Algèbres de Processus Réversibles

Jean Krivine

Thèse de doctorat de l'université Paris 6

Spécialité :

Informatique

Présentée par :

Jean Krivine

Pour obtenir le grade de :

Docteur de l'université Paris 6

Sujet de thèse :

Algèbres de Processus Réversibles

et Programmation Concurrente Déclarative.

Soutenue le 16 novembre 2006 devant le jury composé de :

Jean-Jacques Lévy	*Directeur de thèse*
Robin Milner	*Examinateur*
Catuscia Palamidessi	*Rapporteur*
Iain Phillips	*Rapporteur*
Christian Queinnec	*Président du Jury*
Glynn Winskel	*Examinateur*

Cette page est laissée vide intentionnellement.

Table des matières

Abstract We propose a notion of distributed backtracking built on top of Milner's Calculus of Communicating Systems. This reversible process algebra (RCCS) offers a clear-cut theoretical characterisation of reversibility. In particular, given a process and a past, we show that RCCS allows to backtrack along any *causally equivalent* past. We express behavioural equivalence of reversible processes in terms of simple bisimimulation of causal transition systems. This results in a declarative way of programming distributed transactional systems that can be efficiently model-checked using an event structure based algorithm. Using a categorical abstraction we then show that this method can be generalised for a wide range of concurrent calculus.

Résumé Nous présentons un système de retour arrière distribué basé sur le Calcul des Systèmes Communicants de Robin Milner. L'algèbre de processus réversible ainsi définie (RCCS) nous permet de poser les fondements théoriques du retour arrière dans un calcul concurrent. En particulier, étant donné un processus et un passé, nous montrons que RCCS permet de revenir en arrière dans tout passé *causalement équivalent*. Nous exprimons aussi l'équivalence comportementale associée aux processus réversibles en utilisant une notion de bisimulation mettant en relation les traces causales des processus. Il en résulte une méthode de programmation déclarative de systèmes transactionnels qui peuvent être efficacement vérifiés à l'aide d'un algorithme basé sur des structures d'événements. Par l'intermédiaire d'une construction catégorique, nous montrons que cette méthode peut être généralisée à une large classe de calculs concurrents.

Remerciements

Je voudrais remercier Vincent Danos qui a contribué en grande partie à la formalisation de ces travaux et qui a su trouver les bonnes questions quand je cherchais trop tôt des réponses.

Un grand merci aussi à Pawel Sobocinski pour sa contribution majeure aux résultats du chapitre 7.

Je remercie Jean-Jacques Lévy de m'avoir accueilli au sein de son projet et de m'avoir laissé le libre choix des thèmes de recherche. Ses conseils précieux m'ont permis de rester à flot tout au long de ces trois années de thèse.

Je voudrais aussi remercier notre assistante Sylvie ainsi que toute l'équipe du projet MOSCOVA pour l'environnement de travail exceptionnel tant sur le plan humain que scientifique.

Je saisis l'occasion pour remercier aussi mes enseignants de l'Université Paris 7 et en particulier Marie Ferbus, Serge Griegorieff et Jean-Marie Rifflet.

Enfin, je remercie Soraya de son soutien moral à toute épreuve et mes parents qui ont su me convaincre qu'après footballeur le métier de chercheur était sans doute l'un des plus beaux du monde.

Introduction.

Systèmes distribués.

Un système *concurrent* est composé de processus exécutant des tâches de manière indépendante et pouvant se synchroniser ponctuellement en s'échangeant des messages. Lorsque ces processus ne se trouvent pas chapeautés par un unique ordonnanceur (*scheduler*) on dit aussi que le système est *distribué* ou encore *réparti*. De tels systèmes peuvent être utilisés pour des raisons différentes. En premier lieu on peut vouloir distribuer une tâche par souci d'efficacité : lorsque le calcul contient des parties pouvant être effectuées indépendamment, on repartit la charge de calcul entre des sites différents avant de compiler un résultat global (séquençage d'ADN, *grid computing*). La répartition d'une tâche entre différents agents est aussi un garant de robustesse : la faillite d'un des processus n'entraînant pas forcément la faillite de l'ensemble de la tâche. Par exemple dans le domaine de la biologie moléculaire, qui traite aussi de systèmes biochimiques hautement répartis, il est clair que l'interaction de milliers de protéines chargées de la régulation d'un substrat sera beaucoup plus résistante aux "pannes" qu'une hypothétique super-protéine régulatrice. Enfin on peut aussi voir la distribution d'un système comme la résultante de contraintes topologiques inhérente au problème traité : la commande d'un livre sur Internet implique, par exemple, l'interaction d'un client, d'un service de commande et d'une base de donnée qui se situent probablement dans des lieux géographiques différents.

En fonction du problème distribué qu'on cherche à implémenter on peut vouloir définir plusieurs types de correction. Les propriétés qu'on va exiger d'un programme dépendent bien sûr du degré de sensibilité de l'opération effectuée (simple consultation d'une base de donnée ou virement bancaire) mais aussi de son degré de distribution. Dans le cas où l'architecture générale du système est très fortement répartie il devient difficile de s'assurer que le programme fait bien ce pour quoi il est prévu. En effet, en raison du haut degré de non-déterminisme induit par l'entrelacement des exécutions des

divers composants, il peut être délicat de s'assurer que le système réalise *uniquement* la tâche spécifiée et qu'il ne se bloque jamais. Lorsque le niveau de distribution est moindre, on s'intéressera plus à contrôler la robustesse et l'efficacité du programme concurrent. Dans le cas d'un calcul réparti, cela revient à vérifier que celui-ci résiste bien aux pannes et que ses composants redémarrent dans un état cohérent vis-à-vis du reste du système et avec un minimum de perte de calcul.

Le problème posé est de s'abstraire du langage dans lequel ces systèmes sont programmés et de les décrire dans un formalisme suffisamment expressif pour parler de propriétés qu'on souhaite vérifier.

Le choix du formalisme.

Pour représenter les systèmes distribués, nous avons fait le choix de nous placer dans la tradition des algèbres (ou calculs) de processus. Les termes de ces calculs sont des processus formels et peuvent être vus à la fois comme une spécification d'un système réel et comme un langage de programmation contenant des primitives de communication. Ceci provient du fait que la théorie des algèbres de processus emprunte des concepts au monde des automates (spécification) et à la théorie du λ-calcul (langage).

Il existe de nombreuses variantes d'algèbres de processus, on peut toutefois faire la distinction entre celles qui permettent de parler de concurrence pure et celles qui expriment aussi la possibilité de transmettre des capacités d'interactions entre processus. On retrouve à la base de la première catégorie les *Communicating Sequential Processes* (CSP) de C.A.R Hoare[1] ainsi que le *Calculus of Communicating Systems* (CCS) de Robin Milner[2]. Le π-calcul[3] est quant à lui le formalisme à la base de la deuxième catégorie d'algèbres.

Outre l'avantage d'une correspondance intuitive entre processus formels et processus réels, une motivation importante pour l'usage des algèbres de processus est qu'elles constituent un moyen *modulaire* de décrire des systèmes concurrents. À partir de la formalisation d'un processus p et d'un processus q, il est possible de déduire le comportement de $(p \parallel q)$ correspondant aux processus s'exécutant en parallèle. Cette modularité dans la description des processus est particulièrement souhaitée car elle correspond aussi au caractère modulaire de nombreux systèmes distribués dont les composants peuvent avoir des rôles bien distincts.

[1] [Hoare, 1985]
[2] [Milner, 1989]
[3] [Milner *et al.*, 1992]

Le formalisme principal dans lequel se situent nos travaux est celui de CCS. C'est un langage minimaliste et bien étudié, suffisamment expressif pour décrire des systèmes concurrents non triviaux. Il est toutefois clair que notre approche n'est pas spécifique à ce langage et peut être adaptée à d'autres algèbres de processus comme nous le montrerons avec le π-calcul.

Réversibilité.

Dans les systèmes distribués dits *transactionnels*, plusieurs processus se disputent l'accès à certaines ressources, par exemple des droits en écriture sur un fichier ou simplement la connexion à un site donné. Chaque participant a donc une vue assez claire de l'objectif qu'il souhaite atteindre, mais pour ce faire, il doit émerger un *consensus* correspondant aux règlements des potentiels conflits entre processus, aboutissant à la validation des transactions en cours ou à l'annulation d'une partie d'entre elles quand un consensus général ne peut être trouvé ; on parle alors de verrous (*deadlock*). Lorsqu'un consensus nécessite la rupture d'une symétrie originelle dans la configuration des processus en compétition, on ne peut pas, *a priori*, donner de solution qui exclut systématiquement les verrous durant le déroulement des transactions[4]. On peut se rendre compte de ce fait en considérant un individu A remontant une rue et un individu B venant dans le sens inverse. On peut assimiler les trajectoires de A et B à des transactions qui seront validées une fois que l'objectif final de chacun des piétons sera atteint. Il y a un consensus distribué à résoudre si les trajectoires se rejoignent à un moment donné. Le consensus est résolu directement si A et B choisissent de bifurquer de façon complémentaire ; s'ils choisissent le même côté il faudra retenter un nouveau choix de trajectoires. Des robots mal programmés pourraient tenter de verrouiller à l'avance une décision de trajectoire et on aboutirait à des situations où ils se bloqueraient mutuellement. Il est donc clair que le code de tels robots doit incorporer un mécanisme de changement de décision pour sortir des cas où le consensus n'a pu être trouvé. Une première solution serait de redémarrer les robots dans leurs états initiaux (*restart*), mais le conflit n'arrivant qu'à un point précis de la trajectoire, on perdrait alors inutilement du temps. Une solution plus efficace serait de faire revenir les robots sur leurs traces (*backtrack*) et de retenter, après chaque pas en arrière, de repartir vers l'avant en tentant éventuellement de nouveaux choix locaux de trajectoires. La transaction décrite n'implique pourtant que deux participants, seuls responsables de leurs décisions. Or, la plupart des

[4][Bougé, 1988]

transactions nécessitent l'interaction de plusieurs composants : par exemple l'écriture dans un fichier partagé nécessite l'acquisition de droits demandant un type de consensus plus difficile à obtenir. Le système de retour arrière peut donc être arbitrairement complexe à mettre en œuvre.

