cases} \forall x, y \in X, x =_{G_2} y \Rightarrow x =_{G_1} y; \\ \forall x, y \in X, x \neq_{G_2} y \Rightarrow x \neq_{G_1} y. \end{cases}$$

$x \leq_X^T y$ doit refléter que y est moins précis que x , c.-à-d. que y admet plus de propriétés que x . La relation de pré-ordre dans le domaine est donc l'opposée de l'inclusion des ensembles.

Pour chaque ensemble I , la relation de concrétisation $\gamma_{T_X}^I$ associe à un graphe de T_X un ensemble de fonctions $f \in X \rightarrow I$ défini comme suit :

$$\gamma_{T_X}^I(G) = \left\{ f \in X \rightarrow I \mid \forall x, y \in X, \begin{cases} x =_G y \Rightarrow f(x) = f(y) \\ x \neq_G y \Rightarrow f(x) \neq f(y) \end{cases} \right\}$$

1. *union* : nous définissons un opérateur d'union \sqcup_X^T qui associe à un ensemble non vide $A \in \wp(T_X) \setminus \emptyset$ un élément de T_X défini par :

$$\sqcup_X^T A = (p', E')$$

où

$$\begin{cases} P' &= \{ \bigcap_{G \in A} [x]_G \mid x \in X \} \\ E' &= \{ (\bigcap_{G \in A} [x]_G, \bigcap_{G \in A} [y]_G) \in P'^2 \mid \forall G \in A, x \neq_G y \} \end{cases}$$

2. *jointure* : Nous définissons un opérateur de jointure : pour tout graphe $G_X \in T_X, G_Y \in T_Y$ avec X et Y disjoints, $G_X \otimes^T G_Y \in T_{X \cup Y}$ est le graphe (P', E') constitué des sommets de $G_1 = (P_1, E_1)$ et des sommets de $G_2 = (P_2, E_2)$, les arcs présents dans G_1 ou dans G_2 le sont aussi dans $G_X \otimes^T G_Y$:

$$\begin{cases} P' &= P_1 \cup P_2 \\ E' &= E_1 \cup E_2 \end{cases}$$

3. *projection* : La projection $PROJ_Y^T$ sur Y du graphe $G = (P, E) \in T_X$ est le graphe $G = (P', E') \in T_Y$ défini par :

$$\begin{cases} P' = \{C \cap Y \mid C \in P, C \cap Y \neq \emptyset\}, \\ E' = \{(C_1 \cap Y, C_2 \cap Y) \mid (C_1, C_2) \in E\} \cap (P')^2. \end{cases}$$

4. *renommage* : La primitive $RENAME_g(G)$ applique la bijection g à chaque élément des partitions des sommets de G :

$$\begin{cases} P' = \{g(C) \mid C \in P\}, \\ E' = \{(g(C_1), g(C_2)) \mid (C_1, C_2) \in E\}. \end{cases}$$

5. *vacuité* : Le test de vacuité d'un graphe est défini par la primitive $EMPTY_X \in \wp(T_X)$:

$$G = (P, E) \in EMPTY_X \iff \exists C \in P, (C, C) \in E$$

Relations entre valeurs

Les domaines suivants permettent de représenter les relations d'égalité et d'inégalité entre les valeurs associées aux variables de l'interface d'un processus.

Atome Pour chaque interface V , nous définissons le domaine $Atom_V^{\bar{1}} = T_V$. C'est le graphe d'égalité et d'inégalité décrit précédemment, construit sur l'ensemble des variables de l'interface du point de programme.

Nous définissons la fonction de concrétisation par

Soit $g \in \gamma_{T_V}^{\mathcal{L}}(G)$ tel que :

$$\gamma_V^{\bar{1}}(G) = \{(id, E) \mid fst(E(X)) = g(x)\}.$$

Les opérateurs d'inclusion et d'union sont définis par les opérateurs du domaine des graphes. $\subseteq_V^{\bar{1}} = \leq_V^T$, $\cup_V^{\bar{1}} = \cup_V^T$ et $\perp_V^{\bar{1}} = (\{V\}, (\{V\}, \{V\}))$

Il nous reste à définir les primitives $\epsilon^{\bar{1}}$, $\nu^{\bar{1}}$ et $GC^{\bar{1}}$:

- l'environnement initial est défini par le graphe sans arc et sans sommet :

$$\epsilon^{\bar{1}} = (\emptyset, \emptyset)$$

- l'opérateur $\nu^{\bar{1}}$ représente l'insertion d'une nouvelle variable dans l'atome, chaque variable ayant une valeur différente. Nous ajoutons donc un nouveau nœud contenant uniquement la partition constituée du singleton $\{x\}$. Nous marquons aussi explicitement la différence entre la nouvelle variable et toutes les autres en ajoutant un arc entre chaque sommet et ce nouveau sommet.

$$\nu^{\bar{1}}(x, l, (P, E)) = (P \cup \{\{x\}\}, E \cup (\{\{x\}\} \times P))$$

- la projection est défini en utilisant l'opérateur de projection du domaine des graphes :

$$GC^{=1}(X, G) = PROJ_X^T(G)$$

Molécule Soit la famille de molécules $Molecule_{(V_i)_{1 \leq i \leq n}}^{=1}$ définies sur le n-uplet d'interfaces $(V_i)_{1 \leq i \leq n}$ par le graphe d'égalité et d'inégalité construit sur l'ensemble $\{(X, k) \mid X \in V_k\}$.

- la concrétisation d'une molécule M construite sur la séquence (V_i) d'interfaces par la fonction $\gamma_{(V_i)}^{=1}$ est définie par l'ensemble des couples (id_i, E_i) qui vérifient

$$fst(E_i(v)) = g(v, i)$$

La fonction $g : \{(X, k) \mid X \in V_k\} \rightarrow \mathcal{L}$ doit être dans la concrétisation de M par la fonction $\gamma_{T_{i(X,k) \mid X \in V_k}}^{\mathcal{L}}(M)$

- l'injection associe à un atome, une molécule. Il s'agit donc uniquement de renommer les variables x de l'atome, *i.e.* du graphe construit sur X , en $(x, 1)$ *i.e.* la variable x du premier processus de la molécule. Nous utilisons pour ce faire la primitive *RENAME* du domaine des graphes :

$$INJ^{=1}(A) = RENAME_{[X \mapsto (X,1)]}(A)$$

- la concaténation de deux molécules, $G_1 \bullet^{=1} G_2$, est définie comme la jointure de la première molécule avec la seconde dont les indices de processus auront été décalés :

$$G_1 \otimes^T RENAME_{[(X,k) \mapsto (X,k+m)]}(G_2)$$

où G_1 admet m processus.

- la projection consiste à conserver uniquement le processus désiré en utilisant l'opérateur $PROJ^T$ du domaine des graphes. Puis à renommer ensuite les variables du processus pour retirer l'indice du processus dans la molécule :

$$PROJ^{=1}(k, G) = RENAME_{[(X,k) \mapsto X]}(PROJ_{\{(X,k) \mid X \in V_k \cup \{I\}\}}^T(G))$$

- l'extension d'une molécule $G = (P, E)$ par un ensemble de variables se traduit dans ce domaine par la projection du graphe sur l'ensemble des variables qui ne sont pas étendues. Puis à ajouter, pour chaque variable v dans l'ensemble des variables étendues, un noeud associé à la partition $\{v\}$:

$$(P', E') = PROJ_{\{(x,i) \mid x \in V_i\} \setminus X}^T(P, E)$$

$$NEW_{\bar{\tau}}^{-1}(X, G) = (P' \cup \sigma(X), E')$$

où $\sigma(X) = \{\{x\} \mid x \in X\}$.

- la synchronisation d'une molécule avec un ensemble de contraintes S se réalise en quotientant la partition des variables par les contraintes d'égalité de S et en ajoutant les arcs correspondants aux contraintes d'inégalité de S . Nous définissons dans ce but la relation binaire $=_S$ définie comme la fermeture transitive des relations d'égalité de l'ensemble S . La primitive $SYNC^{=1}$ est alors définie par :

$$SYNC^{=1}(S, (p_i), G) = (P', E')$$

où

$$\begin{cases} P' &= \{C_{=S} \mid C \in P\} \\ E' &= \{(X_{=S}, Y_{=S}) \mid (X, Y) \in E\} \\ &\cup \{(X, Y) \in (P')^2 \mid \exists x \in X, y \in Y, x \neq y \in S\} \end{cases}$$

- le calcul de marqueur ne modifie pas ici la molécule puisque les marqueurs sont complètement abstraits dans ce domaine.

$$FETCH^{=1}((p_i), G) = G$$

Proposition 5.3.6 *Pour tout triplet de points de programme (p_i) , pour toute molécule G ,*

$$\left\{ (\bar{id}, E_i) \left| \begin{array}{l} \forall i \in \llbracket 1; n \rrbracket, (id_i, E_i) \in \gamma_{(V_i)}(G), \\ id_i \neq \bar{id} \\ \forall x \in V_i, y \in \mathcal{L}, (y, \bar{id}) \neq E_i(x) \end{array} \right. \right\} \subseteq \gamma_{(V_i)}(FETCH^{=1}((p_i), G))$$

où $\bar{id} = (p_1.id_1)$.

Preuve 5.3.7 *correction de l'opérateur $FETCH^{=1}$* Soit G une molécule, (p_i) un triplet de points de programme. Soit $\gamma_{(V_i)}^{-1}$ l'ensemble des processus représentés par l'abstraction G et définis par $\{(id_i, E_i) \mid fst(E_i(v)) = g(v)\}$ et $g \in \gamma_{T_{\{(X,k) \mid X \in V_k\}}}^{\mathcal{L}}(G)$.

Soit $\{(id'_i, E'_i)\}$ l'ensemble des processus image de $FETCH^{=1}((p_i), G)$ par la fonction $\gamma_{(V_i)}^{-1}$. $\gamma_{(V_i)}(FETCH^{=1}((p_i), G))$ est donc définie par l'ensemble des couples $\{(id'_i, E'_i)\}$ tels que $fst(E'_i(v)) = g'(v, i)$ avec $g' \in \gamma_{T_{\{(X,k) \mid X \in V_k\}}}^{\mathcal{L}}(FETCH^{=1}((p_i), G))$.

Comme $FETCH^{=1}((p_i), G) = G$, nous pouvons choisir $g' = g$. Nous avons donc pour chaque couple (id_i, E_i) de $\gamma_{(V_i)}(G)$, un couple (id'_i, E'_i) dans $\gamma_{(V_i)}(FETCH^{=1}((p_i), G))$. Aucune condition n'est donnée par la fonction de concrétisation sur les marqueurs du processus ou des variables de son environnement. Les conclusions de la section 3.6.2 nous permettent de choisir ces marqueurs de façon à réaliser les conditions de correction de l'opérateur. \square

Relations entre marqueurs

Les domaines suivants permettent de représenter les égalités entre l'ensemble des marqueurs d'un processus : le marqueur du processus ainsi que ceux associés aux variables de son interface.

Atome Pour chaque interface V , nous définissons le domaine $Atom_V^{\equiv} = T_{V \cup \{I\}}$. C'est le graphe d'égalité et d'inégalité décrit précédemment construit sur l'ensemble des variables de l'interface du point de programme ainsi que du marqueur du processus représenté par la variable I .

