Mémoire de D.E.A.
Laboratoire d'Informatique
de l'École Polytechnique

Conception de π-*sa* : un analyseur statique générique pour le π-calcul

---

Jérôme Feret

---

sous la direction de :

Radhia Cousot
Arnaud Venet

---

soutenu le 14 septembre 1999

Table des matières

1 Résumé

Le π-calcul est un formalisme permettant de décrire des réseaux de processus communicants. Il peut aussi bien servir à représenter un réseau téléphonique, qu'un protocole d'échange de clefs. Cependant, l'analyse des termes du π-calcul n'est que très peu développée. Pourtant, une analyse des termes du π-calcul devrait permettre de montrer la validité de certaines contraintes. Dans le cas d'un réseau, on voudrait s'assurer que les ressources physiques seront suffisantes, pour simuler le réseau virtuel, dans le cas d'un protocole, on voudrait prouver des contraintes de sécurité, et assurer que certains canaux ne communiquent pas entre eux.

Pour différentes raisons, ces objectifs ne peuvent être atteints en gardant telle quelle la sémantique opérationnelle traditionnelle du π-calcul, car celle-ci met en jeu une infinité de processus, et utilise le procédé arbitraire d'α-conversion pour nommer les nouveaux canaux. On définira donc une sémantique non-standard paresseuse, où d'une part, les processus sont dupliqués à la demande, et d'autre part, le nom des nouveaux canaux est bien déterminé.

On utilisera ensuite l'interprétation abstraite pour établir un analyseur générique des termes du π-calcul, que l'on instanciera ensuite, selon les contraintes de communication que l'on veut prendre en compte.

Pour finir, on s'interrogera sur la possibilité d'analyser un sous-processus indépendamment du reste de son terme principal en vue d'une analyse modulaire.

2 Présentation du π-calcul

2.1 Motivation

Le π-calcul est un formalisme servant à représenter des réseaux de processus qui peuvent communiquer entre eux. Le π-calcul est cependant beaucoup plus expressif que CCS [Mil91, MPW92] et a plus d'applications concrètes.

Tout comme CCS, le π-calcul repose sur l'utilisation de processus et de canaux. Cependant, dans le cas du π-calcul, les canaux peuvent communiquer entre eux des vecteurs de canaux. Le résultat de ces communications s'obtient, comme dans le cas du λ-calcul par substitution. D'autre part, le π-calcul, dispose d'un mécanisme de réplication, qui met en concurrence un nombre arbitraire de processus. Ceci permet, par exemple, de dupliquer un processus avant son exécution. Chacun de ces processus, pourra en outre déclarer ses propres variables.

Muni de ces outils, le π-calcul peut aussi bien représenter des réseaux téléphoniques que des protocoles d'échange de clefs. On peut, en outre, abstraire des programmes parallèles réels et réprésenter uniquement l'interaction de leurs processus. Des exemples seront donnés une fois la syntaxe du π-calcul donnée.

2.2 Sémantique traditionnelle

2.2.1 Syntaxe

Le π-calcul met en jeu des processus. On dispose de plusieurs opérateurs pour composer les processus entre eux. Deux processus placés en exécution concurrente peuvent effectuer leurs propres transitions indépendamment l'un de l'autre et communiquer entre eux des messages. Lorsque deux processus sont placés en exécution disjonctive, un seul de ces processus est exécuté, le choix du processus étant non-déterministe. Enfin, on peut mettre une infinité de processus semblables en exécution concurrente, grâce à un opérateur de réplication.

La syntaxe des processus du π-calcul est donnée figure 2.1.

Figure 2.1 Syntaxe

P	::=	action.P	(action)
	|	$(P \mid P)$	(exécution concurrente)
	|	$(P + P)$	(exécution disjonctive)
	|	$*P$	(réplication)
	|	$\emptyset$	(fin d'un processus)

Les processus ne manipulent qu'un seul type de variables, appelées canaux. Les canaux ne servent pas juste, comme dans CCS, à synchroniser les processus. En effet, lors des communications, les canaux échangent des noms de canaux, et transforment les processus en conséquence.

On se donne *Channel*, un ensemble infini dénombrable de noms de canaux.

Par la suite, on appelera *message*, un vecteur formé de noms de canaux. Un processus peut envoyer un *message* sur un canal, un processus peut attendre un *message* sur un canal, un processus peut créer un nouveau canal. L'exécution d'un processus peut aussi être bloquée par des conditions imposées sur les canaux qu'il manipule.

La syntaxe des actions est donnée figure 2.2.

Figure 2.2 Actions

action	::=	$c![x_1, ..., x_n]$	(émission d'un *message*)
	|	$c?[x_1, ..., x_n]$	(attente d'un *message*)
	|	(νx)	(création d'un canal)
	|	$[x = y]$	(filtrage)
	|	$[x \neq y]$	(filtrage)

où c, x_1, ..., x_n, x, $y \in$ *Channel*, *ngeqslant*0

Exemple 2.2.1 On modélise figure 2.3, un serveur de type *ftp*, qui ne peut disposer simultanément que de trois ports.

Figure 2.3 un serveur 3 ports

Allouer := $*$make?[]$(\nu$*canal-entrée*$)(\nu$*canal-sortie*$)(\nu$*info-client*$)$
(ν*canal-entrée*$![$*info-client*$]$
$|$*canal-entrée*$?[$*info-reçue*$].($*canal-sortie*$![$*info-reçue*$]$
$|$ make![]))

Port := make![]

Serveur := **Allouer** | **Port** | **Port** | **Port**

Lorsqu'un client sollicite le serveur, le processus va répliquer la ressource **Allouer**, ce qui ne sera possible que si un sous-processus (make![]) est présent. Dans ce cas, deux canaux virtuels, un canal d'entrée et un canal de sortie, sont alloués, (ce qui est modélisé à chaque fois par la création d'un nouveau canal), le client va alors utiliser le canal d'entrée pour communiquer les informations qu'il veut faire traiter par le serveur. Lorsque celui-ci les reçoit, le port est libéré par le *message* (make![]), ce qui a pour conséquence de permettre une nouvelle réplication, un *message* de trace est alors émis *canal-sortie*![*info-recue*]. On ignore ainsi les détails opérationnels relatifs au traitement effectif de l'information, qui ne concernent pas les communications entre canaux. □

2.2.2 Sémantique opérationnelle

On doit maintenant définir précisement la notion de communication entre les canaux. Une communication entre deux processus mis en concurrence peut avoir

lieu, lorsque l'un envoit un *message* sur un canal, et que l'autre attend un *message* sur ce même canal. Une communication consiste en une série de substitutions.

Pour définir proprement à quelles conditions deux processus peuvent communiquer, et quel est le résultat d'une communication, on a recours aux notions usuelles de variables libres d'un processus ($\mathfrak{FN}$), de variables liées d'un processus ($\mathfrak{BN}$) et de substitution dans un processus. Les seuls lieurs de variables sont l'attente d'un *message* et la création d'un canal. Ainsi dans le processus $c?[x_1, ...x_n]$P (resp. dans $(\nu x)P$), les occurences de x_1, ..., x_n (resp. x) sont liées dans le processus P.

Exemple 2.2.2 Dans le terme **P**:=(νx)(a?$[b]b![x]$ | a![c] | d?$[e]e![e]$),
on a $\mathfrak{FN}$(**P**)={a;c;d} et $\mathfrak{BN}$(**P**)=$\{x;b;e\}$. □

La sémantique du π-calcul est donnée par une relation de congruence et une relation de réduction. La relation de congruence permet de réorganiser la structure d'un processus, pour mettre en évidence les réductions possibles, alors que la relation de réduction décrit les effets de ces réductions. Pour communiquer, deux processus doivent partager les mêmes noms de canaux, l'un des rôles de cette congruence est de sortir les restrictions sur les canaux des sous-processus, en évitant les phénomènes de capture de variables. Ceci est rendu possible par le mécanisme d'α-conversion.

On donne figure 2.4 les règles qui définissent cette congruence.

Figure 2.4 Relation de congruence

$(\nu x)P$	$\equiv$	$(\nu y)P[x \leftarrow y]$ si $y \notin \mathfrak{FN}(P)$	(α-conversion)
$P \mid Q$	$\equiv$	$Q \mid P$	(commutativité)
$P \mid (Q \mid R)$	$\equiv$	$(P \mid Q) \mid R$	(associativité)
$*P$	$\equiv$	$*P \mid P$	(réplication)
$(\nu x)(\nu y)P$	$\equiv$	$(\nu y)(\nu x)P$	(intervertion)
$((\nu x)P) \mid Q$	$\equiv$	$(\nu x)(P \mid Q)$ si $x \notin \mathfrak{FN}(Q)$	(extrusion)

où c, x, $y \in$ *Channel*
(On rappelle qu'une relation de congruence est une relation d'équivalence compatible avec la mise sous contexte.)

On donne figure 2.5 la relation de réduction qui définit la sémantique du π-calcul.

J'att

Exemple 2.2.3 On donne maintenant une suite de réductions pour notre serveur :

(Dans cette exemple on ne précise pas l'utilisation des règles de commutativité et d'associativité)

Figure 2.5 Relation de transition

$$
\begin{array}{rcll}
c![x_1,...,x_n]P \mid c?[y_1,...,y_n]Q & \rightarrow & P \mid Q[y_1 \leftarrow x_1,...,y_n \leftarrow x_n] & \text{(communication)} \\
P+Q & \rightarrow & P & \text{(choix gauche)} \\
P+Q & \rightarrow & Q & \text{(choix droit)} \\
[x=x]P & \rightarrow & P & \text{(filtrage)} \\
[x \neq y]P & \rightarrow & P \text{ si } x \neq y & \text{(filtrage)}
\end{array}
$$

$$
\frac{P \rightarrow Q}{(\nu x)P \rightarrow (\nu x)Q} \qquad \frac{P' \equiv P \;\; P \rightarrow Q \;\; Q \equiv Q'}{P' \rightarrow Q'} \qquad \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}
$$

où c, x_1, ..., x_n, x, y_1, ..., y_n, $y \in$ *Channel*

$$
\begin{aligned}
\mathbf{P_0} := (*&\text{make?}[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
&(\ \textit{canal-entrée}![\textit{info-client}] \\
&\mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
&\qquad\qquad \mid \text{make}![])) \\
&\mid \text{make}![] \mid \text{make}![] \mid \text{make}![])
\end{aligned}
$$

On utilise la règle (réplication) de la congruence, pour libérer une ressource :

$$\mathbf{P_0} \equiv \mathbf{P_1}$$

où

$$
\begin{aligned}
\mathbf{P_1} := (*&\text{make?}[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
&(\ \textit{canal-entrée}![\textit{info-client}] \\
&\mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
&\qquad\qquad \mid \text{make}![])) \\
&\mid \text{make}![] \mid \text{make}![] \mid \text{make}![] \\
&\mid \text{make?}[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
&\qquad (\ \textit{canal-entrée}![\textit{info-client}] \\
&\qquad \mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
&\qquad\qquad\qquad \mid \text{make}![])) \\
&)
\end{aligned}
$$

On peut désormais effectuer une communication :

$$\mathbf{P_1} \rightarrow \mathbf{P_2}$$

où

$$\begin{aligned}
\mathbf{P_2} := {} & (*\text{make?}[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& \quad (\ \textit{canal-entrée}![\textit{info-client}] \\
& \quad \mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
& \qquad\qquad \mid \text{make}![])) \\
& \mid \text{make}![] \mid \text{make}![] \\
& \mid (\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& \quad (\ \textit{canal-entrée}![\textit{info-client}] \\
& \quad \mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
& \qquad\qquad \mid \text{make}![])) \\
&)
\end{aligned}$$

On notera la disparition d'un sous-processus (make![]).
On effectue ensuite la communication sur le canal virtuel :

$$\mathbf{P_2} \rightarrow \mathbf{P_3}$$

où

$$\begin{aligned}
\mathbf{P_3} := {} & (*\text{make?}[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& \quad (\ \textit{canal-entrée}![\textit{info-client}] \\
& \quad \mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
& \qquad\qquad \mid \text{make}![])) \\
& \mid \text{make}![] \mid \text{make}![] \\
& \mid (\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& \quad (\textit{canal-sortie}![\textit{info-client}] \mid \text{make}![]) \\
&)
\end{aligned}$$

Pour une meilleure lisibilité, on réorganise alors notre processus en utilisant la règle d'extrusion.

$$\mathbf{P_3} \equiv \mathbf{P_4}$$

où

$$\begin{aligned}
\mathbf{P_4} := {} & (\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& (*\text{make?}[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& \qquad (\ \textit{canal-entrée}![\textit{info-client}] \\
& \qquad \mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
& \qquad\qquad \mid \text{make}![])) \\
& \mid \text{make}![] \mid \text{make}![] \\
& \mid \textit{canal-sortie}![\textit{info-client}] \mid \text{make}![] \\
&)
\end{aligned}$$

Pour le deuxième client, on ne peut effectuer directement l'étape d'extrusion, il faut auparavant choisir de nouveaux noms pour les canaux.

Considérons le processus $\mathbf{P_5}$ défini ci-dessous :

$$\begin{aligned}
\mathbf{P_5} := {} & (\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& (*\text{make?}[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& \qquad (\ \textit{canal-entrée}![\textit{info-client}] \\
& \qquad \mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
& \qquad\qquad \mid \text{make}![])) \\
& \mid \text{make}![] \mid \text{make}![] \\
& \mid \textit{canal-sortie}![\textit{info-client}] \mid \text{make}![] \\
& \mid (\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& \qquad (\textit{canal-sortie}![\textit{info-client}] \mid \text{make}![]) \\
&)
\end{aligned}$$

On utilise la règle d'α-conversion pour éviter les conflits de noms de canaux :

$$\mathbf{P_5} \equiv \mathbf{P_6}$$

où

$$\begin{aligned}
\mathbf{P_6} := {} & (\nu in_1)(\nu out_1)(\nu info_1) \\
& (*\text{make?}[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& \qquad (\ \textit{canal-entrée}![\textit{info-client}] \\
& \qquad \mid \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
& \qquad\qquad \mid \text{make}![])) \\
& \mid \text{make}![] \mid \text{make}![] \\
& \mid out_1![info_1] \\
& \mid (\nu in_2)(\nu out_2)(\nu info_2) \\
& \quad (\ out_2![info_2] \mid \text{make}![]) \\
&)
\end{aligned}$$

On peut alors effectuer l'extrusion :

$$\mathbf{P_6} \equiv \mathbf{P_7}$$

où

$$\begin{aligned}
\mathbf{P_7} := {} & (\nu in_1)(\nu out_1)(\nu info_1)(\nu in_2)(\nu out_2)(\nu info_2) \\
& (*\text{make?}[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& \qquad (\ \textit{canal-entrée}![\textit{info-client}] \\
& \qquad \mid \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
& \qquad\qquad \mid \text{make}![])) \\
& \mid \text{make}![] \mid \text{make}![] \\
& \mid out_1![info_1] \mid out_2![info_2] \mid \text{make}![] \\
&)
\end{aligned}$$

Remarque 2.2.1 *Dans l'exemple précédent, le recours à l'extrusion n'était pas nécessaire, mais rendait les différentes étapes de la réductions plus lisibles. Cependant, son utilisation est obligatoire, dès qu'un canal, créé par un processus est communiqué hors de ce processus.*

Exemple 2.2.4 Le processus suivant représente aussi un serveur, il met en concurrence une unité de calcul qui traite l'information reçue, et un serveur qui attend d'être sollicité pour envoyer un *message* sur l'unité de travail.

$$\begin{aligned}
\textbf{Traite} &:= *\text{a?}[b]\text{traite}![b] \\
\textbf{Serveur} &:= *\text{make?}[]((\nu c)\ \text{a}![\text{c}] \mid \text{make}![]) \\
\textbf{Init} &:= \text{make}![]
\end{aligned}$$

On présente ici une courte dérivation de ce processus.

$$(\textbf{Traite} \mid \textbf{Serveur} \mid \textbf{Init}) \rightarrow (\textbf{Traite} \mid \textbf{Serveur} \mid \text{make}![] \mid ((\nu c)\ \text{a}![c]))$$

L'unité de calcul ne connait pas le canal c, on doit donc recourir à l'extrusion :

$$(\textbf{Traite} \mid \textbf{Serveur} \mid \text{make}![] \mid ((\nu c)\ \text{a}![c])) \equiv (\nu c)(\textbf{Traite} \mid \textbf{Serveur} \mid \text{make}![] \mid \text{a}![c])$$

La communication est maintenant possible.

2.3 Analyse

Le π-calcul permet donc de décrire précisement des réseaux de communication, il faut maintenant s'interroger sur la possibilité de recueillir des renseignements pertinents sur les processus qu'il représente. L'abstraction d'un programme en processus du π-calcul n'est qu'une étape, on doit ensuite être en mesure d'analyser les termes produits.

On s'interesse ici à deux problèmes d'analyse, que l'on devrait pouvoir traiter sur les termes du π-calcul.

Lorsque les processus du π-calcul représentent un protocole d'échange de clefs, on peut s'interesser à la sécurité de ce protocole, or la sécurité d'un protocole peut s'exprimer sous forme de contraintes sur les communications entre les canaux. Ainsi, dans l'exemple du serveur trois ports, figure 2.3, on veut pouvoir s'assurer que l'information est bien réexpédiée au client sur son canal de sortie et non pas sur un autre.

Lorsque les processus du π-calcul représentent un réseau de télécommunication, on peut s'interesser à majorer l'ensemble des ressources nécessaires pour simuler l'exécution des processus. Pour cela, il faut exprimer des contraintes numériques sur le nombre de sous-processus du processus initial, présents simultanément lors d'une éxécution. Toujours pour l'exemple du serveur trois ports, on voudrait pouvoir s'assurer que le serveur ne peut prendre en charge simultanément que trois clients, ce qui va se traduire par des contraintes sur le nombre de sous-processus de la ressource **allouer**.

Quoi qu'il en soit, ces deux objectifs ne pourront être atteints, en gardant telle quelle la sémantique que l'on a introduite (Cf 2.2). D'une part, il est pour l'instant impossible de chiffrer les ressources nécessaires à la simulation des processus, puisque la relation de congruence permet de répliquer les processus, sans en justifier l'utilité. Ainsi dans l'exemple du serveur, on peut répliquer la ressource **allouer** à l'infini, ce qui empêche tout diagnostique quant aux ressources physiques nécessaires pour la simulation du processus.

Exemple 2.3.1 En effet, le terme donné figure 2.6 est congruent au processus d'origine.

Figure 2.6 Des réplications non-justifiées

$$
\begin{array}{l}
(*\text{make}?[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
\qquad\qquad (\ \textit{canal-entrée}![\textit{info-client}] \\
\qquad\qquad \mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{make}![])) \\
\mid \text{make}![] \mid \text{make}![] \mid \text{make}![] \\
\mid *\text{make}?[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
\qquad\qquad (\ \textit{canal-entrée}![\textit{info-client}] \\
\qquad\qquad \mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{make}![])) \\
\mid *\text{make}?[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
\qquad\qquad (\ \textit{canal-entrée}![\textit{info-client}] \\
\qquad\qquad \mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{make}![])) \\
\mid *\text{make}?[](\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
\qquad\qquad (\ \textit{canal-entrée}![\textit{info-client}] \\
\qquad\qquad \mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{make}![])) \\
)
\end{array}
$$

D'autre part, le mécanisme d'α-conversion, permet de renommer les variables liées à sa guise, ce qui empêche toute analyse sur les noms de canaux. Dans l'exemple du serveur figure 2.3, un *message* de sortie est émis sur un canal de sortie ($\textit{canal-sortie}![\textit{info-reçue}]$). Dans l'exemple 2.2.3, on a mis en évidence, en choisissant habilement le nom des variables, l'invariant selon lequel l'information d'un client lui est rendu sur son propre canal ($out_1![info_1], out_2![info_2]$). Cependant, cet invariant est pour l'instant purement subjectif, vue que l'on peut arbitrairement changer le nom des canaux.

Exemple 2.3.2 Par exemple, le processus $\mathbf{P_6}$ est congruent au processus donné figure 2.7.

Ainsi cette sémantique admet des dérivations chaotiques qui rendent impossible la moindre analyse. Dans un premier temps, on va restreindre cette sémantique, et la transformer en sémantique paresseuse, pour permettre de maîtriser la réplication de ressources. On va ensuite donner un procédé de création de noms de canaux déterministe qui rendra le mécanisme d'α-conversion inutile.

2.4 Sémantique paresseuse

Le but de la sémantique paresseuse, conçue par Turner [Tur95], est de maîtriser l'usage des réplications au cours des dérivations, en vue d'une implantation

Figure 2.7 Des noms de canaux inappropriés

$$
\begin{aligned}
\mathbf{P_6} := {} & (\nu in_1)(\nu out_1)(\nu info_1)(\nu in_2)(\nu out_2)(\nu info_2) \\
& (*\text{make?}[](\nu \textit{canal-entrée})(\nu \textit{canal-sortie})(\nu \textit{info-client}) \\
& \qquad (\ \textit{canal-entrée}![\textit{info-client}] \\
& \qquad \mid\ \textit{canal-entrée}?[\textit{info-reçue}](\textit{canal-sortie}![\textit{info-reçue}] \\
& \qquad\qquad \mid \text{make}![])) \\
& \mid \text{make}![] \mid \text{make}![] \\
& \mid\ \textit{out}_2![\textit{info}_1]) \mid\ \textit{out}_1![\textit{info}_2] \mid \text{make}![] \\
&)
\end{aligned}
$$

réaliste. Pour cela, on n'autorise les réplications que lorsqu'elles sont justifiées par une communication. Cela conduit à quelques modifications sur la syntaxe et sur la sémantique du π-calcul. Comme toute réplication n'est autorisée que si elle conduit à une communication, on impose à tout signe de réplication d'être immédiatement suivi par une attente de *message*. En toute logique, on devrait aussi permettre aux signes de réplications d'être suivi d'une émission de *message*, mais ce ne serait pas réaliste. Cela simulerait un processus qui émettrait une infinité de *messages* sans aucune contrainte, pas même une click d'horloge.

On donne figure 2.8 la nouvelle syntaxe du π-calcul.

Figure 2.8 syntaxe

P	::=	action.P	(action)
	|	(P | P)	(exécution concurrente)
	|	(P + P)	(exécution disjonctive)
	|	$\emptyset$	(fin d'un processus)
action	::=	$c![x_1, ..., x_n]$	(émission d'un *message*)
	|	$c?[x_1, ..., x_n]$	(attente d'un *message*)
	|	$*c?[x_1, ..., x_n]$	(réplication gardée)
	|	(νx)	(création d'un canal)
	|	$[x = y]$	(filtrage)
	|	$[x \neq y]$	(filtrage)

La sémantique opérationnelle se définie simplement, en enlevant la règle de congruence (réplication) et en ajoutant une nouvelle règle de réduction qui factorise l'ancienne règle de congruence (réplication) et la règle de transition (communication). On donne figure 2.9 la nouvelle règle de réduction.

Figure 2.9 dépliage de ressource

$$
c![x_1, ..., x_n]P \mid *c?[y_1, ..., y_n]Q \rightarrow P \mid Q[y_1 \leftarrow x_1, ..., y_n \leftarrow x_n] \mid *c?[y_1, ..., y_n]Q \quad \text{(réplication de ressource)}
$$

Exemple 2.4.1 On montre maintenant une courte dérivation avec cette nouvelle sémantique :

$$
\begin{array}{l}
(*\text{make?}[](\nu\,\mathit{canal\text{-}entrée})(\nu\,\mathit{canal\text{-}sortie})(\nu\,\mathit{info\text{-}client}) \\
\qquad\qquad (\ \mathit{canal\text{-}entrée}![\mathit{info\text{-}client}] \\
\qquad\qquad \mid\ \mathit{canal\text{-}entrée}?[\mathit{info\text{-}reçue}](\mathit{canal\text{-}sortie}![\mathit{info\text{-}reçue}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{make}![])) \\
\mid \text{make}![] \mid \text{make}![] \mid \text{make}![] \\
) \\
\rightarrow \\
(*\text{make?}[](\nu\,\mathit{canal\text{-}entrée})(\nu\,\mathit{canal\text{-}sortie})(\nu\,\mathit{info\text{-}client}) \\
\qquad\qquad (\ \mathit{canal\text{-}entrée}![\mathit{info\text{-}client}] \\
\qquad\qquad \mid\ \mathit{canal\text{-}entrée}?[\mathit{info\text{-}reçue}](\mathit{canal\text{-}sortie}![\mathit{info\text{-}reçue}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{make}![])) \\
\mid \text{make}![] \mid \text{make}![] \\
\mid (\nu\,\mathit{canal\text{-}entrée})(\nu\,\mathit{canal\text{-}sortie})(\nu\,\mathit{info\text{-}client}) \\
\qquad (\ \mathit{canal\text{-}entrée}![\mathit{info\text{-}client}] \\
\qquad \mid\ \mathit{canal\text{-}entrée}?[\mathit{info\text{-}reçue}].(\mathit{canal\text{-}sortie}![\mathit{info\text{-}reçue}] \mid \text{make}![])) \\
) \\
\rightarrow \\
(\nu\,\mathit{in}_1)(\nu\,\mathit{out}_1)(\nu\,\mathit{info}_1) \\
(*\text{make?}[](\nu\,\mathit{canal\text{-}entrée})(\nu\,\mathit{canal\text{-}sortie})(\nu\,\mathit{info\text{-}client}) \\
\qquad\qquad (\ \mathit{canal\text{-}entrée}![\mathit{info\text{-}client}] \\
\qquad\qquad \mid\ \mathit{canal\text{-}entrée}?[\mathit{info\text{-}reçue}](\mathit{canal\text{-}sortie}![\mathit{info\text{-}reçue}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{make}![])) \\
\mid \text{make}![] \mid \text{make}![] \mid \mathit{out}_1![\mathit{info}_1] \mid \text{make}![] \\
) \\
\rightarrow \\
(\nu\,\mathit{in}_1)(\nu\,\mathit{out}_1)(\nu\,\mathit{info}_1)(\nu\,\mathit{in}_2)(\nu\,\mathit{out}_2)(\nu\,\mathit{info}_2) \\
(*\text{make?}[](\nu\,\mathit{canal\text{-}entrée})(\nu\,\mathit{canal\text{-}sortie})(\nu\,\mathit{info\text{-}client}) \\
\qquad\qquad (\ \mathit{canal\text{-}entrée}![\mathit{info\text{-}client}] \\
\qquad\qquad \mid\ \mathit{canal\text{-}entrée}?[\mathit{info\text{-}reçue}](\mathit{canal\text{-}sortie}![\mathit{info\text{-}reçue}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{make}![])) \\
\mid \text{make}![] \mid \mathit{out}_1![\mathit{info}_1] \mid \text{make}![] \\
\qquad\qquad \mid\ \mathit{in}_2![\mathit{info}_2] \\
\qquad\qquad \mid\ \mathit{in}_2?[\mathit{info}_2](\mathit{out}_2![\mathit{info\text{-}reçue}] \mid \text{make}![]) \\
) \\
\rightarrow \\
(\nu\,\mathit{in}_1)(\nu\,\mathit{out}_1)(\nu\,\mathit{info}_1)(\nu\,\mathit{in}_2)(\nu\,\mathit{out}_2)(\nu\,\mathit{info}_2) \\
(*\text{make?}[](\nu\,\mathit{canal\text{-}entrée})(\nu\,\mathit{canal\text{-}sortie})(\nu\,\mathit{info\text{-}client}) \\
\qquad\qquad (\ \mathit{canal\text{-}entrée}![\mathit{info\text{-}client}] \\
\qquad\qquad \mid\ \mathit{canal\text{-}entrée}?[\mathit{info\text{-}reçue}](\mathit{canal\text{-}sortie}![\mathit{info\text{-}reçue}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{make}![])) \\
\mid \text{make}![] \mid \mathit{out}_1![\mathit{info}_1] \mid \text{make}![] \\
\mid \mathit{out}_2![\mathit{info}_2] \mid \text{make}![])
\end{array}
$$

3 Sémantique non-standard

La sémantique standard n'est pas directement analysable. D'une part, elle laisse le choix de renommer les variables, d'autre part elle a recours à une congruence pour réorganiser syntaxiquement les processus avant de mettre en évidence les processus succeptibles de communiquer.

Le but de la sémantique non-standard est d'une part de donner un procédé de génération déterministe de noms pour les nouveaux canaux, d'autre part, d'intégrer la relation de congruence au sein de la relation de transition.