Dans les systèmes transactionnels, la présence de verrous impose donc au programmeur de résoudre des problèmes distincts : d'une part il s'agit de programmer chacun des processus participant à la transaction de sorte que leurs interactions puissent conduire à une solution du problème, d'autre part, il faut aussi s'assurer que le système ne reste pas bloqué dans un état où la requête ne peut plus aboutir. Il faut donc prévoir l'ensemble des verrous possibles induits par le système et programmer une façon d'en sortir, en vérifiant que toute action ayant mené au verrou est correctement annulée (débit d'argent, pré-réservation...). On peut donc distinguer deux étapes de programmation différentes : la première correspond à un mode de programmation "vers l'avant" qui est la plupart du temps intuitif car il met en jeu des interactions entre processus dans l'ordonnancement pensé par le programmeur. La deuxième étape est plus compliquée, car les verrous sont précisément causés par les ordonnancements imprévus et correspond à un mode de programmation "vers l'arrière". Ce dernier est d'autant plus délicat que tout retour en arrière (*backtrack*) d'un calcul distribué se doit de préserver la *cohérence* des états du système. En d'autres termes, l'annulation d'une communication doit être précédée des annulations de toutes les actions qu'elle aurait pu causer. Une question naturelle est alors de savoir s'il est possible de définir de façon générique, une procédure permettant de revenir en arrière sur une partie d'un calcul distribué, tout en préservant la cohérence globale du système. Il serait ainsi possible de se contenter de codes "vers l'avant" décrivant les étapes nécessaires à l'élaboration d'une transaction concurrente sans avoir à expliciter les procédures d'annulation en cas de verrous.

Selon que l'on regarde une algèbre de processus comme un langage de spécification ou comme la base d'un langage de programmation concurrente, le développement d'un calcul de processus réversible présente divers avantages. Dans le cadre d'un langage de programmation cela revient, nous l'avons dit, à automatiser (et donc fiabiliser) la tâche d'implémentation correspondant aux phases d'annulation d'une transaction vouée à l'échec. Dans le cadre d'un langage de spécification on coupe du même coup une partie des preuves à effectuer pour certifier un programme puisque le code de celui-ci à été compressé au strict nécessaire et que la correction du retour arrière est déjà prouvée.

Toutefois, sans préciser ce que l'on entend par retour arrière, la question

posée plus haut est incomplète : il est en effet toujours possible de définir un calcul dans lequel un coordinateur central mémoriserait chaque état intermédiaire du système afin de pouvoir y revenir en cas de besoin. Cette méthode, sans mentionner la lourdeur de sa mise en œuvre, serait contradictoire avec le principe de distribution du calcul. Nous reformulons donc la question de la façon suivante : peut-on définir un langage permettant de faire revenir en arrière le calcul d'un ensemble de processus (ici, des termes de CCS) sans affaiblir le degré de distribution de ce calcul et en préservant la cohérence des états du système ? Dans ce travail, nous proposons de définir les bases théoriques nécessaires à l'élaboration d'un tel langage. Nous en déduirons une méthode de programmation concurrente, dite *déclarative*, permettant d'obtenir des systèmes transactionnels correct à partir de programmes CCS interprétés dans la sémantique réversible que nous auront définie. Nous verrons que cette méthode pourrait être la base d'un nouveau mode de programmation, pour les systèmes transactionnels, dans un langage concurrent intégrant un retour arrière à la manière de PROLOG.

Plan.

Nous présentons brièvement CCS au chapitre 1 ainsi que la notion de correction qui lui est traditionnellement associée. Nous consacrons le chapitre 2 à la définition d'un CCS réversible (RCCS) dont nous montrons la correction au chapitre 3. Nous introduisons les actions irréversibles au chapitre 4 où nous définissons la méthode de programmation concurrente déclarative pour RCCS ainsi que le théorème transactionnel qui en est la base. Nous faisons une brève introduction sur les structures d'événements dans le chapitre 5 et nous définissons un algorithme permettant de vérifier la correction d'un programme déclaratif utilisant une structure d'événements comme représentation interne des processus. Nous décrivons au chapitre 6 les modifications théoriques et pratiques utilisées pour l'implémentation de cet algorithme dans un module du langage Ocaml. Dans le chapitre 7 nous définissons les concepts catégoriques sous-jacents à la méthode de programmation concurrente déclarative ; en particulier nous montrons que la définition d'une telle méthode pour un langage donné se réduit à exhiber un système de factorisation sur la catégorie des calculs du langage. Enfin, nous montrons dans le chapitre 8 que RCCS peut constituer les bases d'un langage pour la biologie des systèmes, en tirant profit du fait que les interactions biologiques sont, pour la plupart, réversibles et fondamentalement concurrentes. Nous concluons ce mémoire par un retour sur les différents concepts abordés ainsi que par l'étude plus détaillée de deux extensions envisagées de RCCS parti-

culièrement importantes dans une optique d'implémentation.

To non french speaking readers. Papers dealing with Chapters 2 and 4 appeared in the proceedings of CONCUR'04 and CONCUR'05[5]. Chapter 7 has also been accepted for publication in the proceedings of EXPRESS'06[6]. The design of biological systems using RCCS detailed in Chapter 8 has been investigated in a paper published in the proceedings of BioConcur'03[7]. On may also have a look at my INRIA research report[8] for a brief overview of Chapters 5 and 6.

[5] [Danos et Krivine, 2004, Danos et Krivine, 2005]
[6] [Danos *et al.*, 2006a]
[7] [Danos et Krivine, 2003]
[8] [Krivine, 2006]

Chapitre 1

Systèmes concurrents et CCS.

Sommaire

Le calcul des systèmes communicants (CCS) a été introduit par Robin Milner comme paradigme de langage concurrent. Il diffère, par exemple, des réseaux de Petri en ce qu'il s'utilise comme un langage de programmation dont les primitives seraient réduites aux simples envois et réceptions sur des canaux. On peut donc lire un terme CCS comme un code abstrait, correspondant à un véritable programme et l'exécuter suivant des règles bien définies. Dans ce chapitre préliminaire, nous présentons en premier lieu la syntaxe de CCS ainsi que la sémantique opérationnelle associée. Dans un second temps nous montrons comment définir une notion de correction pour un processus vis-à-vis d'une spécification de référence. Nous terminons par une discussion sur l'utilité d'une extension du langage autorisant les communications entre processus à être réversibles.

1.1 Définition du langage.

1.1.1 Syntaxe.

Par analogie avec les systèmes qu'ils permettent de décrire, les termes (ou agents) de CCS sont appelés des *processus*. Ces processus possèdent des *capacités* d'interactions qui correspondent à des demandes de communication sur des canaux nommés. Nous donnons, figure 1.1, la syntaxe complète des processus.

Capacité	α	:=	a	Demi-action
		|	τ	Action interne
	a	:=	$\bar{x}, \bar{y}, \bar{z}, \ldots$	Émission
		|	$x, y, z, \ldots$	Réception
Processus	p	::=	$\alpha.p$	Préfixe
		|	$p \parallel p$	Composition parallèle
		|	$p + p$	Choix
		|	$\mathsf{D}(\tilde{x})$	Définition (récursive)
		|	$p \backslash x$	Restriction
		|	0	Processus terminé

FIG. 1.1 – Syntaxe des processus CCS.

Un processus de la forme $p = \alpha.q$ est dit *préfixé* par une capacité d'action α et de *continuation* q. Par exemple, $x.q$ indique un processus attendant une communication sur le canal x avant de passer dans l'état q. La forme $p \parallel q$ exprime la concurrence entre les exécutions de p et q. Sans accès à l'ordonnanceur on ne peut pas trancher lequel des processus réagira en premier, ce qui introduit une forme de non-déterminisme dans la sémantique de CCS. Le choix $p + q$ est une autre façon d'obtenir un comportement non-déterministe : le processus $\bar{x}.p + \bar{y}.q$ peut soit émettre sur le canal x et passer dans l'état p, soit émettre sur y et passer dans l'état q. Les définitions récursives, de la forme $\mathsf{D}(\tilde{x})$, où $\tilde{x}$ est un vecteur de noms de canaux, permettent de modéliser des comportements infinis. On utilise pour cela un ensemble $\Delta := \{\mathsf{D}_i(\tilde{x}) := p_i\}$ qui permet d'associer à chaque constante $\mathsf{D}_i(\tilde{x})$ un appel à un processus p_i de paramètres $\tilde{x}$. On notera que chaque p_i peut contenir lui-même des appels récursifs. On considère que chaque processus p s'exécute dans un environnement statique Δ contenant

un ensemble de définitions récursives. Par exemple dans l'environnement $\Delta := \{\mathsf{A}(x,y) := x.\mathsf{A}(x,y) + \bar{y}.0\}$, le processus $\mathsf{A}(x,y)$ attend un nombre arbitraire de réceptions sur le canal x et peut terminer à tout moment par une émission sur y[1].