Nous définissons la fonction de concrétisation par

$$\gamma_V^{\equiv}(G) = \left\{ (id, E) \mid \begin{array}{l} id = g(I) \\ snd(E(v)) = g(v) \end{array} \right\}$$

avec $g : V \cup \{I\} \rightarrow \mathcal{M}$ et $g \in \gamma_{T_{V \cup \{I\}}}^{\mathcal{M}}(G)$.

Les opérateurs d'inclusion et d'union sont définis par les opérateurs du domaine des graphes. $\subseteq_V^{\equiv} = \leq_{V \cup \{I\}}^T$, $\cup_V^{\equiv} = \cup_{V \cup \{I\}}^T$ et $\perp_V^{\equiv} = (V \cup \{I\}, (V \cup \{I\})^2)$

Il nous reste à définir les primitives ϵ^{\equiv} , ν^{\equiv} et GC^{\equiv} :

- l'environnement initial est défini par le graphe sans arc constitué d'un unique sommet contenant la variable I :

$$\epsilon^{\equiv} = (\{\{I\}\}, \emptyset)$$

- l'opérateur ν^{\equiv} représente l'insertion d'une nouvelle variable dans l'atome. Chaque nouvelle variable ayant le même marqueur que l'identité du processus, nous ajoutons donc la variable dans la partition contenant la variable I .

Soit σ la fonction qui permet d'insérer la variable dans la classe d'équivalence du marqueur identité du processus :

$$\sigma(C) = \begin{cases} C & \text{si } I \neq C \\ C \cup \{x\} & \text{sinon} \end{cases}$$

Et ν^{\equiv} est défini par :

$$\nu^{\equiv}(x, l, (P, E)) = (\{\sigma(C) \mid C \in P\}, \{(\sigma(C_1), \sigma(C_2)) \mid (C_1, C_2) \in E\})$$

- la projection est définie en utilisant l'opérateur de projection du domaine des graphes :

$$GC^{\equiv}(X, G) = PROJ_{X \cup \{I\}}^T(G)$$

Molécule Soit la famille de molécules $Molecule_{(V_i)_{1 \leq i \leq n}}^{\bar{=}2}$ définies sur le n-uplet d'interfaces $(V_i)_{1 \leq i \leq n}$ par le graphe d'égalité et d'inégalité construit sur l'ensemble $\{(X, k) \mid X \in V_k \cup \{I\}\}$.

- la concrétisation d'une molécule M construite sur la séquence (V_i) d'interfaces par la fonction $\gamma_{(V_i)}^{\bar{=}2}$ est définie par l'ensemble des couples (id_i, E_i) qui vérifient : il existe $g \in \{(X, k) \mid X \in V_k \cup \{I\}\} \rightarrow \mathcal{M}$ tel que

$$\begin{cases} id_i = g(I, i) \\ E_i(v) = g(v, i) \end{cases}$$

La fonction g doit être dans la concrétisation de M par la fonction $\gamma_{T_{\{(X,k) \mid X \in V_k \cup \{I\}\}}^{\mathcal{M}}}(M)$

- l'injection associée à un atome, une molécule. Il s'agit donc ici uniquement de renommer les variables x de l'atome, *i.e.* du graphe construit sur X , en $(x, 1)$ *i.e.* la variable x du premier processus de la molécule. Nous utilisons pour ce faire la primitive *RENAME* du domaine des graphes :

$$INJ^{\bar{=}2}(A) = RENAME_{[X \mapsto (X,1)]}(A)$$

- la concaténation de deux molécules, $G_1 \bullet^{\bar{=}2} G_2$, est définie comme la jointure de la première molécule avec la seconde dont les indices de processus auront été décalés :

$$G_1 \otimes^T RENAME_{[(X,k) \mapsto (X,k+m)]}(G_2)$$

- la projection consiste à conserver uniquement le processus désiré en utilisant l'opérateur $PROJ^T$ du domaine des graphes. Puis à renommer ensuite les variables du processus pour retirer l'indice du processus dans la molécule :

$$PROJ^{\bar{=}2}(k, G) = RENAME_{[(X,k) \mapsto X]}(PROJ_{\{(X,k) \mid X \in V_k \cup \{I\}\}}^T(G))$$

- l'extension d'une molécule G par un ensemble de variables se traduit dans ce domaine, par la projection du graphe sur l'ensemble des variables qui ne sont pas étendues, puis par l'ajout, pour chaque variable v dans l'ensemble des variables étendues, d'un noeud associé à la partition $\{v\}$:

$$\begin{aligned} (P', E') &= PROJ_{\{(x,i) \mid x \in V_i \cup \{I\}\} \setminus X}^T G \\ NEW_{\bar{=}2}(X, G) &= (P' \cup X, E') \end{aligned}$$

- la synchronisation d'une molécule, avec un ensemble de contraintes S , se réalise en quotientant la partition des variables par les contraintes d'égalité de S . Les inégalités entre les valeurs dans S ne peuvent pas

nous donner d'informations sur les marqueurs associés. Nous définissons la relation binaire $=_S$ définie comme la fermeture transitive des relations d'égalité de l'ensemble S . La primitive $SYNC^{=2}$ est alors définie par :

$$SYNC^{=2}(S, (p_i), G) = (P', E')$$

où

$$\begin{cases} P' &= \{C_{=S} \mid C \in P\} \\ E' &= \{(X_{=S}, Y_{=S}) \mid (X, Y) \in E\} \end{cases}$$

- le calcul de marqueur extrait des classes d'équivalences les marqueurs identité des processus et les réinsère dans une nouvelle classe d'équivalence. Nous marquons ensuite l'inégalité avec les autres classes en insérant un arc vers chaque autre classe. Nous définissons pour cela la fonction σ :

$$\sigma(C) = C \setminus \{(I, 1), (I, 2), (I, 3)\}$$

Nous pouvons maintenant définir $FETCH^{=2}$:

$$FETCH^{=2}((p_i), (P, E)) = (P', E')$$

où

$$\begin{cases} P' &= \{(I, 1), (I, 2), (I, 3)\} \cup (\{\sigma(C) \mid C \in P\} \setminus \{\emptyset\}), \\ E' &= \left\{ \begin{array}{l} (\sigma(C_1), \sigma(C_2)) \mid (C_1, C_2) \in E, \\ C_1, C_2 \in P \setminus \{(I, 1), (I, 2), (I, 3)\} \end{array} \right\} \\ &\cup \{(\{(I, 1), (I, 2), (I, 3)\}, C) \mid C \in P' \setminus \{(I, 1), (I, 2), (I, 3)\}\} \end{cases}$$

Proposition 5.3.8 *Pour tout triplet de points de programme (p_i) et pour tout graphe G :*

$$\left\{ \begin{array}{l} \overline{id}, E_i \mid \forall i \in \llbracket 1; n \rrbracket, (id_i, E_i) \in \gamma_{(V_i)}(G), \\ id_i \neq \overline{id} \\ \forall x \in V_i, y \in \mathcal{L}, (y, \overline{id}) \neq E_i(x) \end{array} \right\} \subseteq \gamma_{(V_i)}(FETCH^{=2}((p_i), G))$$

où $\overline{id} = (p_1.id_1)$.

Preuve 5.3.9 (correction de l'opérateur $FETCH^{=2}$) *Soit G une molécule, (p_i) un triplet de points de programme. Soit $\gamma_{(V_i)}^{=2}$ l'ensemble des processus représentés par l'abstraction G et définis par $\{(id_i, E_i) \mid id_i = g(I), snd(E_i(v)) = g(v, i)\}$ et $g \in \gamma_{T_{\{(X,k) \mid x \in V_k \cup \{I\}\}}}^{\#}(G)$.*

Soit $\{(id'_i, E'_i)\}$ l'ensemble des processus image de $FETCH^{=2}((p_i), G)$ par la fonction $\gamma_{(V_i)}^{=2}$. $\gamma_{(V_i)}(FETCH^{=2}((p_i), G))$ est donc défini par l'ensemble des couples $\{id'_i, E'_i \mid id'_i = g'(I), snd(E'_i(v)) = g'(v, i)\}$ avec $g' \in \gamma_{T_{\{(X,k) \mid x \in V_k \cup \{I\}\}}}^{\#}(FETCH^{=2}((p_i), G))$

Soit g une fonction de $\gamma_{T_{\{(X,k) \mid x \in V_k \cup \{I\}\}}}^{\#}(G)$, construisons $g' \in \gamma_{T_{\{(X,k) \mid x \in V_k \cup \{I\}\}}}^{\#}(FETCH^{=2}((p_i), G))$ telle que g'

satisfasse les critères de correction pour l'opérateur $FETCH^{=2}$ et $\{(id_i, E_i) \mid id_i = g(I), snd(E_i(v)) = g(v, i)\} = \{(id'_i, E'_i) \mid id'_i = g'(I, i), snd(E'_i(v)) = g(v, i)\}$. Nous avons donc pour chaque couple (id_i, E_i) de $\gamma_{(V_i)}(G)$, un couple (id'_i, E'_i) dans $\gamma_{(V_i)}(FETCH^{=2}((p_i), G))$ qui lui correspond. De plus, comme dans les variables $(I, 1)$, $(I, 2)$ et $(I, 3)$ forment une des partitions du graphe G et que cette partition a un arc vers chacune des autres partitions de G , nous avons donc la seconde condition : $\forall i, j, id'_i \neq id'_j$. Les conclusions de la section 3.6.2 nous permettent de choisir la fonction g' qui renvoie la valeur $p_1.id_1$ pour (I, i) . L'opérateur $FETCH^{=2}$ ne modifie pas les informations quant aux autres variables de G . La dernière condition de correction est donc aussi réalisée. \square

5.3.3 Forme des marqueurs

Nous décrivons dans cette partie, un domaine qui permet de représenter la forme des marqueurs associés à un point de programme ou aux variables d'un point de programme.

Domaine des expressions régulières Un automate sur l'alphabet Σ_m est représenté par un n-uplet (i, f, t, b) où $i, f \in \wp(\Sigma_m)$ représentent respectivement l'ensemble des premières et des dernières lettres des mots reconnus par l'automate. $t \in (\Sigma_m \rightarrow \wp(\Sigma_m))$ est une fonction qui, à chaque lettre de Σ_m , associe l'ensemble des ses lettres successeurs. Enfin, $b \in \{0, 1\}$ permet de préciser si l'automate reconnaît le mot vide.

La fonction de concrétisation $\gamma_{\Sigma_m}^{Reg}$ associe donc à chaque élément $(i, f, t, b) \in Reg_{\Sigma_m}$, l'ensemble des mots u construits sur Σ_m vérifiant :

$$\left\{ \begin{array}{l} |u| > 0 \Rightarrow u_1 \in i \\ |u| > 0 \Rightarrow u_{|u|} \in f \\ \forall i \in \llbracket 1; |u| \rrbracket, |u|_{i+1} \in t(u_i) \\ |u| = 0 \Rightarrow b = 1 \end{array} \right.$$

Deux éléments de Reg_{Σ_m} peuvent reconnaître le même langage, tout en ayant des ensembles i et f ainsi qu'une fonction t différentes. En effet, une lettre peut être présente dans i , n'avoir aucune image par t et ne pas être présente dans f . L'automate ne reconnaît donc pas le mot constitué d'une telle lettre. L'opérateur de réduction ρ^{Reg} permet donc d'obtenir une représentation canonique et minimale de l'automate. Pour cela, nous calculons, dans un premier temps, l'ensemble X_1 des lettres atteignables à partir de i par la transition t . Nous calculons ensuite l'ensemble X_2 des lettres accessibles à partir de $X_1 \cap f$ par la transition inverse de t . L'image de (i, f, t, b) par Reg_{Σ_m} est donc $(i \cap X_2, f \cap X_2, t[x \mapsto t(x) \cap X_2], b)$.