Une telle sémantique a déjà été spécifiée par Arnaud Venet [Ven98], cependant celle-ci n'opère que sur un sous-ensemble du π-calcul. Néanmoins, cette sémantique peut fournir les bases de notre sémantique. Cette sémantique repose en effet sur deux idées essentielles. La première est que si on considère un π-terme donné que l'on appelera P, alors pour toute exécution Q de P, c'est à dire pour tout Q tel que $P \rightarrow^* Q$, Q est congruent à un terme Q' formé par une série de créations de canaux, suivie de la mise en concurrence de sous-processus de P auquel on a appliqué des substitutions. La deuxième idée est que pour nommer un nouveau canal, il suffit de concaténer au nom de son lieur, l'historique des tâches qui ont conduit à cette création. Ces deux idées seront bien sûr explicitées plus loin.

Par la suite, on opère quelques changements anodins sur la syntaxe du π-calcul. On impose à tous les noms de canaux, présent dans un π-terme introduits par les lieurs, d'être deux à deux distincts. De plus on étiquette chaque signe ν, ?, !, +, |, = et $\neq$ par des étiquettes deux à deux distinctes, on utilise pour cela Lbl, un ensemble infini dénombrable d'étiquettes.

On donne figure 3.1 un exemple d'étiquetage pour l'exemple du serveur (Cf figure 2.3).

Figure 3.1 Serveur

$$\begin{aligned}
\textbf{Allouer} := {} & *\text{make}?^{1}[](\nu^{2}\mathit{canal\text{-}entrée})(\nu^{3}\mathit{canal\text{-}sortie})(\nu^{4}\mathit{info\text{-}client}) \\
& (\mathit{canal\text{-}entrée}!^{5}[\mathit{info\text{-}client}] \\
& |^{6}\mathit{canal\text{-}entrée}?^{7}[\mathit{info\text{-}reçue}](\mathit{canal\text{-}sortie}!^{8}[\mathit{info\text{-}reçue}] \\
& \qquad |^{9}\ \text{make}^{10}![])) \\
\textbf{Serveur} := {} & (\nu^{0}\text{make})(\textbf{Allouer}\ |^{11}\ (\text{make}!^{12}[]\ |^{13}\ (\text{make}!^{14}[]\ |^{15}\ \text{make}!^{16}[])))
\end{aligned}$$

3.1 Sémantique petit-pas

3.1.1 Introduction

Cette sémantique manipule des ensembles de tâches, où une tâche est un sous-processus du processus initial, munie d'un identifiant indiquant son historique, et d'un environnement qui lie chacune de ses variables libres à un couple formé

d'un nom de canal qui spécifie quel lieur l'a créée et d'un identifiant qui précise quel était alors l'historique de la tâche correspondante. Les identifiants sont des arbres formés lors des réplications de ressources. L'identifiant des tâches initiales est une feuille vide. Lors d'une réplication, l'identifiant de la tâche créée dynamiquement est formé à partir de l'identifiant de la ressource qui s'est répliquée et de l'identifiant de la tâche qui a envoyé le *message* qui a provoqué cette réplication. Pour les processus simples, lorsque les ressources ne sont pas imbriquées, l'identifiant des ressources est toujours la feuille vide, et ainsi les identifiants sont des séquences.

Les règles de réduction se répartissent en deux principales catégories. D'une part, des règles structurelles serviront à simuler l'ancienne relation de congruence, ainsi cette relation sera orientée dans cette nouvelle sémantique. D'autre part, des règles actives qui correspondent aux règles de communication, de choix et de filtrage.

3.1.2 Définitions

Soit P_0 un processus clos étiqueté.

On note $\begin{array}{lcl} Att(P_0) & = & \{i \in Lbl | ?^i \textit{ sous mot de } P_0\}, \\ Eme(P_0) & = & \{i \in Lbl | !^i \textit{ sous mot de } P_0\}. \end{array}$

On note Id l'ensemble des arbres binaires dont les noeuds sont des couples (i,j) où $i \in Att(P)$ et $j \in Eme(P)$ et les feuilles ne sont pas étiquetées (que l'on note ε). Les arbres seront notés $\mathbf{Noeud}((i,j), \textit{fils-gauche}, \textit{fils-droit})$.

Un environnement est une fonction, dont le domaine est une partie de *Channel* et dont l'image est une partie de $\textit{Channel} \times \textit{Id}$.

Une tâche est un triplet, formé d'un sous-processus P de P_0, d'un identifiant $id \in \textit{Id}$, et d'un environnement dont le domaine est $\mathfrak{FN}(P)$.

L'instance syntaxique de la tâche (P, id, E) est le couple (P, id), l'ensemble des variables de la tâche (P, id, E) est l'image de E.

Une étape d'exécution est un ensemble de tâches.

Si C est une étape d'exécution, on note $\textit{Syn}(C)$ l'ensemble des instances syntaxiques des éléments de C, et $\textit{Var}(C)$ la réunion des ensembles de variables des éléments de C.

3.1.3 Traduction initiale

La traduction du processus est triviale :

On définit $C_0(P_0) = (P_0, \varepsilon, \emptyset)$

3.1.4 Transitions

On définit ensuite la relation de réduction.

– **Promotion**
La règle de promotion est une règle structurelle, qui simule la règle de congruence (associativité). Ainsi, lorsqu'une tâche est composée de deux processus en concurrence, on peut la transformer en deux tâches indépendantes.
Soit C une étape d'exécution,
supposons que $\lambda \in C$, où $\lambda = (P|^i Q, id, E)$
on a alors $C \overset{|^i}{\rightarrow}_1 C'$
où $C' = (C \setminus \{\lambda\}) \cup \{(P, id, E_{|\mathfrak{fFR}(P)}), (Q, id, E_{|\mathfrak{fFR}(Q)})\}$.

– **Création**
La règle de création est la deuxième règle structurelle, elle sert à remplacer la règle d'extrusion. On montrera par le suite sa validité en montrant qu'elle n'engendre pas de conflits entre les noms de variables.
Soit C une étape d'exécution,
supposons que $\lambda \in C$, où $\lambda = ((\nu^i x)P, id, E)$
on a alors $C \overset{\nu^i}{\rightarrow}_1 C'$

$$\text{où } C' = (C \setminus \{\lambda\}) \cup \left\{ \left(P, id, \begin{cases} E' & \text{si } x \in \mathfrak{fFR}(P) \\ E & \text{sinon} \end{cases} \right) \right\}$$

$$\text{avec } E' = \begin{cases} \mathfrak{fFR}(P) & \to \mathfrak{BFR}(P_0) \times Id \\ x & \mapsto (x, id) \\ y & \mapsto E(y) \text{ si } y \neq x \end{cases}$$

– **Choix gauche**
Soit C une étape d'exécution,
supposons que $\lambda \in C$, où $\lambda = (P +^i Q, id, E)$
on a alors $C \overset{+^i_g}{\rightarrow}_1 C'$
où $C' = (C \setminus \{\lambda\}) \cup \{(P, id, E_{|\mathfrak{fFR}(P)})\}$.

– **Choix droit**
Soit C une étape d'exécution,
supposons que $\lambda \in C$, où $\lambda = (P +^i Q, id, E)$
on a alors $C \overset{+^i_d}{\rightarrow}_1 C'$
où $C' = (C \setminus \{\lambda\}) \cup \{(Q, id, E_{|\mathfrak{fFR}(Q)})\}$.

– **Filtrage** $(\diamond \in \{=; \neq\})$
Soit C une étape d'exécution,
supposons que $\lambda \in C$, où $\lambda = ([x \diamond^i y]P, id, E)$
avec $E(x) \diamond E(y)$
on a alors $C \overset{\diamond^i}{\rightarrow}_1 C'$
où $C' = (C \setminus \{\lambda\}) \cup \{(P, id, E_{|\mathfrak{fFR}(P)})\}$.

– **Communication**

Soit C une étape d'exécution,

Supposons que $\lambda, \mu \in C$,

où λ s'écrit $(y?^i[y_1, ..., y_n]P, id_?, E_?)$

et μ s'écrit $(x!^j[x_1, ..., x_n]Q, id_!, E_!)$,

tels que $E_?(y) = E_!(x)$

on a alors $C \overset{(?^i, !^j)}{\rightarrow_1} C'$,

où $C' = (C \setminus \{\lambda, \mu\}) \cup \{(P, id_?, E'); (Q, id_!, E_{!|\mathfrak{FR}(Q)})\}$

avec $E' = \begin{cases} \mathfrak{FR}(P) & \rightarrow \mathfrak{BR}(P_0) \times Id \\ y_k & \mapsto E_!(x_k) \qquad \forall k \in [|1; n|] \\ z & \mapsto E_?(z) \qquad \text{si } z \notin \{x_k | k \in [|1; n|]\} \end{cases}$

– **Ressource**

Lors du dépliage d'une ressource, une tâche est créée dynamiquement. On associe à cette tâche un historique, calculée de manière à ce que les noms de canaux qu'elle créera ne soient pas déjà utilisés par d'autres processus.

Soit C une étape d'exécution,

Supposons que $\lambda, \mu \in C$,

où λ s'écrit $(*y?^i[y_1, ..., y_n]P, id_?, E_?)$

et μ s'écrit $(x!^j[x_1, ..., x_n]Q, id_!, E_!)$,

tels que $E_?(y) = E_!(x)$

on a alors $C \overset{(?^i, !^j)}{\rightarrow_1} C'$,

où $C' = (C \setminus \{\mu\}) \cup \{(P, \mathbf{Noeud}((i, j), id_?, id!), E'); (Q, id_!, E_{!|\mathfrak{FR}(Q)})\}$

avec $E' = \begin{cases} \mathfrak{FR}(P) & \rightarrow \mathfrak{BR}(P_0) \times Id \\ y_k & \mapsto E_!(x_k) \qquad \forall k \in [|1; n|] \\ z & \mapsto E_?(z) \qquad \text{si } z \notin \{x_k | k \in [|1; n|]\} \end{cases}$

3.1.5 Indépendance des variables

On se propose désormais de montrer qu'il ne peut y avoir de conflit entre les noms de variables au sein d'une dérivation non-standard. Pour cela il suffit de prouver le lemme 3.1 selon lequel, au cours d'une dérivation, une instance syntaxique donnée ne peut apparaître qu'une seule fois.

– **Lemme 3.1**

Soit $C_0 \overset{\lambda_1}{\rightarrow_1} ... \overset{\lambda_n}{\rightarrow_1} C_n$, une dérivation non standard, avec $C_0 = C_0(P_0)$,
soit $i \in \mathbb{N}$ et $(P, id) \in Syn(C_i)$,
alors pour $j > i$, $(P, id) \notin Syn(C_j) \Longrightarrow (\forall k > j, (P, id) \notin Syn(C_k))$.

Preuve 1 Pour montrer ce lemme, on associe à chaque instance syntaxique de la dérivation, une autre instance syntaxique,que l'on appelera son père, qui devra nécessairement disparaître pour permettre l'apparition de son fils.

Soit (P, id) une instance syntaxique de le dérivation,

- si $(\nu^i x)P$ est un sous-processus de P_0, $((\nu^i x)P, id)$ est un père de (P, id)
- si $(P +^j Q)$ est un sous-processus de P_0, $(P +^j Q, id)$ est un père de (P, id)
- si $(Q +^j P)$ est un sous-processus de P_0, $(P +^j Q, id)$ est un père de (P, id)
- si $([x \diamond^i y]P)$ est un sous-processus de P_0, $([x \diamond^i y]P, id)$ est un père de (P, id)
- si $x?^i[x_1, ..., x_n]P$ est un sous-processus de P_0, et que $*x?^i[x_1, ..., x_n]P$ n'est pas un sous-processus de P_0, alors $(x?^i[x_1, ..., x_n]P, id)$ est un père de (P, id)
- si $x!^i[x_1, ..., x_n]P$ est un sous-processus de P_0, alors $(x!^i[x_1, ..., x_n]P, id)$ est un père de (P, id).
- si $*x?^i[x_1, ..., x_n]P$ est un sous-processus de P_0, id ne peut être vide, on note $id = \mathbf{Noeud}((i, j), id_?, id_!)$, il existe donc $y!^j[y_1, ..., y_n]Q$ sous-processus de P_0 qui soit père de $*x?^i[x_1, ..., x_n]P$.

Soit maintenant une dérivation non-standard $C_0 \stackrel{\lambda_1}{\rightarrow}_1 ... \stackrel{\lambda_n}{\rightarrow}_1 C_n$ incompatible avec le lemme,
prennons i minimal et $(P, id) \in Syn(C_i)$,
tel que pour $j > i$, $(P, id) \notin Syn(C_j)$ et pour $k > j$ minimal, $(P, id) \in Syn(C_k)$,
soit alors $\mathbf{P}$ le père de (P, id),

$$\text{on a } \begin{cases} \mathbf{P} \in Syn(C_{i-1}) & \text{car } (P, id) \in (Syn(C_i)) \setminus (Syn(C_{i-1})) \\ \mathbf{P} \notin Syn(C_i) & \text{car } \mathbf{P} \text{ est le père de } (P, id) \\ \mathbf{P} \in Syn(C_{k-1}) & \text{car } (P, id) \in (Syn(C_k)) \setminus (Syn(C_{k-1})) \end{cases}$$

(Absurde car i est minimal) □

- **Théorème 3.1 indépendances des variables**

Soit $C_0 \stackrel{\lambda_1}{\rightarrow}_1 ... \stackrel{\lambda_n}{\rightarrow}_1 C_n$ une dérivation non-standard,
soit k tel que $\lambda_k = \nu^i$,
soit $id \in Id$ tel que

$$\begin{cases} ((\nu^i x)P, id) \in Syn(C_{k-1}) \\ (P, id) \in Syn(C_k) \end{cases}$$

on a alors $(x, id) \notin Var(C_{k-1})$.

Preuve 2 Soit $C_0 \stackrel{\lambda_1}{\rightarrow}_1 ... \stackrel{\lambda_n}{\rightarrow}_1 C_n$ une dérivation non-standard,
soit k tel que $\lambda_k = \nu^i$,
soit $id \in Id$ tel que

$$\begin{cases} ((\nu^i x)P, id) \in Syn(C_{k-1}) \\ (P, id) \in Syn(C_k) \\ (x, id) \in Var(C_{k-1}) \end{cases}$$

on a alors

- $\exists l < k \begin{cases} ((\nu x)P, id) \in Syn(C_{l-1}) \\ ((\nu x)P, id) \notin Syn(C_l) \end{cases}$
- $((\nu x)P, id) \in Syn(C_{k-1})$

Ce qui contredit le lemme 3.1. □

3.1.6 Cohérence

Pour montrer la cohérence de la sémantique non-standard vis à vis de la sémantique concrète, on utilise le notion de *bisimulation faible*. Ainsi, après avoir défini la traduction d'une configuration non-standard en un terme du π-calcul, on montrera que les deux systèmes de transition sont identiques, lorsque l'on ignore les transitions structurelles.

Supposons que l'on ait $(\mathfrak{BN}(P_0) \times Id) \in Channel$, pour traduire une configuration standard en un processus concrêt, il suffit d'en déclarer les variables, puis de mettre en concurrence toutes ses tâches, en leur ayant auparavant appliqué leur environnement.

On définit figure 3.2 la fonction de traduction.

Figure 3.2 Traduction

$$\Pi(C) = (\underset{x \in Var(C)}{\nu} x) \quad \underset{(P,a,E) \in C}{|} \quad E(P)$$

Proposition 3.1 Soient C, C' des configurations, λ de la forme $|^i$ ou ν^i, tels que $C_0(P_0) \rightarrow_1^* C \xrightarrow{\lambda}_1 C'$, on a alors $\Pi(C) \equiv \Pi(C')$.

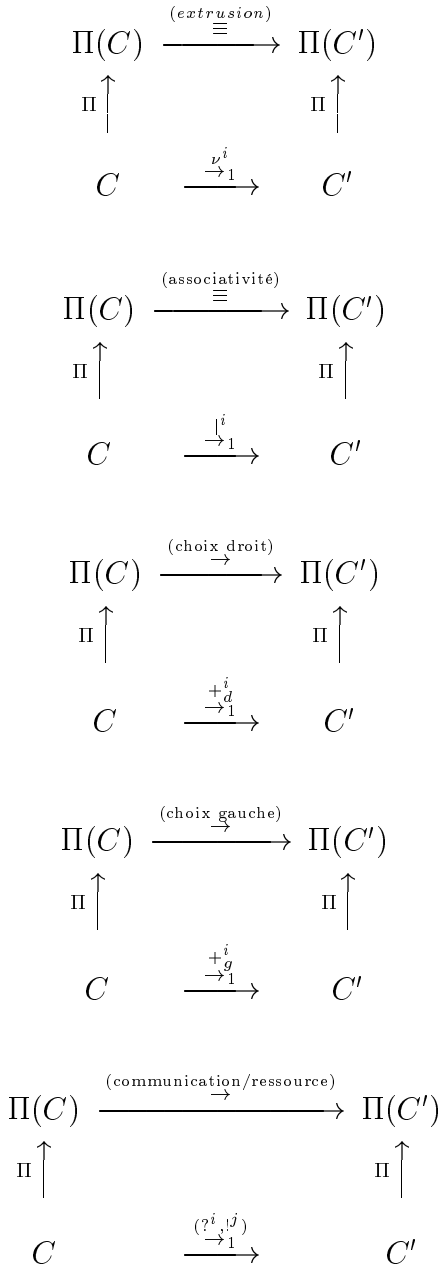
Proposition 3.2 Soient C, C' des configurations, λ de la forme $+_d^i$, $+_g^i$, $\diamond^i$ ou $(?^i, !^j)$, tels que $C_0(P_0) \rightarrow_1{}^* C \xrightarrow{\lambda}_1 C'$ on a alors $\Pi(C) \rightarrow \Pi(C')$.

Preuves 1 Les schémas de la figure 3.3 suffisent à démontrer ces deux propositions.

On déduit de ces deux propositions le théorème 3.2.

Théorème 3.2 La sémantique concrète est une simulation faible de la sémantique non-standard.

Figure 3.3 diagrammes commutatifs



Réciproque 3.1 Soient C une étape d'éxécution, A un processus tels que $C_0(P_0) \rightarrow_1^* C$ et $\Pi(C) \rightarrow_1 A$,

alors il existe D, E des étapes d'éxécutions telles que

$$\begin{cases} C \rightarrow_1^* D & \text{en utilisant uniquement des règles structurelles} \\ D \rightarrow_1 E & \text{en utilisant une règle active} \\ \Pi(E) \equiv A \end{cases}$$

Preuve 3 La restriction de la relation de transition non-standard aux règles structurelles est confluente (elle est en effet neuthérienne et localement confluente), on peut donc définir $\Longrightarrow$ sa limite transitive.

On note D la configuration telle que $C \Longrightarrow D$

D'après la proposition 3.1, le diagramme commutatif suivant est vérifié :

$$\begin{array}{ccc} \Pi(C) & \xrightarrow{\equiv} & \Pi(D) \\ {\scriptstyle \Pi}\uparrow & & {\scriptstyle \Pi}\uparrow \\ C & \xrightarrow{\Longrightarrow} & D \end{array}$$

$\Pi(B)$ peut maintenant être réduit sans avoir recours à la règle de congruence. On peut alors faire agir la règle correspondante, (Cf 3.3) sur B. Le diagramme suivant est donc vérifié.

$$\begin{array}{ccc} \Pi(B) & \xrightarrow{\rightarrow} & \Pi(E) \\ {\scriptstyle \Pi}\uparrow & & {\scriptstyle \Pi}\uparrow \\ B & \xrightarrow{\rightarrow_1} & E \end{array}$$

On a alors $A \equiv \Pi(E)$. □

Exemple 3.1.1 On donne ci-dessous une dérivation non-standard pour l'exemple du serveur figure 3.1. Pour une meilleure lisibilité, on remplacera dans l'écriture des tâches, le nom du sous-processus par l'étiquette de son premier symbole étiqueté.

On a $C_0(\textbf{Serveur}) = \{(0, \varepsilon, \emptyset)\}$

Puis $C_0(\textbf{Serveur})$

$$\xrightarrow{\nu^0}_1 \left\{ \left(11, \varepsilon, \left\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \right) \right\} \right.$$

$$\xrightarrow{|^{11}}_1 \xrightarrow{|^{13}}_1 \xrightarrow{|^{15}}_1 \left\{ \begin{array}{l} \left(1, \varepsilon, \left\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \right) \right. \\ \left(12, \varepsilon, \left\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \right) \right. \\ \left(14, \varepsilon, \left\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \right) \right. \\ \left(16, \varepsilon, \left\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \right) \right. \end{array} \right\}$$

$$\xrightarrow{(1,12)}_1 \left\{ \begin{array}{l} \Big(1, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \Big(2, (1,12), \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \Big(14, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \Big(16, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \end{array} \right\}$$

$$\xrightarrow{\nu^2}_1 \xrightarrow{\nu^3}_1 \xrightarrow{\nu^4}_1 \left\{ \begin{array}{l} \Big(1, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \left(6, (1,12), \begin{cases} \text{make} \mapsto (\text{make}, \varepsilon) \\ \textit{canal-entrée} \mapsto (\textit{canal-entrée}, (1,12)) \\ \textit{canal-sortie} \mapsto (\textit{canal-sortie}, (1,12)) \\ \textit{info-client} \mapsto (\textit{info-client}, (1,12)) \end{cases} \right) \\ \Big(14, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \Big(16, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \end{array} \right\}$$

$$\xrightarrow{|^6}_1 \left\{ \begin{array}{l} \Big(1, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \left(5, (1,12), \begin{cases} \textit{canal-entrée} \mapsto (\textit{canal-entrée}, (1,12)) \\ \textit{info-client} \mapsto (\textit{info-client}, (1,12)) \end{cases} \right) \\ \left(7, (1,12), \begin{cases} \text{make} \mapsto (\text{make}, \varepsilon) \\ \textit{canal-entrée} \mapsto (\textit{canal-entrée}, (1,12)) \\ \textit{canal-sortie} \mapsto (\textit{canal-sortie}, (1,12)) \end{cases} \right) \\ \Big(14, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \Big(16, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \end{array} \right\}$$

$$\xrightarrow{(7,5)}_1 \left\{ \begin{array}{l} \Big(1, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \left(9, (1,12), \begin{cases} \text{make} \mapsto (\text{make}, \varepsilon) \\ \textit{info-reçue} \mapsto (\textit{info-client}, (1,12)) \\ \textit{canal-sortie} \mapsto (\textit{canal-sortie}, (1,12)) \end{cases} \right) \\ \Big(14, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \Big(16, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \end{array} \right\}$$

$$\xrightarrow{|^9}_1 \left\{ \begin{array}{l} \Big(1, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \left(8, (1,12), \begin{cases} \textit{info-reçue} \mapsto (\textit{info-client}, (1,12)) \\ \textit{canal-sortie} \mapsto (\textit{canal-sortie}, (1,12)) \end{cases} \right) \\ \Big(10, (1,12), \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \Big(14, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \\ \Big(16, \varepsilon, \Big\{ \text{make} \mapsto (\text{make}, \varepsilon) \quad \Big) \end{array} \right\}$$

3.1.7 Conclusion

Cette sémantique permet bien de mimer les dérivations standards tout en remédiant aux défauts de la sémantique concrète. Cependant, elle admet trop de règles de dérivations, ce qui conduirait, si on tentait de l'analyser directement, à une analyse trop complexe aussi bien sur le plan de la lisibilité que de la complexité.

3.2 Sémantique économe

3.2.1 Motivation

On se propose dans cette partie d'éliminer les réductions structurelles. En effet, ces réductions ne présentent aucun intéret sur le plan sémantique, car elles n'agissent pas directement sur les processus. Elles ne simulent en effet aucune étape d'exécution.

3.2.2 Principe

On a vu que la restriction du sytème de règles de la sémantique précédente aux règles structurelles était confluente. On se propose donc d'en appliquer la limite après chaque transition liée à une règle d'action.

Il suffit de définir pout λ de la forme $+^i_d$,$+^i_g$,$(?^i, !^j)$,$\diamond^i$:

$$\xrightarrow{\lambda}_2 ::= \xrightarrow{\lambda}_1 \implies$$

3.2.3 Définitions

Soit P_0 un processus clos étiqueté.

Toutes les définitions de la sémantique précédente restent valables (Cf 3.1.2). Il faut néanmoins restreindre l'ensemble des tâches aux processus qui commencent par une attente, une réplication, une émission, un filtrage ou un choix. Ainsi, l'étiquetage des autres oprérateurs devient inutile.

On définit de plus la fonction *Agent* qui étant donné un sous-processus, donne l'ensemble de toutes les tâches mises en concurrence dans celui-ci.

Définition 3.1 *La définition de la fonction Agent est donné figure 3.4.*

3.2.4 Traduction initiale

Pour traduire le processus initial, on a recours à la fonction *Agent*, afin de déterminer l'ensemble des tâches mises en concurrence dans le processus initial.

On définit $C_0(P_0) = \{(p, \varepsilon, E_p) | p \in Agent(P_0)\}$

où $E_p = \begin{cases} \mathfrak{f}\mathfrak{N}(p) & \to \mathfrak{B}\mathfrak{N}(P_0) \times Id \\ x & \mapsto (x, \varepsilon) \end{cases}$

Figure 3.4 la fonction *Agent*

$$
\begin{array}{rcll}
Agent(\emptyset) & = & \{\} & \\
Agent(x!^i[x_1,...,x_n]P) & = & \{x!^i[x_1,...,x_n]P\} & \\
Agent(y?^i[y_1,...,y_n]P) & = & \{y?^i[y_1,...,y_n]P\} & \\
Agent(*y?^i[y_1,...,y_n]P) & = & \{*y?^i[y_1,...,y_n]P\} & \\
Agent(P|Q) & = & Agent(P) \cup Agent(Q) & \\
Agent(P +^i Q) & = & \{P +^i Q\} & \\
Agent((\nu x)P) & = & Agent(P) & \\
Agent([x \diamond^i y]P) & = & \{[x \diamond^i y]P\} & (\diamond \in \{=;\ \neq\})
\end{array}
$$

3.2.5 Transitions

On définit ensuite une nouvelle relation de réduction $\rightarrow_2$. Celle-ci est obtenue à partir de la relation de réduction de la sémantique non-standard petit-pas (Cf 3.1.4). Après chaque transition, l'ensemble des nouvelles tâches est calculé par l'intermédaire de la fonction *Agent*. Par ailleurs, pour chaque nouvelle tâche, on détermine l'ensemble de ses nouveaux canaux, en comparant l'ensemble de ses variables libres à l'ensemble des variables liées du processus dont il est directement issu.

- **Choix gauche**

  Soit C une étape d'exécution,

  supposons que $\lambda \in C$,où $\lambda = (P +^i Q, id, E)$

  on note $f_g : Ag \mapsto \left(Ag, id, \begin{cases} x \mapsto E(x) & si\ x \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(P+Q) \\ x \mapsto (x, id) & si\ x \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(P+Q) \end{cases} \right)$

  on a alors $C \stackrel{+^i_g}{\rightarrow_2} C'$

  où $C' = (C \setminus \{\lambda\}) \cup (f_g(Agent(P)))$.

– **Choix droit**

Soit C une étape d'exécution,

supposons que $\lambda \in C$,où $\lambda = (P +^i Q, id, E)$

on note $f_d : Ag \mapsto \left(Ag, id, \begin{cases} x \mapsto E(x) & si\ x \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(P+Q) \\ x \mapsto (x, id) & si\ x \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(P+Q) \end{cases} \right)$

on a alors $C \overset{+^i_d}{\rightarrow}_2 C'$

où $C' = (C \setminus \{\lambda\}) \cup (f_d(Agent(Q)))$.

– **Filtrage** $(\diamond \in \{=;\ \neq\})$

Soit C une étape d'exécution,

supposons que $\lambda \in C$,où $\lambda = ([x \diamond^i y]P, id, E)$

avec $E(x) \diamond E(y)$

on note $f_\diamond : Ag \mapsto \left(Ag, id, \begin{cases} z \mapsto E(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}([x \diamond y]P) \\ z \mapsto (z, id) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}([x \diamond y]P) \end{cases} \right)$

on a alors $C \overset{\diamond^i}{\rightarrow}_2 C'$

où $C' = (C \setminus \{\lambda\}) \cup (f_\diamond((AgentP)))$.