Comme nous l'avons dit en introduction, un des atouts fondamentaux des algèbres de processus est la possibilité de définir des systèmes de façon modulaire et de les composer ensuite. L'opérateur de restriction $p\backslash x$ permet de spécifier que l'usage du nom x est restreint au processus p. Les conflits de noms avec un potentiel canal x à l'extérieur de la restriction sont gérés par α-conversion en considérant que la restriction $p\backslash x$ lie x dans p. L'ensemble des noms libres dans un processus p est noté $\mathsf{fn}(p)$ et défini inductivement par :

$$\begin{array}{lcl}
\mathsf{fn}(0) & := & \emptyset \\
\mathsf{fn}(\mathsf{D}(\tilde{x})) & := & \bigcup_i x_i \in \tilde{x} \\
\mathsf{fn}(\bar{x}.p) & := & \{x\} \cup \mathsf{fn}(p) \\
\mathsf{fn}(x.p) & := & \{x\} \cup \mathsf{fn}(p) \\
\mathsf{fn}(p+q) & := & \mathsf{fn}(p) \cup \mathsf{fn}(q) \\
\mathsf{fn}(p \parallel q) & := & \mathsf{fn}(p) \cup \mathsf{fn}(q) \\
\mathsf{fn}(p\backslash x) & := & \mathsf{fn}(p) - \{x\}
\end{array}$$

L'*α-convertibilité* $\sim_\alpha$ est la plus petite relation d'équivalence satisfaisant :

$$\frac{y \notin \mathsf{fn}(p)}{p\backslash x \sim_\alpha (p\,\{y/x\})\backslash y} \qquad (\alpha\text{-conv})$$

Par commodité, on définit aussi une congruence structurelle sur les termes de CCS, qui est la plus petite relation d'équivalence satisfaisant :

$$\begin{array}{lcll}
p+0 & \equiv & p & \\
p+q & \equiv & q+p & \\
(p+q)+p' & \equiv & p+(q+p') & \\
p \parallel 0 & \equiv & p & \\
p \parallel q & \equiv & q \parallel p & \\
(p \parallel q) \parallel p' & \equiv & p \parallel (q \parallel p') & \\
\mathsf{D}(\tilde{y}) & \equiv & p\,\{\tilde{y}/\tilde{x}\} & \text{si } (\mathsf{D}(\tilde{x}) := p) \in \Delta \\
p & \equiv & q & \text{si } p \sim_\alpha q
\end{array}$$

1.1.2 Sémantique opérationnelle.

La sémantique opérationnelle des processus CCS est donnée par un *système de transitions étiquetées* (LTS pour *labelled transition system*) dénoté

[1] Les appels aux définitions de processus n'ont pas de continuation et on peut comparer leur usage avec les appels du système UNIX de la forme `exec`.

par un quadruplet $\langle P, p, A, \rightarrow \rangle$ où P représente l'ensemble des processus CCS, p est l'état initial d'un processus et A est l'ensemble des actions étiquetant la relation de transition $\rightarrow \subseteq P \times A \times P$. Nous donnons, figure 1.2, le système de transitions engendré par un processus.

$$\frac{}{\alpha.p + q \xrightarrow{\alpha} p}\ (\mathsf{act})$$

$$(\mathsf{par})\ \frac{p \xrightarrow{\alpha} p'}{p \parallel q \xrightarrow{\alpha} p' \parallel q} \qquad \frac{p \xrightarrow{\alpha} p' \quad \alpha \notin \{x, \bar{x}\}}{p\backslash x \xrightarrow{\alpha} p'\backslash x}\ (\mathsf{res})$$

$$(\mathsf{equiv})\ \frac{p \equiv p' \xrightarrow{\alpha} q' \equiv q}{p \xrightarrow{\alpha} q} \qquad \frac{p \xrightarrow{\bar{a}} p' \quad q \xrightarrow{a} q'}{p \parallel q \xrightarrow{\tau} p' \parallel q'}\ (\mathsf{synch})$$

FIG. 1.2 – Sémantique opérationnelle de CCS.

L'axiome (act) de la sémantique opérationnelle est intuitif : lorsqu'un processus a la possibilité de faire une action, il peut la tenter en renonçant à tout autre choix alternatif. La règle (par), formalise la possibilité d'asynchronie entre processus : si p peut effectuer une action α, il peut toujours le faire sans faire évoluer le contexte. La règle de restriction (res) interdit toute interaction entre p et son contexte sur un canal privé. Le règle (equiv) est un moyen simple de réduire le nombre de règles du LTS en considérant les transitions *modulo* congruence structurelle. Enfin la règle de synchronisation (synch) formalise un rendez-vous entre processus : si à un même instant, deux processus p et q sont capables d'effectuer des actions complémentaires, il est alors possible d'établir une communication. Dans ce cas les deux demi-actions fusionnent en une unique transition interne au processus $p \parallel q$.

1.1.3 Expressivité.

Nous illustrons maintenant l'expressivité de CCS à travers deux exemples inspirés de problèmes de gestion de ressources partagées. Le premier est un système simple de vote distribué, le second correspond à une solution naïve du problème du dîner des philosophes. Nous reviendrons par la suite à l'étude de ces exemples qui permettrons d'illustrer notre démarche.

Exemple 1.1.1 (Système à m votants.) *On considère un système composé d'un ensemble N de processus pouvant émettre un vote sur un canal*

v. On désire implémenter un système dans lequel au plus $m \leq |N|$ processus auront le droit de voter. On obtient un système à m votants en forçant chaque processus à prendre un ticket, dans un ensemble en contenant m, via le canal t avant de pouvoir voter. Le système est donc :

$$\big(\prod_{i=1}^{|N|} \bar{t}.\bar{v}_i.0 \parallel \prod_{1}^{m} t.0\big)\backslash t$$

où $\prod_{i=1}^{k} p_i$ *est une notation pour* $p_1 \parallel \ldots \parallel p_k$.

Un exemple de calcul pour un système à un votant composé de deux processus est :

$$\big(\bar{t}.\bar{v}_1.0 \parallel \bar{t}.\bar{v}_2.0 \parallel t.0\big)\backslash t \xrightarrow{\tau} \big(\bar{t}.\bar{v}_1.0 \parallel \bar{v}_2.0\big)\backslash t$$

$$\xrightarrow{\bar{v}_2} \big(\bar{t}.\bar{v}_1.0\big)\backslash t$$

On note que le processus restant ne peut plus formuler de vote du fait de la restriction du canal t.

Exemple 1.1.2 (Le dîner des philosophes.) *Dans cette exemple proposé par Dijkstra, N philosophes sont placés autour d'une table. Chaque philosophe peut être en train de manger (état* M*) ou bien de penser (état* P*). Un philosophe qui pense peut commencer à manger en prenant les baguettes disposées à sa droite et à sa gauche. La contrainte de partage veut que chaque baguette est partagée entre philosophes voisins ; ainsi le philosophe assis en position i partage sa baguette de gauche avec le philosophe en position $i-1$ et sa baguette de droite avec le philosophe en position $i+1$. La table étant circulaire, on considère toutes les opérations modulo N. Un philosophe en train de manger fini toujours par reposer ses deux baguettes pour se remettre à penser. On modélise le comportement d'un philosophe en fonction de son état :*

$$\Delta := \begin{cases} \mathsf{P}(b_i, b_{i+1}, d_i, f_i) & := \quad \bar{b}_i.\bar{b}_{i+1}.d_i.\mathsf{M}(b_i, b_{i+1}, d_i, f_i) \\ & \quad\; + \; \bar{b}_{i+1}.\bar{b}_i.d_i.\mathsf{M}(b_i, b_{i+1}, d_i, f_i) \\ \mathsf{M}(b_i, b_{i+1}, d_i, f_i) & := \quad f_i.\big(\mathsf{P}(b_i, b_{i+1}, d_i, f_i) \parallel b_i.0 \parallel b_{i+1}.0\big) \end{cases}$$

le canal d_i servant à informer le contexte que le philosophe i commence à manger et le canal f_i signalant la fin de cette période et le retour à l'état pensif. L'état initial du système consiste à mettre en parallèle N philosophes en train de penser, accompagnés de leurs baguettes :

$$P_N := \prod_{i=1}^{N} \big(\mathsf{P}(b_i, b_{i+1}, d_i, f_i) \parallel b_i.0\big)\backslash \tilde{b}$$

Quelques commentaires s'imposent ; même en l'absence d'une spécification formelle du comportement attendu du système, il est clair que le code décrit n'est pas correct. On s'attend à ce qu'un philosophe puisse manger en excluant ses voisins directs (ce qui est effectivement le cas), mais nous avons omis de traiter les verrous qui conduisent au blocage de philosophes voisins dans un état intermédiaire. Ce mécanisme de blocage peut être observé par la transaction suivante dans un système à 2 philosophes :

$$\begin{aligned} P_2 &\xrightarrow{\tau} \big(\bar{b}_2.\bar{d}_1.\mathsf{M}(b_1,b_2,d_1,f_1) \parallel \mathsf{P}(b_2,b_1,d_2,f_2) \parallel b_2.0\big)\setminus\{b_1,b_2\} \\ &\xrightarrow{\tau} \big(\bar{b}_2.\bar{d}_1.\mathsf{M}(b_1,b_2,d_1,f_1) \parallel \bar{b}_1.\bar{d}_2.\mathsf{M}(b_2,b_1,d_2,f_2)\big)\setminus\{b_1,b_2\} \end{aligned}$$

Dans cet état, la règle (res) interdit toute interaction entre b_1 ou b_2 et le contexte (ce qui traduit le fait qu'un philosophe ne peut pas aller chercher de baguette ailleurs). Les deux philosophes sont donc dans un état bloqué sans possibilité pour eux de reposer une baguette ou de manger. Nous reviendrons par la suite à ce problème classique de verrous causé par l'entrelacement de requêtes d'accès à une ressource partagée.

1.2 Bisimulation et notion de correction pour CCS.

Nous avons vu qu'on pouvait utiliser CCS à la manière d'un langage de programmation afin d'implémenter abstraitement des systèmes communicants. Les exécutions de tels systèmes sont alors déduites de la syntaxe des processus CCS qui les composent, par l'intermédiaire de la sémantique opérationnelle donnée sous forme de LTS (Figure 1.2). Étant donnés deux processus CCS, on peut se demander si ceux-ci ont les mêmes capacités d'interactions quelle que soit la façon interne dont ils ont été implémentés. Déterminer une telle propriété permettrait de définir une hiérarchie entre processus, du plus abstrait (spécification) au plus bas niveau (implémentation), en vérifiant que tous les processus intermédiaires ont toujours le même comportement observable. La notion de *bisimilarité* entre processus nous fournit l'outil demandé. Il existe de nombreux types de bisimulations[2], mais nous présentons ici une variante de la bisimulation faible (*weak bisimulation*) basée sur un choix paramétrique d'actions observables. Muni d'une telle notion d'équivalence, nous serons à même de définir la correction d'un processus bas niveau en fonction d'une spécification de référence.