Nous munissons le domaine Reg_{Σ_m} d'une structure de treillis complet $(Reg_{\Sigma_m}, \sqsubseteq_{\Sigma_m}^{Reg}, \sqcap_{\Sigma_m}^{Reg}, \sqcup_{\Sigma_m}^{Reg}, \perp_{\Sigma_m}^{Reg}, \top_{\Sigma_m}^{Reg})$

- $\sqsubseteq_{\Sigma_m}^{Reg}, \sqcap_{\Sigma_m}^{Reg}$ et $\sqcup_{\Sigma_m}^{Reg}$ sont définis en appliquant respectivement les opérateurs \subseteq, \cap et \cup aux ensembles i et f ainsi qu'aux images de la fonction t , et en appliquant respectivement \leq, min et max à b . Dans le cas de l'intersection, on applique ensuite systématiquement l'opérateur ρ^{Reg} pour simplifier la forme de l'automate.
- $\perp_{\Sigma_m}^{Reg} = (\emptyset, \emptyset, [\lambda \mapsto \emptyset], 0)$
- $\top_{\Sigma_m}^{Reg} = (\Sigma_m, \Sigma_m, [\lambda \mapsto \Sigma_m], 1)$

Pour manipuler les éléments du domaine, nous sommes amenés à définir aussi les primitives suivantes :

- la primitive $PREFIX_{\Sigma_m}^{Reg}$ permet d'ajouter une lettre au début des mots reconnus par un automate. Ainsi, soit $\{u \mid u \in \gamma_{\Sigma_m}^{Reg}(i, f, t, b)\}$ alors $\{\lambda.u \mid x.u \in \gamma_{\Sigma_m}^{Reg}(PREFIX_{\Sigma_m}^{Reg}((i, f, t, b), \lambda))\}$.

La primitive $PREFIX_{\Sigma_m}^{Reg}((i, f, t, b), \lambda) = (i', f', t', b')$ est définie par :

$$(\{\lambda\}, f \cup \{\lambda \mid b = 1\}, t[\lambda \mapsto t(\lambda) \cup i], 0)$$

- De façon similaire, la primitive $PUSH_{\Sigma_m}^{Reg}$ permet d'ajouter une lettre à la fin des mots reconnus par un automate. $PUSH_{\Sigma_m}^{Reg}((i, f, t, b), \lambda) = (i', f', t', b')$ est définie par :

$$\begin{cases} i' &= \begin{cases} i \cup \{\lambda\} & \text{si } b = 1 \\ i & \text{sinon} \end{cases} \\ f' &= \begin{cases} \{\lambda\} & \text{si } b = 1 \text{ ou si } i \neq \emptyset \\ \emptyset & \text{sinon} \end{cases} \\ t' &= \left[a \mapsto \begin{cases} t(a) \cup \{\lambda\} & \text{si } a \in f \\ t(a) & \text{sinon} \end{cases} \right] \\ b' &= 0 \end{cases}$$

- la primitive $TAIL_{\Sigma_m}^{Reg}((i, f, t, b)) = (i', f', t', b')$ est définie par :

$$\begin{cases} i' &= \bigcup_{a \in i} t(a) \\ f' &= f \\ t' &= t \\ b' &= \begin{cases} 1 & \text{si } i \cap f \neq \emptyset \\ 0 & \text{sinon} \end{cases} \end{cases}$$

Elle permet de retirer la première lettre de l'automate.

- l'automate qui reconnaît le mot vide est défini par la primitive :

$$EMPTY_{\Sigma_m}^{Reg} = (\emptyset, \emptyset, [\lambda \mapsto \emptyset], 1)$$

Atome Pour chaque interface V , nous définissons le domaine $Atom_V^{Forme} = (V \cup \{I\} \rightarrow Reg_{\mathcal{L}})$. L'automate associé à la variable I abstrait la forme des marqueurs du processus tandis que l'automate associé à une variable de

l'interface reconnaît le langage $\{l.m\}$ où (l, m) est le couple valeur marqueur associé à la variable.

Plus formellement, chaque élément f du domaine est associé à l'ensemble :

$$\gamma_V^{Forme}(f) = \gamma_{\mathcal{L}}^{Reg}(f(I)) \times \prod_{v \in V} \{(l, id) \mid l.id \in \gamma_{\mathcal{L}}^{Reg}(f(v))\}$$

Les opérateurs d'inclusion \sqsubseteq_V^{Forme} et d'union \sqcup_V^{Forme} sont définis par les opérateurs $\sqsubseteq_{\mathcal{L}}^{Ref}$ et $\sqcup_{\mathcal{L}}^{Reg}$. De la même façon, \perp_V^{Forme} est défini par $\perp_{\mathcal{L}}^{Reg}$.

- l'environnement correspond à la fonction qui associe à la variable I l'automate qui reconnaît le mot vide :

$$\epsilon^{Forme} = [I \mapsto EMPTY_{\mathcal{L}}^{Reg}]$$

- l'insertion de la variable v avec comme valeur l est modélisée par l'automate qui reconnaît le mot $l.m$ où m est le marqueur du processus. Nous avons donc :

$$\nu^{Forme}(x, l, f) = f[x \mapsto PREFIX_{\mathcal{L}}^{Reg}(l, f(I))]$$

- la projection est définie par :

$$GC^{Forme}(X, f) = f_{|(V \cap X) \cup \{I\}}$$

Molécule La famille de molécules $Molecule_{(V_i)_{1 \leq i \leq n}}^{Forme}$ définies sur le n-uplet d'interfaces $(V_i)_{1 \leq i \leq n}$ par $\{(X, i) \mid 1 \leq i \leq n, X \in V_i \cup \{I\}\} \rightarrow Reg_{\mathcal{L}}$.

Nous associons à chaque élément $f \in Molecule_{(V_i)_{1 \leq i \leq n}}^{Forme}$, l'ensemble $\gamma_{(V_i)}^{Forme}(f)$ défini par :

$$\left\{ (id_i, E_i) \in \prod_{1 \leq i \leq n} Env_{V_i}^{\mathcal{L}} \left| \begin{array}{l} \forall i, \\ id_i \in \gamma_{\mathcal{L}}^{Reg}(f((I, i))) \\ \forall v, E_i(v) = (l, id_v) \implies l.id_v \in \gamma_{\mathcal{L}}^{Reg}(f((v, i))) \end{array} \right. \right\}$$

- l'injection est définie comme le renommage de chaque variable v en $(v, 1)$

$$INJ^{Forme}(f) = [(X, 1) \mapsto f(X)]$$

- la concaténation $f_1 \bullet^{Forme} f_2$ est définie comme suit, où f_1 est une molécule de m processus :

$$\begin{cases} (X, i) \mapsto f_1(X, i) & \text{si } 1 \leq i \leq m \\ (X, i) \mapsto f_2(X, i - m) & \text{si } m + 1 \leq i \leq m + n \end{cases}$$

- l'opérateur de projection restreint simplement les variables aux variables du processus concerné et supprime l'indice du processus :

$$PROJ^{Forme}(k, f) = [v \in V_k \cup \{I\} \mapsto f(v, k)]$$

- l'extension $NEW_{\top}^{Forme}(X, f)$ consiste à remplacer dans f les automates associés aux variables de l'ensemble X par l'automate $\top_{\mathcal{L}}^{Reg}$:

$$NEW_{\top}^{Forme}(X, f) = f[v \in X \mapsto \top_{\mathcal{L}}^{Reg}]$$

- le calcul de la synchronisation d'une molécule par un ensemble de contraintes comme effectué par l'opérateur $SYNC^{Forme}(S, (p_i)_i, f)$ revient à calculer le langage intersection des variables égales par S . Les contraintes pouvant être de la forme $(x, k) = (I, l)$, il nous faut ici comparer d'une part les mots de la forme $l_x.id_x$ qui représentent les valeurs possibles du couple $(l_x, id_x) \in \mathcal{L}_p \times \mathcal{M}$ dans l'environnement d'un processus et, d'autre part, les mots de la forme $p_l.id_l$ où de la même façon $p_l \in \mathcal{L}$ est un point de programme qui représente le l -ème processus et id_l son marqueur. La fonction g joue ce rôle en rajoutant au début des automates associés à (I, k) la lettre constituée du point de programme du k -ème processus.

$$g((p_i), x, k) \triangleq \begin{cases} PREFIX_{\mathcal{L}}^{Reg}(p_k, f(x, k)) & \text{si } x = I, \\ f(x, k) & \text{sinon} \end{cases}$$

La fonction h calcule elle l'intersection des automates associés aux variables égales par la relation $=_S$ qui est construite comme la fermeture transitive des relations d'égalité de S .

$$h((p_i), S, x, k) \triangleq \begin{cases} \cap_{\mathcal{L}}^{Reg} \{g((p_i), y, l) \mid (y, l) \in (x, k)_{=S}\} & \text{si } x \neq I \\ TAIL_{\mathcal{L}}^{Reg}(\cap_{\mathcal{L}}^{Reg} \{g((p_i), y, l) \mid (y, l) \in (x, k)_{=S}\}) & \text{si } x = I \end{cases}$$

Enfin, nous définissons l'opérateur de synchronisation. Pour éviter des calculs inutiles et coûteux, le calcul de l'intersection des automates comme effectué par l'opérateur h n'a lieu que si aucune des variables de la molécule n'est associée à l'automate $\perp_{\mathcal{L}}^{Reg}$:

$$SYNC^{Forme}(S, (p_i), f)(x, k) \triangleq \begin{cases} \perp_{\mathcal{L}}^{Reg} & \text{si } \exists (y, l), g(y, l) = \perp_{\mathcal{L}}^{Reg} \\ h((p_i), S, x, k) & \text{sinon} \end{cases}$$

- le calcul des nouveaux marqueurs consiste, dans de domaine, à rajouter, devant les automates associés aux marqueurs des processus, la lettre du deuxième processus.

$$FETCH^{Forme}((p_i), f) \triangleq f[(I, k) \mapsto PUSH_{\mathcal{L}}^{Reg}(p_1, f(I, 1))]$$

Proposition 5.3.10 *Pour tout triplet (p_i) et toute molécule f :*

$$\left\{ (\bar{id}, E_i) \left| \begin{array}{l} \forall i \in \llbracket 1; n \rrbracket, (id_i, E_i) \in \gamma_{(V_i)}(f), \\ id_i \neq \bar{id} \\ \forall x \in V_i, y \in \mathcal{L}, (y, \bar{id}) \neq E_i(x) \end{array} \right. \right\} \subseteq \gamma_{(V_i)}(FETCH^{Forme}((p_i), f))$$

où $\overline{id} = (p_1.id_1)$.