– **Communication**

Soit C une étape d'exécution,

Supposons que $\lambda, \mu \in C$,

où λ s'écrit $(y?^i[y_1, ..., y_n]P, id_?, E_?)$

et μ s'écrit $(x!^j[x_1, ..., x_n]Q, id_!, E_!)$,

tels que $E_?(y) = E_!(x)$

on note

$$f_? : Ag \mapsto \left(Ag, id_?, \begin{cases} z \mapsto E_?(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(y?^i[y_1, ..., y_n]P) \\ y_k \mapsto E_!(x_k) & \text{si } y_k \in \mathfrak{FR}(Ag) \\ z \mapsto (z, id_?) & \text{si } \begin{cases} z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(y?^i[y_1, ..., y_n]P) \\ z \notin \{y_k | k \in [|1; n|]\} \end{cases} \end{cases} \right)$$

et

$$f_! : Ag \mapsto \left(Ag, id_!, \begin{cases} z \mapsto E(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(x!^j[x_1, ..., x_n]Q) \\ z \mapsto (z, id_!) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(x!^j[x_1, ..., x_n]Q) \end{cases} \right)$$

on a alors $C \overset{(?^i,!^j)}{\rightarrow}_2 C'$,

où $C' = (C \setminus \{\lambda, \mu\}) \cup (f_?(Agent(P))) \cup (f_!(Agent(Q)))$

– **Ressource**

Soit C une étape d'exécution,

Supposons que $\lambda, \mu \in C$,

où λ s'écrit $(*y?^i[y_1, ..., y_n]P, id_?, E_?)$

et μ s'écrit $(x!^j[x_1, ..., x_n]Q, id_!, E_!)$,

tels que $E_?(y) = E_!(x)$

on définit $id_* = \mathbf{Noeud}((i,j), id_?, id_!)$

on note

$$f_* : Ag \mapsto \left(Ag, id_*, \begin{cases} z \mapsto E_?(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(y?^i[y_1, ..., y_n]P) \\ y_k \mapsto E_!(x_k) & \text{si } y_k \in \mathfrak{FR}(Ag) \\ z \mapsto (z, id_*) & \text{si } \begin{cases} z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(y?^i[y_1, ..., y_n]P) \\ z \notin \{y_k | k \in [|1; n|]\} \end{cases} \end{cases} \right)$$

et

$$f_! : Ag \mapsto \left(Ag, id, \begin{cases} z \mapsto E(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(x!^j[x_1, ..., x_n]Q) \\ z \mapsto (z, id_!) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(x!^j[x_1, ..., x_n]Q) \end{cases} \right)$$

on a alors $C \overset{(?^i, !^j)}{\rightarrow_2} C'$,

où $C' = (C \setminus \{\mu\}) \cup (f_*(Agent(P))) \cup (f_!(Agent(Q)))$

Exemple 3.2.1 On donne ci-dessous un exemple de dérivation pour l'exemple du serveur redéfini figure 3.5

Figure 3.5 Serveur

$$\begin{aligned} \mathbf{Allouer} := {} & *\text{make}?^1[](\nu \textit{canal-entrée})(\nu \textit{canal-sortie})(\nu \textit{info-client}) \\ & (\textit{canal-entrée}!^2[\textit{info-client}] \\ & |\textit{canal-entrée}?^3[\textit{info-reçue}](\textit{canal-sortie}!^4[\textit{info-reçue}] \\ & \quad | \text{ make}!^5[])) \\ \mathbf{Serveur} := {} & (\nu \text{make})(\mathbf{Allouer} \mid \text{make}!^6[] \mid \text{make}!^7[] \mid \text{make}!^8[]) \end{aligned}$$

$$\text{On a } C_0(\mathbf{Serveur}) = \left\{ \begin{pmatrix} 1, \varepsilon, \{\text{make} \mapsto (\text{make}, \varepsilon)\} \end{pmatrix} \begin{pmatrix} 6, \varepsilon, \{\text{make} \mapsto (\text{make}, \varepsilon)\} \end{pmatrix} \begin{pmatrix} 7, \varepsilon, \{\text{make} \mapsto (\text{make}, \varepsilon)\} \end{pmatrix} \begin{pmatrix} 8, \varepsilon, \{\text{make} \mapsto (\text{make}, \varepsilon)\} \end{pmatrix} \right\}$$

Puis $C_0(\mathbf{Serveur})$

$$\xrightarrow{(1,6)}_2 \left\{ \begin{array}{l} \left(1, \varepsilon, \left\{ \text{make} \quad \mapsto (\text{make}, \varepsilon) \right\}\right) \\ \left(7, \varepsilon, \left\{ \text{make} \quad \mapsto (\text{make}, \varepsilon) \right\}\right) \\ \left(8, \varepsilon, \left\{ \text{make} \quad \mapsto (\text{make}, \varepsilon) \right\}\right) \\ \left(2, (1,6), \left\{ \begin{array}{ll} \textit{canal-entrée} & \mapsto (\textit{canal-entrée}, (1,6)) \\ \textit{info-client} & \mapsto (\textit{info-client}, (1,6)) \end{array} \right.\right) \\ \left(3, (1,6), \left\{ \begin{array}{ll} \textit{canal-entrée} & \mapsto (\textit{canal-entrée}, (1,6)) \\ \textit{canal-sortie} & \mapsto (\textit{canal-sortie}, (1,6)) \\ \text{make} & \mapsto (\text{make}, \varepsilon) \end{array} \right.\right) \end{array} \right\}$$

$$\xrightarrow{(3,2)}_2 \left\{ \begin{array}{l} \left(1, \varepsilon, \left\{ \text{make} \quad \mapsto (\text{make}, \varepsilon) \right\}\right) \\ \left(7, \varepsilon, \left\{ \text{make} \quad \mapsto (\text{make}, \varepsilon) \right\}\right) \\ \left(8, \varepsilon, \left\{ \text{make} \quad \mapsto (\text{make}, \varepsilon) \right\}\right) \\ \left(4, (1,6), \left\{ \begin{array}{ll} \textit{info-reçue} & \mapsto (\textit{info-client}, (1,6)) \\ \textit{canal-sortie} & \mapsto (\textit{canal-sortie}, (1,6)) \end{array} \right.\right) \\ \left(5, (1,6), \left\{ \text{make} \quad \mapsto (\text{make}, \varepsilon) \right\}\right) \end{array} \right\}$$

3.2.6 Cohérence

Théorème 3.3 La fonction *identité* induit une *bisimulation faible* entre le système de transition $\rightarrow_1$ et $\rightarrow_2$, en prenant comme règles muettes les règles structurelles.

La démonstration est triviale.

3.2.7 Conclusion

On a ainsi défini une deuxième sémantique non-standard qui permet de s'affranchir des règles structurelles. La sémantique obtenue est donc plus simple et moins complexe à analyser et ce, sans perte de cohérence (Cf Théorème 3.3). On peut maintenant se demander s'il est possible d'aller encore plus loin dans cette direction, en factorisant les règles liées aux choix non-déterministes.

3.3 Sémantique relativiste

3.3.1 Principe

Si l'on s'intéresse uniquement aux transitions qui concernent les communications, il peut être gènant d'avoir dans son système de transition des règles liées aux choix non-déterministes.

On se propose donc de normaliser les dérivations admises par notre sémantique. On peut soit restreindre l'ensemble des dérivations à celles qui réalisent un minimum de choix non-déterministes, lorsqu'ils sont indispensables avant une communication, soit au contraire restreindre l'ensemble des dérivations à celles qui réalisent un maximum de choix non-déterministes après chaque communication. Dans les deux cas, on factorisera les choix non-déterministes avec la règle de communication.

Pour ce faire, on utilisera des marqueurs et une relation d'exclusion. Chaque tâche portera un ensemble de marqueurs, qui spécifiera ses contraintes d'exclusion avec les autres tâches. Dans le premier cas, les tâches seront insérées avec leurs marqueurs, mais lors d'une communication, toutes les tâches qui agissent élimineront les tâches portant des marqueurs concurrents, dans le deuxième cas, après une communication, on choisira un marqueur pour chaque classe d'exclusion, et seules les tâches portant exclusivement ces marqueurs seront insérées dans la nouvelle configuration.

3.3.2 Relation d'exclusion

Soit P_0 un processus clos étiqueté.

On note *Marqueur* un ensemble dénombrable infini, munie d'une fonction de choix $h_{Marqueur} : \wp(Marqueur) \to Marqueur$.

Définition 3.2 *Soit $A \subset Marqueur$,*

On appèle ensemble de classes d'exclusion sur A tout partitionnement de A.

Définition 3.3 *On appèlera tâche marquée de P_0, tout couple formé d'une tâche et d'un ensemble de marqueurs.*

Définition 3.4 *Soit C un ensemble de classes d'exclusion sur A, pour $x, y \in A$, on dira que x est en exclusion avec y, noté $x \not\forall y$ si et seulement si*

$$\begin{cases} x \neq y \\ \exists c \in C, x \in c \text{ et } y \in c \end{cases}$$

Etant donné un ensemble de classes d'exclusion sur $A \subseteq Marqueur$, on effectue des choix sur un ensemble de tâches marquées, en choisissant un représentant pour chaque classe d'exclusion. On oublie alors toutes les tâches qui portent un marqueur concurrent à un représentant.

On définit maintenant la fonction *Agent*, qui étant donné un ensemble de classes d'exclusion C sur un ensemble de marqueurs A et un sous-processus marqué P, donne un nouvel ensemble de classes d'exclusion C', avec $C \subseteq C'$ sur un nouvel ensemble de marqueurs, et l'ensemble des tâches mises en concurrence ou

en exclusion mutuelle dans le sous-processus P, les nouvelles exclusions mutuelles étant marquées par les nouveaux marqueurs.

Définition 3.5 *La définition de Agent est donné figure 3.6.*

Figure 3.6 la fonction *Agent*

$$
\begin{array}{rcl}
Agent(A,(\emptyset,S)) & = & (A,\{\}) \\
Agent(A,(x![x_1,...,x_n]P,S)) & = & (A,\{(x![x_1,...,x_n]P,S)\} \\
Agent(A,(y?[y_1,...,y_n]P,S)) & = & (A,\{(y?[y_1,...,y_n]P,S)\} \\
Agent(A,(*y?[y_1,...,y_n]P,S)) & = & (A,\{(*y?[y_1,...,y_n]P,S)\} \\
Agent(A,([x\diamond y]P,S)) & = & (A,\{([x\diamond y]P,S\}) \\
Agent(A,((\nu x)P,S)) & = & Agent(A,(P,S))
\end{array}
$$

$$
\begin{array}{l}
Agent(A,(P|Q,S)) = (A'',T_P \cup T_Q) \\
\text{où } (A',T_P) = Agent(A,(P,S)) \\
\text{et } (A'',T_Q) = Agent(A',(Q,S))
\end{array}
$$

$$
\begin{array}{l}
Agent(A,(P+Q,S)) = (A'',T_P \cup T_Q) \\
\text{où } \begin{cases}
g = h_{Marqueur}(Marqueur \setminus \bigcup A) \\
d = h_{Marqueur}(Marqueur \setminus ((\bigcup A) \cup \{d\})) \\
(A',T_P) = Agent(A \cup \{\{d;g\}\},(P,S \cup \{g\})) \\
(A'',T_Q) = Agent(A',(Q,S \cup \{d\}))
\end{cases}
\end{array}
$$

Définition 3.6 *Soit C un ensemble d'exclusion et T un ensemble de tâches marquées, On appelle valuation de C, toute fonction totale $\sigma : C \to \bigcup C$ telle que $\forall a \in C,\ \sigma a \in a$.*

On définit alors la restriction de T à σ :

$$\sigma(T) = \{t | \exists S, (t,S) \in T \text{ et } (\sigma C) \subseteq S\}$$

Exemple 3.3.1 On prend $(\mathbb{N}^*, min)$ comme ensemble de marqueurs,
on considère le processus $\mathbf{P}$::=(a$?^1$[]b$!^2$[])+(a$!^4$[]|(b$?^5$[]+b$!^6$[])),
on a alors $Agent(\emptyset,\mathbf{P}) = \left(\left\{ \begin{array}{l} \{1;2\} \\ \{3;4\} \end{array} \right\}, \left\{ \begin{array}{l} (a?^1[]b!^2[],\{1\}) \\ (a!^4[],\{2\}) \\ (b?^5[],\{2;3\}) \\ (b!^6[],\{2;4\}) \end{array} \right\} \right)$
On considère ensuite $\sigma : \begin{cases} \{1;2\} \mapsto 2 \\ \{3;4\} \mapsto 3 \end{cases}$
On a ainsi $\sigma(\mathbf{P})$={a$!^4$[];b$?^5$[]} □

3.3.3 Définitions

Avec cette sémantique, seuls les opérateurs ?, !, = et $\neq$ sont étiquetés.

Soit P_0 un processus clos étiqueté.

Une étape d'exécution est une paire (A, T) où A est un ensemble de classes d'exclusion et T un ensemble de tâches marquées par des éléments des classe de A.

Les autres définitions de la sémantique petit-pas (Cf 3.1.2) restent valables.

3.3.4 Traduction initiale

On définit l'état initial de notre sémantique :

On définit $C_0(P_0) = (A, \{((p, \varepsilon, E_p), S) | (p, S) \in T\})$

où $(A, T) = Agent(\emptyset, P_0)$ et $E_p = \begin{cases} \mathfrak{FR}(p) & \to \mathfrak{BR}(P_0) \times Id \\ x & \mapsto (x, \varepsilon) \end{cases}$

3.3.5 Transitions

On définit ensuite la relation de réduction de la sémantique relativiste.

– **Filtrage** $(\diamond \in \{=;\ \neq\})$

Soit (A, C) une étape d'exécution,

supposons que $\lambda \in C$,où $\lambda = (([x \diamond^i y]P, id, E), M)$

avec $E(x) \diamond E(y)$

on note $Exclu = \{a \in \bigcup A | \exists b \in M,\ a \not\Upsilon b\}$

on note $f_\diamond : (Ag, S) \mapsto \left(\left(Ag, id, \begin{cases} z \mapsto E(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}([x \diamond y]P) \\ z \mapsto (z, a) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}([x \diamond y]P) \end{cases} \right), S \right)$

on a alors $(A, C) \overset{\diamond^i}{\to}_3 (A', C')$

où $C' = \{(P, S \setminus M, id, E))|(T, S) \in (C \setminus \lambda) \text{ et } S \cap Exclu = \emptyset\} \cup (f_\diamond(T))$

et $(A', T) = Agent(A \setminus M, (P, \emptyset))$.

– **Communication**

Soit (A, C) une étape d'exécution,

Supposons que $\lambda, \mu \in C$,

où λ s'écrit $((y?^i[y_1, ..., y_n]P, id_?, E_?), M_?)$

et μ s'écrit $((x!^j[x_1, ..., x_n]Q, id_!, E_!), M_!)$,

tels que $E_?(y) = E_!(x)$ et $\forall (a, b) \in M_? \times M_!,\ a \uparrow b$

on note $M = M_? \cup M_!$

on note $Exclu = \{a \in \bigcup A | \exists b \in M,\ a \not\Upsilon b\}$

on note

$$f_? : Ag \mapsto \left(Ag, id_?, \begin{cases} z \mapsto E_?(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(y?^i[y_1, ..., y_n]P) \\ y_k \mapsto E_!(x_k) & \text{si } y_k \in \mathfrak{FR}(Ag) \\ z \mapsto (z, id_?) & \text{si } \begin{cases} z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(y?^i[y_1, ..., y_n]P) \\ z \notin \{y_k | k \in [|1; n|]\} \end{cases} \end{cases} \right)$$

et

$$f_! : (Ag, S) \mapsto \left(\left(Ag, id_!, \begin{cases} z \mapsto E(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(x!^j[x_1, ..., x_n]Q) \\ z \mapsto (z, id_!) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(x!^j[x_1, ..., x_n]Q) \end{cases} \right), S \right)$$

on a alors $(A, C) \overset{(?^i\ !^j)}{\to}_3 (A'', C')$,

$$\text{où } \begin{cases} (A', T) = Agent(A \setminus M, (P, \emptyset)) \\ (A'', T') = Agent(A', (Q, \emptyset)) \\ C' = \{(T, S \setminus M)|(T, S) \in (C \setminus \{\lambda, \mu\}) \text{ et } S \cap Exclu = \emptyset\} \cup (f_?T) \cup (f_!T') \end{cases}$$

– **Ressource**

Soit (A, C) une étape d'exécution,

Supposons que $\lambda, \mu \in C$,

où λ s'écrit $((*y?^i[y_1, ..., y_n]P, id_?, E_?), M_?)$

et μ s'écrit $((x!^j[x_1, ..., x_n]Q, id_!, E_!), M_!)$,

tels que $E_?(y) = E_!(x)$ et $\forall (a, b) \in M_? \times M_!,\ a \uparrow b$

on note $M = M_? \cup M_!$

on note $Exclu = \{a \in \bigcup A | \exists b \in M,\ a \not\gamma b\}$

on définit $id_* = \mathbf{Noeud}((i, j), id_?, id_!)$

on note

$$f_* : Ag \mapsto \left(Ag, id_*, \begin{cases} z \mapsto E_?(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(y?^i[y_1, ..., y_n]P) \\ y_k \mapsto E_!(x_k) & \text{si } y_k \in \mathfrak{FR}(Ag) \\ z \mapsto (z, id_*) & \text{si } \begin{cases} z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(y?^i[y_1, ..., y_n]P) \\ z \notin \{y_k | k \in [|1; n|]\} \end{cases} \end{cases} \right)$$

et

$$f_! : (Ag, S) \mapsto \left(\left(Ag, id_!, \begin{cases} z \mapsto E(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(x!^j[x_1, ..., x_n]Q) \\ z \mapsto (z, id_!) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(x!^j[x_1, ..., x_n]Q) \end{cases} \right), S \right)$$

on a alors $(A, C) \overset{(?^i, !^j)}{\rightarrow_3} (A'', C')$,

où $\begin{cases} (A', T) = Agent(A \setminus M, (P, \emptyset)) \\ (A'', T') = Agent(A', (Q, \emptyset)) \\ C' = \{(T, S \setminus M) | (T, S) \in (C \setminus \{\mu\}) \text{ et } S \cap Exclu = \emptyset\} \cup (f_* T) \cup (f_! T') \end{cases}$

Exemple 3.3.2 On donne figure 3.8 le graphe de toutes les dérivations possibles pour le processus donné figure 3.7,

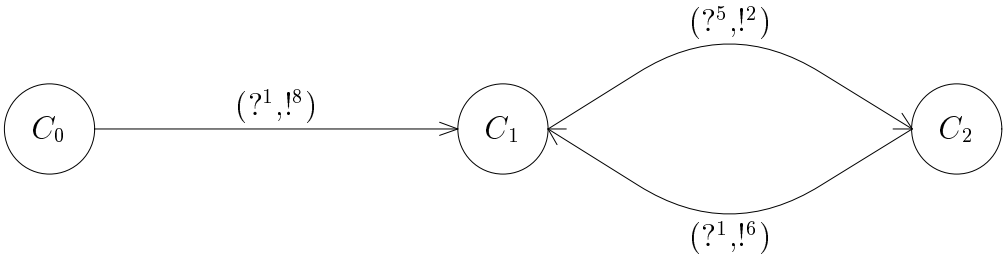
Figure 3.7 deux processus altérnés

A := $*$a?[1][x](x![2][a] + c?[3][u]d![4][u])
B := $*$b?[5][x](x![6][b] + c![7][e])
C := a![8][b]
P := **A** | **B** | **C**

avec

$$C_0 = C_0(\mathbf{P}) = \left(\{\}, \left\{ \begin{array}{l} \left(\left(1, \varepsilon, \begin{cases} \text{a} & \mapsto (\text{a}, \varepsilon) \\ \text{c} & \mapsto (\text{c}, \varepsilon) \\ \text{d} & \mapsto (\text{d}, \varepsilon) \end{cases} \right), \{\} \right) \\ \left(\left(5, \varepsilon, \begin{cases} \text{b} & \mapsto (\text{b}, \varepsilon) \\ \text{c} & \mapsto (\text{c}, \varepsilon) \\ \text{e} & \mapsto (\text{e}, \varepsilon) \end{cases} \right), \{\} \right) \\ \left(\left(8, \varepsilon, \begin{cases} \text{a} & \mapsto (\text{a}, \varepsilon) \\ \text{b} & \mapsto (\text{b}, \varepsilon) \end{cases} \right), \{\} \right) \end{array} \right\} \right)$$

Figure 3.8 graphe des dérivations



$$C_1 = \left(\{\{1;2\}\}, \left\{ \begin{array}{l} \left(\left(1, \varepsilon, \left\{ \begin{array}{ll} \mathrm{a} & \mapsto (\mathrm{a}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{d} & \mapsto (\mathrm{d}, \varepsilon) \end{array} \right. \right), \{\} \right) \\ \left(\left(5, \varepsilon, \left\{ \begin{array}{ll} \mathrm{b} & \mapsto (\mathrm{b}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{e} & \mapsto (\mathrm{e}, \varepsilon) \end{array} \right. \right), \{\} \right) \\ \left(\left(2, \varepsilon, \left\{ \begin{array}{ll} \mathrm{a} & \mapsto (\mathrm{a}, \varepsilon) \\ x & \mapsto (\mathrm{b}, \varepsilon) \end{array} \right. \right), \{1\} \right) \\ \left(\left(3, \varepsilon, \left\{ \begin{array}{ll} \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{d} & \mapsto (\mathrm{d}, \varepsilon) \end{array} \right. \right), \{2\} \right) \end{array} \right\} \right)$$

$$C_2 = \left(\{\{1;2\}\}, \left\{ \begin{array}{l} \left(\left(1, \varepsilon, \left\{ \begin{array}{ll} \mathrm{a} & \mapsto (\mathrm{a}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{d} & \mapsto (\mathrm{d}, \varepsilon) \end{array} \right. \right), \{\} \right) \\ \left(\left(5, \varepsilon, \left\{ \begin{array}{ll} \mathrm{b} & \mapsto (\mathrm{b}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{e} & \mapsto (\mathrm{e}, \varepsilon) \end{array} \right. \right), \{\} \right) \\ \left(\left(6, \varepsilon, \left\{ \begin{array}{ll} \mathrm{b} & \mapsto (\mathrm{b}, \varepsilon) \\ x & \mapsto (\mathrm{a}, \varepsilon) \end{array} \right. \right), \{1\} \right) \\ \left(\left(7, \varepsilon, \left\{ \begin{array}{ll} \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{e} & \mapsto (\mathrm{e}, \varepsilon) \end{array} \right. \right), \{2\} \right) \end{array} \right\} \right)$$

3.3.6 Cohérence

La sémantique relativiste n'est pas en *bisimulation* avec $\rightarrow$. En effet, le choix non-déterministe n'est pas une action muette, car elle engendre des conséquences irrémédiables.

Cependant, on peut montrer que $\rightarrow_2$ simule faiblement $\rightarrow_3$, de plus $\rightarrow_2$ et $\rightarrow_3$

ont la même sémantique collectrice.

Définition 3.7 *On note Σ, l'ensemble des étiquettes du système de transitions $\rightarrow_2$,*

On note Π_Σ la fonction de projection qui oublie toutes les occurences des étiquettes concernant les choix déterministes.

$$\Pi_\Sigma \ : \ \begin{cases} \Sigma & \rightarrow \Sigma \\ +^i_j & \mapsto \varepsilon \\ \lambda & \mapsto \lambda \end{cases}$$

On étend triviallement Π_Σ en Π_{Σ^*}

$$\Pi_{\Sigma^*} \ : \ \begin{cases} \Sigma^* & \rightarrow \Sigma^* \\ u_1...u_n & \mapsto \Pi_\Sigma(u_1)...\Pi_\Sigma(u_n) \end{cases}$$

Définition 3.8 *On note $\mathfrak{S}_2$ la sémantique collectrice de $\rightarrow_2$:*

$$\mathfrak{S}_2 = \left\{ (\Pi_{\Sigma^*}(u), C) \;\middle|\; \begin{array}{l} C_0(P_0) \stackrel{u}{\rightarrow}{}^*_2 C \\ \forall (P, id, E) \in C, P \text{ n'est pas de la forme } Q_1 +^i Q_2 \end{array} \right\}$$

Définition 3.9 *On note $\mathfrak{S}_3$ la sémantique collectrice de $\rightarrow_3$:*

$$\mathfrak{S}_3 = \left\{ (u, \sigma T) \;\middle|\; \begin{array}{l} C_0(P_0) \stackrel{u}{\rightarrow}{}^*_3 (A, T) \\ \sigma \text{ une valuation totale sur } A \end{array} \right\}$$

Théorème 3.4 $\mathfrak{S}_2 = \mathfrak{S}_3$

3.3.7 Conclusion

Cette sémantique est compliquée à formaliser, cependant elle offre un avantage certain. Ainsi, l'ensemble des configurations non-standard donne une bonne intuition du comportement d'un processus (Cf figure 3.8), car toutes les étapes intermédiaires dues à des règles parasites sont éliminées, néanmoins cette sémantique a tendance à cacher les *interblocages*, ainsi lorsqu'un choix non-déterministe peut conduire à un *interblocage*, celui-ci n'est jamais activé.

De plus, la présence des marqueurs est un nouvel obstacle pour l'abstraction de cette sémantique.

3.4 Sémantique gloutonne

3.4.1 Principe

Dans la sémantique gloutonne, les choix sont effectués immédiatement derrière les autres réductions actives. Dès lors, au cours d'une exécution, les tâches présentes dans les configurations ne portent pas de marqueurs. Les marqueurs servent juste à définir les étapes de transition. Ainsi, après chaque transition, on calcule l'ensemble des tâches étiquetées sous-jacent. On choisit alors une valuation totale de cet ensemble pour séléctionner les tâches qui resteront dans la nouvelle configuration.

3.4.2 Définitions

Avec cette sémantique, seuls les opérateurs ?, !, $=$ et $\neq$ sont étiquetés.

Une étape d'exécution est un ensemble de tâche non-marquées.

Les autres définitions de la sémantique relativiste (Cf 3.3.3), notamment la fonction *Agent*(figure 3.6) et la notion de classes d'exclusion (Cf 3.3.2), restent valables.

3.4.3 Traduction initiale

La sémantique gloutonne admet plusieurs états initiaux. Ainsi, si P_0 commence par un opérateur de choix non-déterministe, choisir un état initial revient à effectuer ce choix.

On définit l'ensemble des états initiaux de notre sémantique :

On note $(A_0, T) = Agent(\emptyset, (P_0, \emptyset))$.

$C_0(P_0) = \{\{(p, \varepsilon, E_p) \,|p \in \sigma(T)\} \,|\sigma$ valuation totale de $A_0\}$

et $E_p = \begin{cases} \mathfrak{F}\mathfrak{N}(p) & \to \mathfrak{B}\mathfrak{N}(P_0) \times Id \\ x & \mapsto (x, \varepsilon) \end{cases}$

3.4.4 Transitions

La relation de réduction se définit essentiellement comme celle de la sémantique relativiste. Cependant, après chaque transition, on choisit un ensemble d'agents grâce à une valuation totale :

– **Filtrage** $(\diamond \in \{=;\ \neq\})$

Soit C une étape d'exécution,

supposons que $\lambda \in C$,où $\lambda = ([x \diamond^i y]P, id, E)$

avec $E(x) \diamond E(y)$

on note $(A, T) = Agent(\emptyset, (P, \emptyset))$

soit alors σ une valuation totale sur A

on note $f_\diamond : Ag \mapsto \left(Ag, id, \begin{cases} z \mapsto E(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}([x \diamond y]P) \\ z \mapsto (z, a) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}([x \diamond y]P) \end{cases} \right)$

on a alors $C \xrightarrow{\diamond^i}_4 C'$

où $C' = (C \setminus \{\lambda\}) \cup (f_\diamond(\sigma T))$.