[2] [Sangiorgi, 1995, Galpin, 2000]

1.2.1 Notion d'équivalence.

On considère des systèmes de transitions étiquetées dans A muni d'un ensemble $K \subseteq A$ d'étiquettes *observables*. Pour le LTS d'un processus CCS on prend traditionnellement $K = A\backslash\{\tau\}$ mais nous nous en tiendrons ici à cette version paramétrique. On considère aussi l'ensemble complémentaire de K dans A que nous désignons par K^c. Enfin on note $\rightarrow_w^*$ une séquence de transitions étiquetées avec $w \in A^*$, où A^* désigne l'ensemble des mots engendrés par l'alphabet A. La relation de (bi)simulation est définie de la façon suivante (voir aussi Figure 1.3) :

Définition 1.2.1 *Soient $\mathcal{S}_1 = \langle S_1, s_1, A, \rightarrow\rangle$ et $\mathcal{S}_2 = \langle S_2, s_2, A, \rightarrow\rangle$ des systèmes de transitions étiquetées munis d'un ensemble $K \subseteq A$ d'observables. Une relation $\mathcal{R} \subseteq S_1 \times S_2$ est une* simulation faible *entre $\mathcal{S}_1$ et $\mathcal{S}_2$ si $s_1\ \mathcal{R}\ s_2$ et pour tout $s \in S_1$ et tout $t \in S_2$, si $s\ \mathcal{R}\ t$ alors :*

- *$s \xrightarrow{a} s'$, $a \in K^c$ implique $t \rightarrow_w^* t'$ avec $w \in (K^c)^*$ et $s'\ \mathcal{R}\ t'$*
- *$s \xrightarrow{a} s'$, $a \in K$ implique $t \rightarrow_w^* t'$ avec $w \in (K^c)^* a (K^c)^*$ et $s'\ \mathcal{R}\ t'$*

De plus $\mathcal{R}$ est une bisimulation faible *si $\mathcal{R}^{-1}$ est aussi une simulation.*

Pour se représenter la notion de simulation on peut imaginer un jeu entre l'opposant $\mathcal{S}_1$ et le joueur $\mathcal{S}_2$. À chaque transition de $\mathcal{S}_1$, le joueur $\mathcal{S}_2$ doit pouvoir répondre par une série de transitions portant la même observation et avoir toujours une stratégie gagnante dans l'état d'arrivée quels que soient les coups suivants de $\mathcal{S}_1$. Le jeu de bisimulation est plus difficile à gagner puisque l'opposant à le droit de changer de système après chaque réponse du joueur. Par exemple si on prend $K = A\backslash\{\tau\}$, le processus $p := a.0 + b.0$ est bien simulé par $q := \tau.a.0 + \tau.b.0$, en revanche ces processus ne sont pas bisimilaires car après une τ-transition de q, l'opposant peut faire le choix de jouer l'action visible à laquelle q a renoncé.

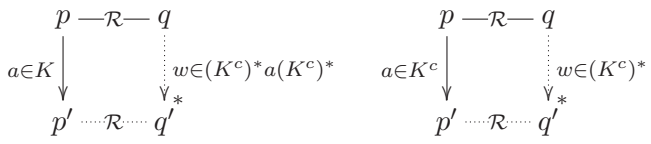


FIG. 1.3 – Relation de (bi)simulation.

Il est clair que notre notion d'équivalence est d'autant plus discriminante que K est grand. Dans les cas limites, si $K = A$ on obtient la notion de

bisimulation forte et prendre $K = \emptyset$ revient à mettre en relation tous les systèmes de transitions.

1.2.2 Spécification.

Nous avons construit les exemples de la section 1.1.3 en faisant correspondre un processus CCS à chaque agent à modéliser (un votant, un philosophe) ; le système final étant une simple composition parallèle de chacun de ces processus de base. Avec une telle méthode il est clair que le grain de distribution du système formel est le même que celui du système réel. Toutefois, lorsqu'on veut spécifier un problème concurrent on ne se soucie plus, en principe, de préserver ce grain et on se contente d'un système qui génère les interactions attendues du processus avec l'environnement de façon minimale. On donne généralement la spécification d'un système sous forme de LTS, mais on peut aussi se servir de CCS comme langage de spécification. La notion de correction "à la Milner" pour les processus CCS est la suivante : étant donné un LTS de spécification $\mathcal{S}$ et une implémentation p, on dira que cette dernière est correcte vis à vis de $\mathcal{S}$ s'il existe une relation de bisimulation entre $\mathcal{S}$ et le LTS engendré par p. Dans la suite nous donnerons alternativement des spécifications directement sous forme de LTS ou bien sous forme de processus CCS. Reprenons les exemples 1.1.1 et 1.1.2 afin de les analyser à la lumière de ces nouvelles notions. Étant donné un ensemble N de processus, on souhaite autoriser $m \leq |N|$ votants. On donne la spécification suivante :

$$\mathcal{S}_{vote}(m+1, N \uplus \{i\}) \xrightarrow{v_i} \mathcal{S}_{vote}(m, N) \tag{1.1}$$

Cette spécification, d'apparence raisonnable, impose en fait des contraintes assez fortes sur l'implémentation. En effet, elle requiert qu'à partir du moment où il reste encore des tickets de vote disponibles, *tout* processus n'ayant pas encore voté a la possibilité de le faire. Nous allons voir que cette propriété n'est pas satisfaite par le code donné dans l'exemple 1.1.1. Pour savoir si notre implémentation $\big(\prod_{i\in N} \bar{t}.\bar{v}_i.0 \parallel \prod_{1}^{m} t.0\big)\backslash t$ est correcte il nous faudrait exhiber une relation de bisimulation avec $\mathcal{S}_{vote}(m, N)$, en prenant comme ensemble d'observables $K = \{\tilde{v}\}$. Or les deux LTS ne sont clairement pas bisimilaires ; pour s'en persuader, nous montrons, Figure 1.4, que la spécification du système à deux processus et un votant n'est déjà pas correctement implémentée. Ce cas est un exemple typique de choix partiel (*partial commitment*) qui entraîne la non-correction du programme : dans l'état initial, le processus "voit" un certain nombre de solutions et il existe un chemin

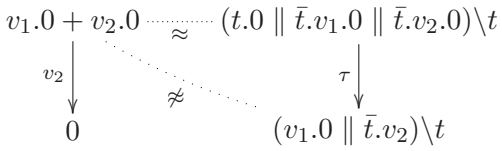


FIG. 1.4 – Les systèmes ne sont pas bisimilaires.

silencieux, partant de l'état initial, qui force à renoncer à certains de ces choix. Afin de rester conforme à la spécification des m-votants, il est nécessaire d'introduire de la divergence pour retarder au maximum le choix du vote[3]. Le principe est que toute transition silencieuse nécessaire pour trouver un consensus (ici vérifier qu'au plus m votes sont effectués) ne doit pas entraîner de choix définitifs. Autrement dit la partie correspondant à la recherche d'une solution doit être annulable tant que la solution en question n'a pas été validée.

Exemple 1.2.1 (Les m-votants sans choix partiels.) *On crée un système dans lequel il est impossible de prévoir le résultat d'un vote avant que celui-ci ait réellement eut lieu. Pour cela on rend* réversible *la phase d'acquisition d'un ticket par de la récursion :*

$$\Delta_{vote} := \big\{\mathsf{Vote}(t,v) := \bar{t}.\big(v.0 + \tau.(\mathsf{Vote}(t,v) \parallel t.0)\big)\big\}$$

et le système à m votants autorisés devient :

$$V_{m,N} := \big(\prod_{i=1}^{|N|}\mathsf{Vote}(t,v_i) \parallel \prod_{1}^{m} t.0\big)\backslash t$$

Proposition 1.2.1 *Le processus $V_{m,N}$ est correct, i.e $V_{m,N} \approx \mathcal{S}_{vote}(m,N)$ avec $K = A\backslash\{\tau\}$.*

La figure 1.5 se contente de montrer la correction de cette nouvelle implémentation pour deux processus et un votant autorisé. □

On procède de même avec l'exemple 1.1.2. On numérote les positions autour de la table de 1 à N et on partitionne l'ensemble des philosophes entre ceux qui mangent M et ceux qui pensent P. Le système de transitions étiquetées

[3][Nestmann et Pierce, 1996, Palamidessi et Herescu, 2005]

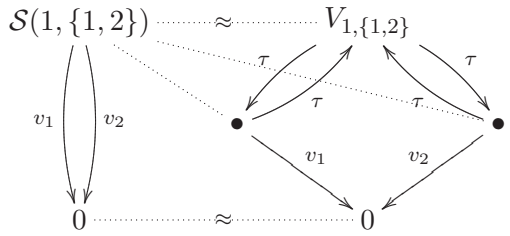


FIG. 1.5 – Correction des m votants avec retour arrière.

correspondant à la spécification du problème des philosophes est donné par :

$$\begin{array}{lcl} \mathcal{S}_{phil}(P \cup \{i-1, i, i+1\}, M) & \xrightarrow{d_i} & \mathcal{S}_{phil}(P \cup \{i-1, i+1\}, M \cup \{i\}) \\ \mathcal{S}_{phil}(P, M \cup \{i\}) & \xrightarrow{f_i} & \mathcal{S}_{phil}(P \cup \{i\}, M) \end{array} \tag{1.2}$$

Comme nous l'avons mentionné plus tôt, l'implémentation de cette spécification, donnée dans l'exemple 1.1.2, n'est pas correcte car elle contient des verrous. Ces états bloqués ne pouvant être mis en relation avec aucun état de la spécification, il en résulte qu'on ne peut pas trouver de relation de bisimulation entre la spécification donnée, dès deux philosophes. Le problème que nous rencontrons est similaire à celui qui a entraîné l'incorrection du code non réversible des m-votants ; l'erreur provenant de ce que des choix partiels étaient opérés avant le vote final. Dans le cas des philosophes, le fait de saisir une baguette réduit aussi l'espace des solutions accessibles : la prise de la baguette de droite par le philosophe i empêche toute chance de manger aux philosophes $i-1$ et $i+1$. Dans les cas limites, la réduction de l'espace de choix est telle qu'on aboutit à des verrous. Il est possible d'éviter ces choix partiels en rendant réversible toute action entrant dans l'élaboration du consensus.