Preuve 5.3.11 (correction de l'opérateur $FETCH^{Forme}$) Soit f une molécule, (p_i) un triplet de points de programme. Soit γ_V^{Forme} l'ensemble des processus représentés par l'abstraction f et définis par

$$\left\{ (id_i, E_i) \in \prod_{1 \leq i \leq n} Env_{V_i}^{\mathcal{A}} \left| \begin{array}{l} \forall i, \\ id_i \in \gamma_{\mathcal{L}}^{Reg}(f((I, i))) \\ \forall v, E_i(v) = (l, id_v) \implies l.id_v \in \gamma_{\mathcal{L}}^{Reg}(f((v, i))) \end{array} \right. \right\}$$

Soit $\{(id'_i, E'_i)\}$ l'ensemble des processus image de $FETCH^{Forme}((p_i), f)$ par la fonction γ_V^{Forme} . $\gamma_{(V_i)}(FETCH^{Forme}((p_i), f))$ est donc défini par :

$$\left\{ (id_i, E_i) \in \prod_{1 \leq i \leq n} Env_{V_i}^{\mathcal{A}} \left| \begin{array}{l} \forall i, \\ id_i \in \gamma_{\mathcal{L}}^{Reg}(PUSH_{\mathcal{L}}^{Reg}(p_1, f(I, 1))) \\ \forall v, E_i(v) = (l, id_v) \implies l.id_v \in \gamma_{\mathcal{L}}^{Reg}(f((v, i))) \end{array} \right. \right\}$$

Nous avons bien pour chaque couple (id_i, E_i) de $\gamma_{(V_i)}(f)$, un couple (id'_i, E'_i) dans $\gamma_{(V_i)}(FETCH^{Forme}((p_i), f))$ avec $id_i \neq id'_i$, $id'_i = p_1.id_1$ et $E_i = E'_i$. Les marqueurs associés aux valeurs de l'environnement E_i n'ont donc pas été modifiées et, d'après les conclusions de la section 3.6.2, les conditions de correction sont réalisées. □

5.3.4 Relations globales entre variables et marqueurs

Le domaine introduit par Karr en 1976 permet de représenter les relations affines entre les valeurs associées aux variables d'un programme. Un élément du domaine est donc représenté par une matrice dont les colonnes sont indexées par les variables et par un vecteur colonne qui correspond au second membre de l'équation $A \times X = b$. Chaque ligne contient une relation affine entre une ou plusieurs variables. Ce domaine représente donc pour n variables l'enveloppe affine, ou l'espace affine de dimension n des solutions de cette matrice. Karr a ensuite défini l'opérateur d'union sur ce domaine. Il a donc décrit un algorithme permettant de calculer le plus petit espace affine contenant deux autres espaces affines; dit autrement $x \sqsubseteq^{\mathcal{K}} y$ tel que $x \sqsubseteq^{\mathcal{K}} x \sqcup^{\mathcal{K}} y$ et $y \sqsubseteq^{\mathcal{K}} x \sqcup^{\mathcal{K}} y$. Les matrices sont représentées sous une forme normale, nous effectuons d'abord un pivot de GAUSS pour obtenir une matrice triangulaire supérieure, ensuite nous divisons chaque ligne par la première valeur non nulle. Enfin, nous éliminons de chaque ligne les relations qui sont exprimées par des lignes inférieures. Nous obtenons ainsi une forme normale. Il est aisé de définir un opérateur de projection pour conserver uniquement l'information pour certaines variables : il suffit de mettre la matrice en forme normale après avoir interverti les colonnes pour obtenir comme premières colonnes, celles qui correspondent aux variables à supprimer. Nous

pouvons ensuite supprimer ces colonnes sans perdre d'informations sur les autres variables.

Pour représenter les contraintes entre les valeurs et les marqueurs associés aux variables, nous abstrayons chaque mot $m \in \mathcal{M}$ par son vecteur de Parikh pour ensuite représenter dans le domaine de Karr les relations entre les occurrences de chaque lettre.

Atome Pour chaque interface V , nous définissons par \mathcal{K}_V l'ensemble des variables $\{P_\lambda \mid \lambda \in \mathcal{L}\} \cup \{V_{(\lambda,v)} \mid \lambda \in \mathcal{L}, v \in V\} \cup \{B_{l,v} \mid l \in \mathcal{L}, v \in V\}$. Les variables P_λ représentent l'occurrence de la lettre λ dans le marqueur du processus. Les variables $V_{\lambda,v}$ représentent l'occurrence de la lettre λ dans le marqueur associé à la variable v dans l'environnement du processus. Enfin, les variables $B_{l,v}$ permettent de représenter le fait que la variable v prend la valeur l dans l'environnement du processus.

Le domaine $Atom_V^{Glob}$ est défini comme le domaine de Karr construit sur l'ensemble des variables \mathcal{K}_V .

La fonction de concrétisation est définie par :

$$\gamma_V^{Glob}(K) = \left\{ (id, E) \left| \left[\begin{array}{ll} P_\lambda^i & \rightarrow |\psi(id)|_\lambda, \\ V_{(\lambda,v)}^i & \rightarrow |\psi(id_v)|_\lambda \text{ où } (l', id_v) = E(v), \\ B_{(l,v)}^i & \rightarrow \delta_l^{l'} \text{ où } (l', id_v) = E(v). \end{array} \right] \right\}$$

Les opérateurs d'inclusion \sqsubseteq_V^{Glob} et d'union \sqcup_V^{Glob} sont définis par les opérateurs équivalents dans le domaine de Karr.

L'élément \perp_V^{Glob} du domaine est constitué de la matrice contenant la relation $1 = 0$.

- l'environnement initial est décrit par la matrice dans laquelle nous avons inséré uniquement les variables P_λ . Chaque occurrence de la lettre λ dans le marqueur est nulle puisque le marqueur est vide :

$$\epsilon^{Glob} = \{P_\lambda = 0, \forall \lambda \in \mathcal{L}\}$$

- l'insertion d'une variable par l'opérateur $\nu^{Glob}(x, l, K)$ est définie en insérant dans le système les contraintes entre le marqueur associé à la variable x (représenté par les variables $V_{(\lambda,x)}$) et le marqueur du processus (représenté par P_λ) et entre la valeur de la variable représentée par les variables $B_{(\lambda,x)}$ et l :

$$V_{(\lambda,x)} = P_\lambda \quad B_{(\lambda,x)} = \delta_l^{l'}$$

où δ_x^y est le symbole de Kronecker.

- la projection de l'atome sur un ensemble de variables X est définie par l'opérateur $GC^{Glob}(X, A)$ comme la projection du domaine de Karr des variables $\{P_\lambda \mid \lambda \in \mathcal{L}\} \cup \{V_{(\lambda,v)} \mid \lambda \in \mathcal{L}, v \in V \cap X\} \cup \{B_{l,v} \mid l \in \mathcal{L}, v \in V \cap X\}$.

Molécule Nous définissons pour chaque n-uplet d'interfaces de processus V_i , l'ensemble $\mathcal{H}_{(V_i)}$ des variables $\{P_\lambda^i \mid \lambda \in \mathcal{L}\} \cup \{V_{(\lambda,v)}^i \mid \lambda \in \mathcal{L}, v \in V_i\} \cup \{B_{(l,v)}^i \mid l \in \mathcal{L}, v \in V_i\}$.

Le domaine $Molecule^{Glob}$ est défini comme le domaine de Karr sur les variables de $\mathcal{H}_{(V_i)}$.

La fonction de concrétisation est définie par :

$$\gamma_{(V_i)}^{Glob}(K) = \left\{ (id_i, E_i) \left| \left[\begin{array}{ll} P_\lambda^i & \rightarrow |\psi(id_i)|_\lambda, \\ V_{(\lambda,v)}^i & \rightarrow |\psi(id_v)|_\lambda \text{ où } (l', id_v) = E_i(v), \\ B_{(l,v)}^i & \rightarrow \delta_{l'}^i \text{ où } (l', id_v) = E_i(v). \end{array} \right. \right\}$$

- l'injection $INJ^{Glob}(K)$ d'un atome dans la molécule consiste à renommer les variables P_λ en P_λ^i , $V_{(\lambda,v)}$ en $V_{(\lambda,v)}^i$ et $B_{(l,v)}$ en $B_{(l,v)}^i$
- la concaténation $K_1 \bullet^{Glob} K_2$ de deux molécules est obtenue en mettant ensemble les contraintes des deux systèmes affines. Il faut d'abord renommer les variables de K_2 pour tenir compte du nombre de processus de K_1 . Ensuite, il faut insérer les nouvelles contraintes dans la première molécule.
- la projection $PROJ^{Glob}(k, f)$ est définie comme la projection du domaine de Karr pour conserver uniquement les variables P_λ^k , $V_{(\lambda,v)}^k$ et $B_{(\lambda,v)}^k$. Ces variables sont ensuite renommées pour obtenir un atome.
- l'extension $NEW_{\top}^{Glob}(X, A)$ consiste à retirer l'information sur les variables de X de la molécule A . Il s'agit donc, à la manière de la projection, d'échanger les colonnes pour avoir les variables de X à gauche, puis de mettre la matrice en forme normale. Nous pouvons alors retirer les lignes où les variables de X ont des coefficients non nuls sans risquer de perdre de l'information sur les autres variables.
- soit la fonction aff qui associe à chaque couple $(x, k) \in (V_k \cup \{I\}) \times \llbracket 1; n \rrbracket$ les couples de variables suivant :

$$aff(X, \lambda) = \begin{cases} (\delta_\lambda^{pk}, P_\lambda^k) & \text{si } X = (I, k) \\ (B_{(\lambda,v)}^k, V_{(\lambda,v)}^k) & \text{si } X = (v, k) \text{ avec } v \in V_k \end{cases}$$

la synchronisation $SYNC^{Glob}(S, (p_i), K)$ correspond donc à l'insertion dans la molécule décrite par le système affine K d'une contrainte sur la valeur et d'une contrainte sur le marqueur des variables égales par la fermeture transitive des contraintes d'égalité de l'ensemble S . Pour chaque X, Y tels que $aff(X) = (x_1, x_2)$, $aff(Y) = (y_1, y_2)$, nous insérons donc les contraintes $x_1 = y_1$ et $x_2 = y_2$ dans le système K .

- le calcul des marqueurs $FETCH^{Glob}((p_i), K)$ se fait ici en quatre étapes :

1. nous ajoutons, pour chaque $\lambda \in \mathcal{L}$, une variable P_λ^0 au système K et nous ajoutons la contrainte $P_\lambda^0 = P_\lambda^1$;

2. nous utilisons le même mécanisme que NEW_{\top}^{Glob} pour supprimer les informations à propos des variables P_{λ}^1 , P_{λ}^2 et P_{λ}^3 ;
3. nous insérons les contraintes $P_{\lambda}^i = P_{\lambda}^0 + \delta_{p_1}^{\lambda}$ pour tout $i \in \llbracket 1; 3 \rrbracket$;
4. finalement, nous projetons pour supprimer les variables P_{λ}^0 .

Proposition 5.3.12 *Soit un triplet de points de programme (p_i) et un système K :*

$$\left\{ (\bar{id}, E_i) \left| \begin{array}{l} \forall i \in \llbracket 1; n \rrbracket, (id_i, E_i) \in \gamma_{(V_i)}(K), \\ id_i \neq \bar{id} \\ \forall x \in V_i, y \in \mathcal{L}, (y, \bar{id}) \neq E_i(x) \end{array} \right. \right\} \subseteq \gamma_{(V_i)}(FETCH^{Glob}((p_i), K))$$

où $\bar{id} = (p_1.id_1)$.