– **Communication**

Soit C une étape d'exécution,

Supposons que $\lambda, \mu \in C$,

où λ s'écrit $(y?^i[y_1, ..., y_n]P, id_?, E_?)$

et μ s'écrit $(x!^j[x_1, ..., x_n]Q, id_!, E_!)$,

tels que $E_?(y) = E_!(x)$

on note $(A_?, T_?) = Agent(\emptyset, (P, \emptyset))$

soit alors $\sigma_?$ une valuation totale sur $A_?$

on note $(A_!, T_!) = Agent(\emptyset, (Q, \emptyset))$

soit alors $\sigma_!$ une valuation totale sur $A_!$

on note

$$f_? : Ag \mapsto \left(Ag, id_?, \begin{cases} z \mapsto E_?(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(y?^i[y_1, ..., y_n]P) \\ y_k \mapsto E_!(x_k) & \text{si } y_k \in \mathfrak{FR}(Ag) \\ z \mapsto (z, id_?) & \text{si } \begin{cases} z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(y?^i[y_1, ..., y_n]P) \\ z \notin \{y_k | k \in [|1; n|]\} \end{cases} \end{cases} \right)$$

et

$$f_! : Ag \mapsto \left(Ag, id_!, \begin{cases} z \mapsto E(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(x!^j[x_1, ..., x_n]Q) \\ z \mapsto (z, id_!) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(x!^j[x_1, ..., x_n]Q) \end{cases} \right)$$

on a alors $C \xrightarrow{(?^i, !^j)}_4 C'$,

où $C' = (C \setminus \{\lambda, \mu\}) \cup (f_?(\sigma_? T_?)) \cup (f_!(\sigma_! T_!))$

– **Ressource**

Soit C une étape d'exécution,

Supposons que $\lambda, \mu \in C$,

où λ s'écrit $(*y?^i[y_1, ..., y_n]P, id_?, E_?)$

et μ s'écrit $(x!^j[x_1, ..., x_n]Q, id_!, E_!)$,

tels que $E_?(y) = E_!(x)$

on note $(A_?, T_?) = Agent(\emptyset, (P, \emptyset))$

soit alors $\sigma_?$ une valuation totale sur $A_?$

on note $(A_!, T_!) = Agent(\emptyset, (Q, \emptyset))$

soit alors $\sigma_!$ une valuation totale sur $A_!$

on définit $id_* = \mathbf{Noeud}((i, j), id_?, id_!)$

on note

$$f_* : Ag \mapsto \left(Ag, id_*, \begin{cases} z \mapsto E_?(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(y?^i[y_1, ..., y_n]P) \\ y_k \mapsto E_!(x_k) & \text{si } y_k \in \mathfrak{FR}(Ag) \\ z \mapsto (z, id_*) & \text{si } \begin{cases} z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(y?^i[y_1, ..., y_n]P) \\ z \notin \{y_k | k \in [|1; n|]\} \end{cases} \end{cases} \right)$$

et

$$f_! : Ag \mapsto \left(Ag, id, \begin{cases} z \mapsto E(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(x!^j[x_1, ..., x_n]Q) \\ z \mapsto (z, id_!) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(x!^j[x_1, ..., x_n]Q) \end{cases} \right)$$

on a alors $C \overset{(?^i, !^j)}{\rightarrow_4} C'$,

où $C' = (C \setminus \{\mu\}) \cup (f_*(\sigma_? T_?)) \cup (f_!(\sigma_! T_!))$

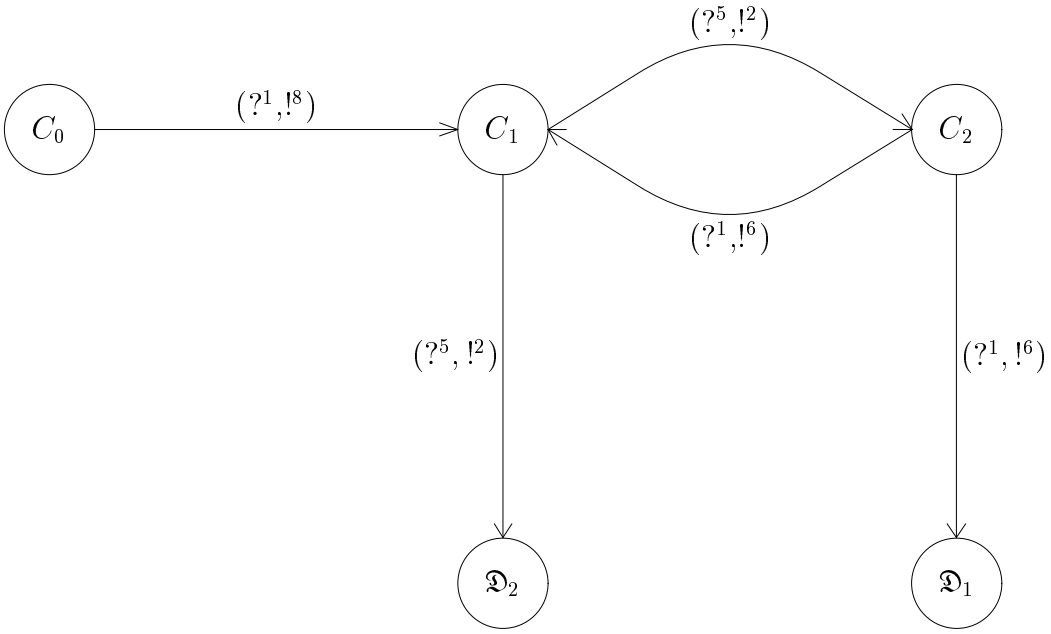
Exemple 3.4.1 On donne figure 3.9 le graphe des dérivations possible pour le processus donné figure 3.7.

avec

C_0, l'unique état initial de $C_0(\mathbf{P})$,

$$C_0 = \left\{ \begin{array}{l} \left(1, \varepsilon, \begin{cases} \mathrm{a} & \mapsto (\mathrm{a}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{d} & \mapsto (\mathrm{d}, \varepsilon) \end{cases} \right) \\ \left(5, \varepsilon, \begin{cases} \mathrm{b} & \mapsto (\mathrm{b}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{e} & \mapsto (\mathrm{e}, \varepsilon) \end{cases} \right) \\ \left(8, \varepsilon, \begin{cases} \mathrm{a} & \mapsto (\mathrm{a}, \varepsilon) \\ \mathrm{b} & \mapsto (\mathrm{b}, \varepsilon) \end{cases} \right) \end{array} \right\}$$

Figure 3.9 graphe des dérivations



$$C_1 = \left\{ \begin{array}{l} \left(1, \varepsilon, \begin{cases} \mathrm{a} & \mapsto (\mathrm{a}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{d} & \mapsto (\mathrm{d}, \varepsilon) \end{cases}\right) \\ \left(5, \varepsilon, \begin{cases} \mathrm{b} & \mapsto (\mathrm{b}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{e} & \mapsto (\mathrm{e}, \varepsilon) \end{cases}\right) \\ \left(2, \varepsilon, \begin{cases} \mathrm{a} & \mapsto (\mathrm{a}, \varepsilon) \\ x & \mapsto (\mathrm{b}, \varepsilon) \end{cases}\right) \end{array} \right\}$$

$$C_2 = \left\{ \begin{array}{l} \left(1, \varepsilon, \begin{cases} \mathrm{a} & \mapsto (\mathrm{a}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{d} & \mapsto (\mathrm{d}, \varepsilon) \end{cases}\right) \\ \left(5, \varepsilon, \begin{cases} \mathrm{b} & \mapsto (\mathrm{b}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{e} & \mapsto (\mathrm{e}, \varepsilon) \end{cases}\right) \\ \left(6, \varepsilon, \begin{cases} \mathrm{b} & \mapsto (\mathrm{b}, \varepsilon) \\ x & \mapsto (\mathrm{a}, \varepsilon) \end{cases}\right) \end{array} \right\}$$

$$\mathfrak{D}_1 = \left\{ \begin{array}{l} \left(1, \varepsilon, \begin{cases} \mathrm{a} & \mapsto (\mathrm{a}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{d} & \mapsto (\mathrm{d}, \varepsilon) \end{cases}\right) \\ \left(5, \varepsilon, \begin{cases} \mathrm{b} & \mapsto (\mathrm{b}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{e} & \mapsto (\mathrm{e}, \varepsilon) \end{cases}\right) \\ \left(4, \varepsilon, \begin{cases} \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{d} & \mapsto (\mathrm{d}, \varepsilon) \end{cases}\right) \end{array} \right\}$$

$$\mathfrak{D}_2 = \left\{ \begin{array}{l} \left(1, \varepsilon, \begin{cases} \mathrm{a} & \mapsto (\mathrm{a}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{d} & \mapsto (\mathrm{d}, \varepsilon) \end{cases}\right) \\ \left(5, \varepsilon, \begin{cases} \mathrm{b} & \mapsto (\mathrm{b}, \varepsilon) \\ \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{e} & \mapsto (\mathrm{e}, \varepsilon) \end{cases}\right) \\ \left(7, \varepsilon, \begin{cases} \mathrm{c} & \mapsto (\mathrm{c}, \varepsilon) \\ \mathrm{e} & \mapsto (\mathrm{e}, \varepsilon) \end{cases}\right) \end{array} \right\}$$

3.4.5 Cohérence

La sémantique gloutonne n'est en *bisimulation* ni avec la sémantique économe (Cf 3.2) ni avec la sémantique relativiste (Cf 3.3), car les choix non-déterministes ont des conséquences irrémédiables quant aux réductions qui les suivent.

Cependant, on peut montrer que $\rightarrow_2$ simule faiblement $\rightarrow_4$, de plus $\rightarrow_2$ et $\rightarrow_4$ ont la même sémantique collectrice.

Définition 3.10 *On note $\mathfrak{S}_4$ la sémantique collectrice de $\rightarrow_3$:*

$$\mathfrak{S}_4 = \left\{ (u, P) \;\middle|\; \exists i \in C_0(P_0),\ i \xrightarrow{u}{}^*_4 P \right\}$$

Théorème 3.5 $\mathfrak{S}_2 = \mathfrak{S}_4$

3.4.6 Conclusion

La sémantique gloutonne est toute aussi difficile à formaliser que la sémantique relativiste. Elle offre cependant plus d'avantages que cette dernière. En effet, l'ensemble de ses configurations non-standard donne une bonne intuition du comportement d'un processus (Cf figure 3.9), mais en plus il fait apparaître les *interblocages*. De plus, les marqueurs opèrent au sein des transitions et n'apparaissent plus aux côtés des sous-processus, ce qui facilitera l'élaboration d'une

abstraction de cette sémantique. Enfin, les valuations que l'on peut associer à chaque transition sont connues une fois pour toute, ce qui permettra une implantation efficace.

4 Interprétation Abstraite

4.1 Une théorie d'approximation discrète

L'interprétation abstraite [Cou78, Cou81, CC92a] est une théorie qui fournit un certain nombre d'outils pour définir et calculer des approximations de spécifications sémantiques concrètes.

Une spécification concrète est habituellement donnée par une structure partiellement ordonnée $\mathfrak{D}$, représentant un domaine sémantique dans lequel s'expriment des propriétés concrètes, un endomorphisme F de $\mathfrak{D}$ et un élément $\perp$ de $\mathfrak{D}$ tel qu'il existe un plus petit point fixe de F supérieur ou égal à $\perp$, noté $lfp_{\perp}\ F$. Cependant, $lfp_{\perp}\ F$ n'est pas toujours calculable, ni même finiment représentable. Pour y remédier, on a recours à des domaines abstraits, qui sont eux aussi des structures ordonnées, dans lesquels on peut concevoir une sémantique abstraite, qui sera une approximation de la sémantique concrète. Ainsi, cette sémantique passera sous silence certaines propriétés de la sémantique concrète. En contre-partie, la sémantique abstraite présentera l'avantage d'être calculable.

L'interprétation abstraite permet d'établir les liens entre les sémantiques concrètes et abstraites. Elle explique comment calculer des sémantiques abstraites, d'après une spécification concrète.

4.2 Sémantique concrète

On se place dans le cas où la sémantique concrète est donnée par un ordre partiel complet $(\mathfrak{D}, \sqsubseteq, \sqcup, \perp)$, et F est un endomorphisme continu au sens de Scott.

Définition 4.1 *Un ordre partiel $(\mathfrak{D}, \sqsubseteq)$ est complet si et seulement si toute partie de $\mathfrak{D}$ totalement ordonnée possède une borne supérieure dans $\mathfrak{D}$. On note alors $\sqcup$ l'opérateur qui, à toute partie totalement ordonnée de $\mathfrak{D}$, associe sa borne supérieure.*

Proposition 4.1 Un ordre partiel complet $(\mathfrak{D}, \sqsubseteq, \sqcup)$ possède un plus petit élément que l'on note $\perp$.

Définition 4.2 *Un endomorphisme F d'un ordre complet dans lui même est dit continu au sens de Scott si il conserve les bornes supérieures des parties totalement ordonnées.*

Dans ce cadre, le théorème de Kleene assure que F admet un plus petit point fixe. De plus, on a $lfp_{\perp}F = \bigsqcup_{n\in\mathbb{N}} F^n(\perp)$

Exemple 4.2.1 Soit $\searrow$ un système de transition sur un ensemble A, et i un élément de A,

la sémantique collectrice § du triplet $(A, \searrow, i)$ peut se définir comme le plus petit point fixe sur l'ordre partiel complet $(\wp(A), \subseteq, \cup, \emptyset)$ de l'endomorphisme continu

$$F = X \mapsto \{i\} \cup \{x' | \exists x \in X : x \searrow x'\}$$

On a alors

$$\S = \bigcup_{n \in \mathbb{N}} F^n(\bot)$$

□

Cependant, le calcul de $lfp_\bot F$ n'est pas effectif. Le but de l'interprétation abstraite est donc de calculer une approximation effective de ce point fixe. On utilise pour cela une sémantique abstraite, moins précise que la sémantique concrète. La sémantique abstraite oublie certaines propriétés exprimées par la sémantique concrète, en contre-partie la sémantique abstraite sera calculable. L'interprétation abstraite fournit alors les outils pour retranscrire la sémantique concrète en sémantique abstraite, et indique comment interpréter en terme d'approximation les résultats obtenus avec la sémantique abstraite.

4.3 Sémantique abstraite

On choisit pour domaine abstrait un ordre partiel noté $(\mathfrak{D}^\sharp, \precsim)$, qui admet un plus petit élément noté $\bot^\sharp$. Ce domaine abstrait est relié au domaine concret par une fonction de concrétisation γ.

- $\bot = \gamma(\bot^\sharp)$
- $\forall d_1^\sharp, d_2^\sharp \in \mathfrak{D}^\sharp, d_1^\sharp \precsim d_2^\sharp \implies \gamma(d_1^\sharp) \subseteq \gamma(d_2^\sharp)$

4.4 Transfert de point fixe

On se propose maintenant de mimer le calcul de la sémantique concrète dans le domaine abstrait. Pour cela on a recours à une fonction abstraite $F^\sharp$ croissante qui vérifie la condition de cohérence suivante :

$$\forall d^\sharp \in \mathfrak{D}^\sharp,\ [F \circ \gamma](d^\sharp) \sqsubseteq [\gamma \circ F^\sharp](d^\sharp)$$

Le théorème 4.1 établit maintenant la correspondance entre la sémantique concrète $lfp_\bot(F)$ et la sémantique abstraite :

Théorème 4.1 Sous ces conditions, on a $\mathrm{lfp}_\bot(F\ ) \subseteq \left(\bigsqcup_{n \in \mathbb{N}} \left(\gamma \left({F^\sharp}^n(\bot^\sharp) \right) \right) \right)$.

4.5 Opérateur d'élargissement

Un opérateur d'élargissement [Cou81, CC92a] est un accélérateur de convergence qui permet de calculer, à partir de la sémantique abstraite, une approximation de la sémantique concrète de manière effective.

Définition 4.3 *Un opérateur* $\nabla : \mathfrak{D}^\sharp \times \mathfrak{D}^\sharp \to \mathfrak{D}^\sharp$, *est un opérateur d'élargissement si et seulement si il vérifie les conditions suivantes :*

- $\forall d_1^\sharp,\ d_2^\sharp \in \mathfrak{D}^\sharp,\ d_1\sharp \precsim d_1^\sharp \nabla d_2^\sharp$ *et* $d_2\sharp \precsim d_1^\sharp \nabla d_2^\sharp$.
- $\forall (d_n)$ *suite croissante d'éléments de* $\mathfrak{D}^\sharp$ *la suite* (d_n^∇) *définie ci dessous :*

$$\begin{cases} d_0^\nabla = d_0^\sharp \\ d_{n+1}^\nabla = d_n^\nabla \nabla d_{n+1}^\sharp \end{cases}$$

est stationnaire.

On définit maintenant l'itération abstraite avec élargissement (F_n^∇) ci dessous :

$$\begin{cases} F_0^\nabla & = \bot^\sharp \\ F_{n+1}\nabla & = \begin{cases} F_n^\nabla & \text{si } F^\sharp(F_n^\nabla) \precsim F_n^\nabla \\ F_n^\nabla \nabla F^\sharp(F_n^\nabla) & \text{sinon} \end{cases} \end{cases}$$

Théorème 4.2 Itération Abstraite [CC92a, CC92b])
L'itération abstraite avec élargissement est stationnaire. De plus si l'on note $\S^\sharp$ sa limite, on a $\S \sqsubseteq \gamma(\S^\sharp)$

Ce théorème permet donc de concevoir une approximation effective de la sémantique concrète.

4.6 Algèbre des domaines

On se propose maintenant de fournir des outils pour construire des domaines abstraits à partir d'autres domaines abstraits.

4.6.1 Produit

Soit $(\mathfrak{D}_1^\sharp, \precsim_1, \bot_1^\sharp)$, $(\mathfrak{D}_2^\sharp, \precsim_2, \bot_2^\sharp)$ deux ordres partiels pointés reliés à un ordre complet $(\mathfrak{D}, \subseteq, \sqcup, \bot)$ par deux fonctions de concrétisation γ_1 et γ_2.
On note :

- $\mathfrak{D}^\sharp = \mathfrak{D}_1^\sharp \times \mathfrak{D}_2^\sharp$
- $\precsim$ défini par $(a_1, a_2) \precsim (b_1, b_2)$ si et seulement si $a_1 \precsim_1 b_1$ et $a_2 \precsim_1 b_2$

- $\bot^\sharp = (\bot_1^\sharp, \bot_2^\sharp)$
- $\gamma(a_1, a_2) = \gamma_1(a_1) \cap \gamma_2(a_2)$

Théorème 4.3 $(\mathfrak{D}^\sharp, \precsim, \bot^\sharp)$ est un ordre partiel pointé et γ est une fonction de concrétisation.

De plus si ∇_1 et ∇_2 sont respectivement des opérateurs d'élargissement pour $\mathfrak{D}_1^\sharp$ et $\mathfrak{D}_2^\sharp$, alors ∇ défini par $(a_1, a_2)\nabla(b_1, b_2) = (a_1 \nabla_1 b_1, a_2 \nabla_2 b_2)$ est un opérateur d'élargissement pour $\mathfrak{D}^\sharp$.

4.6.2 Partitionnement

Soit $(\mathfrak{D}^\sharp, \precsim, \bot^\sharp)$ un ordre partiel relié à un ordre complet pointé $(\mathfrak{D}, \subseteq, \sqcup, \bot)$ par une fonction de concrétisation γ.

Soit A un ensemble fini.

On note :

- $\mathfrak{D}_A^\sharp = \mathcal{F}(A \to \mathfrak{D}^\sharp)$
- $\precsim_A$ défini par $f_1^\sharp \precsim f_2^\sharp$ si et seulement si $\forall a \in A,\ f_1^\sharp(a) \precsim f_2^\sharp(a)$
- $\bot_A^\sharp = (a \mapsto \bot^\sharp)$
- $\gamma_A(f^\sharp) = \bigcup_{a \in A} \gamma(a)$

Théorème 4.4 $(\mathfrak{D}_A^\sharp, \precsim_A, \bot_A)$ est un ordre partiel pointé et γ_A est une fonction de concrétisation.

De plus si ∇ est un opérateur d'élargissement pour $\mathfrak{D}^\sharp$, alors ∇_A défini par

$$[(f_1^\sharp)\nabla_A(f_2^\sharp)](a) = [f_1^\sharp(a)]\nabla[f_2^\sharp(a)]$$

est un opérateur d'élargissement pour $\mathfrak{D}^\sharp$.

Le partitionnement permet de séparer le résultat des étapes de l'itération abstraite, ce qui conduit généralement à une analyse plus fine. Néanmoins, le partitionnement requiert un espace mémoire plus conséquent.

4.6.3 Réduction

Soit $(\mathfrak{D}^\sharp, \precsim, \bot^\sharp)$ un ordre partiel pointé relié à un ordre complet $(\mathfrak{D}, \subseteq, \sqcup, \bot)$ par une fonction de concrétisation γ. Soient F un endomorphisme de $\mathfrak{D}$ et $F^\sharp$ un endormorphisme de $\mathfrak{D}^\sharp$ vérifiants les conditions de 4.1

Définition 4.4 *On appelle opérateur de réduction, tout endomorphisme ρ de $\mathfrak{D}$ qui vérifie les conditions suivantes :*

- $\forall d^\sharp \in D^\sharp,\ \gamma(d^\sharp) \subseteq \gamma(\rho d^\sharp)$

– $\forall d^\sharp \in D^\sharp,\ \rho(d^\sharp) \precsim (d^\sharp)$

Théorème 4.5 Soit ρ un opérateur de réduction, L'endomorphisme $\rho \circ F^\sharp$ vérifie alors les conditions de 4.1. De plus, l'inégalité suivante est vérifiée :

$$lfp_\perp(F\ ) \subseteq \left(\bigsqcup_{n\in\mathbb{N}} \left(\gamma\left([\rho \circ F^\sharp]^n \perp^\sharp\right)\right)\right) \subseteq \left(\bigsqcup_{n\in\mathbb{N}} \left(\gamma\left({F^\sharp}^n \perp^\sharp\right)\right)\right)$$

5 Sémantique abstraite

5.1 Sémantique collectrice

Soit P_0 un processus du π-calcul étiqueté selon les règles de la sémantique gloutonne.

La sémantique collectrice d'un système représente l'ensemble des états que peut prendre ce système.

On note $\mathfrak{Conf}$ l'ensemble des étapes d'exécutions possibles associées à ce terme,

$$\mathfrak{Conf} = \wp\left(\left\{(P, id, E) \;\middle|\; \begin{array}{l} P \text{ est un sous processus de } P_0 \\ E \text{ est un environnement de domaine } \mathfrak{fn}(P) \end{array} \right\}\right)$$

On rappelle que Σ désigne l'alphabet de toutes les étiquettes associées au système de transition $\rightarrow_4$.

Définition 5.1 *On appelle sémantique collectrice du processus P_0, l'ensemble suivant :*

$$\mathfrak{S} = \{(u, C) \in (\Sigma^* \times \mathfrak{Conf}) | \exists c_0 \in C_0(P_0)\ c_0 \stackrel{u}{\rightarrow}_4 C\}$$

On peut définir $\mathfrak{S}$ comme le plus petit point fixe de l'endomorphisme continu F sur le treillis complet $(\wp(\mathfrak{Conf}), \subseteq, \cup, \bot, \cap, \top)$ défini ci-dessous :

$$F(X) = \{(\varepsilon, c_0) | c_0 \in C_0(P_0)\} \cup \{(u.w, c) | \exists c' \in \mathfrak{Conf},\ (u, c') \in X \text{ et } c' \stackrel{w}{\rightarrow}_4 c\}$$

Conformément à la méthodologie introduite précédemment (Cf 4), on utilise un domaine abstrait pour approximer $\mathfrak{S}$.

5.2 Domaines abstraits

Le domaine abstrait que l'on va utiliser sera générique, ce qui permettra selon les problèmes que l'on se pose sur les termes à analyser de l'instancier avec les domaines abstraits adéquats.

Pour définir notre analyse, on a recours à une fonction de partitionnement et à deux domaines abstraits. La fonction de partitionnement définit un critère d'observation sur les étapes d'éxécution en vue de séparer les différentes étapes de calculs (Cf 4.6.2). Le premier domaine abstrait représentera l'interaction entre les canaux, il décrira ainsi le lien entre les identifiants des processus et leurs environnements. Le deuxième domaine décrit quant à lui la syntaxe des étapes d'éxécutions, il servira ainsi à compter les sous-processus présents.

Notre analyse dépend uniquement des paramètres suivants :

- f, une fonction de partitionnement $f : \mathfrak{Conf} \rightarrow A$ où A est un ensemble fini quelconque. Le rôle de cette fonction est de partitionner les calculs

de l'itération abstraite selon des critères observables sur les configurations non-standards.

Exemple 5.2.1 On peut prendre $A = \{0; 1\}$ et $f \ : \ C \mapsto \mathit{Card}(C) \ \mathit{mod}\ 2$ □

- $Id_1^\sharp$ et $Id_2^\sharp$, deux treillis pour représenter les identifiants.
  - $(Id_1^\sharp, \sqsubseteq_1^\sharp, \sqcup_1^\sharp, \bot_1^\sharp, \sqcap_1^\sharp, \top_1^\sharp)$
    $Id_1^\sharp$ est un treillis utilisé pour abstraire des ensembles d'identifiants. $Id_1^\sharp$ est relié à Id par une fonction de concrétisation γ_1.

    $$\gamma_1 \ : \ Id_1^\sharp \mapsto \wp(Id)$$

  - $(Id_2^\sharp, \sqsubseteq_2^\sharp, \sqcup_2^\sharp, \bot_2^\sharp, \sqcap_2^\sharp, \top_2^\sharp)$
    $Id_2^\sharp$ est un treillis utilisé pour abstraire des ensembles de couples d'identifiants. $Id_2^\sharp$ est relié à Id par la fonction de concrétisation γ_2.

    $$\gamma_2 \ : \ Id_2^\sharp \mapsto \wp(Id \times Id)$$

- $\mathfrak{V}$, un treillis pour représenter l'aspect syntaxique des étapes d'exécutions.
  $\mathfrak{V}$ est relié à $\wp(\Sigma^*, \mathfrak{Conf})$ par la fonction de concrétisation $\gamma_{\mathfrak{V}}$.

  $$\gamma_{\mathfrak{V}} \ : \ \mathfrak{V} \to \wp(\Sigma^*, \mathfrak{Conf})$$

  En pratique, ce domaine abstrait ignore toutes les propriétés qui peuvent porter sur les identifiants des tâches et sur les identifiants des variables, il sert ainsi à représenter des multi-ensembles de sous-processus. Son rôle est alors de compter les processus présents au cours des différentes éxécutions du processus analysé.

On construit maintenant notre domaine abstrait à partir de ces données génériques :

On note $\mathfrak{Pro}$ l'ensemble de sous-processus de P_0.

Le domaine abstrait $C_{Pro}^\sharp$ associe à chaque sous-processus l'ensemble des identifiants que l'on peut lui affecter au cours d'une exécution.

$$C_{Pro}^\sharp = \mathfrak{Pro} \to Id_1^\sharp$$

On note $\mathfrak{Can}$ l'ensemble des triplets (p,x,y) où $p \in \mathfrak{Pro}$, x$\in \mathfrak{FN}(p)$ et y$\in \mathfrak{BN}(P_0)$. $\mathfrak{Can}$ est l'ensemble de toutes les relations syntaxiques entre les canaux, ainsi la relation syntaxique (p,x,y), désigne le fait que l'occurence syntaxique du canal x dans le sous-processus p peut être affecté par un canal créé par le lieur νy.

Le domaine abstrait $C^\sharp_{Com}$ associe à chaque relation syntaxique (P, x, y) un couple d'identifiants. La première composante représente l'ensemble des identifiants du processus dont x est variable libre, la deuxième donne l'ensemble des identifiants des tâches qui sont succeptibles d'avoir créé le canal y.

$$C^\sharp_{Com} = \mathfrak{Can} \to Id^\sharp_2$$

On définit enfin $C^\sharp$ notre treillis abstrait principal :

$$\mathfrak{C}^\sharp \ : \ A \to (\mathfrak{V} \times C^\sharp_{Pro} \times C^\sharp_{Com})$$

On définit ensuite la fonction de concrétisation Γ :

$$\Gamma : \left\{ \begin{array}{ll} \mathfrak{C}^\sharp \to & \wp(\Sigma^* \times \mathfrak{Conf}) \\ f^\sharp \mapsto & \left\{ (w,c) \left| \begin{array}{l} (w,c) \in \gamma_v(v) \\ (P,a,E) \in c \text{ et } E(x) = (y,b) \implies (a,b) \in \gamma_2(com(P,x,y)) \\ (P,a,E) \in c \implies a \in \gamma_1(pro(P)) \\ \text{où } (v, pro, com) = f^\sharp(f(w,c)) \end{array} \right. \right\} \end{array} \right.$$

5.3 Primitives abstraites

Afin de mimer la fonction concrète dans notre treillis abstrait, on a recours à des primitives abstraites génériques, qui opèrent sur les treillis $Id^\sharp_1$, $Id^\sharp_2$, $\mathfrak{V}$ ainsi que sur la fonction de partitionnement. La cohérence de la sémantique abstraite sera assurée par des contraintes que l'on impose à ces primitives.