Exemple 1.2.2 (Les philosophes sans verrous.) *Dans le nouveau système, la prise d'une baguette devient une opération réversible gérée par l'ensemble de définitions récursives :*

$$\Delta_{phil} = \left\{ \begin{array}{lcl} \mathsf{P}'(b_1, b_2, d, f) & := & \bar{b}_1.\big(\mathsf{Q}(b_1, b_2, d, f) + \tau.(\mathsf{P}'(b_1, b_2, d, f) \parallel b_1)\big) \\ \mathsf{Q}(b_1, b_2, d, f) & := & \bar{b}_2.\big(\mathsf{M}(b_1, b_2, d, f) + \tau.(\mathsf{P}'(b_1, b_2, d, f) \parallel b_1 \parallel b_2)\big) \\ \mathsf{M}(b_1, b_2, d, f) & := & f.\big(\mathsf{P}'(b_1, b_2, d, f) \parallel b_1 \parallel b_2\big) \end{array} \right.$$

On obtient un système sans verrou avec l'état initial :

$$P'_N := \prod_{i=1}^{N} \Big(\big(\mathsf{P}'(b_i, b_{i+1}, d_i, f_i) + \mathsf{P}'(b_{i+1}, b_i, d_i, f_i) \big) \parallel b_i.0 \Big) \backslash \tilde{b}$$

Proposition 1.2.2 *Le processus P'_N est correct, i.e $P'_N \approx \mathcal{S}_{phil}(N, \emptyset)$ avec $K = A \backslash \{\tau\}$.*

Le diagramme de la figure 1.6 prouve la proposition pour $N = \{1, 2\}$ (pour simplifier, on a identifié les états intermédiaires correspondant à un choix différent dans la prise de la première baguette). □

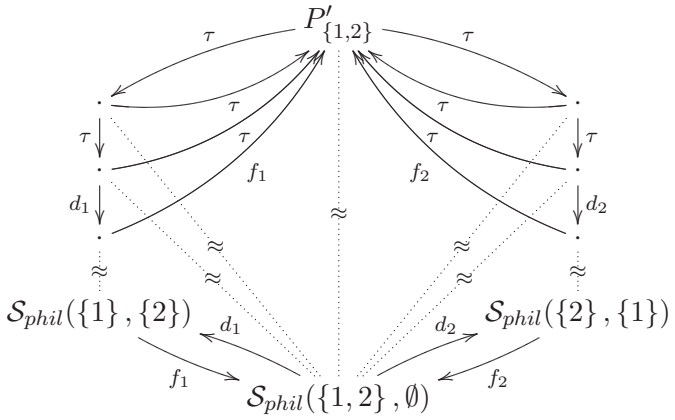


FIG. 1.6 – Correction du système à 2 philosophes avec retour arrière.

1.3 Discussion.

Lorsqu'on compare les exemples 1.1.1 et 1.1.2 avec leurs correspondants réversibles, on s'aperçoit que l'obtention de la correction s'est faite au prix de la lisibilité et de la simplicité du code. Par ailleurs, la méthode que nous avons employée pour implémenter le retour arrière par l'intermédiaire de processus récursifs n'est en rien générique. Dans le cas des exemples choisis, le retour arrière est rendu aisé par le faible degré d'imbrication des actions nécessaires à l'établissement du consensus. Ainsi, reposer une baguette ou un ticket est une opération triviale dans le sens qu'elle ne requiert pas à son tour de consensus pour être effectuée. Toutefois un système transactionnel

arbitraire n'a pas de raison d'avoir ces bonnes propriétés. Les mécanismes transactionnels sont pourtant riches en potentiels verrous et choix partiels du fait de la compétition entre processus concurrents. Dans de nombreux cas les opérations de recherche d'un consensus doivent pouvoir être annulées afin de tenter d'autres alternatives ou d'obtenir un ordonnancement plus favorable. Du point de vue de la programmation, l'ajout de tels mécanismes se révèle délicat et pour le moins contre-intuitif. Pourtant, à partir d'un code contenant des verrous indésirables, il est assez facile de distinguer les actions devant être réversibles des actions définitives. Basés sur ce constat nous allons mettre au point, dans les prochains chapitres, une sémantique opérationnelle permettant de faire repartir en arrière des processus de façon cohérente.

Chapitre 2

Systèmes à communications réversibles.

Sommaire

Nous présentons dans ce chapitre le *Reversible Calculus of Communicating Systems* (RCCS), une extension de CCS dans laquelle les communications sont par défaut réversibles[1]. La prise en compte d'un mécanisme de retour arrière dans une algèbre de processus de type CCS est à mettre en relation avec plusieurs études antérieures étudiant des propriétés de tolérance aux pannes[2] ou la volonté de définir un modèle transactionnel basé sur une algèbre de processus avec retour arrière[3]. Tandis que ces approches reposent sur une représentation explicite des fautes gérées par un système d'exceptions, nous avons opté pour une sémantique moins déterministe, dans laquelle une demande de retour arrière peut être formulée à tout instant sans avoir à prévoir de mécanisme de gestion *ad-hoc*. Bien que les mécanismes d'exceptions présentent l'intérêt de correspondre à l'implémentation de nombreux systèmes, notre approche offre une base théorique plus claire et correspond à l'idée qu'on ne cherche pas à prévoir les blocages mais à

[1] [Danos et Krivine, 2004]
[2] [Prasad, 1987, Janowski, 1994, Amadio et Prasad, 1994, Riely et Hennessy, 2001]
[3] [Bergstra *et al.*, 1994]

définir un mécanisme générique de retour arrière pouvant être déclenché à tout instant. Les transitions de RCCS sont donc intrinsèquement réversibles à la manière d'une réaction chimique $H + O + H \leftrightarrow H_2O$. Cette intuition, couplée à la remarque que la plupart des interactions bio moléculaires sont aussi réversibles, a motivé une approche permettant de modéliser des systèmes biologiques à l'aide de RCCS[4] ; ce champ particulier d'application sera détaillé au chapitre 8. Après avoir présenté la syntaxe des processus RCCS, nous définirons la propriété de cohérence syntaxique d'un système. Nous présenterons ensuite la sémantique associée aux systèmes réversibles et nous montrerons qu'elle préserve bien la cohérence syntaxique des processus. Nous finirons par discuter de la nécessité d'introduire une notion de cohérence plus calculatoire.

2.1 Un CCS avec retour arrière.

L'idée pour implémenter un mécanisme de retour arrière est d'équiper chaque processus d'une pile mémoire capable d'enregistrer ses interactions. À chaque nouvelle transition vers l'avant, l'information nécessaire pour revenir en arrière étant ajoutée sur la pile. Deux contraintes ont guidé notre choix du système de mémoire. Comme nous l'avons dit en introduction, la machinerie nécessaire pour implémenter le retour arrière ne doit pas nuire à la distribution du système. Il existe donc autant de piles mémoires qu'il y a de processus s'exécutant en parallèle. On peut voir chacune de ces mémoires comme un processus dédié à l'annulation des communications du programme CCS dont il est le tuteur. Deuxièmement, le système de retour arrière ne doit pas être trop flexible au risque de perdre la cohérence du système, c'est à dire de revenir dans des états ne correspondant pas à un passé accessible depuis l'état initial. Cette notion de cohérence, pour le moment informelle, va être exprimée dans ce chapitre de façon *syntaxique* : nous définissons l'ensemble des systèmes RCCS grammaticalement corrects puis nous montrons que cette cohérence est préservée par la sémantique opérationnelle. Nous verrons, dans le chapitre suivant, une autre notion de cohérence plus *opérationnelle* qui définit ce qu'on entend par retour arrière dans un passé possible du système.

2.1.1 Syntaxe.

Les éléments fondamentaux de RCCS sont appelés des *processus*. À la différence de CCS, ils ne peuvent être eux même composés de sous processus

[4] [Danos et Krivine, 2003]

s'exécutant en parallèle, auquel cas on parlera ici de *systèmes*. Un processus RCCS est de la forme $m \triangleright p$ où m est une *pile mémoire* dont le sommet est à gauche et le fond à droite et où p est un processus CCS classique appelé *processus simple*. On utilisera aussi l'anglicisme *processus monitoré par* m pour désigner le processus simple dont m se charge d'enregistrer les actions.

Processus simples.

$$\begin{array}{lll} \alpha & ::= & a \mid \bar{a} \mid \tau \qquad\qquad \text{Actions CCS} \\ \\ p, q & ::= & 0 \mid \sum \alpha_i.p_i \mid (p \parallel q) \mid p\backslash x \mid D(\tilde{x}) \quad \text{Processus CCS} \end{array}$$

La syntaxe des processus simples correspond à celle des processus CCS dite à somme gardée. La notation $\sum \alpha_i.p_i$ pour la somme indique en effet que chaque branche de choix doit pouvoir être déterminée par le choix d'une action α_i. On s'autorise toutefois à utiliser $p + 0$ pour indiquer une branche de choix morte. Les processus simples de RCCS ne sont plus considérés à congruence structurelle près pour la composition parallèle ; en revanche on conserve les règles classiques pour la somme, la restriction et l'α-conversion (voir Section 1.1.1) qu'on désigne par $\equiv_{+,\backslash,\alpha}$. Cette rigidité dans la structure des systèmes RCCS nous permet d'identifier de façon simple les processus intervenant dans une transition.