Preuve 5.3.13 (correction de l'opérateur $FETCH^{Glob}$) *Soit K une molécule, (p_i) un triplet de points de programme. Soit $\gamma_{(V_i)}^{Glob}$ l'ensemble des processus représentés par l'abstraction K et définis par*

$$\left\{ (id_i, E_i) \left| \left[\begin{array}{ll} P_{\lambda}^i & \rightarrow |\psi(id_i)|_{\lambda}, \\ V_{(\lambda, v)}^i & \rightarrow |\psi(id_v)|_{\lambda} \text{ où } (l', id_v) = E_i(v), \\ B_{(l', v)}^i & \rightarrow \delta_{l'}^{l'} \text{ où } (l', id_v) = E_i(v). \end{array} \right. \right. \right\}$$

Soit $\{(id'_i, E'_i)\}$ l'ensemble des processus image de $FETCH^{Glob}((p_i), K)$ par la fonction $\gamma_{(V_i)}^{Glob}$. $\gamma_{(V_i)}(FETCH^{Glob}((p_i), K))$ est donc défini par :

$$\left\{ (id'_i, E'_i) \left| \left[\begin{array}{ll} P_{\lambda}^i & \rightarrow |\psi(id_i)|_{\lambda} + \delta_{p_1}^{\lambda}, \\ V_{(\lambda, v)}^i & \rightarrow |\psi(id_v)|_{\lambda} \text{ où } (l', id_v) = E_i(v), \\ B_{(l', v)}^i & \rightarrow \delta_{l'}^{l'} \text{ où } (l', id_v) = E_i(v). \end{array} \right. \right. \right\}$$

L'opérateur $FETCH^{Glob}$ n'a pas changé la cardinalité du nombre de processus représentés. Nous avons bien pour chaque couple (id_i, E_i) de $\gamma_{(V_i)}(K)$, un couple (id'_i, E'_i) dans $\gamma_{(V_i)}(FETCH^{Glob}((p_i), K))$. Par contre, tous les processus ont vu leur marqueur identité modifié. Pour tout processus (id'_i, E'_i) , $P_{p_1}^i = |\psi(id_1)|_{p_1} + 1$. Alors que le même processus dans f a pour composante p_1 du vecteur de Parikh de son marqueur identité : $|\psi(id_1)|_{p_1}$. Nous avons bien $id_i \neq id'_i$ et $id'_i = p_1.id_1$. D'autre part, aucune autre composante n'a été modifiée et donc $E_i = E'_i$. Les conditions de correction sont réalisées, d'après les conclusions de la section 3.6.2. \square

5.3.5 Produit réduit

Les quatre domaines définis précédemment $=_1$, $=_2$, *Forme* et *Glob* représentent le flot de contrôle du système. Nous utilisons un produit réduit de ces domaines pour construire notre domaine principal. Nous définissons dans ce but, la fonction de réduction ρ qui va être appelée avant et après chaque

transition simulée dans le domaine abstrait. Cette fonction va permettre de raffiner les éléments des domaines qui compose le produit réduit afin d’injecter des informations dans les autres domaines. Nous appelons aussi cette fonction après chaque synchronisation afin de ne calculer que les transitions possibles. Nous construisons donc le produit cartésien 4.3.2 des domaines $=_1$, $=_2$, *Forme* et *Glob*. Un élément de ce domaine est donc un quadruplet d’éléments de ces domaines.

La fonction ρ est donc un endomorphisme de $(Atom_V^{\bar{=}_1} \times Atom_V^{\bar{=}_2} \times Atom_V^{Forme} \times Atom_V^{Glob})$. Elle effectue un ensemble d’actions jusqu’à atteindre un point fixe :

- si deux variables ont des marqueurs différent, elles sont différentes. Pour chaque arc d’inégalité marquée entre deux partitions A et B de l’élément du domaine $=_2$, nous ajoutons un arc dans l’élément du domaine $=_1$ entre deux partitions qui contiennent des éléments de A d’une part et de B de l’autre.
- tous les éléments présents dans la même partition de $=_2$ doivent partager le même marqueur. Nous raffinons donc le marqueur associé à ces variables dans *Forme*. Nous faisons de même pour les valeurs associées aux variables dans *Forme* en utilisant $=_1$.
- enfin, nous utilisons l’élément du domaine *Forme* pour inférer des égalités entre marqueurs pour $=_2$. Plus de détails pourront être trouver à ce sujet dans [Fer05b, sect. 8.3.3.2].

5.4 Domaines pour le dénombrement

5.4.1 Domaine générique

À la manière du domaine pour le flot de contrôle, nous décrivons ici un domaine générique, introduit dans [Fer01], permettant d’observer des propriétés numériques sur les termes analysés et de calculer les transitions. Ce domaine est paramétré par un autre domaine qui représente les propriétés du terme. Soit \mathcal{V}_c l’ensemble constitué de l’ensemble des étiquettes de point de programme et de l’ensemble des étiquettes de transition. Comme dans la section précédente, et comme décrit dans la section 3.6.2, l’ensemble \mathcal{T} des étiquettes de transition est ici une copie de \mathcal{L}_p où l’étiquette p de la transition désigne le second processus, *i.e.* celui qui correspond au comportement de l’acteur qui reçoit le message. Les éléments de \mathcal{L}_p et de \mathcal{T} sont différenciés dans \mathcal{V}_c pour ne pas confondre l’occurrence du processus p avec l’occurrence de p dans les étiquettes de transitions menant à l’état courant.

Le domaine $\mathbb{N}^{\mathcal{V}_c}$

Une première approximation d’un système de processus est donnée par l’abstraction $\Pi_{\mathbb{N}^{\mathcal{V}_c}}$ qui associe à chaque couple $(u, C) \in \Sigma^* \times \mathcal{C}(\mathcal{S})$, la famille

des entiers naturels, indexée par l'ensemble \mathcal{V}_c , définie comme suit :

$$(\Pi_{\mathbb{N}^{\mathcal{V}_c}}(u, C))_v = \begin{cases} \text{Card}(\{(p, id, E) \in C \mid p = v\}) & \text{si } v \in \mathcal{L}_p \\ |\psi(u)|_v & \text{si } v \in \mathcal{T} \end{cases}$$

Nous associons donc à chaque point de programme le nombre de processus de ce point de programme dans la configuration, ainsi que le nombre de fois que la lettre $\lambda \in \mathcal{T} = \mathcal{L}_p$ apparaît dans le mot u .

La concrétisation d'un ensemble d'entiers naturels indexée par un ensemble \mathcal{V}_c est définie comme :

$$\gamma_{\mathbb{N}^{\mathcal{V}_c}} = \{ (u, C) \in \Sigma^* \times \mathcal{C}(\mathcal{S}) \mid \Pi_{\mathbb{N}^{\mathcal{V}_c}}(u, C) \in A^\# \}$$

Le domaine $\mathcal{N}_{\mathcal{V}_c}$

Nous introduisons maintenant le domaine générique construit sur l'ensemble $\mathcal{N}_{\mathcal{V}_c}$ défini comme un pré-ordre sur les familles d'entiers naturels indexées par \mathcal{V}_c . Ce domaine doit définir les primitives suivantes respectant ces propriétés :

1. l'élément $\perp_{\mathcal{N}_{\mathcal{V}_c}}$ doit être tel que $\gamma_{\mathcal{N}_{\mathcal{V}_c}}(\perp_{\mathcal{N}_{\mathcal{V}_c}}) = \emptyset$ et $\forall a \in \mathcal{N}_{\mathcal{V}_c}, \perp_{\mathcal{N}_{\mathcal{V}_c}} \sqsubseteq_{\mathcal{N}_{\mathcal{V}_c}}^\# a$. La concrétisation est donc stricte.
2. l'union de plusieurs éléments doit être supérieure à chacun de ses éléments : $\forall A \in \wp_{finite}(\mathcal{N}_{\mathcal{V}_c}), \cup_{\mathcal{N}_{\mathcal{V}_c}}^\#(A) \in \mathcal{N}_{\mathcal{V}_c}$ et $\forall a \in A, a \sqsubseteq_{\mathcal{N}_{\mathcal{V}_c}}^\# \cup_{\mathcal{N}_{\mathcal{V}_c}}^\#(A)$
3. pour assurer la convergence de l'union, nous définissons un opérateur d'élargissement $\nabla_{\mathcal{N}_{\mathcal{V}_c}}$ qui doit vérifier les propriétés 4.2.1
4. l'addition des deux familles d'entiers effectuée par l'opérateur $+^\#$ doit vérifier $\forall a^\#, b^\# \in \mathcal{N}_{\mathcal{V}_c}, a^\# +^\# b^\# \in \mathcal{N}_{\mathcal{V}_c}$ et $\{(a_v + b_v)_{v \in \mathcal{V}_c} \mid a \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a^\#), b \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(b^\#)\} \subseteq \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a^\# +^\# b^\#)$. Elle doit donc représenter dans l'abstrait le pendant de l'addition dans les entiers
5. la soustraction est définie de façon similaire, elle doit représenter dans l'abstrait le pendant de la soustraction dans les entiers naturels. Elle doit donc vérifier : $\forall a^\#, b^\# \in \mathcal{N}_{\mathcal{V}_c}, (a^\# -^\# b^\#) \in \mathcal{N}_{\mathcal{V}_c}$ et

$$\left\{ (a_v - b_v)_{v \in \mathcal{V}_c} \mid \begin{array}{l} a \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a^\#), b \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(b^\#), \\ \forall v \in \mathcal{V}_c, a_v \geq b_v \end{array} \right\} \subseteq \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a^\# -^\# b^\#)$$

6. la condition de synchronisation impose seulement que les processus participant à l'interaction soient présents dans la configuration : $\forall a^\# \in \mathcal{N}_{\mathcal{V}_c}, t \in \mathbb{N}^{\mathcal{V}_c}, \text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, a^\#) \in \mathcal{N}_{\mathcal{V}_c}$ et $\{a \mid a \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a^\#), a_v \geq t_v, \forall v \in \mathcal{V}_c\} \subseteq \gamma_{\mathcal{N}_{\mathcal{V}_c}}(\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, a^\#))$
7. la primitive $0_{\mathcal{N}_{\mathcal{V}_c}}$ représente la famille d'entiers indexés par \mathcal{V}_c et tous égaux à 0 : $0_{\mathcal{N}_{\mathcal{V}_c}} \in \mathcal{V}_c$ et $(0)_{i \in \mathcal{V}_c} \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(0_{\mathcal{N}_{\mathcal{V}_c}})$

8. la primitive $1_{\mathcal{N}_{\mathcal{V}_c}(v)}$ correspond à la famille d'entiers indexés par \mathcal{V}_c où tous les entiers sont nuls sauf celui indexé par $v : \forall v \in \mathcal{V}_c, 1_{\mathcal{N}_{\mathcal{V}_c}(v)} \in \mathcal{N}_{\mathcal{V}_c}$ et $(\delta_i^v)_{i \in \mathcal{V}_c} \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(1_{\mathcal{N}_{\mathcal{V}_c}(v)})$ avec δ le symbole de Kronecker.