5.3.1 Partitionnement

– *possible* : $A \times \mathfrak{V} \times \Sigma \to \wp(A)$
Intuitivement, $possible(a, v, \lambda)$ est l'ensemble des images par f de toutes les configurations concrètes que l'on peut obtenir en effectuant une transition λ à partir d'une configuration c compatible avec le critère d'observation a (i.e. $f(c) = a$) et avec l'information v, acquise sur la struture syntaxique de la configuration c (i.e $c \in \gamma(v)$).

La fonction *possible* devra ainsi respecter la contrainte de cohérence suivante :

$$\left\{ f(u.\lambda, c') \left| \exists u \in \Sigma^*, \ \exists \lambda \in \Sigma, \exists C', C \in \mathfrak{Conf} \text{ tel que } \begin{array}{l} f(u, C) = a \\ (u, C) \in \gamma_V(v) \\ C \xrightarrow{\lambda}_4 C' \end{array} \right. \right\}$$

$$\subseteq possible(a, v, \lambda)$$

5.3.2 Primitives Logiques

Des fonctions de logiques usuelles opèrent sur les treillis $Id_1^\sharp$ et $Id_2^\sharp$:

- $\Pi_1 : Id_2^\sharp \to Id_1^\sharp$

  $\Pi_1(c)$ donne l'ensemble des premières composantes des couples représentés par c.

$$\{u \in Id \,|\, \exists v \in Id \text{ tel que } (u,v) \in \gamma_2(c)\} \subseteq \gamma_1(\Pi_1(c))$$

- $\Pi_2 : Id_2^\sharp \to Id_1^\sharp$

  $\Pi_2(c)$ donne l'ensemble des deuxièmes composantes des couples représentés par c.

$$\{v \in Id \,|\, \exists u \in Id \text{ tel que } (u,v) \in \gamma_2(c)\} \subseteq \gamma_1(\Pi_2(c))$$

- $Inj_1 : Id_1^\sharp \to Id_2^\sharp$

  $Inj_1(c)$ donne l'ensemble des couples dont la première composante est un élément réprésenté par c.

$$\{(u,v) \in Id \times Id \,|\, u \in \gamma_1(c)\} \subseteq \gamma_2(Inj_1(c))$$

- $Inj_2 : Id_1^\sharp \to Id_2^\sharp$

  $Inj_2(c)$ donne l'ensemble des couples dont la deuxième composante est un élément représenté par c.

$$\{(u,v) \in Id \times Id \,|\, v \in \gamma_1(c)\} \subseteq \gamma_2(Inj_2(c))$$

- $\underset{= \to}{\bowtie} : Id_2^\sharp \times Id_2^\sharp \to Id_2^\sharp$

  $\underset{= \to}{\bowtie}(c,c')$ détermine l'ensemble des couples de la forme (a,b) tels qu'il existe c tel que (c,a) (resp. (c,b)) soit un élément représenté par c (resp. c')

$$\left\{(u,v) \in Id \times Id \,\middle|\, \exists w \in \Sigma^* \;\begin{array}{c}(w,u) \in \gamma_2(c)\\(w,v) \in \gamma_2(c')\end{array}\right\} \subseteq \gamma_2(\underset{= \to}{\bowtie}(c,c'))$$

– $\underset{\leftarrow =}{\bowtie} : Id_2^\sharp \times Id_2^\sharp \to Id_2^\sharp$

$\underset{\leftarrow =}{\bowtie}(c, c')$ détermine l'ensemble des couples de la forme (a, b) tels qu'il existe c tel que (b, c) (resp. (a, c)) soit un élément représenté par c (resp. c')

$$\left\{(u,v) \in Id \times Id \,\middle|\, \exists w \in Id \text{ tel que } \begin{array}{l}(v,w) \in \gamma_2(c)\\(u,w) \in \gamma_2(c')\end{array}\right\} \subseteq \gamma_2(\underset{\leftarrow =}{\bowtie}(c, c'))$$

– *existe-différent* $: Id_2^\sharp \times Id_2^\sharp \to Id_1^\sharp$

existe-différent(c, c') représente l'ensemble des identifiants a, tels qu'il existe deux identifiants distincts b et b' tels que (a, b) (resp. (a, b')) soit un élément représenté par c (resp. c').

$$\left\{ u \in Id \,\middle|\, \exists v \in Id,\ \exists w \in Id,\ \text{ tel que } \begin{array}{l}(u,v) \in \gamma_2(c)\\(u,w) \in \gamma_2(c')\\v \neq w\end{array}\right\}$$

$$\subseteq \gamma_2(\textit{existe-différent}(c, c'))$$

5.3.3 Formation des identifiants

Les primitives suivantes servent à créer les identifiants initiaux :

– $\varepsilon_1^\sharp \in Idp$

$\varepsilon_1^\sharp$ représente le singleton feuille vide :

$$\{\varepsilon\} \subseteq \gamma_1(\varepsilon_1^\sharp)$$

– $\varepsilon_2^\sharp \in Idc$

$\varepsilon_2^\sharp$ représente le singleton dont les deux composantes sont la feuille vide :

$$\{(\varepsilon, \varepsilon)\} \subseteq \gamma_2(\varepsilon_2^\sharp)$$

Alors que les primitives suivantes servent à construire les identifiants par récurrence :

– $push_1^{(i,j)} : Id_1^\sharp \times Id_1^\sharp \to Id_1^\sharp$

$push_1^{(i,j)}(c_g, c_d)$ forme l'ensemble des arbres ayant pour noeud (i, j), pour fils gauche un élément représenté par c_g et pour fils droit un élément représenté par c_d.

$$\left\{\mathbf{Noeud}((i,j),u,v) \,\middle|\, \begin{array}{l}u \in \gamma_1(c_g)\\v \in \gamma_1(c_d)\end{array}\right\} \subseteq \gamma_1(push_1^{(i,j)}(c_g, c_d))$$

– $push_2^{(i,j)} \ : \ Id_1^\sharp \times Id_2^\sharp \to Id_2^\sharp$

$push_2^{(i,j)}(c_g, c_d)$ forment l'ensemble des couples d'arbres dont la première composante a pour noeud (i,j), pour fils gauche un élément représenté par c_g, et pour fils droit la première composante d'un élément représenté par $c-d$, et dont la deuxième composante est la deuxième composante du même élément représenté par c_d.

$$\left\{ (\mathbf{Noeud}((i,j),u,v),w) \;\middle|\; \begin{array}{l} u \in \gamma_1(c_g) \\ (v,w) \in \gamma_2(c_d) \end{array} \right\} \subseteq \gamma_2(push_2^{(i,j)}(c_g, c_d))$$

– $dpush \ : \ Id_1^\sharp \to Id_2^\sharp$

$dpush(c)$ forme l'ensemble des couples dont les deux composantes sont égales à un même élément représenté par c

$$\{(u,u) | u \in \gamma_1(c)\} \subseteq \gamma_2(dpush(c))$$

5.3.4 Compteur de tâche

Une primitive abstraite forme les compteurs initiaux :

– $Init_{\mathfrak{V}} \ : \ \wp_{\text{finies}}(\mathfrak{Conf}) \to \mathfrak{V}$

$Init_{\mathfrak{V}}$ est une fonction de concrétisation.

$$\{(\varepsilon, c) | c \in C\} \subseteq \gamma_{\mathfrak{V}}(Init_{\mathfrak{V}}(C))$$

Deux primitives agissent sur le domaine $\mathfrak{V}$.

– $nonzero \ : \mathfrak{V} \times A \to \wp(\mathfrak{Pro})$

$nonzero(v,a)$ donne l'ensemble des sous-processus qui peuvent être présents dans une étape d'exécution représentée par v et compatible avec le critère d'observation a

$$\left\{ p \in \mathfrak{Pro} \;\middle|\; \exists (u,C) \in \gamma_V(v),\ \exists id, E \text{ tels que } \begin{cases} f(u,C) = a \\ (p, id, E) \in C \end{cases} \right\}$$

$$\subseteq nonzero(v,a)$$

– $trans$: $(\wp(\mathfrak{Pro}) \times \wp(\mathfrak{Pro}) \times \wp(\mathfrak{Pro}) \times \Sigma \times \mathfrak{V} \times A) \to \mathfrak{V}$

Son rôle est de détecter si une transition est possible à partir d'une étape d'exécution, lorsque l'on connait une abstraction de cette dernière dans $\mathfrak{V}$. Elle donne ensuite une abstraction de l'ensemble des étapes d'exécution qui peuvent résulter d'une telle transition et qui sont compatibles avec un critère d'observation donné.

L'éficacité de ce domaine dépand essentiellement des propriétés que l'on peut calculer dans ledomaine $\mathfrak{V}$.

$$\left\{ (u.\lambda, C) \;\middle|\; \exists C' \begin{cases} (u, C') \in \gamma_{\mathfrak{V}}(v) \\ \forall p \in P_{req},\ \exists id, E,\ (p, id, E) \in C' \\ \forall p \in \mathfrak{Pro} \begin{cases} \sharp(p)(C) = \sharp(p)(C') & \text{si } p \notin P_{lost} \cup P_{new} \\ \sharp(p)(C) + 1 = \sharp(p)(C') & \text{si } p \in P_{lost} \setminus P_{new} \\ \sharp(p)(C) = \sharp(p)(C') + 1 & \text{si } p \in P_{new} \setminus P_{lost} \\ \sharp(p)(C) = \sharp(p)(C') & \text{si } p \in P_{lost} \cap P_{new} \end{cases} \\ f(u.\lambda, C) = a \end{cases} \right\}$$

$$\subseteq \gamma_{\mathfrak{V}}(trans(P_{req}, P_{lost}, P_{new}, \lambda, v, a))$$

où $\sharp(p)(c)$ désigne le nombre d'instance syntaxique portant sur le sous-processus p dans l'étape d'exécution c.

5.4 Etat initial

On définit maintenant $f_0^\sharp$, la représentation de l'ensemble des états initiaux associés au processus P_0, pour la sémantique gloutonne.

$$f_0^\sharp(a) = (v_0^\sharp(a), p_0^\sharp(a), c_0^\sharp(a))$$

où

– $v_0^\sharp(a) = Init_{\mathfrak{V}}(C_a)$

– $p_0^\sharp(a) = \begin{cases} \mathfrak{Pro} \to Id_1^\sharp \\ P \mapsto \varepsilon_1^\sharp & \text{si } \exists c \in C_a,\ \exists E,\ (P, \varepsilon, E) \in c \\ P \mapsto \bot_1^\sharp & \text{sinon} \end{cases}$

– $c_0^\sharp(a) = \begin{cases} \mathfrak{Can} \to Id_2^\sharp \\ (P,x,y) \mapsto \varepsilon_2^\sharp \quad \text{si } \exists c \in C_a,\ \exists E,\ \begin{cases} (P,\varepsilon,E) \in c \\ x \in \mathfrak{FR}(P) \\ x = y \end{cases} \\ (P,x,y) \mapsto \bot_2^\sharp \quad \text{sinon} \end{cases}$

avec

$$C_a = \{c \in C_0(P_0) | f(\varepsilon, c) = a\}$$

5.5 Tables de transitions

– **Test (=)**
Soit $(v^\sharp, c_p^\sharp, c_c^\sharp) = f^\sharp(a)$, $u \in Can$ et $[x =^i y]P$ un sous-processus,
et $a' \in possible(a, v^\sharp, =^i)$ tels que :
$c_c^\sharp([x =^i y]P, x, u) = id_\sharp$
$c_c^\sharp([x =^i y]P, y, u) = id'_\sharp$
$id_\cap = id_\sharp \cap id'_\sharp \neq \bot_2$
On note $(M,T) = Agent(P,\emptyset)$
Soit σ une valuation totale sur M,
$v' = trans(\{[x =^i y]P\}, \{[x =^i y]P\}, \sigma T, =^i, v^\sharp, a') \neq \bot_V$
On note $\mathfrak{P} = nonzero(v', a')$
on a alors :
$f^\sharp \rightharpoonup (a' \mapsto (v', c'_p, c'_c))$
où

$$c'_p = \begin{cases} & \{p \mapsto id \in c_p^\sharp | p \in \mathfrak{P}\} \\ \bigcup & \{p \mapsto \Pi_1 id_\cap | p \in \mathfrak{P} \cap (\sigma T)\} \end{cases}$$

$$c'_c = \begin{cases} & \{(p,z,t) \mapsto id \in c_c^\sharp | p \in \mathfrak{P}\} \\ \bigcup & \left\{ (p,z,u) \mapsto id_\cap \;\middle|\; \begin{array}{l} p \in \mathfrak{P} \cap (\sigma T) \\ z \in \{x;y\} \cap \mathfrak{FR}(P) \end{array} \right\} \\ \bigcup & \left\{ (p,z,z) \mapsto dpush(\Pi_1 id_\cap) \;\middle|\; \begin{array}{l} p \in \mathfrak{P} \cap (\sigma T) \\ z \in \mathfrak{BR}(P) \cap \mathfrak{FR}(p) \} \end{array} \right\} \\ \bigcup & \left\{ (p,z,t) \mapsto c_c^\sharp([x =^i y]P, z, t) \cap Inj_1(\Pi_1 id_\cap) \;\middle|\; \begin{array}{l} p \in \mathfrak{P} \cap (\sigma T) \\ z \in (\mathfrak{FR}(P) \cap \mathfrak{FR}(p)) \setminus \{x;y\} \end{array} \right\} \end{cases}$$

– **Test ($\neq$)**

Soit $(v^\sharp, c_p^\sharp, c_c^\sharp) = f^\sharp(a)$, $u, v \in Can$ et $[x \neq^i y]P$ un sous-processus avec $x \neq y$ et $a' \in possible(a, v^\sharp, \neq^i)$ tels que :

$c_c^\sharp([x \neq^i y]P, x, u) = id_\sharp$

$c_c^\sharp([x \neq^i y]P, y, v) = id'_\sharp$

$$\perp_1 \neq id_\cap = \begin{cases} (\Pi_1 id_\sharp) \cap (\Pi_1 id'_\sharp) & \text{si } u \neq v \\ existe\text{-}différent(id_\sharp, id'_\sharp) & \text{sinon} \end{cases}$$

On note $(M, T) = Agent(P, \emptyset)$

Soit σ une valuation totale sur M,

$v' = trans(\{[x \neq^i y]P\}, \{[x \neq^i y]P\}, \sigma T, \neq^i, v^\sharp, a') \neq \perp_V$

On note $\mathfrak{P} = nonzero(v', a')$

on a alors :

$f^\sharp \rightharpoonup (a' \mapsto (v', c'_p, c'_c))$

$$\text{où} \quad c'_p = \begin{cases} & \{p \mapsto id \in c_p^\sharp | p \in \mathfrak{P}\} \\ \bigcup & \{p \mapsto id_\cap | p \in \mathfrak{P} \cap (\sigma T)\} \end{cases}$$

$$\text{et} \quad c'_c = \begin{cases} & \{(p,z,t) \mapsto id \in c_c^\sharp | p \in \mathfrak{P}\} \\ \bigcup & \left\{ (p,z,z) \mapsto dpush(id_\cap) \;\middle|\; \begin{array}{l} p \in \mathfrak{P} \cap (\sigma T) \\ z \in \mathfrak{BN}(P) \cap \mathfrak{FN}(p) \end{array} \right\} \\ \bigcup & \left\{ (p,z,t) \mapsto c_c^\sharp([x \neq^i y]P, z, t) \cap Inj_1(id_\cap) \;\middle|\; \begin{array}{l} p \in \mathfrak{P} \cap (\sigma T) \\ z \in \mathfrak{FN}(P) \cap \mathfrak{FN}(p) \end{array} \right\} \end{cases}$$

– **Communication**

Soit $(v^\sharp, c_p^\sharp, c_c^\sharp) = f^\sharp(a)$, $u \in Can$ et $y?^i[y_1,...,y_n]P$, $x!^j[x_1,...,x_n]Q$ deux sous-processus,

et $a' \in possible(a, v^\sharp, \neq^i)$ tels que :

$c_c^\sharp(y?^i[y_1,...y_n]P, y, u) = id_\sharp^?$

$c_c^\sharp(x!^j[x_1,...x_n]Q, x, u) = id_\sharp^!$

$\bot_1 \neq id_{can} = (\Pi_2(id_\sharp^?)) \cap (\Pi_2(id_\sharp^!))$

On note $(M_?, T_?) \triangleq Agent(P, \emptyset)$

et $(M_!, T_!) \triangleq Agent(Q, \emptyset)$

Soit $\sigma_?$ (resp. $\sigma_!$) une valuation totale sur $M_?$ (resp. $M_!$),

$v' \triangleq trans(\{\lambda;\mu\}, \{\lambda;\mu\}, (\sigma_? T_?) \cup (\sigma_! T_!), (?^i, !^j), v^\sharp, a') \neq \bot_V$

où $\begin{cases} \lambda = y?^i[y_1,...y_n]P \\ \mu = x!^j[x_1,...x_n]Q \end{cases}$

On note $\mathfrak{P} \triangleq nonzero(v', a')$

$idpro^? \triangleq \Pi_1(id_\sharp^? \cap Inj_2 id_{can})$

$idpro^! \triangleq \Pi_1(id_\sharp^! \cap Inj_2 id_{can})$

$id_k^t \triangleq \underset{=\,\rightarrow}{\bowtie} (\underset{\leftarrow\,=}{\bowtie} (id_\sharp^?, id_\sharp^!), c_c^\sharp(x!^j[x_1,...,x_n]Q, x_k, t)) \cap Inj_1(idpro^?)$

on a alors :

$f^\sharp \rightharpoonup (a' \mapsto (v', c_p', c_c'))$

où $c_p' = \begin{cases} \{p \mapsto id \in c_p^\sharp | p \in \mathfrak{P}\} \\ \bigcup \{p \mapsto idpro^? | p \in \mathfrak{P} \cap (\sigma_? T_?)\} \\ \bigcup \{q \mapsto idpro^! | q \in \mathfrak{P} \cap (\sigma_! T_!)\} \end{cases}$

et $c_c' = \begin{cases} \{(p,z,t) \mapsto id \in c_c^\sharp | p \in \mathfrak{P}\} \\ \bigcup \left\{ (p,z,z) \mapsto dpush(idpro^?) \;\middle|\; \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_? T_?) \\ z \in (\mathfrak{BR}(P) \cap \mathfrak{FR}(p)) \setminus \{y_i\} \end{array} \right\} \\ \bigcup \left\{ (p, y_k, t) \mapsto id_k^t \;\middle|\; \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_? T_?) \\ y_k \in \mathfrak{BR}(P) \cap \mathfrak{FR}(p) \end{array} \right\} \\ \bigcup \left\{ (q,z,z) \mapsto dpush(idpro^!) \;\middle|\; \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_! T_!) \\ z \in \mathfrak{BR}(Q) \cap \mathfrak{FR}(q) \end{array} \right\} \\ \bigcup \left\{ (p,z,t) \mapsto c_c^\sharp(y?^i[y_1,...y_n]P, z, t) \cap Inj_1(idpro^?) \;\middle|\; \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_? T_?) \\ z \in (\mathfrak{FR}(P) \cap \mathfrak{FR}(p)) \setminus \{y\} \end{array} \right\} \\ \bigcup \left\{ (p,y,u) \mapsto c_c^\sharp(y?^i[y_1,...y_n]P, y, u) \cap Inj_2(id_{can}) \;\middle|\; \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_? T_?) \\ y \in (\mathfrak{FR}(P) \cap \mathfrak{FR}(p)) \end{array} \right\} \\ \bigcup \left\{ (q,z,t) \mapsto c_c^\sharp(x!^i[x_1,...,x_n]Q, z, t) \cap Inj_1(idpro^!) \;\middle|\; \begin{array}{l} q \in \mathfrak{P} \cap (\sigma_! T_!) \\ z \in (\mathfrak{FR}(Q) \cap \mathfrak{FR}(q)) \setminus \{x\} \end{array} \right\} \\ \bigcup \left\{ (q,x,u) \mapsto c_c^\sharp(x!^i[x_1,...,x_n]Q, x, u) \cap Inj_2(id_{can}) \;\middle|\; \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_! T_!) \\ x \in (\mathfrak{FR}(Q) \cap \mathfrak{FR}(q)) \end{array} \right\} \end{cases}$

– **Ressource**

Soit $(v^\sharp, c_p^\sharp, c_c^\sharp) = f^\sharp(a)$, $u \in Can$ et $*y?^i[y_1,...,y_n]P$, $x!^j[x_1,...,x_n]Q$ deux sous-processus,

et $a' \in possible(a, v^\sharp, \neq^i)$ tels que :

$c_c^\sharp(y?^i[y_1,...y_n]P, y, u) = id_\sharp^?$

$c_c^\sharp(x!^j[x_1,...x_n]Q, x, u) = id_\sharp^!$

$\bot_1 \neq id_{can} = (\Pi_2(id_\sharp^?)) \cap (\Pi_2(id_\sharp^!))$

On note $(M_?, T_?) \triangleq Agent(P, \emptyset)$

et $(M_!, T_!) \triangleq Agent(Q, \emptyset)$

Soit $\sigma_?$ (resp. $\sigma_!$) une valuation totale sur $M_?$ (resp. $M_!$),

$v' = trans(\{y?^i[y_1,...y_n]P; x!^j[x_1,...x_n]Q\}, \{x!^j[x_1,...x_n]Q\}, (\sigma_? T_?) \cup (\sigma_! T_!), (?^i, !^j), v^\sharp, a')$

$\neq \bot_V$

On note $\mathfrak{P} \triangleq nonzero(v', a')$

$idpro^? \triangleq \Pi_1(id_\sharp^? \cap Inj_2 id_{can})$

$idpro^! \triangleq \Pi_1(id_\sharp^! \cap Inj_2 id_{can})$

$id_{s,t} \triangleq Inj_1(idpro^!) \cap \underset{=\rightarrow}{\bowtie} (\underset{\leftarrow=}{\bowtie} (id^!, id^?), c_c^\sharp(y?^i[y_1,...y_n]P, s, t))$

$id_k^t \triangleq c_c^\sharp(x!^i[x_1,...,x_n]Q, x_k, t)$

on a alors :

$f^\sharp \rightharpoonup (a' \mapsto (v', c_p', c_c'))$

où $c_p' = \left\{ \begin{array}{ll} & \{p \mapsto id \in c_p^\sharp | p \in \mathfrak{P}\} \\ \bigcup & \{p \mapsto push_1^{(i,j)}(idpro^?, idpro^!) | p \in \mathfrak{P} \cap (\sigma_? T_?)\} \\ \bigcup & \{q \mapsto idpro^! | q \in \mathfrak{P} \cap (\sigma_! T_!)\} \end{array} \right.$

et $c_c' = \left\{ \begin{array}{ll} & \{(p,z,t) \mapsto id \in c_c^\sharp | p \in \mathfrak{P}\} \\ \bigcup & \left\{ (p,z,z) \mapsto dpush(push_1^{(i,j)}, idpro^?, idpro^!) \left| \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_? T_?) \\ z \in (\mathfrak{BR}(P) \cap \mathfrak{FR}(p)) \setminus \{y_i\} \end{array} \right. \right\} \\ \bigcup & \left\{ (p, y_k, t) \mapsto push_2^{(i,j)}(idpro^?, id_k^t) \left| \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_? T_?) \\ y_k \in \mathfrak{BR}(P) \cap \mathfrak{FR}(p) \end{array} \right. \right\} \\ \bigcup & \left\{ (q,z,z) \mapsto dpush(idpro^!) \left| \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_! T_!) \\ z \in \mathfrak{BR}(Q) \cap \mathfrak{FR}(q) \end{array} \right. \right\} \\ \bigcup & \left\{ (p,z,t) \mapsto push_2^{(i,j)}(idpro^?, id_{z,t}) \left| \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_? T_?) \\ z \in (\mathfrak{FR}(P) \cap \mathfrak{FR}(p)) \setminus \{y\} \end{array} \right. \right\} \\ \bigcup & \left\{ (p,y,u) \mapsto push_2^{(i,j)}(idpro^?, id_{y,u}) \cap Inj_2(id_{can}) \left| \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_? T_?) \\ y \in (\mathfrak{FR}(P) \cap \mathfrak{FR}(p)) \end{array} \right. \right\} \\ \bigcup & \left\{ (q,z,t) \mapsto c_c^\sharp(x!^i[x_1,...,x_n]Q, z, t) \cap Inj_1(idpro^!) \left| \begin{array}{l} q \in \mathfrak{P} \cap (\sigma_! T_!) \\ z \in (\mathfrak{FR}(Q) \cap \mathfrak{FR}(q)) \setminus \{x\} \end{array} \right. \right\} \\ \bigcup & \left\{ (q,x,u) \mapsto c_c^\sharp(x!^i[x_1,...,x_n]Q, x, u) \cap Inj_2(id_{can}) \left| \begin{array}{l} p \in \mathfrak{P} \cap (\sigma_! T_!) \\ x \in (\mathfrak{FR}(Q) \cap \mathfrak{FR}(q)) \end{array} \right. \right\} \end{array} \right.$

5.6 Cohérence

On définit alors la fonction abstraite $F^\sharp$:

$$F^\sharp(f^\sharp) = f_0^\sharp \sqcup \bigsqcup \{f'^\sharp | f^\sharp \rightharpoonup f'^\sharp\}$$

Théorème 5.1 Les conditions de cohérences imposées aux primitives abstraites suffisent à vérifier l'équation suivante :

$$\Gamma \circ F \sqsubseteq F^\sharp \circ \Gamma$$

Ce qui assure la cohérence de l'approximation.

6 Problèmes d'analyse et domaines abstraits

On présente maintenant divers problèmes d'analyse. Pour chaque problème, on montrera comment instancier notre analyseur générique pour pouvoir le traiter.

6.1 Analyse du flux de controle

6.1.1 Principe

L'analyse de flux est la plus simple des abstractions de la sémantique du π-calcul. L'analyse de flux consiste à calculer pour chaque canal syntaxique, quel opérateur νx a pu les créer. Cette analyse ne repose que sur des considérations purement syntaxiques. On ne fera donc pas la différence entre les deux noms de canaux crées par deux instances syntaxiques (P, id) et (P, id'), lorsque $id \neq id'$. Une telle analyse à déjà été proposée par Bodei, Degano, Nielson et Nielson [BDNN98]. Notre analyseur générique peut effectuer une analyse de flux, et ce avec des domaines abstraits très simples.

6.1.2 Domaines abstraits

On peut effectuer une analyse de flux sans partitionnement et sans compter les instances syntaxiques des sous-processus. On peut en outre se contenter du treillis $\{\bot; \top\}$ pour représenter les identificateurs :

- **partitionnement**

  On prend A égal à un singleton $\{a\}$.