Systèmes.

$$\begin{array}{lll} r, s & ::= & m \triangleright p \quad \text{Processus} \\ & \mid & (r \parallel s) \quad \text{Composition parallèle} \\ & \mid & r\backslash x \quad \text{Restriction} \end{array}$$

Les systèmes RCCS sont composés de processus monitorés mis en parallèle et pouvant se trouver dans le champ d'une restriction. Les systèmes sont considérés à congruence structurelle près, définie par :

$$\begin{array}{lcll} (r \parallel s) & \equiv & (s \parallel r) & \\ (r \parallel s) \parallel s' & \equiv & r \parallel (s \parallel s') & \\ m \triangleright p & \equiv & m \triangleright p' \quad \text{si } p \equiv_{+,\backslash,\alpha} p' & \\ m \triangleright (p \parallel q) & \equiv & (\langle 1 \rangle \cdot m \triangleright p) \parallel (\langle 2 \rangle \cdot m \triangleright q) & \textsf{(dist)} \\ m \triangleright (p\backslash x) & \equiv & (m \triangleright p)\backslash x \quad \text{si } x \notin m & \textsf{(perm)} \end{array}$$

La règle (dist) s'interprète naturellement de gauche à droite ; un processus simple $p = p_1 \parallel p_2$, monitoré par une mémoire m, peut lancer l'exécution de p_1 et p_2 après duplication de m en $\langle 1 \rangle \cdot m$ et $\langle 2 \rangle \cdot m$. Ces dernières deviennent

les mémoires gérant respectivement p_1 et p_2. Dans le sens inverse on obtient que deux processus $\langle 1\rangle \cdot m \triangleright p_1$ et $\langle 2\rangle \cdot m \triangleright p_2$ peuvent fusionner en un père unique $m \triangleright p$. Noter que les piles ne divergent qu'en leurs sommets, valant $\langle 1\rangle$ et $\langle 2\rangle$. La règle (perm) indique qu'un processus simple restreint sur un canal x doit être exécuté dans un système où x est aussi restreint. La condition $x \notin m$ peut toujours être satisfaite par α-conversion et permet d'éviter des captures de noms. On considère aussi une congruence supplémentaire permettant de déplacer les restrictions à l'extérieur d'un système. Cette règle n'étant pas utile pour la définition de la sématique opérationnelle de RCCS, nous la laissons délibérément à part. Elle se révélera ultérieurement nécessaire pour la définition de la cohérence syntaxique.

$$(r\backslash x) \parallel s \equiv (r \parallel s)\backslash x \quad \text{si } x \notin s \qquad \text{(scope)}$$

Mémoires. Comme nous l'avons dit, les mémoires RCCS sont des piles (dont le sommet se lit à gauche) contenant une information locale permettant de faire revenir en arrière un processus.

m, m'	$::=$	$\langle \star, \alpha, p\rangle \cdot m$	Demi-synchronisation.
	$\mid$	$\langle m', a, p\rangle \cdot m$	Synchronisation
	$\mid$	$\langle 1\rangle \cdot m$	Identifiant fils gauche
	$\mid$	$\langle 2\rangle \cdot m$	Identifiant fils droit
	$\mid$	$\langle\rangle$	Fond de pile

En plus des identifiants de fils $\langle 1\rangle$ et $\langle 2\rangle$ il existe deux autres types de cellules pouvant être empilées sur une mémoire. Elles correspondent aux informations nécessaires pour l'annulation d'une interaction. Les cellules de la forme $\langle \star, \alpha, p\rangle$ indiquent que le processus vient d'effectuer une action α en renonçant au processus simple alternatif p et on notera simplement $\langle m, a\rangle$ au lieu de $\langle m, a, 0\rangle$. Le symbole $\star$ indique que le processus partenaire, ayant effectué l'action complémentaire $\bar{\alpha}$ n'est pas encore connu. Il s'agit en fait d'une sémantique retardée (*late semantics*) car le symbole $\star$ sera remplacé par l'identifiant mémoire du processus partenaire au moment de l'application de la règle de synchronisation (voir Section 2.1.2) ; dans ce cas on obtient une cellule de la forme $\langle m, a, p\rangle$ indiquant que le processus a effectué l'action a avec un processus de mémoire m.

Exemple. Sans sémantique opérationnelle on peut d'ores et déjà noter que les processus RCCS possèdent une information syntaxique concernant les interactions passées. Ainsi nous verrons que le système

$$\langle\langle 2\rangle, \bar{x}\rangle\langle 1\rangle \triangleright p \parallel \langle\langle 1\rangle, x\rangle\langle 2\rangle \triangleright q$$

provient d'un terme unique $\langle\rangle \triangleright (\bar{x}.p \parallel x.q)$. Un système RCCS contient donc à tout instant la même quantité d'information, en se contentant de faire passer les capacités d'interactions du côté futur (processus simple) au côté passé (mémoires) et *vice versa*. Un système RCCS correspond donc à une photographie (*snapshot*) d'un programme concurrent pris à un temps donné. Dans les cas limites, le programme n'a pas encore réagi, le système est alors de la forme $\langle\rangle \triangleright p$ ou bien n'a plus de réactions vers l'avant possibles, par exemple $m \triangleright 0$. Tous les processus RCCS ne correspondent pas à une photographie cohérente. Considérons par exemple le système :

$$r = \langle\langle 2\rangle, x\rangle \cdot \langle 1\rangle \triangleright p \parallel \langle 2\rangle \triangleright q$$

dans lequel la mémoire du processus de gauche indique la réception d'un message sur le canal x en provenance d'un processus dont la mémoire valait $\langle 2\rangle$, tandis que la mémoire de ce dernier n'a enregistré aucun envoi. Nous caractérisons donc dans la prochaine section, l'ensemble des systèmes dont les mémoires contiennent une information cohérente.

2.1.2 Sémantique réversible.

À la manière de CCS, nous définissons la sémantique opérationnelle de RCCS à l'aide d'un LTS (voir Figure 2.1). Les transitions sont des quadruplets $t = \langle r, \mu, \zeta, s\rangle$ où r et s sont, respectivement, la source et le but de la transition, μ est l'ensemble des mémoires impliquées dans la transition et $\zeta ::= \alpha \mid \alpha^-$ est une action *dirigée*.

$$\frac{}{m \triangleright \alpha.p + q \xrightarrow{m:\alpha} \langle\star, \alpha, q\rangle \cdot m \triangleright p}\ (\mathsf{act}) \qquad \frac{}{\langle\star, \alpha, q\rangle \cdot m \triangleright p \xrightarrow{m:\alpha^-} m \triangleright \alpha.p + q}\ (\mathsf{act}^-)$$

$$\frac{r \xrightarrow{m:\bar{a}} r' \quad s \xrightarrow{m':a} s'}{r \parallel s \xrightarrow{m,m':\tau} r'_{m'@m} \parallel s'_{m@m'}}\ (\mathsf{synch}) \qquad \frac{r \xrightarrow{m:\bar{a}^-} r' \quad s \xrightarrow{m':a^-} s'}{r_{m'@m} \parallel s_{m@m'} \xrightarrow{m,m':\tau^-} r' \parallel s'}\ (\mathsf{synch}^-)$$

$$\frac{r \xrightarrow{\mu:\zeta} r'}{r \parallel s \xrightarrow{\mu:\zeta} r' \parallel s}\ (\mathsf{par}) \qquad \frac{r \equiv r' \xrightarrow{\mu:\zeta} s' \equiv s}{r \xrightarrow{\mu:\zeta} s}\ (\equiv) \qquad \frac{r \xrightarrow{\mu:\zeta} r' \quad \zeta \neq x, \bar{x}, x^-, \bar{x}^-}{r\backslash x \xrightarrow{\mu:\zeta} r'\backslash x}\ (\mathsf{res})$$

FIG. 2.1 – Sémantique opérationnelle de RCCS.

Axiomes. On définit tout d'abord les transitions basiques concernant les processus :

$$\text{(act)}\ \frac{}{m \triangleright \alpha.p + q \xrightarrow{m:\alpha} \langle \star, \alpha, q \rangle \cdot m \triangleright p} \qquad \frac{}{\langle \star, \alpha, q \rangle \cdot m \triangleright p \xrightarrow{m:\alpha^-} m \triangleright \alpha.p + q}\ \text{(act}^-\text{)}$$

La règle (act) est une transition vers l'avant qui empile une nouvelle case au sommet de sa mémoire. Celle-ci contient le choix alternatif q qui pourra être rétabli lors de l'application de la règle arrière (act$^-$) qui dépile la case au sommet de la mémoire. Lorsqu'un processus est de la forme $m \triangleright \alpha.p$ on pourra toujours considérer qu'il est structurellement égal à $m \triangleright \alpha.p + 0$ pour appliquer l'axiome et on notera simplement $\langle \star, \alpha \rangle \cdot m$ pour $\langle \star, \alpha, 0 \rangle \cdot m$.

Règles de synchronisation. Les règles de synchronisation sont aussi bi-directionnelles. Dans le sens vers l'avant la règle suit le comportement du processus CCS tout en enregistrant l'adresse mémoire du processus partenaire par l'intermédiaire de l'opérateur @ décrit plus bas :

$$\frac{r \xrightarrow{m:\bar{a}} r' \quad s \xrightarrow{m':a} s'}{r \parallel s \xrightarrow{m,m':\tau} r'_{m'@m} \parallel s'_{m@m'}}\ \text{(synch)}$$

L'instanciation du symbole $\star$, stocké à l'adresse m, par m' est défini par :

$$\begin{array}{lcll} (r \parallel s)_{m'@m} & = & r_{m'@m} \parallel s_{m'@m} & \\ (r\backslash x)_{m'@m} & = & (r_{m'@m})\backslash x & \text{si } x \notin m' \\ (\langle \star, \alpha, q \rangle \cdot m \triangleright p)_{m'@m} & = & \langle m', \alpha, q \rangle \cdot m \triangleright p & \\ r_{m'@m} & = & r & \text{dans les autres cas.} \end{array}$$

La règle de dé-synchronisation est symétrique :

$$\frac{r \xrightarrow{m:\bar{a}^-} r' \quad s \xrightarrow{m':a^-} s'}{r_{m'@m} \parallel s_{m@m'} \xrightarrow{m,m':\tau^-} r' \parallel s'}\ \text{(synch}^-\text{)}$$

Remarquons que la règle (act$^-$) précise bien que le symbole en tête de pile doit être $\star$ pour autoriser un retour arrière. Il est donc impossible de décider unilatéralement de dépiler une mémoire de la forme $\langle m', \alpha, p \rangle \cdot m$ seule la règle (synch$^-$) permet de le faire. Considérons le système $\langle\langle 2\rangle, \bar{x}\rangle\langle 1\rangle \triangleright 0 \parallel \langle\langle 1\rangle, x\rangle\langle 2\rangle \triangleright 0$. On peut appliquer la règle (synch$^-$) de la manière suivante :

$$\frac{\text{(act}^-\text{)}\ \dfrac{}{\langle \star, \bar{x}\rangle\langle 1\rangle \triangleright 0 \xrightarrow{\langle 1\rangle:\bar{x}^-} \langle 1\rangle \triangleright \bar{x}.0} \qquad \dfrac{}{\langle \star, x\rangle\langle 2\rangle \triangleright 0 \xrightarrow{\langle 2\rangle:x^-} \langle 2\rangle \triangleright \bar{x}.0}\ \text{(act}^-\text{)}}{\langle\langle 2\rangle, \bar{x}\rangle\langle 1\rangle \triangleright 0 \parallel \langle\langle 1\rangle, x\rangle\langle 2\rangle \triangleright 0 \xrightarrow{\langle 1\rangle,\langle 2\rangle:\tau^-} \langle 1\rangle \triangleright \bar{x}.0 \parallel \langle 2\rangle \triangleright x.0}\ \text{(synch}^-\text{)}$$

Tandis que les communications vers l'avant d'un processus sont choisies de façon non-déterministe lorsqu'il y a plusieurs partenaires potentiels, le retour arrière est lui totalement déterministe : il se fait entre les deux processus qui disposent de mémoires complémentaires.