Nous définissons maintenant trois primitives génériques qui utilisent les primitives définies précédemment et vont nous permettre ensuite de définir la sémantique opérationnelle d'un tel domaine :

- la primitive $\Sigma^\#$ permet de calculer la somme d'éléments abstraits. Elle est définie par induction :

$$\Sigma^\# X = \begin{cases} 0_{\mathcal{N}_{\mathcal{V}_c}} & \text{si } X = \emptyset \\ \min(X) +^\# \Sigma^\#(X \setminus \min(X)) & \text{sinon avec } X \\ & \text{totalement ordonné} \end{cases}$$

- la primitive $\chi^\#(V)$ associe 1 à chaque élément de V et 0 sinon :

$$\chi^\#(V) = \Sigma^\#(1_{\mathcal{N}_{\mathcal{V}_c}(v)})_{v \in V}$$

- la primitive $\beta^\#$ permet de représenter dans l'abstrait le démarrage des continuations. Ainsi $\beta^\#$ associe à chaque continuation A , $\chi^\#(A)$ et effectue l'union de toutes les continuations possibles :

$$\beta^\# : \begin{cases} \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L})) & \rightarrow \mathcal{N}_{\mathcal{V}_c} \\ Ct & \mapsto \chi^\#(Ct) \end{cases}$$

Sémantique opérationnelle

Nous utilisons maintenant les primitives définies dans la sous-section précédente pour décrire la sémantique opérationnelle abstraite. C'est à dire la façon dont nous allons simuler dans le domaine abstrait une transition dans le concret.

L'état initial, c'est à dire l'élément abstrait du domaine correspondant au terme CAP que nous cherchons à analyser, est obtenu en exécutant les continuations du terme initial. Soit $C_0 \in \mathcal{C}_{env}$ l'abstraction des états initiaux définie par :

$$C_0^{\mathcal{N}_{\mathcal{V}_c}} = \beta^\#(init_s)$$

où $init_s$ est l'ensemble des couples (p, E_s) où p est un point de programme correspondant à un processus présent dans la configuration initiale et E_s son environnement statique associé.

Le calcul d'une étape de transition se fait en quatre étapes :

- il faut d'abord trouver le triplet des processus qui vont interagir,
- il faut ensuite vérifier la présence des processus dans le système
- nous calculons ensuite les continuations possibles

- il faut enfin ajouter un processus pour chaque processus présent dans les continuations, supprimer les processus acteur et message qui interagissent. Enfin, il faut ajouter le second membre de l'étiquette de transition au système.

Théorème 5.4.1 $(\mathcal{N}_{\mathcal{V}_c}, \sqsubseteq^{\#}_{\mathcal{N}_{\mathcal{V}_c}}, \cup^{\#}_{\mathcal{N}_{\mathcal{V}_c}}, \perp^{\#}_{\mathcal{N}_{\mathcal{V}_c}}, \gamma_{\mathbb{N}^{\mathcal{V}_c}} \circ \gamma_{\mathcal{N}_{\mathcal{V}_c}}, C_0^{\mathcal{N}_{\mathcal{V}_c}}, \rightsquigarrow_{\mathcal{N}_{\mathcal{V}_c}}, \nabla_{\mathcal{N}_{\mathcal{V}_c}})$ est une abstraction.

Preuve 5.4.2 La preuve peut être trouvée dans [Fer05b, App. D]. □

5.4.2 Intervalles

Le premier domaine numérique est le domaine fonctionnel des intervalles. Nous associons à chaque élément de \mathcal{V}_c un intervalle. C'est ce domaine qui va nous permettre d'observer la plupart des propriétés d'intérêt. Les opérations du domaine des intervalles sont définies composantes par composantes.

Le domaine abstrait \mathcal{I} est associé à $\wp(\mathbb{N}^{\mathcal{V}_c})$ par la fonction de concrétisation $\gamma_{\mathcal{I}}$ définie par

$$\gamma_{\mathcal{I}}(f) = \{u \in \mathbb{N}^{\mathcal{V}_c} \mid \forall i \in \mathcal{V}_c, u_i \in f(i)\}$$

- l'union $\cup_{\mathcal{I}}$ est définie composante par composante par l'union des intervalles :

$$f \cup_{\mathcal{I}} g = [x \mapsto f(x) \cup g(x)]$$

$$\text{avec } \llbracket a; b \rrbracket \cup \llbracket c; d \rrbracket = \llbracket \min(a, c); \max(b, d) \rrbracket$$

- le domaine des intervalles n'étant pas de hauteur bornée : il existe une chaîne croissante infinie entre un élément du domaine autre que \top et \top , nous devons définir un opérateur d'élargissement $\nabla_{\mathcal{I}}^n$:

$$[f \nabla_{\mathcal{I}}^n g](x) = f(x) \nabla^n g(x)$$

$$\text{où } \llbracket a; b \rrbracket \nabla_{\mathcal{I}}^n \llbracket c; d \rrbracket = \begin{cases} \llbracket \min(a, c); +\infty \rrbracket & \text{si } d > \max(b, n) \\ \llbracket \min(a, c); \max(b, d) \rrbracket & \text{sinon} \end{cases}$$

Le paramètre n est un argument de l'analyse, plus il est petit plus l'analyse sera rapide. Dans notre cas, $n=0$ suffit puisque les éléments du domaines sont ensuite raffinés par un opérateur de réduction.

- l'opérateur $+$ _{\mathcal{I}} est défini comme l'addition des intervalles associés à chaque élément de \mathcal{V}_c :

$$(f +_{\mathcal{I}} g) = [x \mapsto f(x) + g(x)]$$

- l'opérateur $-$ _{\mathcal{I}} est défini de façon similaire par :

$$(f -_{\mathcal{I}} g) = [x \mapsto f(x) - g(x) \cup \llbracket 0; +\infty \rrbracket]$$

$$\text{où } \llbracket a; b \rrbracket \nabla_{\mathcal{I}}^n \llbracket c; d \rrbracket = \begin{cases} \llbracket \perp \rrbracket & \text{si } \max(a, c) > \min(b, d) \\ \llbracket \max(a, c); \min(b, d) \rrbracket & \text{sinon} \end{cases}$$

- la synchronisation $SYNC_{\mathcal{G}}(f, t)$ vérifie que l'élément abstrait f décrit une configuration comportant au moins les processus décrit par t :

$$SYNC_{\mathcal{G}}(t, f) = [x \mapsto f(x) \cup \llbracket t(x); +\infty \rrbracket]$$

- $0_{\mathcal{G}} = [x \mapsto \llbracket 0; 0 \rrbracket]$
- $1_{\mathcal{G}} = \left[\begin{cases} x \mapsto \llbracket 1; 1 \rrbracket & \text{si } x = v \\ x \mapsto \llbracket 0; 0 \rrbracket & \text{sinon} \end{cases} \right]$

5.4.3 Contraintes numériques globales

Le second domaine \mathcal{K} est un domaine de Karr[Kar76] dont les variables sont les éléments de \mathcal{V}_c . Les éléments de ce domaine sont liés aux éléments du domaine $\wp(\mathbb{N}^{\mathcal{V}_c})$ par la fonction $\gamma_{\mathcal{K}}$ qui associe à chaque système affine son ensemble de solutions.

- l'union $a \cup_{\mathcal{K}} b$ est l'union telle que définie dans le domaine de Karr, c'est le plus petit espace affine qui contient les deux espaces affines a et b .
- Comme le treillis $(\mathcal{K}, \sqsubseteq_{\mathcal{K}})$ est de hauteur bornée, l'opérateur $\nabla_{\mathcal{K}}$ est défini ici comme $\cup_{\mathcal{K}}$.
- l'addition $+_{\mathcal{K}}$ est définie ici par :

$$(O_1 + \overline{H_1}) +_{\mathcal{K}} (O_2 + \overline{H_2}) = (0_1 + {}^v 0_2) + (\overline{H_1} \cup_{\mathcal{K}} \overline{H_2})$$

avec $0_1 + {}^v 0_2$ l'addition des vecteurs.

- la soustraction $-_{\mathcal{K}}$ est définie de façon similaire :

$$(O_1 + \overline{H_1}) -_{\mathcal{K}} (O_2 + \overline{H_2}) = (0_1 - {}^v 0_2) + (\overline{H_1} \cup_{\mathcal{K}} \overline{H_2})$$

avec $0_1 - {}^v 0_2$ la soustraction des vecteurs.

- la synchronisation n'est pas calculée dans ce domaine mais dans celui des intervalles. Nous avons donc

$$SYNC_{\mathcal{K}}(I, K) = K$$

- l'élément $0_{\mathcal{K}}$ est défini comme le système affine sans contraintes

$$0_{\mathcal{K}} = \{x = 0, \forall x \in \mathcal{V}_c\}$$

- la primitive $1_{\mathcal{K}}(v)$ est définie par le système qui associe 1 à v et 0 aux autres variables de \mathcal{V}_c :

$$1_{\mathcal{K}}(v) = \begin{cases} x = 1 & \text{si } x = v \\ x = 0 & \text{sinon} \end{cases}$$

5.4.4 Produit réduit

Nous utilisons ensuite un produit réduit pour construire notre domaine de dénombrement de processus. Comme dans la section 5.3.5, nous construisons ici notre domaine de dénombrement comme le réduit du produit cartésien des domaines définis précédemment, à savoir, \mathcal{I} et \mathcal{K} . Soit ρ la fonction de réduction.

La concrétisation $\gamma_{\mathcal{N}_{\mathcal{V}_c}}(i, k)$ est définie comme l'intersection des images de i et k par leur fonction de concrétisation respective : $\gamma_{\mathcal{I}}(i) \cap \gamma_{\mathcal{K}}(k)$.

La synchronisation est définie par

$$SYNC_{\mathcal{N}_{\mathcal{V}_c}}(A, (i, k)) = \rho(i', k)$$

$$\text{où } \begin{cases} i'(x) = i(x) \cap \llbracket 1; +\infty[& \forall x \in A \\ i'(x) = i(x) & \forall x \in \mathcal{L}_p \setminus A \end{cases}$$

Enfin, les primitives autres que celles décrites ci-dessus sont définies composantes par composantes.

La réduction ρ doit vérifier le critère de correction. Elle réalise le raffinement d'une part des intervalles associés aux variables de \mathcal{V}_c et l'insertion de contraintes dans le système affine de \mathcal{K} .

La réduction s'effectue en trois parties, après avoir supprimé les formes indéterminées de la matrice, c.-à-d. les variables apparaissant avec un coefficient positif dans une contrainte et un coefficient négatif dans une autre, nous raffinons les intervalles infinis de \mathcal{I} en utilisant \mathcal{K} puis nous raffinons les intervalles finis en appliquant l'algorithme suivant :

- trianguler la matrice par élimination gaussienne;
- Pour chaque variable pivot (première variable non nulle de chaque ligne) :
 - ★ remplacer la borne supérieur de l'intervalle associé à la variable par l'intervalle obtenu en utilisant la contrainte affine et les bornes inférieures des variables de la contrainte;
 - ★ supprimer la variable pivot en remplaçant la contrainte affine par une contrainte dont le second membre est un intervalle;
- Parcourir alors les variables dans le sens inverse de l'ordre précédent. Pour chaque variable :
 - ★ remplacer dans \mathcal{I} l'intervalle associé à la variable par l'intervalle obtenu;
 - ★ remplacer dans les contraintes associées à des intervalles la variable par l'intervalle obtenu;
- Enfin, insérer dans le système linéaire les contraintes $x = l$ pour chaque variable x associée dans \mathcal{I} à un intervalle de la forme $\llbracket l; l \rrbracket$.