- **identificateurs**

  On prend $Id_p = (\{\bot_1, \top_1\}$ et $Id_c = (\{\bot_2, \top_2\}$

  Les fonctions de concrétisation sont

$$\gamma_p = \begin{cases} \bot_1 \mapsto \emptyset \\ \top_1 \mapsto Id \end{cases} \quad \text{et } \gamma_c = \begin{cases} \bot_2 \mapsto \emptyset \\ \top_2 \mapsto Id \times Id \end{cases}$$

- **compteur de processus**

  On prend V égal à un singleton $\{\top_v\}$.

$$\gamma_v = \left\{ \top_v \mapsto \Sigma^* \times \mathfrak{Conf} \right.$$

6.1.3 Primitives abstraites

Les primitives abstraites se définissent trivialement :

- **partitionnement**
  - *possible* est la fonction constante égale à a
- **Primitives logiques**
  -
  $$\Pi_1 = \Pi_2 = \begin{cases} \bot_2 \mapsto \bot_1 \\ \top_2 \mapsto \top_1 \end{cases}$$
  -
  $$Inj_1 = Inj_2 = \begin{cases} \bot_1 \mapsto \bot_2 \\ \top_1 \mapsto \top_2 \end{cases}$$
  -
  $$\underset{= \rightarrow}{\bowtie} = \underset{\leftarrow =}{\bowtie} = \begin{cases} \bot_2, \bot_2 \mapsto \bot_2 \\ \top_2, \bot_2 \mapsto \bot_2 \\ \bot_2, \top_2 \mapsto \bot_2 \\ \top_2, \top_2 \mapsto \top_2 \end{cases}$$
  -
  $$\textit{existe-différent} = \begin{cases} \bot_2, \bot_2 \mapsto \bot_1 \\ \top_2, \bot_2 \mapsto \bot_1 \\ \bot_2, \top_2 \mapsto \bot_1 \\ \top_2, \top_2 \mapsto \top_1 \end{cases}$$
- **Formation des identifiants**
  -
  $$\varepsilon_1^\sharp = \top_1$$
  -
  $$\varepsilon_2^\sharp = \top_2$$

–

$$push_1^{(i,j)} = \begin{cases} \bot_1, \bot_1 \mapsto \bot_1 \\ \top_1, \bot_1 \mapsto \bot_1 \\ \bot_1, \top_1 \mapsto \bot_1 \\ \top_1, \top_1 \mapsto \top_1 \end{cases}$$

–

$$push_2^{(i,j)} = \begin{cases} \bot_1, \bot_2 \mapsto \bot_2 \\ \top_1, \bot_2 \mapsto \bot_2 \\ \bot_1, \top_2 \mapsto \bot_2 \\ \top_1, \top_2 \mapsto \top_2 \end{cases}$$

–

$$dpush = \begin{cases} \bot_1 \mapsto \bot_2 \\ \top_2 \mapsto \top_2 \end{cases}$$

– **Compteur de tâche**

  – $Init_{\mathfrak{V}}$ est la fonction constante égale à $\top_v$
  – *nonzero* est la fonction constante égale à $\mathfrak{Pro}$
  – *trans* est la fonction constante égale à $\top_v$

6.1.4 Résultats

L'analyse de flux n'est pas précise. Elle ne peut distinguer les processus récursifs que s'ils n'exportent pas les canaux qu'ils créent vers les autres processus, de plus, cette analyse ne permet de gérer des processus en exclusion mutuelle.

Néanmoins, notre analyse est plus précise que celle de Degano, Bodei, Nielson et Nielson [BDNN98]. En effet, le point fixe de la fonction abstraite $F^\sharp$ est calculé de manière exacte (sans élargissement), ainsi la sémantique abstraite est le plus petit point fixe de $F^\sharp$. Or ce plus petit point fixe peut être vu comme la plus petite solution d'un système de contraintes. Ceci nous permet de comparer les deux analyses, car toute contrainte induite par $F^\sharp$ est nécessairement une contrainte du système de la sémantique de Bodei, Degano, Nielson et Nielson. La plus petite solution de notre système est donc plus petite que la plus petite solution de leur système d'équations.

Exemple 6.1.1 Pour le processus $((\nu\mathrm{a})(\nu\mathrm{b})(\mathrm{a}?[x](\mathrm{a}![\mathrm{b}]|x![\mathrm{a}]))$,

– L'analyse de flux de Bodei, Degano, Nielson et Nielson ne permet pas de montrer que la variable x n'est jamais affectée au canal b.

– Notre analyse détecte que la variable x n'est jamais affectée au moindre canal. □

En fait, ces deux analyses mettent en jeu exactement les mêmes contraintes, seule la formulation est différente. Cependant, notre analyse construit dynamiquement l'ensemble des contraintes au cours de l'itération de la fonction abstraite. Celle de Degano, Bodei, Nielson et Nielson construit l'ensemble de contraintes directement en permettant à tous les rédex de se rencontrer.

6.2 Contraintes d'exclusion mutuelle

6.2.1 Présentation

Au cours d'une exécution d'un processus du π-calcul, il peut arriver que deux sous-processus ne soient jamais présents simultanément dans la même étape d'exécution. Ces processus ne peuvent donc pas à forciori communiquer entre eux. Cela pose un problème d'analyse, en effet, au cours d'une itération abstraite, les différentes étapes d'exécution sont mélangées et si l'itérateur ne détecte pas une contrainte d'exclusion mutuelle, il permet à ces deux processus de communiquer et l'analyse n'est pas précise.

Les contraintes d'exclusion mutuelle sont fortement liées à l'opérateur de choix non-déterministe, cependant certains processus peuvent comporter des exclusions mutuelles sans pour autant comporter cet opérateur.

Exemple 6.2.1 Le processus (a?[]**A** | a?[]**B** | a![]) se comporte exactement comme le processus **A** + **B**, pourvu que a ne soit pas une variable libre ni du processus **A** ni du processus **B**. □

Le problème d'exclusion mutuelle n'est traité ni par l'analyse de Bodei, Degano, Nielson et Nielson où l'opérateur d'exclusion a la même sémantique que l'opérateur de mise en concurrence, ni par celle d'Arnaud Venet [Ven98] où cet opérateur n'est pas présent dans la syntaxe.

Exemple 6.2.2 On donne figure 6.1 un exemple de processus dans lequel un problème d'exclusion mutuelle se pose :

Figure 6.1 deux processus alternés

$\mathbf{A} := *\mathrm{a}?^{1}[\mathrm{x}](\mathrm{x}!^{2}[\mathrm{a}] + \mathrm{c}?^{3}[u]\mathrm{d}!^{4}[u])$
$\mathbf{B} := *\mathrm{b}?^{5}[\mathrm{x}](\mathrm{x}!^{6}[\mathrm{a}] + \mathrm{c}!^{7}[\mathrm{e}])$
$\mathbf{C} := \mathrm{a}!^{8}[\mathrm{b}]$
$\mathbf{P} := \mathbf{A} \mid \mathbf{B} \mid \mathbf{C}$

En effet, les ressources **A** et **B** ne peuvent se répliquer simultanément et le communication entre $\mathrm{c}?^{3}[u]\mathrm{d}!^{4}[u]$ et $\mathrm{c}!^{7}[\mathrm{e}]$ n'a jamais lieu. □

Cependant, ni l'analyse de Bodei, Degano, Nielson et Nielson, ni celle d'Arnaud Venet ne détecte l'impossibilité de cette communication. □

6.2.2 Relation d'égalités linéaires entre variables

Pour résoudre ce problème, il faut pouvoir trouver une formulation pour les contraintes d'exclusion mutuelle. Il faut donc être capable d'exprimer le fait que de deux processus, un seul peut être présent à la fois. Ceci peut se faire gràce à des équations linéaires. On pourra en notant x et y le nombres d'occurences textuelles de deux processus dictincts au cours d'une exécution capturer la contrainte $x+y = 1$.

Le domaine de Karr [Kar76] est le domaine abstrait des espaces affines de dimension finie. Etant donné un ensemble de variables fixé, le domaine de Karr fournit un ensemble d'algorithmes pour manipuler des contraintes linéaires entre ces variables.

Définition 6.1 *soit V un ensemble de variables, On note N_V le domaine des contraintes linéaires entre ces variables.*

Chaque élément du domaine $\mathrm{N_V}$ est représenté par des matrices à valeurs dans $\mathbb{Q}$.

Exemple 6.2.3 On note V=$\{v_1, v_2, v_3, v_4\}$. La matrice $\mathbf{M}$ représente l'espace affine donné par le système d'equations cartésiennes $\mathbf{S}$.

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 4 & 5 \end{pmatrix} \quad \mathbf{S} \ : \ \begin{cases} v_1 + v_3 + 2.v_4 &= 3 \\ v_2 + 2.v_3 + 4.v_5 &= 5 \end{cases}$$

□

En opérant sur les lignes des matrices, on peut calculer une forme normale pour représenter les espaces affines. Les matrices normales sont appelées matrices échelonnées.

Définition 6.2 *Soit $\boldsymbol{M}$ une matrice ayant m lignes et n colonnes,*
Pour toutes lignes i, on note $p(i)$ le plus petit indice, s'il existe, tel que $\boldsymbol{M}_{i,p(i)}$ soit non nul.

Une matrice est échelonnée si elle vérifie les deux conditions suivantes :

- $\forall i \in [|1; m|],\ p(i)$ *existe et* $p(i) = 1$
- $\forall i \in [|1; m|],\ \forall k < i, \boldsymbol{M}_{k,p(i)} = 0$

Le domaine de Karr fournit alors les primitives suivantes :

- **l'intersection**

  L'intersection, $\cap_{\mathrm{N_V}} : \mathrm{N_V} \times \mathrm{N_V} \mapsto \mathrm{N_V}$, est l'opérateur qui associe à deux espaces affines leur intersection.

- **l'union affine**

  L'union affine, $\cup_{\mathrm{N_V}} : \mathrm{N_V} \times \mathrm{N_V} \mapsto \mathrm{N_V}$, est l'opérateur qui associe à deux espaces affines leur enveloppe affine.

- **le test d'inclusion**

  On note le test d'inclusion $\subseteq_{\mathrm{N_V}}$.

- **l'incrémentation**

  L'incrémentation $inc^{v_i}_{\mathrm{N_V}} : \mathrm{N_V} \to \mathrm{N_V}$, définit la translation des espaces affines par un vecteur unité positif. Ainsi $inc^{v_i}_{\mathrm{N_V}}(A)$ représente l'espace affine obtenu en incrémentant de un la variable v_i dans tous les points de l'espace A.

- **la décrémentation**

  La décrémentation $dec^{v_i}_{\mathrm{N_V}} : \mathrm{N_V} \to \mathrm{N_V}$ définit la translation des espaces affines par un vecteur unité négatif. Ainsi $dec^{v_i}_{\mathrm{N_V}}(A)$ représente l'espace affine obtenu en décrémentant de un la variable v_i dans tous les points de l'espace A

- **la projection**

  La projection $proj^{v_i}_{\mathrm{N_V}} : \mathrm{N_V} \to \mathrm{N_V}$ définit l'oubli d'une variable. Ainsi $proj^{v_i}_{\mathrm{N_V}}(A)$ représente de plus petit espace affine contenant A et stable par le vecteur v_i

Remarque 6.2.1 *Il est en fait possible d'effectuer tout changement de variable affine.*

6.2.3 Intervalles

Cependant un problème subsiste. Comment interpréter des contraintes linéaires pour calculer des contraintes d'exclusion mutuelle. En fait, les contraintes d'exclusion mutuelle du type $x + y = 1$ n'apparaissent pas telles quelles sdans les matrices.

Une résolution exacte du système d'équations relève de *l'arithmétique de Pressburger* et serait en coût exponentiel, ce que l'on ne peut accepter.

Pour remédier à cela, on propose de réaliser le produit réduit (Cf 4.6.1) entre le treillis de Karr et un domaine non relationnel d'intervalle.

Définition 6.3 *On définit* $(I, \subseteq_I, \cup_I, \bot_I, \cap_I, \top_I)$ *le domaine abstrait suivant :*

- I *est l'ensemble des intervalles de* $[|0; \infty|[$
- $\subseteq_I$ *est l'inclusion d'intervalles*
- $\cup_I$ *est l'union d'intervalles*
- $\bot_I = \emptyset$
- $\cap_I$ *est l'intersection d'intervalles*
- $\top_I = [|0; \infty|[$

On munit ce domaine de deux primitives abstraites :

- inc_I

  inc_I *est l'incrémentation.*

$$inc_I = \begin{cases} \emptyset & \mapsto \emptyset \\ [|a;b|] & \mapsto [|a+1; b+1|] \\ [|a;\infty|[& \mapsto [|a+1;\infty|[\end{cases}$$

- dec_I

  dec_I *est la décrémentation.*

$$dec_I = \begin{cases} \emptyset & \mapsto \emptyset & \\ [|a;b|] & \mapsto \emptyset & si\ b < 1 \\ [|a;b|] & \mapsto [|Max\{a-1;0\}; b-1|] & sinon \\ [|a;\infty|[& \mapsto [|Max\{a-1;0\};\infty|[& \end{cases}$$

Le domaine des intervalles n'est pas de hauteur de bornée, on a donc recours à un opérateur d'élargissement.

Définition 6.4 *On définit ci-dessous est un opérateur d'élargissement* ∇_I^n *:*

$$\begin{cases} [|a;b|] & \nabla_I^n & \emptyset & = [|a;b|] & \\ \emptyset & \nabla_I^n & [|a;b|] & = [|a;b|] & \\ [|a;b|] & \nabla_I^n & [|c;d|] & = [|min\{a;c\}; max\{b;d\}|] & si\ d \leqslant n \\ [|a;b|] & \nabla_I^n & [|c;d|] & = [|min\{a;c\};\infty|[& si\ d > n \\ [|a;\infty|[& \nabla_I^n & [|c;d|] & = [|min\{a;c\};\infty|[& \\ [|a;\infty|[& \nabla_I^n & [|c;\infty|[& = [|min\{a;c\};\infty|[& \\ [|a;b|] & \nabla_I^n & [|c;\infty|[& = [|min\{a;c\};\infty|[& \end{cases}$$

Définition 6.5 *Pour V, un ensemble de variables, on définit le treillis fonctionnel* $I_V = (V) \to I$ *où les opérations* $(\subseteq_{I_V}, \cup_{I_V}, \bot_{I_V}, \cap_{I_V}, \top_{I_V}, \nabla)$ *sont définies point par point.*

On munit I_V *des primitives suivantes :*

- $inc_{I_V}^{v_i} I^\sharp = \begin{cases} v \mapsto inc_I(I^\sharp(v)) & si\ v = v_i \\ v \mapsto I^\sharp(v) & sinon \end{cases}$

- $dec_{I_V}^{v_i} I^\sharp = \begin{cases} v \mapsto dec_I(I^\sharp(v)) & si\ v = v_i \\ v \mapsto I^\sharp(v) & sinon \end{cases}$

6.2.4 Réduction

Il faut maintenant définir la réduction entre le domaine des relations affines et le domaine des intervalles. En effet, le domaine des relations affines permet d'exprimer des contraintes de la forme $x + y = 1$, le domaine des intervalles permet d'exprimer des contraintes de la forme $x \geqslant 1,\ y \geqslant 1$. La réduction permettrait donc de détecter si ces contraintes sont incompatibles.

Pour effectuer la réduction, on va définir un ensemble de contraintes redondantes. En effet, la forme normale de Karr donne des équations portant sur peu de variables, ce qui est intéressant, car cela évite le phénomène de résonnance entre ces variables.

Cependant, cela facilite les formes indeterminées infinitaires. Pour remédier à cela, on calcule une forme *positive* de la matrice. Les contraintes de celle-ci portent sur plus de variables, mais comporte essentiellement des termes positifs. Les contraintes engendrées permettront ainsi de déterminer si la valeur d'une variable peut être nulle, et si cette valeur est bornée.

Pour obtenir cette forme *positive*, on procède colonne par colonne : si une colonne admet des termes négatifs et des termes positifs, on annule les lignes négatives, en opérant sur les termes de la matrice, si une colonne ne contient que des termes positifs, on ne fait rien et ainsi de suite.

Exemple 6.2.4 On donne ci dessous $\mathbf{M}_1$ une matrice normale et $\mathbf{M}_2$ sa matrice positive correspondante.

$$\mathbf{M}_1 = \begin{pmatrix} 1 & 0 & 0 & 1 & 2 & 3 \\ 0 & 1 & 0 & -1 & -3 & 5 \\ 0 & 0 & 1 & 0 & 0 & 2 \end{pmatrix} \quad \mathbf{M}_2 = \begin{pmatrix} 1 & 0 & 0 & 1 & 2 & 3 \\ 2 & 1 & 0 & 1 & 1 & 11 \\ 0 & 0 & 1 & 0 & 0 & 2 \end{pmatrix}$$

□

On se sert de ces contraintes linéaires pour raffiner l'information que l'on dispose sur les intervalles.

Exemple 6.2.5 Si on a les contraintes

$$\begin{cases} x = y + z + 1 \\ \begin{cases} x \in [|8; 10|] \\ y \in [|4; 5|] \\ z \in [|4; 10|] \end{cases} \end{cases}$$

on peut déduire la contrainte supplémentaire

$$x \in [|8; 10|] \cap [|4 + 4 + 1; 5 + 10 + 1|] = [|9; 10|]$$

puis le système de contraintes suivant

$$\begin{cases} x = y + z + 1 \\ \begin{cases} x \in [|9; 10|] \\ y \in [|4; 5|] \\ z \in [|6; 10|] \end{cases} \end{cases}$$

□

Réciproquement, on peut enrichir le système d'équation lorsque l'information permet de déduire la valeur d'une variable.

Remarque 6.2.2 *La réduction présentée ci-dessous n'est pas exacte, ce qui lui permet d'être rapide, elle détecte cependant qu'un système d'equation et d'inégalité du type* $x \geqslant 1,\ y \geqslant 1$ *est absurde.*

6.2.5 Domaines abstraits

On se contente simplement de définir le treillis $\mathfrak{V}$, les autres domaines sont les mêmes que pour l'analyse de flux (Cf 6.1).

On fixe n un entier qui sera un paramètre du domaine,

L'ensemble de variables pour chaque étape d'exécution (u, C) sera, d'une part, le nombre d'occurences de chaque étiquette dans le mot u, et d'autre part, le nombre d'occurences syntaxiques de chaque sous-processus dans l'ensemble C. On prend ainsi $\mathrm{V} = \Sigma + \mathfrak{Pro}$

On choisit ainsi $\mathfrak{V} = \mathrm{N_V} \times \mathrm{I_V}$ On définit ensuite les deux primitives dec^{v_i} et inc^{v_i} composantes par composantes :

- $dec^{v_i}(k, i) = (dec^{v_i}_{\mathrm{N_V}}(k), dec^{v_i}_{\mathrm{I_V}}(i))$
- $inc^{v_i}(k, i) = (inc^{v_i}_{\mathrm{N_V}}(k), inc^{v_i}_{\mathrm{I_V}}(i))$

On note ensuite ρ un opérateur de réduction pour ce domaine.

6.2.6 Primitives abstraites

– $Init_{\mathfrak{A}}$

$$Init_{\mathfrak{A}}(c) = (k, i)$$

où

$$k = \bigcup_{c \in C} \mathfrak{A} \begin{cases} \lambda = 0 & \forall \lambda \in \Sigma \\ p = 1 & \text{si } \exists (p, id, E) \in c \\ p = 0 & sinon \end{cases}$$

$$i = \bigcup_{c \in C} \mathfrak{A} \begin{cases} \lambda \in [|0;0|] & \forall \lambda \in \Sigma \\ p = [|1;1|] & \text{si } \exists (p, id, E) \in c \\ p = [|0;0|] & sinon \end{cases}$$

– *nonzero*

La présence ou l'absence d'un processus se lit directement dans le treillis des intervalles :

$$nonzero((k, i), a) = \{p \in \mathfrak{Pro} | [|1; \infty|[\cap_{\mathrm{I}} (i(p)) \neq \emptyset\}$$

– *trans*

La translation est la primitive la plus intéressante de ce treillis, elle se construit en deux étapes : la première étape consiste à vérifier que les sous-processus nécéssaires à la réduction sont présents, la deuxième réalise une translation sur la valeur abstraite calculée par cette réduction.

$$trans(P_{req}, P_{lost}, P_{new}, \lambda, (k, i), a) = [(\bigcirc_{p \in P_{lost}} dec^p) \circ (\bigcirc_{p \in P_{new}} inc^p) \circ inc^{\lambda}](v^{\rho})$$

où

$$v^{\rho} = \rho \left(k, i \cap_{\mathrm{I_V}} \begin{cases} x \mapsto [|0; \infty|[& \text{si } x \notin P_{req} \\ x \mapsto [|1; \infty|[& \text{sinon} \end{cases} \right)$$

Le signe $\bigcirc$ désigne la composition d'une famille de fonctions qui commutent deux à deux.

Remarque 6.2.3 *Un recours à l'opérateur de réduction ρ dans la définition de la primitive nonzero aurait été inutile, vu que le premier argument de la primitive nonzero est toujours le résultat d'un appel à la primitive trans, qui a déjà effectué cette réduction.*

6.2.7 Résultats

Outre le fait de pouvoir détecter des contraintes d'exclusion mutuelle, ce domaine permet de compter efficacement les occurrences syntaxiques des sous-processus. On donne ci-dessous deux exemples d'application.

Exemple 6.2.6 Dans l'exemple des processus alternés figure 6.1, l'analyseur détecte que la communication entre $c?^3[u]d!^4[u]$ et $c!^7[e]$ n'a jamais lieu. □

Exemple 6.2.7 Dans l'exemple du serveur figure 3.5, l'analyseur détecte le sous-processus dont le symbole de tête est étiqueté par $?^3$ ne peut apparaître au plus que trois fois, ce qui montre qu'au plus trois clients peuvent être traités simultanément. □

Exemple 6.2.8 Dans l'exemple d'un serveur erroné donné figure 6.2, où les ports physiques seraient libérés avant que la communication *serveur-client* ne soit effectuée, l'analyseur détecte que le sous-processus dont le symbole de tête est étiqueté par $?^3$ peut apparaître une infinité de fois, on peut de plus déduire de la matrice de Karr que ce nombre est égal à la différence entre le nombre de réplications de la ressource (**Allouer**) et le nombre de communications ($?^3$,$!^2$). □

Figure 6.2 un serveur erroné

$$
\begin{aligned}
\textbf{Allouer} := {} & *\text{make}?^1[][(\nu\, \textit{canal-entrée})(\nu\, \textit{canal-sortie})(\nu\, \textit{info-client}) \\
& (\textit{canal-entrée}!^2[\textit{info-client}] \\
& |\textit{canal-entrée}?^3[\textit{info-reçue}]\textit{canal-sortie}!^4[\textit{info-reçue}] \\
& | \text{ make}!^5[])
\end{aligned}
$$

$$
\textbf{Serveur} := (\nu \text{make})(\textbf{Allouer} \mid \text{make}!^6[] \mid \text{make}!^7[] \mid \text{make}!^8[])
$$

6.2.8 Partitionnement

Le partitionnement permet de séparer au cours de l'analyse différentes étapes d'exécution. Pour cela, il faut trouver un critère qui permette de les séparer.

Un critère simple serait de partitionner les étapes d'exécution en fonction de l'ensemble (et non multi-ensemble) des sous-processus qui y sont présents. Notre analyse permet de faire un tel partitionnement. En effet, l'absence ou la présence d'un sous-processus se traduit par une contrainte sur les intervalles. L'opérateur de réduction permet alors de dire si un système de contraintes portant sur l'absence et la présence d'un sous-processus est satisfiable ou non.

Cependant, un tel partitionnement est couteux, puisqu'il requiert en espace un facteur exponentiel supplémentaire.

6.3 Contraintes temporelles

6.3.1 Présentation

Lorsqu'un processus contient des ressources (i.e. une tâche commençant par un symbole de réplication), il a la possibilité de créer dynamiquement des nouveaux processus. Chacun de ces processus peut alors créer ses propres canaux, puis les exporter vers les autres processus. Pour qu'une analyse soit précise, il faut qu'elle puisse faire la différence entre deux canaux créés par deux sous-processus différents mais issus de la même ressource. Jusqu'à maintenant, seule l'analyse d'Arnaud Venet [Ven98] était capable de distinguer deux instances d'une tâche récursive, mais son analyse ne porte que sur un sous-ensemble du π-calcul. L'analyse que nous proposons, étend celle d'Arnaud Venet à l'ensemble des termes du π-calcul.

Exemple 6.3.1 On donne figure 6.3 la syntaxe d'un processus qui crée un anneau. Ainsi, à chaque réplication de la ressource **Link**, une arête (edge) est créée entre un nouveau sommet (*next*) et le sommet précédent (*last*). L'anneau est initialisé par le sous-processus **Init** qui donne le premier sommet (first) et bouclé par le sous-processus **close** qui lie le dernier sommet au premier.

Figure 6.3 un anneau de communication

Link ::= *make?^{1}[*last*](ν*next*)(edge!^{4}[*last*,*next*]||make!^{5}[*left*]||[*last*=^{6} *next*]d!^{7}[])
Close ::= make?^{2}[*last*](edge!^{8}[*last*,first])
Init ::= make!^{3}[first]
Ring ::= (νmake)(νedge)(νfirst)(**Link** | **Close** | **Init**)

Le filtrage [*last* = *next*] n'est jamais validé car même si les deux canaux (*last*) et (*next*) ont tous deux été créer par le même lieur (ν*next*) ils ne peuvent pas avoir été créés par la même réplication du processus **Link**. □

Le problème des contraintes temporelles n'est traité ni par l'analyse de Bodei, Degano, Nielson et Nielson, ni par l'analyse de la partie précédente (Cf 6.2).

Pour résoudre ce problème, il faut pouvoir abstraire précisément l'historique des processus, et plus particulièrement la différence entre l'identifiant des canaux qui indique quelle tâche les a créés et l'identifiant de la tâche dans laquelle ils sont présents. Cette abstraction doit être assez robuste pour conserver l'information après les opérations de jointure lors des étapes de communication et de réplication.

Pour simplifier, on propose d'abstraire les arbres par les mots formés par les étiquettes de leurs peignes droits. En effet, les noeuds sont créés par des réplications. Ces réplications agissent sur un couple d'arbres et un arbre. L'arbre seul est greffé en sous-arbre gauche de la première composante du couple d'arbres, alors qu'il n'a à priori aucun rapport avec celui-ci. Ainsi, on ne peut pas à la fois obtenir une information relationnelle sur le couple ainsi formé et garder une information globale sur l'arbre.

6.3.2 Domaine de graphes

Pour abstraire un ensemble de mots sur un alphabet Σ, on utilise un domaine de graphes orientés sur Σ, muni d'un ensemble d'états initiaux et d'un ensemble d'états finals.

Un graphe orienté représente alors le langage des mots qui forment un chemin dans ce graphe. Cependant, les graphes ne peuvent reconnaître le mot vide. On associe donc à chaque graphe un booléen, qui spécifie si un langage reconnait ou non, le mot vide.

Définition 6.6 *Soit $G = (\Sigma, \rightarrow, I, F, b)$ un graphe orienté muni de I et F deux parties de Σ et d'un booléen $b \in \{0; 1\}$.*

On définit le langage reconnu par G ci-dessous :

$$\mathcal{L}(G) = \left\{ u \in \Sigma^* \middle| u = u_0 ... u_n \text{ et } \begin{cases} u_0 \in I \\ u_n \in F \\ \forall i \in [|0; n|[,\ u_i \rightarrow u_{i+1} \end{cases} \right\} \cup \{\varepsilon | b = 1\}$$

Pour Σ fixé, on note L_G l'ensemble des langages reconnus par un tel graphe.

$\mathrm{L_G}$ est un modèle simpliste pour représenter des langages. Il possède cependant beaucoup de propriétés qui facilitent son utilisation.

Proposition 6.1 Si Σ est fini, $\mathrm{L_G}$ est un treillis complet fini, de hauteur $1 + (Card(\Sigma))^2$.

On a de plus $\top_{\mathrm{L_G}} = (G, G \times G, G, G, 1)$

Définition 6.7 *Soit $(G, \rightarrow, I, F, b)$ un graphe,*

On dira que G est élagué si et seulement si l'assertion suivante est réalisée :

$$\rightarrow \subseteq \{g \in G | \exists i \in I, \exists f \in F,\ \textit{tels que } i \rightarrow^* g \rightarrow^* F\}^2.$$

Proposition 6.2 Tout langage reconnaissable par un graphe est représenté par un unique graphe élagué.

De plus, l'implantation de toutes les primitives nécessaire à l'analyseur sont faciles à implanter.

- **le mot vide**
  Le graphe $\varepsilon_{\mathrm{L_G}} = (G, \emptyset, \emptyset, \emptyset, 1)$ ne reconnait que le mot vide.

- **l'intersection**
  L'intersection, $\cap_{\mathrm{L_G}} : \mathrm{L_G} \times \mathrm{L_G} \mapsto \mathrm{L_G}$ est l'opérateur qui associe à deux langages leur intersection. Cette opération est obtenue en faisant l'intersection des deux relations de transition, des deux ensembles d'états inititiaux, des deux ensembles d'états finals, et la conjonctions des deux booléens.

- **l'union**
  L'union affine, $\cup_{\mathrm{L_G}} : \mathrm{L_G} \times \mathrm{L_G} \mapsto \mathrm{L_G}$ est l'opérateur qui associe à deux langages le plus petit langage reconnaissable par un graphe contenant l'union des deux langages donnés en argument. Là encore, cette opération est obtenue en faisant l'union des deux relations de transition, des deux ensembles d'états inititiaux, des deux ensembles d'états finals, et la disjonction des deux booléens.