Règles de passage au contexte. Les règles de passage au contexte sont identiques à celles de CCS, en gardant à l'esprit qu'elles s'appliquent aussi bien à des actions avant qu'arrière.

$$(\mathsf{par})\ \frac{r \xrightarrow{\mu:\zeta} r'}{r \parallel s \xrightarrow{\mu:\zeta} r' \parallel s} \qquad \frac{r \equiv r' \xrightarrow{\mu:\zeta} s' \equiv s}{r \xrightarrow{\mu:\zeta} s}\ (\equiv)$$

$$(\mathsf{res})\ \frac{r \xrightarrow{\mu:\zeta} r' \quad \zeta \neq x, \bar{x}, x^-, \bar{x}^-}{r\backslash x \xrightarrow{\mu:\zeta} r'\backslash x}$$

La règle (equiv) ne relève pas ici d'une simple commodité pour décrire le LTS, mais elle permet aussi de s'assurer que les mémoires suivent la structure arborescente d'un processus. Par exemple, le processus $m \triangleright (a.p \parallel b.q)$ ne peut réagir tel quel sans appliquer tout d'abord la règle de congruence :

$$\frac{m \triangleright (a.p \parallel b.q) \equiv \langle 1\rangle \cdot m \triangleright a.p \parallel \langle 2\rangle \cdot m \triangleright q \xrightarrow{m:a} \langle \star, a\rangle\langle 1\rangle \cdot m \triangleright p \parallel \langle 2\rangle \cdot m \triangleright b.q}{m \triangleright (a.p \parallel b.q) \xrightarrow{m:a} \langle \star, a\rangle\langle 1\rangle \cdot m \triangleright p \parallel \langle 2\rangle \cdot m \triangleright b.q}$$

De même, il n'existe pas d'axiome permettant de gérer un processus de la forme $m \triangleright (p\backslash x)$ directement. On doit donc auparavant appliquer la congruence (perm) afin de faire permuter la restriction avec la mémoire : $m \triangleright (p\backslash x) \equiv (m \triangleright p)\backslash x$ et appliquer ensuite une transition à $m \triangleright p$ capable de passer la restriction, sous les conditions fixées par la règle (res).

2.1.3 Systèmes cohérents.

On définit le *contexte d'évaluation* $C[\bullet]$ d'un processus de la façon suivante :

$$C[\bullet] := (C[\bullet] \parallel s) \mid (r \parallel C[\bullet]) \mid C[\bullet]\backslash x \mid \bullet \tag{2.1}$$

La *relation de préfixe* sur les mémoires est notée $m \sqsubset m'$ et signifie qu'il existe une mémoire m'' telle que $m'' \cdot m = m'$. Une mémoire m *apparaît* dans un système r, noté $m \sqsubseteq r$ s'il existe un contexte $C[\bullet]$ tel que $r \equiv C[m \triangleright p]$ pour un certain p. Si $m \sqsubseteq m'$ et $m' \sqsubseteq r$ on notera aussi $m \lessdot r$.

Définition 2.1.1 *Un système r est dit* homogène *s'il est de la forme $s\backslash\tilde{x}$ où s est un système sans restriction.*

On peut toujours homogénéiser un système en lui appliquant, autant de fois qu'il est nécessaire, la congruence structurelle (scope) après d'éventuelles α-conversions. Notons qu'une forme homogène d'un système n'est pas unique, par exemple $m \triangleright (p\backslash x)$ et $(m \triangleright p)\backslash x$ sont deux formes homogènes d'un même processus.

Définition 2.1.2 *Soit $r = s\backslash\tilde{x}$ un système homogène où s est sans restriction. Une mémoire $m \Lsh s$ est* compatible *avec r, noté $m \uparrow r$, si :*

1. $m = \langle\rangle$.
2. $m = \langle 1\rangle \cdot m' \Rightarrow \langle 2\rangle \cdot m' \Lsh r$ *et* $m' \uparrow r$.
3. $m = \langle 2\rangle \cdot m' \Rightarrow \langle 1\rangle \cdot m' \Lsh r$ *et* $m' \uparrow r$.
4. $m = \langle m_1, \bar{a}, p\rangle \cdot m_2 \Rightarrow \langle m_2, a, q\rangle \cdot m_1 \Lsh r$ *et* $m_2 \uparrow r$.
5. $m = \langle \star, \bar{x}, p\rangle \cdot m' \Rightarrow x \notin \tilde{x}$ *et* $m' \uparrow r$.
6. $m = \langle \star, x, p\rangle \cdot m' \Rightarrow x \notin \tilde{x}$ *et* $m' \uparrow r$.
7. $m = \langle \star, \tau, p\rangle \cdot m' \Rightarrow m' \uparrow r$.

La notion de cohérence des mémoires garantit qu'aucun processus n'a enregistré d'émission ou de réception complète sans que son partenaire ne l'ait aussi enregistrée et qu'un identifiant de fils est toujours accompagné d'un identifiant partenaire. La répétition de la condition $x \notin \tilde{x}$, dans les règles 5 et 6, permet de s'assurer qu'une mémoire n'a pas enregistré une demi-action sur un canal restreint ; ainsi la mémoire du système $(\langle \star, \bar{x}\rangle.m \triangleright p)\backslash x$ sera considérée comme non compatible. La contrainte d'homogénéité empêche de considérer des mémoires comme étant compatibles alors qu'elles ne sont pas dans le champ d'une même restriction. Par exemple, avec le système suivant :

$$r = \langle 2, \bar{x}\rangle\langle 1\rangle \triangleright 0 \parallel (\langle 1, x\rangle\langle 2\rangle \triangleright 0)\backslash x$$

sans la condition d'homogénéité, on pourrait conclure que les mémoires de r sont compatibles, ce qui ne serait plus vrai après α-conversion :

$$r \sim_\alpha \langle 2, \bar{x}\rangle\langle 1\rangle \triangleright 0 \parallel (\langle 1, y\rangle\langle 2\rangle \triangleright 0)\backslash y$$

Définition 2.1.3 *Les mémoires m et m' sont dites* apparentées *si $m \frown m'$ où $\frown$ est la plus petite relation symétrique satisfaisant :*

$$\frac{}{\langle 1\rangle \cdot m \frown \langle 2\rangle \cdot m} \qquad \frac{m \frown m'}{m_1 \cdot m \frown m_2 \cdot m'}$$

En d'autres termes le branchement des mémoires doit avoir une structure d'arbre. Cette condition suffit à garantir l'unicité des mémoires dans les systèmes où elles sont toutes apparentées, car la relation est clairement anti-réflexive. Cette propriété nous assure que le choix des mémoires comme identifiants de transition est correct.

Définition 2.1.4 (Cohérence syntaxique) *Un système r est* cohérent *si ses mémoires sont compatibles et apparentées deux à deux, i.e :*

$$\begin{array}{ll} (1) & \forall m \sqsubseteq r : m \uparrow r \\ (2) & \forall m, m' \sqsubseteq r : m \neq m' \Rightarrow m \frown m' \end{array}$$

Propriété 2.1.1 *La cohérence est préservée par congruence structurelle.*

Seule la règle (dist) modifie les mémoires. En particulier $m' \frown m$ implique $m' \frown \langle i \rangle \cdot m$ pour tout i. Enfin $\langle 1 \rangle \cdot m$ et $\langle 2 \rangle \cdot m$ sont, par définition, apparentés. De plus si m est compatible avec son système, alors après application de (dist) on a bien $\langle 1 \rangle \cdot m$ et $\langle 2 \rangle \cdot m$ compatibles avec le système équivalent. □

Nous appelons *systèmes racines*, les systèmes composés du seul processus $\langle\rangle \triangleright p$. Notons que le système $r = C[\langle\rangle \triangleright p]$ est cohérent si et seulement si c'est aussi une racine, c'est à dire que le contexte C est vide.

Projection et relèvement. Soient P et R désignant respectivement l'ensemble des processus CCS et des systèmes RCCS. Le calcul introduit est une "décoration" des termes de CCS pouvant être effacée par la *projection* $\varphi : R \to P$ suivante :

$$\begin{array}{l} \varphi(m \triangleright p) = p \\ \varphi(r \parallel s) = \varphi(r) \parallel \varphi(s) \\ \varphi(r \backslash x) = \varphi(r) \backslash x \end{array}$$

De même, tout processus CCS peut être envoyé dans un système RCCS par une opération de *relèvement*, $\ell_{(m_i)_{i \in I}} : P \to R$, paramétrée par une liste de mémoires et qui envoie un processus simple de la forme $\prod_i p_i$, vers un système $\prod_i m_i \triangleright p_i$. On s'intéresse en fait au sous-ensemble des relèvements qui dotent un processus CCS de mémoires cohérentes et en particulier au relèvement initial $\ell(p) = \langle\rangle \triangleright p$. S'il est clair que l'application $\varphi \circ \ell$ correspond à l'identité sur P, en revanche $\ell \circ \varphi$, qui efface les capacités d'interactions vers l'arrière d'un système, n'est pas l'identité sur R.