Soit $C^\# \in \mathcal{C}_{env}^\#$ une configuration abstraite,
soit $(3, components, compatibility, v_passing)$ une règle de réduction,
soit $(p_k)_{1 \leq k \leq 3} \in \mathcal{L}_p^3$ un triplet d'étiquettes de points de programme et
 $(pi_k)_{1 \leq k \leq 3} = (s_k, (parameter), (bd), constraints, continuation_k)$ un triplet
d'interactions partielles,
nous définissons par $t \in \mathbb{N}^c$ le n-uplet tel que t_v soit le nombre d'occurrences
de v dans la séquence $(p_k)_{1 \leq k \leq 3}$.

Si

1. $\forall k \in \llbracket 1; 3 \rrbracket, pi_k \in interaction(p_k)$;
2. $SYNC_{\mathcal{N}_{\mathcal{V}_c}}(t, C^\#) \neq \perp_{\mathcal{N}_{\mathcal{V}_c}}$

alors :

$$C \xrightarrow{(p_k)_k} \#SYNC_{\mathcal{N}_{\mathcal{V}_c}}(t, C^\#) +\# Transition +\# Launched -\# Consumed$$

où :

- $Transition = 1_{\mathcal{N}_{\mathcal{V}_c}}(p_1)$;
- $Launched = \Sigma^\# ((\beta^\#(continuation_k))_k)$;
- $Consumed = \Sigma^\# (1_{\mathcal{N}_{\mathcal{V}_c}}(p_k))_{2 \leq k \leq 3}$

FIG. 5.2 – Sémantique opérationnelle abstraite pour l'analyse de dénombrement.

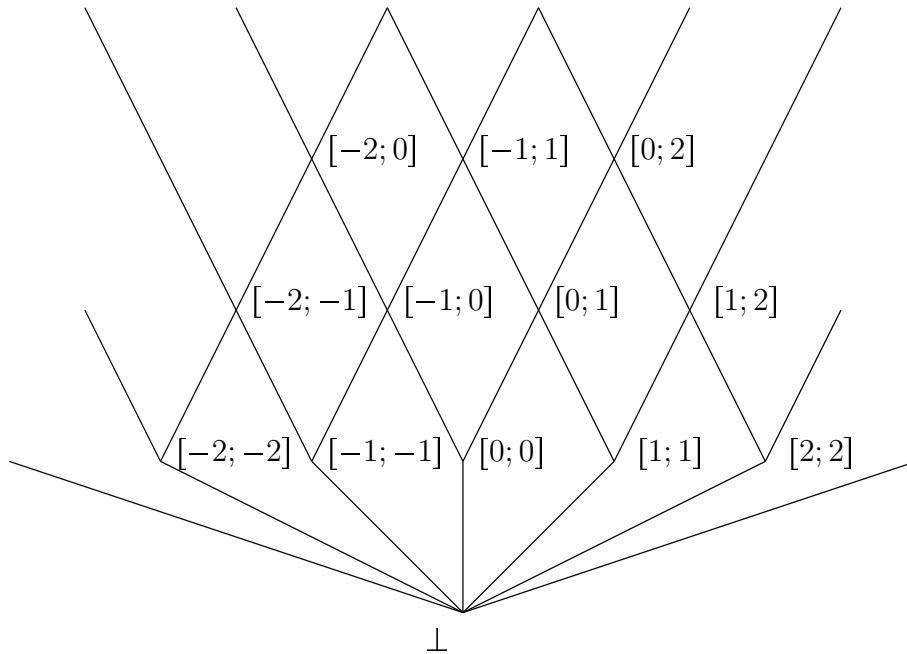


FIG. 5.3 – Le treillis des intervalles de \mathbb{N}^2

Chapitre 6

PAC-SA

Nous avons réalisé l'analyse exprimée dans le chapitre 5 par les domaines de flot de contrôle et de dénombrement dans un analyseur écrit en OCaml. Nous avons développé notre analyseur en utilisant le mécanisme des foncteurs de OCaml pour permettre plus de souplesse dans l'intégration de futurs domaines. La réalisation a duré plus de deux mois. Elle est nécessaire pour traiter des exemples de taille plus significative, mesurer la qualité de l'approximation réalisée en fonction des domaines utilisés. De plus, le co-développement des constructions formelles et pratiques permet une meilleure compréhension de la sémantique des domaines.

Notre programme est donc constitué de trois parties :

- une première partie qui prend un terme de CAP et l'étiquette de façon automatique. La syntaxe acceptée par le programme est la syntaxe telle que définie dans le chapitre 2 dans laquelle ν devient $\tilde{\nu}$, ζ devient $\$$ et \emptyset , le terme vide devient \mathcal{L} . L'ensemble des interactions partielles et des interfaces (ensemble de variables) associées à chaque point de programme sont évaluées dans cette partie. Nous déterminons aussi l'ensemble des comportements définis de façon syntaxique. Enfin, nous calculons l'ensemble des processus présents dans l'état initial. dans la méta-syntaxe. *[1000 loc]*.
- une seconde partie consiste en l'itérateur abstrait, le cœur de notre analyseur. L'itérateur abstrait projette l'état initial dans le domaine abstrait par lequel il est paramétré. Il calcule statiquement l'ensemble des triplets (a, b, m) de transitions possibles en se basant uniquement sur l'égalité d'étiquette de message entre le processus message et le processus comportement, ceci pour éviter des calculs inutiles manipulant un élément abstrait. Il parcourt ensuite cette liste de transition en simulant les transitions dans l'abstrait jusqu'à atteindre le point fixe de l'analyse.
Plus précisément, nous avons choisi d'étiqueter chaque élément de la

liste de transition par un drapeau indiquant si le point fixe a été atteint pour les éléments de la transition. Lorsqu’une transition est effectuée dans l’abstrait, si elle est possible, nous comparons l’itéré, c.-à-d. l’élément abstrait image par la transition de l’élément abstrait courant, avec son antécédent par la fonction de transition abstraite. Si il ne lui est pas inférieur ou égal, nous indiquons dans la liste des transitions que toutes les transitions qui contiennent un des points de programme correspondant à un processus présent dans les continuations de la transition n’ont pas atteint le point fixe. Et nous répétons cette transition jusqu’à atteindre un point fixe local (c’est à dire pour la transition considérée). Au bout de n itérations, si le point fixe n’a pas été atteint, nous utilisons alors l’opérateur d’élargissement pour accélérer la convergence. Nous passons alors à la prochaine transition de la liste dont l’étiquette stipule qu’elle n’a pas atteint le point fixe. Lorsque nous atteignons la fin de la liste des étiquettes de transitions, nous reparcourons cette liste depuis le début en ne considérant que les éléments n’ayant pas le drapeau “point fixe atteint”.

Comme l’élément abstrait courant croît entre chaque série d’itération pour chaque transition, au bout d’un temps fini, nous atteignons l’élément abstrait post point fixe de la sémantique du terme analysé. [350 *loc*].

- enfin une troisième et dernière partie du programme consiste en la réalisation des différents domaines décrits précédemment. Nous avons donc définis les signatures pour les domaines de flot de contrôle et pour les domaines de dénombrement que doivent respecter les domaines implantés. Et de la même façon, pour les domaines de flot de contrôle, nous avons défini la signature qu’un module doit respecter pour planter un atome ou une molécule. [4500 *loc*].

Un soin particulier a été consacré à la réalisation du domaine des égalités affines de Karr [Kar76] et à celle du domaine des graphes d’égalité et d’inégalité.

Nous avons implanté le premier en tenant compte des remarques faites dans [Fer99]. Une matrice est donc représentée sous forme creuse par une liste de bloc. Il est donc aisé de permuter tel ou tel ensemble de colonnes pour ensuite projeter la matrice sur un sous-espace en normalisant la matrice pour conserver l’ensemble des contraintes contenues dans les colonnes que nous voulons supprimer. Les valeurs de la matrice sont représentées par des entiers rationnels de précision arbitraire en utilisant le module Num de OCaml.

Le second domaine a été construit en utilisant la librairie OCamlGraph développée par l’équipe DEMONS du LRI et permettant de manipuler des graphes en utilisant des structures de donnée impératives ou fonctionnelles. Nous avons choisi ici de représenter nos graphes par une structure persistante.

Les domaines de flot de contrôle d'égalité et d'inégalité entre marqueurs et entre valeurs ainsi que le domaine de contraintes globales entre marqueurs et valeurs et le domaines de contraintes numériques globales sont donc implantés comme des foncteurs paramétrés par la représentation de, respectivement, un graphe d'égalité et d'inégalité et une matrice.

Chapitre 7

Propriétés

Nous décrivons dans ce chapitre les propriétés que nous cherchons à étudier sur des termes de CAP. Pour chaque propriété, nous donnerons un exemple de la propriété réalisée ainsi qu'un contre-exemple.

7.1 Linéarité

La linéarité est une propriété qui exprime le fait que tous les acteurs présents simultanément dans une configuration donnée et dans ses évolutions sont “installés” sur des adresses distinctes deux à deux, sur des noms liés par des opérateurs ν distincts ou leurs instances récursives. Elle permet de représenter les adresses des acteurs comme des ressources. Cette propriété est parfois observable très aisément, parfois moins. Dans l'exemple 2.7 page 12, nous voyons aisément que lorsque le terme ne peut plus être réduit, il y a dans le système deux acteurs sur l'adresse a . Par contre, la configuration décrite page 13 par l'exemple 2.9 est plus complexe, et, seule une analyse précise peut montrer que tous les acteurs présents dans la configuration ainsi que ceux qui vont être créés par la réduction du terme initial respecteront la propriété de linéarité.

L'utilisation des domaines de dénombrement suffit pour observer cette propriété. Si tous les acteurs du terme sont associés soit à l'intervalle $\llbracket 0; 1 \rrbracket$ s'ils sont présents dans le système dans l'état initial, soit à l'intervalle $\llbracket 1; 1 \rrbracket$ sinon, et si les acteurs qui peuvent être associés aux mêmes noms avec les mêmes marqueurs sont liés par des contraintes d'exclusion mutuelle dans le domaine des contraintes numériques globales de la forme $\sum_{i=1\dots n} p_i = 1$ où les variables p_i représentent les étiquettes des points de programme, alors le terme vérifie la propriété de linéarité.

7.2 File d'attente bornée

Dans CAP, lorsqu'il y a un envoi de message, les messages sont dans le système. Contrairement aux langages synchrones où le passage de paramètre s'effectue en une seule étape, il se déroule ici en deux temps : l'envoi du message d'abord, lorsqu'il fait partie d'une continuation ; et la réception ensuite par un acteur d'un message présent dans le système, soit parce qu'il a été envoyé, soit parce qu'il était présent dans la configuration initiale. Il peut être intéressant d'étudier si une configuration donnée peut diverger, c.-à-d. si le terme peut lorsqu'il est réduit générer plus de messages qu'il ne pourra en traiter par la suite. Nous voulons donc essayer de borner la file d'attente du système.

Par exemple, le terme suivant génère à chaque réduction deux messages $m()$ mais n'en traite qu'un seul. Il y a donc au bout de n itérations, n messages $m()$ dans le système :

$$\nu a^\alpha, a \triangleright^1 [m^2() = \zeta(e, s)(e \triangleright^3 s \parallel a \triangleleft^4 m() \parallel a \triangleleft^5 m())] \parallel a \triangleleft^6 m()$$

Par contre, dans la configuration décrite par le terme

$$\nu a^\alpha, a \triangleright^1 [m^2() = \zeta(e, s)(e \triangleright^3 s \parallel a \triangleleft^4 m())] \parallel a \triangleleft^6 m()$$

bien que le terme ait une dérivation infinie, et donc qu'il génère une infinité de messages $m()$, il y a dans le système au plus un message $m()$.