- **le test d'inclusion**
  On note le test d'inclusion $\subseteq_{\mathrm{L_G}}$. Ce test ce fait simplement en comparant les deux relations de transition, les deux ensembles d'etats inititiaux, les deux ensembles d'états finals, ainsi que les deux booléens.

- **l'ajout d'une lettre**

  L'ajout de la lettre λ, $concat^{\lambda}_{\mathrm{L_G}} : \mathrm{L_G} \to \mathrm{L_G}$, est l'opérateur qui retourne le plus petit langage reconnaissable par un graphe contenant tous les mots du langage donné en argument suivis de la lettre λ.

  $concat^{\lambda}_{\mathrm{L_G}}((G, \to, I, F, b) = (G, \to \cup\{(f, \lambda) | f \in F\}, I, \{\lambda\}, 0) \cup_{\mathrm{L_G}} \mathit{mot\text{-}vide}$

  où

  $$\mathit{mot\text{-}vide} = \begin{cases} (G, \emptyset, \{\lambda\}, \{\lambda\}, 0) & \text{si } b = 1 \\ (G, \emptyset, \emptyset, \emptyset, 0) & \text{sinon} \end{cases}$$

Ce noyaux de primitives est suffisant pour définir notre domaine abstrait.
On définit de plus $\mathrm{L^2_G}$ le domaine abstrait des couples de graphes, défini composante par composante.

6.3.3 Domaine relationnel

Notre objectif est d'exprimer la différence entre les identifiants des tâches et les identifiants des processus. On ne peut l'atteindre sans générer de l'information

relationnelle. Il faut en fait pouvoir générer une information qui caractérisera la distance entre deux identifiants.

Pour cela on utilise le domaine des relations affines (Cf 6.2.2). On abstrait en effet un couple de mots, par l'ensemble des contraintes linéaires qui lient les occurrences de chaque lettre de chaque mot.

Soit $\underline{\Sigma}$ une copie de Σ, on utilise les deux treillis N_{Σ} et $\mathrm{N}_{\Sigma+\underline{\Sigma}}$. Les éléments de Σ désignent les lettres des identifiants des tâches actives, alors que ceux de $\underline{\Sigma}$ désignent les identifiants des tâches qui ont créé les canaux.

On relie N_{Σ}, et $\mathrm{N}_{\Sigma+\underline{\Sigma}}$ par des primitives logiques :

- $proj^1$

  $proj^1$(k), pour $k \in \mathrm{N}_{\Sigma+\underline{\Sigma}}$ est le système d'équations formé par toutes les contraintes induites par le système k sur l'ensemble de variables Σ.

  Ce système s'obtient facilement à partir de la forme normale de k, après avoir permuté ses colonnes deux à deux, de sorte que les colonnes relatives à l'ensemble de variables $\underline{\Sigma}$ soient placées devant celles relatives à l'ensemble de variables Σ.

- $proj^2$

  $proj^2$(k), pour $k \in \mathrm{N}_{\Sigma+\underline{\Sigma}}$ est le système d'équations formées par toutes les contraintes induites par le système k sur l'ensemble de variables $\underline{\Sigma}$.

  Ce système s'obtient facilement à partir de la forme normale de k.

- inj^1

  Cette primitive transforme un système d'équations portant sur l'ensemble de variable Σ en un système d'équations identique portant sur l'ensemble de variables $\Sigma + \underline{\Sigma}$, sans en changer les contraintes, le système ainsi obtenu ne contiendra pas de contraintes sur les variables de l'ensemble $\underline{\Sigma}$.

- inj^2

  Cette primitive transforme un système d'équations portant sur l'ensemble de variables Σ en un système d'équations identique portant sur l'ensemble de variables $\Sigma + \underline{\Sigma}$, en remplacant chaque occurrence de variable $\lambda \in \Sigma$, par son homologue $\underline{\lambda} \in \underline{\Sigma}$. Le système ainsi obtenu ne contiendra pas de contraintes sur les variables de l'ensemble Σ.

- $\overset{\mathrm{N}_{\Sigma+\underline{\Sigma}}}{\underset{=\,\rightarrow}{\bowtie}}$

  $\overset{\mathrm{N}_{\Sigma+\underline{\Sigma}}}{\underset{=\,\rightarrow}{\bowtie}}(k, k')$ est le système d'équations obtenu en réalisant la jointure selon l'ensemble de variables Σ, des deux systèmes d'équations k et k'.

  Pour l'obtenir, on renomme les variables $\underline{\lambda}$ du système d'équations k (resp. k') par $\underline{\lambda^1}$ (resp. $\underline{\lambda^2}$). On calcule ensuite l'ensemble de toutes les contraintes

portant exclusivement sur des variables de la forme $\underline{\lambda^1}$ et $\underline{\lambda^2}$. On renomme ensuite dans le système obtenu, toutes les variables de la forme $\underline{\lambda^1}$ (resp. $\underline{\lambda^2}$) en λ (resp. $\underline{\lambda}$).

- $\overset{\mathrm{N}_{\Sigma+\underline{\Sigma}}}{\underset{\leftarrow =}{\bowtie}}$

  $\overset{\mathrm{N}_{\Sigma+\underline{\Sigma}}}{\underset{\leftarrow =}{\bowtie}}(k, k')$ est le système d'équations obtenu en réalisant la jointure selon l'ensemble de variables $\underline{\Sigma}$, des deux systèmes d'équations k et k'.

  Pour l'obtenir, on renomme les variables λ du système d'équations k (resp. k') par λ^1 (resp. λ^2). On calcule ensuite l'ensemble de toutes les contraintes portant exclusivement sur des variables de la forme λ^1 et λ^2. On renomme ensuite dans le système obtenu, toutes les variables de la forme λ^1 (resp. λ^2) en $\underline{\lambda}$ (resp. λ).

6.3.4 Réduction

On peut définir une réduction entre le domaine des relations affines et le domaine des graphes. En effet, il est possible de déduire des contraintes linéaires en observant un graphe.

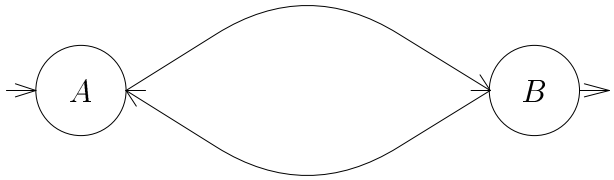
Proposition 6.3 Soit $G = (\Sigma, \rightarrow, I, F, b)$ un graphe,
Soit V un ensemble de variables distinctes comprenant

- une variable $x^i_\lambda,\ \forall \lambda \in \Sigma$
- une variable $x^f_\lambda,\ \forall \lambda \in \Sigma$
- une variable $x_{(\lambda,\lambda')},\ \forall \lambda, \lambda' \in \Sigma^2$ tel que $\lambda \rightarrow \lambda'$
- une variable $x_\lambda,\ \forall \lambda \in \Sigma$

On associe à chaque chemin valide $(u_0 ... u_k)$ dans G une valuation pour V :

- $x_\lambda = Card\{i \in [|0; k|] | u_i = \lambda\}$
- $x^i_\lambda = \begin{cases} 0 & \text{si } u_0 \neq \lambda \\ 1 & \text{sinon} \end{cases}$
- $x^k_\lambda = \begin{cases} 0 & \text{si } u_k \neq \lambda \\ 1 & \text{sinon} \end{cases}$
- $x_{(\lambda,\lambda')} = Card\{i \in [|0; k|[| u_i = \lambda \text{ et } u_{i+1} = \lambda'\}$

Figure 6.4 le langage A(BA)*



Les mots non-vides reconnus par G vérifient le système de contraintes suivant :

$$\begin{cases} \left(\sum_{\lambda\in\Sigma} x^i_\lambda\right) = 1 \\ \left(\sum_{\lambda\in\Sigma} x^f_\lambda\right) = 1 \\ x_\lambda = x^i_\lambda + \sum_{\lambda'\in\Sigma} x_{(\lambda',\lambda)} = x^f_\lambda + \sum_{\lambda'\in\Sigma} x_{(\lambda,\lambda')} \end{cases}$$

Preuves 2 Par analogie avec le loi de Kirchoff.

Remarque 6.3.1 *Lorsque G reconnait le mot vide, on calcule l'union affine entre l'espace solution du système d'équations induit par la loi de Kirchoff, et le point associé au mot vide ($\{x_\lambda = 0\ \forall\lambda \in G$)*

Ainsi, on peut associer à chaque graphe un ensemble de contraintes, puis le projeter pour ne garder que des contraintes portant sur les variables de l'ensemble $\{x_\lambda\}$. Ceci s'obtient en un coût quadratique en fonction du nombre de variables, en permutant les colonnes de la matrice de Karr, puis en normalisant celle-ci.

Exemple 6.3.2 On considère de graphe donné figure 6.4. Ce graphe est par exemple obtenu au cours de l'analyse des processus alternés figure 6.1.

La loi de Kirchoff appliquée à ce graphe induit le système de contraintes

suivantes :

$$\begin{cases} x_A^i = 1 \\ x_A^f = 0 \\ x_B^i = 0 \\ x_B^f = 1 \\ x_A = x_A^i + x_{(B,A)} \\ x_A = x_A^f + x_{(A,B)} \\ x_B = x_B^i + x_{(A,B)} \\ x_B = x_B^f + x_{(B,A)} \end{cases}$$

Ce système d'équations est représenté par la matrice **M**, en prenant comme ordre de variables : (x_i^A, x_f^A, x_i^B, x_f^B, $x_{(A,B)}$, $x_{(B,A)}$, x_A, x_B).

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & -1 & 0 \end{pmatrix}$$

On note **M'**, la forme normale de **M**.

$$\mathbf{M}' = \left(\begin{array}{cccccc|ccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & -1 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \end{array}\right)$$

On a ainsi capturé la contrainte $\left\{ x_A = x_B \right.$ □

6.3.5 Domaines abstraits

On se contente simplement de définir les treillis $Id_1^\sharp$ et $Id_2^\sharp$, les autres domaines sont les mêmes que pour l'analyse des exclusions mutuelles (Cf 6.2.5).

On choisit ainsi $Id_1^\sharp = (\mathrm{L}_\Sigma \times \mathrm{N}_\Sigma)$ et $Id_1^\sharp = (\mathrm{L}_\Sigma^2 \times \mathrm{N}_{\Sigma+\underline{\Sigma}})$

On note ensuite ρ un opérateur de réduction pour ce domaine.

6.3.6 Primitives abstraites

- **Primitives logiques**
  - Π_1

$$\Pi_1((g_1, g_2), k) = \big(g_1, proj^1(k)\big)$$

  - Π_2

$$\Pi_1((g_1, g_2), k) = \big(g_2, proj^2(k)\big)$$

  - Inj_1

$$Inj_1(g_1, k) = \big((g_1, \top_\Sigma), inj^1(k)\big)$$

  - Inj_2

$$Inj_2(g_2, k) = \big((\top_\Sigma, g_2), inj^2(k)\big)$$

  - $\underset{= \rightarrow}{\bowtie}$

$$\underset{= \rightarrow}{\bowtie}\,(((g_1, g_2), k), ((g_1', g_2'), k')) = \left((g_2, g_2'), \overset{\mathrm{N}_{\Sigma+\Sigma}}{\underset{= \rightarrow}{\bowtie}}\,(k, k')\right)$$

  - $\underset{\leftarrow =}{\bowtie}$

$$\underset{\leftarrow =}{\bowtie}\,(((g_1, g_2), k), ((g_1', g_2'), k')) = \left((g_2', g_2), \overset{\mathrm{N}_{\Sigma+\Sigma}}{\underset{\leftarrow =}{\bowtie}}\,(k, k')\right)$$

  - *existe-différent*

    Cette primitive n'a pas été étudiée par manque de temps. On propose ici une solution grossière :

$$\textit{existe-différent}(A, A') = (\Pi_1 A) \cap (\Pi_2 A')$$

Remarque 6.3.2 *Pour concevoir une primitive plus fine, il faut analyser en détails la structure des graphes. Lorsque les graphes ne comportent pas de cycles imbriqués, et les mots correspondants aux cycles sont représentés dans le domaine des relations affines par des vecteurs libres. On peut conclure sur l'unicité des mots reconnus par un tel graphe, sachant le nombre d'occurrence de chaque lettre. Cependant, notre domaine de graphe est peu propice à la formation de tels graphes.*

– **Formation des identifiants**

– ε_1

$$\varepsilon_1 = \left(\varepsilon_{\mathrm{L}_\Sigma}, \left\{ \lambda = 0,\ \forall \lambda \in \Sigma \quad \right.\right)$$

– ε_2

$$\varepsilon_2 = \left((\varepsilon_{\mathrm{L}_\Sigma}, \varepsilon_{\mathrm{L}_\Sigma}), \begin{cases} \lambda = 0,\ \forall \lambda \in \Sigma \\ \underline{\lambda} = 0,\ \forall \underline{\lambda} \in \underline{\Sigma} \end{cases} \right)$$

– $push_1^{(i,j)}$

$$push_1^{(i,j)}((g,k),(g',k')) = \left(concat_{\mathrm{L}_\Sigma}^{(i,j)}(g'), inc_{\mathrm{N}_\Sigma}^{(i,j)}(k') \right)$$

– $push_2^{(i,j)}$

$$push_2^{(i,j)}((g,k),((g'_1,g'_2),k')) = \left((concat_{\mathrm{L}_\Sigma}^{(i,j)}(g'_1), g'_2), inc_{\mathrm{N}_{\Sigma+\underline{\Sigma}}}^{(i,j)}(k') \right)$$

– $dpush$

$$dpush(g,k) = \left((g,g), k \cap_{\mathrm{N}_{\Sigma+\underline{\Sigma}}} \left\{ x = \underline{x} \quad \right.\right)$$

6.3.7 Résultats

Ce domaine abstrait permet bien de distinguer deux instances d'une tâche récursive. Il permet en plus de calculer des contraintes de sécurité, en indiquant sous quelles conditions sur leurs identifiants id et id', un canal syntaxique d'une tâche ayant pour identifiant id a pu être affecté à un autre canal créé par un opérateur (νx) présent sur une tâche dont l'identifiant était id'.

Exemple 6.3.3 Pour l'exemple de l'anneau (Cf figure 6.3), l'analyseur donne la contrainte suivante :

$$(\text{edge!}^4[last,next], last, next) \mapsto ((g_1, g_2), k)$$

où

$$\begin{cases} g_1 \text{ représente la langage } (1,3).(1,5)^+ \\ g_2 \text{ représente la langage } (\underline{1,3}).(\underline{1,5})^* \\ k \text{ contient la contrainte } (1,5) = (\underline{1,5}) + 1 \end{cases}$$

Cela montre que le canal syntaxique *last* est affecté du nom du canal *next* qui avait été lors de la réplication précédente de la tâche **Link**.

L'analyseur peut donc en déduire que le filtrage $[last = next]$ n'est jamais valide ce qui se traduit par le fait que le processus $d![]$ n'est présent dans aucune étape d'exécution. En effet, l'analyse montre que le nombre d'instance de ce processus est toujours égal à zéro. □

Exemple 6.3.4 Dans l'exemple du serveur (Cf 3.5), l'analyseur donne la contrainte suivante :

$$(\textit{canal-sortie}!^4[\textit{info-reçue}], \textit{info-reçue}, \textit{info-client}) \mapsto ((g_1, g_2), k)$$

où

$$\begin{cases} g_1 \text{ représente la langage } ((1,6)|(1,7)|(1,8)).(1,5)^* \\ g_2 \text{ représente la langage } ((1,6)|(1,7)|(1,8)).(1,5)^* \\ k \text{ contient les contraintes } \begin{cases} \underline{(1,5)} = (1,5) \\ \underline{(1,6)} = (1,6) \\ \underline{(1,7)} = (1,7) \\ \underline{(1,8)} = (1,8) \end{cases} \end{cases}$$

Or ces contraintes sont suffisantes pour démontrer que le canal syntaxique *info-reçue* est lié au canal *info-client* qui avait été créé par la tâche active, ce qui assure la sécurité de ce protocole. □

6.3.8 Limites

L'analyse présentée précédemment est au moins aussi précise que celle d'Arnaud Venet. Elle présente néanmoins certains défauts.

Ainsi, l'analyse temporelle ne permet de comparer que l'identifiant des canaux à l'identifiant des tâches. Cela est insuffisant pour générer l'information relationelle entre deux canaux qui seraient exportés vers un autre processus, par l'intermédiaire d'une même communication.

Exemple 6.3.5 On considère le processus figure 6.5, composé du processus de l'anneau mis en concurrence avec un processus de filtrage dont le but est de déterminer à quelles conditions une arête peut relier deux sommets identiques.

L'analyseur est incaple ici de détecter que le filtrage $[x{=}y]$ ne peut être validé que si $x = y = (\text{first}, \varepsilon)$. □

On peut cependant remédier à ce problème, en utilisant un domaine abstrait relationnel plus général. Ce domaine permettrait de relier tous les identifiants des variables libles d'un processus entre eux. Un tel domaine serait donc un domaine fonctionnel qui associerait à tout couple (p, e) où p est un sous-processus et e une

Figure 6.5 un anneau de communication et un test

Link ::= *make?^{1}[*last*](*νnext*)(edge!^{4}[*last*,*next*]||make!^{5}[*left*]|[*last*=^{6} *next*]d!^{7}[])
Close ::= make?^{2}[*last*](edge!^{8}[*last*,first])
Init ::= make!^{3}[first]
Ring ::= (*ν*make)(**Link** | **Close** | **Init**)
Test ::= mon?^{9}[*x*,*y*][*x*=^{10} *y*][*x*≠^{11}first] test!^{12}[]
P ::= (*ν*test)(*ν*edge)(**Ring** | **Test**)

fonction de $\mathfrak{FN}(p)$ dans $\mathfrak{BN}(P_0)$ une abstraction du n-uplet formé par les identifiants des processus actifs, et des identifiants des tâches des canaux pointés. Pour qu'un tel domaine soit concevable, il faut utiliser des domaines moins précis pour décrire les ensembles d'identifants. On peut par exemple abstraire un ensemble d'identifiants en l'ensemble des tailles des fils droits de ses identifiants. Un tel domaine permettrait de prouver que le sous-processus $test!^{11}[]$ est inaccessible. Une telle modification requiert l'ajout d'un nouveau domaine abstrait pour la sémantique abstraite.

6.4 Problèmes d'échappement

6.4.1 Présentation

Un sous-processus déclare ses propres canaux. Ces derniers, tant qu'ils ne sont pas exportés par l'intermédiaire d'une communication, ne peuvent être connus par les autres processus qui ne peuvent alors pas communiquer sur ces canaux. Dès lors, la localité de certains canaux induit de nouvelles contraintes et peut empêcher une transition de s'effectuer.

Exemple 6.4.1 Dans le processus figure 6.6, les variables a et b ne peuvent s'échapper du sous-processus $\mathbf{P}_1$, les différentes réplications du sous-processus $\mathbf{P}_1$ ne peuvent donc pas interagir entre elles. L'analyse du sous-processus $\mathbf{P}_1$ seul, détermine que le sous-processus $c!^5[]$ est inaccessible, alors que l'analyse du processus $\mathbf{P}$ ne calcule pas cette contrainte. En effet, l'analyse du processus dans sa globalité mélange l'information relative à chaque réplication du sous-processus $\mathbf{P}_1$, ainsi, elle ne peut détecter la contrainte selon laquelle le sous-processus $b?^4[]c!^5[]$ et le sous-processus $b!^7[]$ ne peuvent être présent simultanément, avec le même identifiant pour le canal b. L'analyse du sous-processus $\mathbf{P}_1$ seul, permet de générer la bonne contrainte d'exclusion mutuelle. □

Remarque 6.4.1 *L'exemple 6.4.1 est simpliste, car le sous-processus concerné n'avait aucune interaction avec les autres sous-processus.*

On montre dans cette partie comment analyser le comportement d'un sous-processus en ayant une information réduite sur le reste de son processus principal.

Figure 6.6 un processus sans échapement

$\mathbf{A} ::= a?^3[]b?^4[]\mathrm{c}!^5[]$
$\mathbf{B} ::= a?^6[]b!^7[]$
$\mathbf{C} ::= a!^8[]$
$\mathbf{P}_1 ::= (\nu a)(\nu b)(\mathbf{A} \mid \mathbf{B} \mid \mathbf{C})$
$\mathbf{P} ::= ((*\mathrm{make}?^0[](\mathbf{P}_1 \mid \mathrm{make}!^1[])) \mid \mathrm{make}!^2[])$

Pour ce faire, on introduit une notion d'échappement, qui permet de majorer l'ensemble des variables connues par des processus autres que le sous-processus que l'on analyse actuellement, afin de majorer l'action des autres processus sur ce sous-processus.

6.4.2 Sémantique ouverte

On modifie quelque peu la sémantique du π-calcul.

Pour des raisons d'ordre technique, on se place directement dans le cadre de la sémantique gloutonne (Cf 3.4). On s'affranchit ainsi des différents problèmes liés aux α-conversions, et aux extrusions.

On associe à chaque étape d'exécution un ensemble de variables $\mathfrak{E}$ qui représente l'ensemble des variables connues par le monde extérieur. On majore ensuite l'action du monde extérieur sur notre sous-processus, en considérant toutes les opérations d'espionnage et de brouillage portant sur les canaux de $\mathfrak{E}$. On considère ainsi que tout sous-processus de la forme $x![x_1, ..., x_n]$ ou de la forme $y?[y_1, ..., y_n]$ (où $x, y, x_1, y_1, ..., x_n, y_n \in \mathfrak{E}$) est succeptible d'être mis en concurrence avec notre processus. Les actions d'espionnage peuvent ensuite enrichir l'ensemble des variables connues par le monde extérieur, alors que les actions de brouillages permettent de perturber de manière chaotique l'exécution du sous-processus.

Par ailleurs, pour assurer la cohérence de la sémantique et conserver l'indépendance des variables (Cf théorème 3.1), on compte le nombre d'opérations de brouillage effectuées sur des ressources, on utilise alors ce compteur pour former les identifiants.

Définitions On définit *Id* comme étant l'ensemble des arbres binaires dont les noeuds sont des couples (i,j) où $i \in Att(P)$ et $j \in Eme(P)$, et les feuilles sont étiquetées par ε ou un entier naturel.

Les étapes d'exécution sont ici formées de triplets $(C, \mathfrak{E}, n)$ où C est un ensemble de tâches, $\mathfrak{E} \in (Channel \times Id)$ est un ensemble de variables et n est un compteur.

Les autres définitions de la sémantique gloutonne restent valables (Cf 3.4.2).

Traduction initiale On considère que toutes les variables libres du processus P_0 sont connues du monde extérieur.

On définit alors l'ensemble des états initiaux de notre sémantique :

On note $(A_0, T) = Agent(\emptyset, (P_0, \emptyset))$.

$C_0(P_0) = \{(\{(p, \varepsilon, E_p) | p \in \sigma(T)\}, \{(x, \varepsilon) | x \in \mathfrak{FN}(P_0)\}, 0) \,|\sigma \text{ valuation totale de } A_0\}$

Relation de transition Le système de transitions comporte trois nouvelles règles de transitions, une transition *espionnage* qui correspond à l'écoute d'un message par un processus extérieur et deux transitions *brouillage* qui correspondent à l'envoi d'un message formé de canaux échappés sur un canal échappé.

On définit maintenant la relation de réduction de la sémantique ouverte.

- **Anciennes transitions**

  Les anciennes transitions sont sûres et ne laissent pas échapper de variables :

$$\frac{C \xrightarrow{\lambda}_4 C'}{(C, \mathfrak{E}, n) \overset{\lambda}{\hookrightarrow} (C', \mathfrak{E}, n)}$$

- **Espionnage**

  Cette transition permet d'écouter des canaux privés, sur un canal que l'on connait :

  Soit $(C, \mathfrak{E}, n)$ une étape d'exécution,

  supposons que $\lambda \in C$, où $\lambda = (x^{!^i}[x_1, ..., x_n]P, id, E)$

  avec $E(x) \in \mathfrak{E}$

  on note $(A, T) = Agent(\emptyset, (P, \emptyset))$

  soit alors σ une valuation totale sur A

  on note $f : Ag \mapsto \left(Ag, id, \begin{cases} z \mapsto E(z) & \text{si } z \in \mathfrak{FN}(Ag) \cap \mathfrak{FN}([x \diamond y]P) \\ z \mapsto (z, a) & \text{si } z \in \mathfrak{FN}(Ag) \cap \mathfrak{BN}([x \diamond y]P) \end{cases} \right)$

  on a alors $(C, \mathfrak{E}, n) \overset{spy^i}{\hookrightarrow} (C', \mathfrak{E}', n)$

  où $C' = (C \setminus \{\lambda\}) \cup (f(\sigma T))$.

  et $\mathfrak{E}' = \mathfrak{E} \cup \{E(x_k) | k \in [|1; n|]\}$

– **Communication brouillée**

Soit $(C, \mathfrak{E}, n)$ une étape d'exécution,

Supposons que $\lambda \in C$,

où λ s'écrit $(y?^i[y_1, ..., y_n]P, id_?, E_?)$

tels que $E_?(y) \in \mathfrak{E}$

on note $(A_?, T_?) = Agent(\emptyset, (P, \emptyset))$

soit alors $\sigma_?$ une valuation totale sur $A_?$

on choisit x_1,...,$x_n \in \mathfrak{E}$,

on note

$$f_? : Ag \mapsto \left(Ag, id_?, \begin{cases} z \mapsto E_?(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(y?^i[y_1, ..., y_n]P) \\ y_k \mapsto E_!(x_k) & \text{si } y_k \in \mathfrak{FR}(Ag) \\ z \mapsto (z, id_?) & \text{si } \begin{cases} z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(y?^i[y_1, ..., y_n]P) \\ z \notin \{y_k | k \in [|1; n|]\} \end{cases} \end{cases} \right)$$

on a alors $(C, \mathfrak{E}, n) \overset{(?^i, @)}{\hookrightarrow} (C', \mathfrak{E}, n)$,

où $C' = (C \setminus \{\lambda\}) \cup (f_?(\sigma_? T_?))$

– **Réplication brouillée**

Lors d'une réplication brouillée, on ne connait pas le processus qui envoie un *message*. Il faut pourtant lui assigner un identifiant, pour pouvoir former celui de la tâche dupliquée, tout en préservant la cohérence de la sémantique qui est essentiellement assurée par le lemme 3.1. On utilise alors le compteur, en l'incrémentant par la suite, pour ne pas provoquer de conflits de variables.

Soit $(C, \mathfrak{E}, n)$ une étape d'exécution,

Supposons que $\lambda \in C$,

où λ s'écrit $(*y?^i[y_1, ..., y_n]P, id_?, E_?)$

tels que $E_?(y) \in \mathfrak{E}$

on note $(A_?, T_?) = Agent(\emptyset, (P, \emptyset))$

soit alors $\sigma_?$ une valuation totale sur $A_?$

on choisit x_1,...,$x_n \in \mathfrak{E}$,

on définit $id_* = \mathbf{Noeud}((i, j), id_?, n)$

on note

$$f_? : Ag \mapsto \left(Ag, id_*, \begin{cases} z \mapsto E_?(z) & \text{si } z \in \mathfrak{FR}(Ag) \cap \mathfrak{FR}(y?^i[y_1, ..., y_n]P) \\ y_k \mapsto E_!(x_k) & \text{si } y_k \in \mathfrak{FR}(Ag) \\ z \mapsto (z, id_*) & \text{si } \begin{cases} z \in \mathfrak{FR}(Ag) \cap \mathfrak{BR}(y?^i[y_1, ..., y_n]P) \\ z \notin \{y_k | k \in [|1; n|]\} \end{cases} \end{cases} \right)$$

on a alors $(C, \mathfrak{E}, n) \overset{(?^i, @)}{\hookrightarrow} (C', \mathfrak{E}, n+1)$,

où $C' = C \cup (f_?(\sigma_? T_?))$

Cohérence Le lemme 3.1 reste valide, on peut adapter la preuve simplement.

En effet, la notion de père reste valable, excepté pour les instances syntaxiques de la forme $(x?^i[x_1, ..., x_n], \mathbf{Noeud}((?^i, @), id_?, n))$, cependant, l'apparition d'une telle instance syntaxique est accompagnée d'une incrémentation du compeur, ce qui assure que le lemme est aussi vérifié par les instances syntaxiques de cette forme.

6.4.3 Sémantique abstraite

La sémantique abstraite associée à la sémantique ouverte n'a pas été rédigée, cependant, elle ne poserait pas de difficulté particulière.

On donne ici les lignes directrices pour concevoir une telle sémantique :

- Les compteurs dans les identifiants seront tous abstraits en la même constante. Cette constante représentera une boîte, que l'on pourra remplir par la suite en étudiant le reste du processus.
- On utilise le domaine $Id_1^\sharp$ pour former un domaine fonctionnel, noté $\mathfrak{E}^\sharp$, : $(\mathfrak{E}^\sharp : (\mathfrak{BN}(P_0) \cup \mathfrak{FN}(P_0)) \rightarrow Id_1^\sharp)$. Ce domaine associe à chaque canal x une majoration de l'ensemble des identifiants id, tels que la variable (x, id) puisse échapper au cours d'une éxécution.

6.4.4 Résultats

L'analyse ouverte, permet de détecter que certaines transitions sont impossibles au sein d'un sous-processus.

Ainsi, l'ensemble des transitions d'*espionnage* et de *brouillage* majorent toujours l'ensemble des transitions qui peuvent résulter de l'interaction du sous-processus analysé avec un sous-processus extérieur. Aussi, lorsqu'une transition est impossible avec la sémantique ouverte, elle est aussi impossible avec la sémantique gloutonne, on peut ainsi récupérer l'information calculée, pour détecter les transitions impossibles et les interdire lors d'un calcul ultérieur.

La sémantique ouverte n'a pas été implantée. Un calcul à la main donne les résultats attendus.

Exemple 6.4.2 Pour le processus donné figure 6.6, l'analyse du sous-processus $\mathbf{P}_1$, montre, en utilisant les contraintes d'exclusion mutuelle, que la transition $(?^4, !^7)$ n'a jamais lieu. On peut donc en déduire quel que soit le contexte dans lequel $\mathbf{P}_1$ est plongé, le sous-processus $c!^5[]$ n'est jamais activé. □

Par ailleurs, la sémantique ouverte peut être utilisée pour exprimer des contraintes de sécurité en assurant que des variables sont privées au sein d'un processus.

7 Implémentation

L'itérateur abstrait (Cf 5) et les trois domaines présentés précédemment (Cf 6.1,6.2,6.3) ont été implantés en CAML light.

7.1 Modules

Un mois de travail a été consacré à l'implantation de l'analyseur, pour un total d'environ *9 000 lignes*.

Les différents modules de π-*sa* se répartissent de la manière suivante :

Pré-calcul	3000 lignes
Itérateur abstrait	2000 lignes
Matrices creuses	2000 lignes
Analyse de flux	300 lignes
Exclusion mutuelle	700 lignes
Contraintes temporelles	1000 lignes

7.1.1 Pré-calcul

Le module pré-calcul fournit une fonction qui lit un fichier texte dans lequel est écrit un terme du π-calcul, et rend un terme sous forme non-standard, qui est en fait un jeu de fonctions permettant d'agir efficacement sur ce terme. Ce jeu contient des fonctions de hachage pour numéroter les canaux et les sous-processus, des fonctions $\mathfrak{FN}$, $\mathfrak{BN}$, *Agent* et σ qui indiquent les canaux libres, les canaux liés, les agents et les valuations relatifs à un processus. Ce module se charge aussi de l'étiquetage des processus, pour permettre par la suite de rendre l'information calculée intelligible. L'étiquetage est soit inséré par l'utilisateur dans la syntaxe des termes, soit inféré automatiquement.

7.1.2 Itérateur abstrait

Le module itérateur abstrait définit une fonction qui prend en argument un processus sous forme non-standard et un domaine abstrait muni de ses primitives, ainsi qu'une fonction de sortie qui indique ce que doit rendre l'analyse. Cet itérateur est calqué sur les règles de transitions abstraites (Cf 5).

7.1.3 Graphes et matrices

Ces deux modules donnent la représentation des graphes et des matrices, ainsi que les primitives usuelles qui agissent sur eux.

7.1.4 Domaines abstraits

Ces modules contiennent les primitives abstraites qui agissent sur les domaines abstraits, ils dépendent généralement des modules utilisés pour représenter les graphes et les matrices.

7.2 Application

La fonction *main* prend en argument le nom d'un fichier ASCII contenant un terme partiellement étiqueté. Dans ce fichier, les caractères ♯ et % représentent respectivement les caractères ν et $\neq$, alors que les symboles les chaînes $(? : eti)$, $(! : eti)$ représentent les opérateurs étiquetés $?^{eti}$ $!^{eti}$.

La fonction *main* réalise une première itération, avec le domaine de l'analyse de flux, ceci permet de majorer l'ensemble des processus accessibles et de l'ensemble des transitions possibles. Cette information sera utilisée pour réduire la taille des matrices pour la deuxième itération.

La deuxième itération est effectuée avec le domaine des exclusion mutuelles en fixant le paramètre à 0. Ceci réduit encore l'ensemble des variables de l'analyse.

La troisième itération est la plus complexe, elle consiste d'une part à l'analyse des contraintes temporelles et d'autre part à l'analyse des exclusions mutuelles en fixant le paramètre au plus grand entier obtenu comme borne supérieure finie d'intervalle calculé au cours de l'itération précédente.

7.3 Optimisation

Afin de réduire les temps d'exécution et l'occupation mémoire, un effort d'implantation a été effectué pour représenter les matrices.

Cette représentation permet d'effectuer des opération sur les lignes des matrices et des opérations de jointures à un coût réduit.

7.3.1 Des matrices par blocs

Les matrices sont représentées par blocs de colonnes. Une matrice est ainsi une liste de blocs. Les blocs sont découpés de manière à réduire les opérations de jointures et de projections. Ainsi par exemple pour le domaine $\mathrm{N}_{\Sigma+\underline{\Sigma}}$, on utilisera trois blocs, un pour représenter les variables de Σ, un autre pour représenter les variables de $\underline{\Sigma}$. Il est ainsi facile de faire commuter ces deux blocs, pour débuter le processus de normalisation sur les variables de $\underline{\Sigma}$, plutôt que sur celle de Σ.

Pour les jointures, on déplace les blocs facilement et on insère de nouveau blocs vides sans problème.

Exemple 7.3.1 On présente ici deux matrices par blocs M_1 et M_2, on détaille ensuite le calcul de leur jointure selon la première composante :

$$M_1 = \left(\begin{array}{ccc|ccc|c} 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{array}\right)$$

$$M_2 = \left(\begin{array}{ccc|ccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 & 1 & 0 & 2 \end{array}\right)$$

On forme la matrice suivante bloc par bloc :

$$\left(\begin{array}{ccc|ccc|ccc|c} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 \end{array}\right)$$

On normalise

$$\left(\begin{array}{ccc|ccc|ccc|c} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right)$$

Puis on projette indépendamment du premier bloc :

$$\left(\begin{array}{ccc|ccc|c} 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right)$$

□

7.3.2 Des lignes creuses

Les lignes au sein de chaque bloc sont creuses. Elles sont représentées par un couple constitué de la liste des clefs ordonnées des éléments non nuls, et une table de hachage qui permet d'accéder à ces éléments.

Ainsi toutes les opérations sur les lignes sont peu coûteuses, par exemple additionner deux lignes revient à faire la fusion de deux listes.

7.3.3 Conclusion

Ces optimisations sont utiles, car la majorité des matrices que l'on rencontre au cours de l'analyse sont essentiellement diagonnales. Ainsi, le nombre d'éléments non nuls d'une matrice est presque linéaire et non plus quadratique. L'occupation mémoire d'une telle matrice devient donc linéaire ainsi que les opérations élémentaires sur les lignes. Les permutations de colonnes sont maintenant en coût constant, mais pour une collection restreinte de séries de permutations.

7.4 Analyse des résultats

Le résultat de l'analyse n'est pas encore directement lisible par l'utilisateur. Il est pour l'instant donné sous forme d'une matrice donnant les relations d'exclusions, d'un tableau d'intervalle pour compter les processus. L'analyse des identifiants est donnée par une table de hachage.

Par la suite, un *script CGI* sera réalisé. Il calculera à partir d'un fichier texte, une page *html* qui contiendra toute l'information calculée par l'analyse.

Pour les compteurs de tâches, l'information non-relationnelle sera donnée telle quelle, alors que l'information relationnelle sera redondante. On donnera en effet deux matrices : la matrice normale permet d'établir des contraintes de causes à effets et la matrice *positive* permet d'établir des contraintes d'exclusion mutuelle.

Pour les contraintes temporelles, on obtiendra l'information en cliquant sur les étiquettes de tâches, ou sur les noms de canaux. On obtiendra alors les graphes sous formes de grammaire linéaire gauche, et contraintes sur les occurences, ces contraintes seront simplifiées en éliminant toute contrainte du type $x = k$ lorsqu'elles peuvent s'obtenir trivialement par la loi de Kirchoff.

L'utiliseur peut accèder à l'information calculée par le biais de tests insérés directement dans le processus. Il peut aussi manipuler les matrices de Karr pour extraire l'ensemble des contraintes qui ne portent que sur un sous-ensemble choisis de variables.

8 Conclusion : Vers une analyse modulaire

L'analyse des problèmes d'échappement de variables, présentée succintement laisse entrevoir l'opportunité de concevoir une analyse modulaire pour étudier les processus du π-calcul. Ceci, permettrait d'étudier des programmes conséquents et donc de passer à l'échelle.

Cependant, il faut rester conscient que l'analyse d'un sous-processus n'apporte pour l'instant qu'une information très restreinte quant au reste du processus. Elle permet en effet de détecter d'une part que des transitions sont impossibles et d'autre part que certains sous-processus sont inaccessibles. L'information qu'elle donne sur les identifiants est partielle, et on se sait pas comment remplir les trous qui représentent les interactions du sous-processus avec le monde extérieur.

Une fois ces problèmes résolus, il sera temps d'adapter la méthodologie développée lors de la conception de π-*sa* , à des langages réels, qui utilisent des modèles sémantique proche du π-calcul, comme par exemple le langage *ERLANG*, utilisé par *Ericsson* en téléphonie mobile.

A Processus analysés avec π-*sa*

On présente dans cette appendice quelques exemples de processus analysés avec succes par π-*sa*. Pour chaque exemple, les résultats viennent directement de l'analyseur et aucun calcul n'a été effectué à la main.

A.1 Les processus alternés

A.1.1 Fichier source

A :=*a?[x](x![a] + c?[u]d![u]);
B :=*b?[x](x![b] + c![e]);
C :=a![b];

(A|B|C);;

A.1.2 Accessibilité

$(\nu$a$)(\nu$b$)(\nu$c$)(\nu$d$)(\nu$e$)$
$(*\text{a}?^1[x](x!^2[\text{a}] + \text{c}?^3[u]\cancel{\text{d}!^4[u]})$
$|*\text{b}?^5[x](x!^6[\text{a}] + \text{c}!^7[\text{e}])$
$|\text{a}!^8[\text{b}])$

A.1.3 Compteur de processus

On note $\sharp(i)$ le nombre d'instances du processus P tel que i soit l'étiquette du premier symbole de P, au cours des différentes étapes d'exécution.

π-*sa* calcule notamment les contraintes suivantes :

$$\begin{cases} \sharp(1) = \sharp(5) = 1 \\ \sharp(4) = 0 \\ \sharp(i) \in [|0;1|],\ \forall i \in \{2;3;6;7;8\} \\ \sum\limits_{i \in \{2;3;6;7;8\}} \sharp(i) = 1 \end{cases}$$

π-*sa* a détecté une exclusion mutuelle entre le sous-processus $(\text{c}?^3[u]\text{d}!^4[u])$ et le sous-processus $(\text{c}!^7[\text{e}])$.

Cette contrainte suffit à montrer que ces deux sous-processus ne peuvent pas communiquer entre eux. Elle est de plus trouvée très rapidement à l'issue de la deuxième itération.

A.2 Le serveur trois ports

A.2.1 Fichier source

```
((*make?[](♯socket)(♯out)(♯infoclient)
              ((socket?[in](out![in]|make![]))
                |socket![infoclient]))
| make![]
| make![]
| make![]
| make?[]make?[]make?[]make?[]test![]);;
```

A.2.2 Accessibilité

$(\nu\text{make})(\nu\text{test})$
$\quad((*\text{make}?^{1}[](\sharp socket)(\sharp out)(\sharp infoclient)$
$\qquad\qquad(socket?^{2}[in](out!^{3}[in]|\text{make}!^{4}[]))$
$\qquad\qquad|socket!^{5}[infoclient]))$
$\quad|\ \text{make}!^{6}[]$
$\quad|\ \text{make}!^{7}[]$
$\quad|\ \text{make}!^{8}[]$
$\quad|\ \text{make}?^{9}[]\text{make}?^{10}[]\text{make}?^{11}[]\text{make}?^{12}[]\cancel{\text{test}!^{13}[]})$

π-*sa* a permis de détecter que le sous-processus (test$!^{13}$) est inaccessible. Ainsi, il ne peut jamais y avoir simultanément plus de trois *message* (make![]) au cours d'une exécution.

A.2.3 Compteur de processus

On note $\sharp(i)$ le nombre d'instances du processus P étiqueté i, au cours des différentes étapes d'exécution.

π-*sa* calcule entre autres les contraintes suivantes :

$$\begin{cases} \sharp(1) = 1 \\ \sharp(13) = 0 \\ \sharp(i) \in [|0;3|],\ \forall i \in \{2;3;4;5\} \\ \sharp i \in [|0;1|],\ \forall i \in [|6;12|] \\ \sharp(4) + \sharp(6) + \sharp(7) + \sharp(8) = 3 - \sharp(2) - \sharp(10) - 2 \times \sharp(11) - 3 \times \sharp(12) \end{cases}$$

La contrainte d'exclusion mutuelle détectée par π-*sa*, permet de déduire par réduction que le sous-processus étiqueté 13 est inaccessible. En effet, celui-ci n'est accessible qu'à partir d'une configuration qui contiendrait un *message* (make![]) et un sous processus étiqueté 12.

Or $\sharp(12) \geqslant 1 \implies \sharp(4) + \sharp(6) + \sharp(7) + \sharp(8) \leqslant 0$.
Puis $\sharp(12) \geqslant 1 \implies \sharp(4) = \sharp(6) = \sharp(7) = \sharp(8) = 0$.

A.2.4 Sécurité

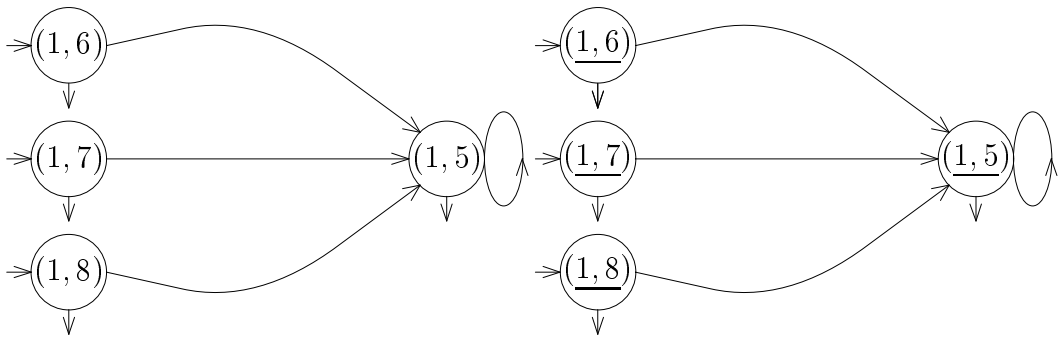
π-*sa* prouve aussi la sécurité du protocole, en montrant que l'information émise par le client ne peut être renvoyée que sur son propre canal de sortie.

Pour cela, on examine les environnements que l'on peut associer au processus 3 :

$$\begin{cases} (3, \mathit{out}, \mathit{out}) & \mapsto \left((g, \underline{g}),\ \{\lambda = \underline{\lambda},\ \forall \lambda \in \Sigma \right. \\ (3, \mathit{in}, \mathit{infoclient}) & \mapsto \left((g, \underline{g}),\ \{\lambda = \underline{\lambda},\ \forall \lambda \in \Sigma \right. \end{cases}$$

où g et $\underline{g}$ sont données figure A.1

Figure A.1 graphes des identifiants



Ces contraintes suffisent à montrer que dans une tâche de la forme ($\mathit{out}![\mathit{in}],\mathit{id}$), l'identifiant du canal *out* auquel est affecté la variable *out*, et l'identifiant du canal *infoclient* auquel est affecté la variable *in*, sont tous deux égaux à *id*.

Ainsi, lorsqu'un *message infoclient* est émis sur un canal *out*, ces deux canaux ont nécessairement été créés par la même réplication du serveur. Ils appartiennent donc au même client.

A.3 Le serveur erroné

A.3.1 Fichier source

```
((*make?[](♯socket)(♯out)(♯infoclient)
              (socket?[in]out![in]
               |make![]
               |socket![infoclient]))
| make![]
| make![]
| make![]
| make?[]make?[]make?[]make?[]test![]);;
```

A.3.2 Accessibilité

$(\nu\text{make})(\nu\text{test})$
$((*\text{make}?^{1}[](\sharp socket)(\sharp out)(\sharp infoclient)$
$(socket?^{2}[in]out!^{3}[in]$
$|socket!^{4}[infoclient]))$
$|\text{make}!^{5}[]$
$|\ \text{make}!^{6}[]$
$|\ \text{make}!^{7}[]$
$|\ \text{make}!^{8}[]$
$|\ \text{make}?^{9}[]\text{make}?^{10}[]\text{make}?^{11}[]\text{make}?^{12}[]\text{test}!^{13}[])$

A.3.3 Compteur de processus

On considère une étape d'exécution, on note $\sharp(i)$ le nombre d'instances du processus P étiqueté i et $\underline{\sharp}(\lambda)$ le nombre de transitions étiquetée λ, effectuées à partir d'une configuration initiale pour atteindre cette étape.

π-sa calcule entre autres les contraintes suivantes :

$$\begin{cases} \sharp(1) = 1 \\ \sharp(13) = 0 \\ \sharp(i) \in [|0;\infty|[,\ \forall i \in \{2;3;4;5\} \\ \sharp i \in [|0;1|],\ \forall i \in [|6;12|] \\ \sharp(2) = \underline{\sharp}(1,5) + \underline{\sharp}(1,6) + \underline{\sharp}(1,7) + \underline{\sharp}(1,8) - \sharp(3) \end{cases}$$

π-sa signale qu'il ne peut majorer le nombre d'instances simultanées du sous-processus étiqueté par 2. Ce n'est bien sûr qu'un avertissement, puisque l'analyse effectue une approximation supérieure de la sémantique collectrice concrète.

π-*sa* a donc su faire la différence entre un protocole réaliste qui n'utilise qu'un nombre limité de ressources physiques, et un protocole qui en utilise un nombre arbitraire.

Par ailleurs, π-*sa* donne une piste pour trouver la source de l'erreur potentielle, sous la forme d'une contrainte linéaire. Celle-ci exprime les liens de causes à effets entre le nombre de sous-processus présents et celui des transitions effectuées.

A.3.4 Sécurité

π-*sa* prouve la sécurité du protocole. En montrant que l'information émise par le client ne peut être réémise que sur son propre canal de sortie.

Les contraintes calculées sur les identifiants sont les mêmes que pour le protocole du serveur 3 ports (Cf A.2.4).

A.4 L'anneau de communication

A.4.1 Fichier source

```
init        :=make![left0];
link(x,y)   :=edge![x,y];
close(x)    :=edge![x,first];
call(x)     :=make![x];

(*make?[last](♯next)
            ( link(last,next)
            | call(next)
            | [last=next]test1![])
| *make?[last]close(last)
| init
| *edge?[x,y][x=y]test2![x]);;
```

A.4.2 Accessibilité

$(\nu\text{make})(\nu\text{edge})(\nu\text{first})(\nu\text{test1})(\nu\text{test2})$
$(*\text{make}?^{1}[last](\nu next)$
$(\text{edge}!^{2}[last,next]$
$|\text{make}!^{3}[left]$
$|[last=^{4}next]\text{test1}!^{5}[])$
$|*\text{make}?^{6}[last](\text{edge}!^{7}[last,\text{first}])$
$|\text{make}!^{8}[\text{first}]$
$|*\text{edge}?^{9}[x,y][x=^{10}y]\ \text{test2}!^{11}[x]$

A.4.3 Compteurs de processus

On note $\sharp(i)$ le nombre d'instances du processus P étiqueté par i soit l'étiquette du premier symbole de P, au cours des différentes étapes d'exécution.

π-sa calcule entre autres les contraintes suivantes :

$$\begin{cases} \sharp(1) = \sharp(6) = \sharp(9) = 1 \\ \sharp(5) = 0 \\ \sharp(i) \in [|0;1|],\ \forall i \in \{3;4;7;8\} \\ \sharp(i) \in [|0;\infty|[,\ \forall i \in [|2;10;11|] \\ \underline{\sharp}(9,2) + \sharp(2) = \underline{\sharp}(1,8) + \underline{\sharp}(1,3) \\ \underline{\sharp}(9,7)\sharp(3) + \sharp(7) + \sharp(8) = 1 \end{cases}$$

Remarque A.4.1 *L'analyse n'est pas aussi précise, lorsque l'on ne change pas le paramètre n du domaine $\mathfrak{B}$, pour la troisième itération. En effet, si l'on garde $n = 0$, la réduction n'est pas assez fine pour contrebalancer l'imprécision provoquée par l'opérateur d'élargissement, et l'analyseur trouve alors la contrainte $\sharp(7) \in [|0; 2|]$, au lieu de la contrainte $\sharp(7) \in [|0; 1|]$.*

A.4.4 Sécurité

π-sa prouve la sécurité du protocole, en montrant que chaque sommet est soit connecté au sommet initial, soit connecté à son successeur.

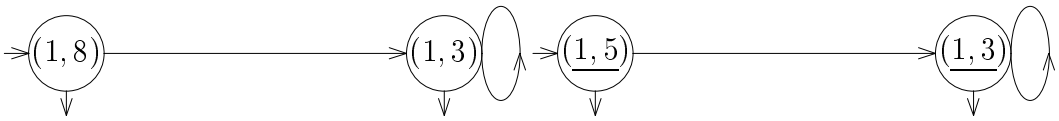
Pour cela, on examine les environnements que l'on peut associer aux processus 2 et 7.

On traite par exemple le cas où la connection est assurée par le sous-processus (edge![2][*last*,*next*]) et que le canal first n'est pas affecté à la variable *last*.

$$\begin{cases} (2, last, next) & \mapsto \Big((g, \underline{g}), \big\{(1,3) = (\underline{1,3}) + 1 \quad \big) \\ (2, next, next) & \mapsto \Big((g, \underline{g}), \big\{(1,3) = (\underline{1,3}) \quad \big) \end{cases}$$

où g et $\underline{g}$ sont données figure A.2

Figure A.2 graphes des identifiants



Ces contraintes suffisent à montrer que dans une tâche de la forme (edge![*last*,*next*],*id*), lorsque le canal first n'est pas affecté à la variable *last*, l'identifiant du canal *next* auquel est affecté la variable *last* est obtenu en concaténant la lettre $(1, 3)$ à *id*, alors que l'identifiant du canal *next* auquel est affecté la variable *next*, est égal à *id*. Ainsi, la variable à laquelle est affecté canal *next* a été créée par la réplication de ressource qui a succédé à celle qui avait créé la variable à laquelle est affecté le canal *last*

A.4.5 Topologie du réseau

On sait désormais, que chaque sommet est soit connecté à son successeur, soit connecté au sommet initial.

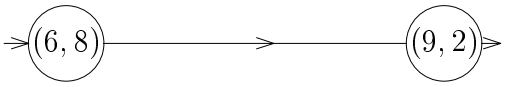
Par ailleurs, *π-sa* nous informe que seul le sommet initial peut éventuellement peut être connecté à lui même.

Pour cela, on examine tous les environnements que l'on peut associer aux processus 11.

$$\begin{cases} (11, test2, test2) & \mapsto \left((g', \varepsilon_1), \begin{cases} (6,8) = 1 \\ (9,2) = 1 \end{cases} \right) \\ (11, x, first) & \mapsto \left((g', \varepsilon_1), \begin{cases} (6,8) = 1 \\ (9,2) = 1 \end{cases} \right) \end{cases}$$

où g' est donné figure A.3

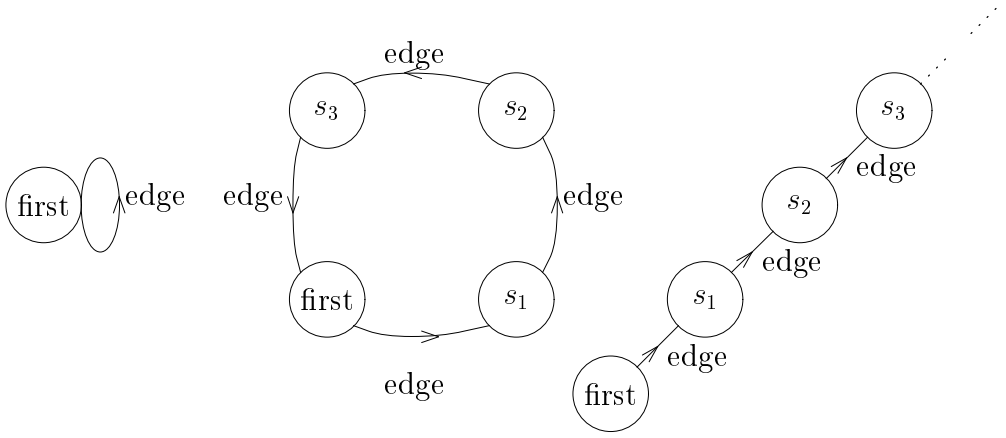
Figure A.3 graphe de l'identifiants



D'autre part, *π-sa* nous avertit qu'il se peut que des dérivations infinies existent. En effet, il ne peut trouver de borne au nombre de transitions étiquetés $(1, 3)$ que peut comporter une exécution. Cette information est obtenue par le compteur de tâches : $(\sharp(2) \in [|0; \infty|[)$.

On décrit alors figure A.4 les trois formes que peut prendre un réseau de communication formé au cours d'une exécution du processus **ring**.

Figure A.4 topologies des réseaux engendrés par le processus **ring**



B Prévisions

B.1 Un terme ouvert

B.1.1 Fichier source

```
P:=( a?[]b?[]test![]
   | a?[]b![]
   | a![]) ;

(*make[]((♯ a)(♯ b)P | make![])
| make![])
```

B.1.2 Accessibilité

$(\nu\text{make})(\nu\text{test})$
$\quad(*\text{make}?^{1}[](\nu a)(\nu b)$
$\qquad\qquad(a?^{2}[]b?^{3}[]\sout{\text{test}!^{4}[]}$
$\qquad\qquad|a?^{5}[]b!^{6}[]$
$\qquad\qquad|a!^{7}[]$
$\quad)$

L'analyse ouverte du sous-processus **P** indépendamment de son contexte permet de déduire que le sous-processus $(\text{test}!^{4}[])$ est inaccessible.

En effet, l'analyseur détecte que seuls les canaux (make) et (test) sont publiques. Ainsi, le contexte du sous-processus **P** ne peut interférer avec celui-ci ni sur le canal a, ni sur le canal b. Puis, l'analyse des exclusions mutuelles montre que sous ces conditions, le sous-processus $(\text{test}!^{4}[])$ est inaccessible.

Références

[BDNN98] C. Bodei, P. Degano, F. Nielson, and H. R. Nielson. Control flow analysis for the π-calculus. In *Proc. CONCUR'98*, number 1466 in Lecture Notes in Computer Science, pages 84–98. Springer-Verlag, 1998.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, August 1992.

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening-narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming, Proceedings of the Fourth International Symposium, PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 1992.

[Cou78] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* PhD thesis, Université Scientifique et Médicale de Grenoble, 1978.

[Cou81] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, 1981.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, pages 133–151, 1976.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. In *Proceedings of the International Summer School on Logic and Algebra of Specification.* Springer Verlag, 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[Tur95] D. N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation.* PhD thesis, Edinburgh University, 1995.

[Ven98] A. Venet. Automatic determination of communication topologies in mobile systems. In *Proceedings of the Fifth International Static Analysis Symposium SAS'98*, volume 1503 of *Lecture Notes in Computer Science*, pages 152–167. Springer-Verlag, 1998.