Un fois un terme analysé, nous pouvons observer l'élément plus petit point fixe de la sémantique collectrice du terme. En particulier, si le terme admet la propriétés suivante, alors il a une file d'attente bornée : pour chaque $l \in \mathcal{V}_c$ correspondant à un processus :

- soit l'intervalle associé à l est borné ;
- soit il y a, dans le domaine des contraintes numériques globales, une contrainte de la forme $l = l'$ avec $l' \in \mathcal{V}_c$ correspondant à une étiquette de transition.

Ainsi même si le processus apparaît infiniment souvent, il disparaît autant de fois.

7.3 Comportements inaccessibles

Un ensemble de comportements peut contenir des comportements qui ne seront jamais utilisés. Mais par contre, la notion d'ordre supérieure induite par le passage de comportement permet à certains comportements d'être utilisés, non pas sur l'acteur sur lequel ils sont définis syntaxiquement, mais dans d'autres acteurs auxquels ils sont passés en paramètre.

Ainsi dans le terme suivant, le comportement $m_2()$ est utilisé seulement dans l'acteur c

$$\begin{aligned}
& \nu a^\alpha, b^\beta, c^\gamma, \\
& a \triangleright^1 [\\
& \quad m_0^2() = \zeta(e, s)(b \triangleleft^3 n_1(s) \parallel b \triangleleft^4 m_1(c)), \\
& \quad m_1^5(dest) = \zeta(e, s)(dest \triangleleft^6 m_2()), \\
& \quad m_2^7() = \zeta(e, s)(\emptyset) \\
&] \parallel \\
& b \triangleright^8 [n_1^9(self) = \zeta(e, s)(e \triangleright^{10} self \parallel c \triangleleft^{11} n_2(self))] \parallel \\
& c \triangleright^{12} [n_2^{13}(self) = \zeta(e, s)(e \triangleright^{14} self)] \parallel \\
& a \triangleleft^{15} m_0()
\end{aligned}$$

L'analyse du terme avec un flot de contrôle précis permet de détecter de façon précise les utilisations de chaque comportement. Nous pouvons donc détecter les comportements qui ne seront jamais utilisés et donc alléger le terme en supprimant ces comportements. Nous observons pour cela l'élément du domaine des intervalles du plus petit point fixe de l'analyse. Tous les comportements inaccessibles y seront associé avec un intervalle de la forme $\llbracket 0; 0 \rrbracket$.

7.4 Messages orphelins

Un orphelin est un message qui ne peut pas être pris en compte dans un chemin d'exécution. Nous distinguons deux genres de messages orphelins :

- les orphelins de *sûreté* : ce sont les messages qui n'appartiennent pas ou plus au potentiel de traitement de l'acteur. L'acteur n'adoptera plus, dans chacune de ses dérivations possibles, un comportement permettant de prendre en compte un tel message ;
- les orphelins de *vivacité* : ce sont les messages qui peuvent être traités par un des futurs comportements de l'acteur, mais l'évolution du programme est telle que l'acteur n'adoptera jamais ce comportement. Par exemple, pour atteindre ce comportement, il faut simplement recevoir un message qui n'est jamais envoyé à l'acteur.

Il existe plusieurs analyses, dans la bibliographie dont certaines développées au sein de notre équipe, permettant de détecter statiquement certains messages orphelins de sûreté et d'annoter les termes pour permettre lors de l'exécution d'éliminer certains messages orphelins de sûreté. Par contre, il n'y a actuellement aucune analyse permettant de traiter les messages orphelins de vivacité.

Pour l'instant, notre analyse ne permet pas d'identifier tous les messages orphelins d'un terme. Nous pouvons certainement détecter certains orphelins triviaux en faisant un test d'accessibilité sur une continuation factice des processus messages, mais une telle analyse n'est pas implanté pour l'instant dans notre analyseur.

Conclusion

Bilan des contributions

Dans ce mémoire, nous avons représenté le langage CAP dans le cadre générique d'analyse de systèmes mobiles décrit dans [Fer05b]. Nous avons exprimé dans ce cadre le passage de comportement dû à la notion d'ordre supérieur de CAP. Nous avons ainsi permis d'exprimer dans ce système un langage de second ordre.

Notre approche a permis de représenter de façon exacte le comportement d'un système mobile décrit en CAP. Notre sémantique fusionne pour l'instant les instances récursives des mêmes processus et ne permet pas de déterminer de façon systématique quel acteur a créé quel élément et à quel moment.

L'analyse d'un terme exprimé dans une telle sémantique a été ensuite réalisée, en utilisant le cadre de l'interprétation abstraite, pour sur-approximer de façon correcte la sémantique collectrice du terme et en observer les propriétés numériques sur l'occurrence des différents éléments qui le constituent.

Nous avons utilisé pour cela le produit réduit de deux domaines définis dans [Fer05b] pour approximer, d'une part, le flot de contrôle du terme et d'autre part, ses propriétés numériques.

Nous avons pu ainsi observer, dans l'élément abstrait plus petit point fixe de l'analyse, des propriétés essentiellement numériques sur les configurations que peut prendre le terme analysé comme la linéarité des configurations possibles, borner le nombre d'éléments présents au même moment dans le système ou détecter les comportements inaccessibles.

Cette analyse a été implantée dans un prototype en OCaml qui, étant donné un terme en CAP, le traduit en un élément abstrait correspondant et calcule ensuite l'élément abstrait post-point fixe de l'analyse.

Perspectives

Nous voulons différencier les instances récursives du même processus, nous envisageons donc d'utiliser un domaine supplémentaire pour partitionner les propriétés des processus par rapport à leur marqueurs.

Nous voulons pouvoir observer plus de propriétés sur le langage CAP, en

particulier, la détection de messages orphelins. Pour ce faire, nous voulons définir une analyse en arrière du terme pour pouvoir approximer l'ensemble des comportements futurs des différents acteurs.

D'autre part, notre prototype est fonctionnel mais peu performant, nous envisageons d'améliorer la représentation et la manipulation des domaines les plus coûteux.

Le langage CAP tel que défini actuellement permet à un acteur de décrire par une variable l'ensemble des comportements qui le caractérisent. Il peut ensuite passer en paramètre d'un message cet ensemble de comportements. La variable, contenant la valeur désignant cet ensemble de comportements, peut ensuite être passée d'acteurs en acteurs pour être ensuite utilisée, soit pour insérer dans le système un nouvel acteur sur ce comportement, soit pour remplacer le comportement d'un acteur déjà présent. Ce mécanisme ne permet pas de construire de nouveau comportement, mais seulement de récupérer ceux qui sont définis syntaxiquement dans le terme. Puisque notre analyse de flot de contrôle est précise, nous envisageons d'étendre la syntaxe et la sémantique de CAP pour pouvoir construire de façon dynamique des ensembles de comportement avec des opérateurs ensemblistes.

Enfin nous voudrions analyser la partie concurrente de langages réels comme le langage JAVA^{ACT}, une API de programmation proche de CAP pour le langage JAVA.

Bibliographie

- [Agh92] Gul Agha. Hal : A high-level actor language and its distributed implementation. In *21st International Conference on Parallel Processing (ICPP'92)*, volume 2, pages 158–165, August 1992.
- [AH87] Gul Agha and Carl Hewitt. Actors : A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. 1987.
- [AMST92] Gul Agha, Ian. A. Mason, Scott Smith, and Carolyn L. Talcott. Towards a theory of actor computation. In Springer-Verlag, editor, *Concur '92*, volume 630 of *LNCS*, pages 565–579, 1992.
- [CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
- [CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.
- [CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4) :511–547, August 1992.
- [CPDS99] Jean-Louis Colaço, Marc Pantel, Fabien Dagnat, and Patrick Sallé. Static safety analysis for non-uniform service availability in Actors . In *Formal Methods for Open Object-based Distributed Systems , Florence*, pages 371–386, Pays-Bas, 15-18 février 1999. Kluwer Academic Publisher.
- [CPS96] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. An actor dedicated process calculus. In *ECOOP'96 Workshop on Proof Theory of Concurrent Object-Oriented Programming*, -, page? -, mai 1996.
- [CPS97a] Jean-Louis Colaco, Marc Pantel, and Patrick Sallé. A set-constraint-based analysis of actors. In *Proceedings of the 1997*

- IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, -, pages 107–122. Chapman and Hall, juillet 1997.
- [CPS97b] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. Analyse de linéarité par typage dans un calcul d’acteurs. In *Actes des Journées Francophones des Langages Applicatifs*, -, page? -, janvier 1997.
- [CPS98] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. From set based to multiset based analysis : A practical approach . In *Workshop on set constraints and set based analysis* , Pise, pages 1–10. Pise Univ., 28 octobre 1998.
- [CPS00] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. Static analysis of behavior changes in Actor languages. In Jean-Paul Bartsch, Takanobu Baba, Jean-Pierre Briot, and Akinori Yonezawa, editors, *Object-Oriented Parallel and Distributed Programming*, pages 53–72. Hermès Science, 8, quai du Marché-Neuf, 75004 Paris, France, janvier 2000.
- [Dal99] Silvano Dal Zilio. *Le calcul bleu : types et objets*. PhD thesis, Université de Nice - Sophia-Antipolis, 1999.
- [Fer99] Jérôme Feret. Conception de π -sa : un analyseur statique générique pour le π -calcul. Master’s thesis, École polytechnique, Paris, France, september 1999.
- [Fer01] Jérôme Feret. Occurrence counting analysis for the pi-calculus. *Electronic Notes in Theoretical Computer Science*, 39.2, 2001. Workshop on GEometry and Topology in COncurrency theory, PennState, USA, August 21, 2000.
- [Fer02] Jérôme Feret. Dependency analysis of mobile systems. In *European Symposium on Programming (ESOP’02)*, number 2305 in LNCS. Springer-Verlag, 2002. © Springer-Verlag.
- [Fer05a] Jérôme Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming*, 63.1, 2005. special issue on pi-calculus, 2005.
- [Fer05b] Jérôme Feret. *Analyse des systèmes mobiles par interprétation abstraite*. PhD thesis, École polytechnique, Paris, France, february 2005.
- [FGL⁺96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR’96)*, pages 406–421. Springer-Verlag, 1996.
- [HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

- [Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6 :133 – 151, 1976.
- [KNY95] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *Second International Static Analysis Symposium (SAS'95)*, volume 983, pages 225–242. Springer-Verlag, 1995.
- [PS93] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proceedings 8th IEEE Logics in Computer Science*, pages 376–385, Montreal, Canada, 1993.
- [RV97] António Ravara and Vasco T. Vasconcelos. Behavioral types in a calculus of concurrent objects. In *Euro-Par'97*, volume 1300 of *LNCS*, pages 54–561. Springer Verlag, 1997.
- [Ven98] Arnaud Venet. *Static Analysis of Dynamic Graph Structures in Untyped Languages*. PhD thesis, École polytechnique, Paris, France, december 1998.
- [VT93] Vasco Thudichum Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *ISOTAS*, pages 460–474, 1993.