2.2 Propriétés.

Avant d'énoncer les propriétés qui garantissent le bon comportement du LTS décrit plus haut, nous rappelons quelques définitions standard.

Notations. Deux transitions sont *co-initiales* si elles ont même source, *co-finales* si elles ont même but et *composables* si le but de la première est aussi source de la deuxième. Une transition RCCS est dite *positive* si elle est orientée vers l'avant et *négative* sinon. Les transitions sont dénotées par t, t', et t^- désigne la transition inverse de t, avec $(t^-)^- = t$. Les séquences de transitions composables deux à deux sont appelées *traces de calcul* ou simplement traces et seront dénotées par des lettres grecques $\gamma, \delta, \sigma, \ldots$. Les notions définies plus haut pour les transitions s'étendent naturellement aux traces. En particulier, une trace sera dite *positive* (resp. *négative*) si les transitions qui la composent sont toutes positives (resp. *négatives*). Une trace toujours positive ou toujours négative est dite *uniforme* et on désigne la trace vide par ϵ. Une transition *dérivable* de source r est une transition dont la preuve peut être déduite du LTS présenté Figure 2.1. Comme nous l'avons vu, les *identifiants* μ de telles transitions sont soit un couple de mémoires (dans le cas d'une synchronisation), soit une mémoire seule. Grâce à l'unicité des mémoires (section 2.1.3), on peut traiter ces identifiants comme des ensembles et nous les manipulerons à l'aide d'opérations ensemblistes.

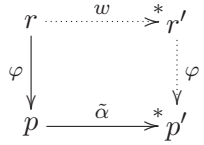
Lemme 2.2.1 (Réversibilité) *Pour toute transition positive,* $t : r \xrightarrow{\mu:\alpha} r'$ *il existe une transition négative* $t^- : r' \xrightarrow{\mu:\alpha^-} r$ *et réciproquement.*

Par induction sur le LTS. Il suffit de noter qu'une règle du LTS s'applique à des prémices si et seulement si la règle négative s'applique aux conclusions. □

Comme nous l'avons dit plus haut, les systèmes RCCS correspondent à des processus CCS enrichis de l'information nécessaire pour revenir en arrière. Il est donc normal qu'on puisse simuler le comportement de CCS avec des traces RCCS vers l'avant. Soit w un mot de la forme $(\mu_1 : \alpha_1) \ldots (\mu_n : \alpha_n)$, issu d'une trace RCCS. On définit l'extension de φ en posant $\varphi(w) := \alpha_1 \ldots \alpha_n$.

Lemme 2.2.2 (Simulation) *Un système RCCS peut* simuler *un processus*

CCS, i.e :



with $\varphi(w) = \tilde{\alpha}$.

LEMME 2.2.2 – Simulation.

On vérifie tout d'abord que $r \equiv s \Rightarrow \varphi(r) \equiv \varphi(s)$. C'est évident pour les règles standard concernant la composition parallèle et la restriction. On vérifie que $\varphi(m \triangleright p \parallel q) = p \parallel q$ qui est bien égal à $\varphi(\langle 1\rangle \cdot m \triangleright p \parallel \langle 2\rangle \cdot m \triangleright q)$. De même $\varphi(m \triangleright p\backslash x) = \varphi(m \triangleright p)\backslash x$. Enfin on conclut par une induction sur le LTS de RCCS, restreint aux règles de transitions positives, que $r \xrightarrow{\mu:\alpha} r'$ implique $\varphi(r) \xrightarrow{\alpha} \varphi(r')$ dans CCS ; il suffit pour cela de remarquer qu'en effaçant les mémoires on retrouve exactement le LTS de CCS. □

Proposition 2.2.1 (Correction) *Le LTS préserve la cohérence des systèmes.*

Par induction sur le LTS : l'application des axiomes ajoute ou enlève des cellules de la forme $\langle \star, \alpha, q\rangle$ des mémoires, ce qui préserve la relation de parenté entre les mémoires du système. De plus, si m est compatible avec le système et que $m \triangleright a.p + q$ se réduit en $\langle \star, a, q\rangle \cdot m \triangleright p$ par la règle (act), par application de la règle de contexte (res), a ne peut être restreint. Par conséquent $\langle \star, a, q\rangle \cdot m$ est aussi compatible. Pour l'axiome (act$^-$), si $\langle \star, a, q\rangle \cdot m$ est compatible, alors par définition m l'est aussi. Mis à part la règle (equiv), les règles de contexte ne modifient pas les piles mémoires et nous avons vu que la cohérence était préservée par congruence (Propriété 2.1.1). Enfin les règles de synchronisation ne font que modifier les mémoires avec l'opérateur @ qui préserve la relation de parenté. On doit donc vérifier que les mémoires restent compatibles. Dans le cas d'une application de la règle (synch), la transition est de la forme $r \parallel s \xrightarrow{m,m':\tau} r'_{m'@m} \parallel s'_{m@m'}$. Par hypothèse d'induction on a $\langle \star, \bar{a}, p\rangle \cdot m \uparrow r'$ et $\langle \star, a, q\rangle \cdot m' \uparrow s'$. On a donc, par définition, $m \uparrow r'$ et $m' \uparrow s'$. Par conséquent $\langle m', \bar{a}, p\rangle \cdot m \uparrow (r'_{m'@m} \parallel s'_{m'@m})$ car $\langle m, a, q\rangle \cdot m' \sqsubseteq (r'_{m'@m} \parallel s'_{m'@m})$ et donc $\langle m, a, q\rangle \cdot m' \uparrow (r'_{m'@m} \parallel s'_{m'@m})$. Le cas de la règle (synch$^-$) est traité de façon identique. □

Une conséquence de la proposition 2.2.1 est que tout système provenant d'un système racine est cohérent. La réciproque de ce lemme garantit que

tout système cohérent provient bien d'un système racine. Cela veut dire en particulier qu'un système cohérent, qui n'est pas une racine, peut *toujours* repartir en arrière.

Proposition 2.2.2 *Toute trace négative partant d'un système cohérent termine sur un unique système racine.*

Soit $r \equiv (m_1 \rhd p_1 \parallel \ldots \parallel m_n \rhd p_n)\backslash\tilde{x}$. Posons $\langle m_1, a, p\rangle \cdot m_2 \simeq \langle m_2, \bar{a}, q\rangle \cdot m_1$ pour tout m_1, m_2, a, p, q et $\langle 1\rangle \cdot m \simeq \langle 2\rangle \cdot m$ pour tout m. Soit l'ordre partiel strict $\prec = (\simeq, \sqsubset, \simeq)^+$; $\prec$ est bien fondé car $m \prec m'$ implique $|m| < |m'|$. Posons $M := \{m_i \sqsubseteq r \mid \forall m_j \sqsubseteq r : m_i \not\prec m_j\}$, l'ensemble des mémoires maximales pour la relation $\prec$ dans r. On montre le lemme par induction sur la taille de M. Si $M = \emptyset$ alors la seule possibilité est $r = \langle\rangle \rhd p$, pour un certain p, qui est un système racine. Soit donc $m_i \in M$ et étudions tous les cas possibles :
— $m_i = \langle m'_k, a\rangle \cdot m'_i$. Par cohérence de r, on a $\langle m'_i, a\rangle \cdot m'_k \preccurlyeq r$ et donc il existe $m_k \sqsubseteq r$ et m_0 telles que $m_k = m_0 \cdot \langle m'_i, \bar{a}\rangle \cdot m'_k$. Supposons que $m_0 \neq \langle\rangle$. Dans ce cas $\langle m'_i, a\rangle \cdot m'_k \sqsubset m_k$ et puisque $\langle m'_i, \bar{a}\rangle \cdot m'_k \simeq m_i$ on en déduit $m_i \prec m_k$ ce qui contredit l'hypothèse $m_i \in M$. En appliquant l'axiome (act^-), on a :

$$\langle \star, a\rangle \cdot m'_i \rhd p_i \xrightarrow{m'_i : a^-} m'_i \rhd a.p_i \qquad (1)$$
$$\langle \star, \bar{a}\rangle \cdot m'_k \rhd p_k \xrightarrow{m'_k : a} m'_k \rhd \bar{a}.p_k \qquad (2)$$

et par (synch^-) on a $r \xrightarrow{m'_i, m'_k : \tau^-} r'$ qui se réduit en un unique système racine par hypothèse d'induction.
— $m_i = \langle 1\rangle \cdot m'_i$. Par cohérence de r on sait que $\langle 2\rangle \cdot m'_i \preccurlyeq r$. Par conséquent il existe $m_k \sqsubseteq r$ et m_0 telles que $m_k = m_0 \cdot \langle 2\rangle \cdot m'_i$. En utilisant le même argument que précédemment, on a nécessairement que $m_0 = \langle\rangle$ et donc en appliquant la congruence (dist) on obtient $r \equiv r'$ qui se réduit en un unique système racine par hypothèse d'induction.
— $m_i = \langle \star, a\rangle \cdot m'_i$. On peut appliquer directement l'axiome (act^-) et $r \xrightarrow{m'_i : a^-} r'$ car $a \notin \tilde{x}$ par cohérence de r. On applique de nouveau l'hypothèse d'induction pour conclure que r' se réduit bien sur un unique système racine. □

2.3 Discussion.

Nous avons vu qu'il est possible de déterminer statiquement si un système RCCS correspond à une photographie possible d'un calcul partant d'un état initial donné. En effet, si un système r est cohérent, la proposition 2.2.2

indique l'existence d'une trace négative σ^- permettant de faire revenir r vers un unique état racine r_0. On déduit du lemme de réversibilité que le système r est la photographie de r_0 après la trace de calcul σ.

Par la suite nous supposerons que les systèmes considérés sont toujours cohérents. Toutefois, cette notion de cohérence syntaxique n'est pas suffisante pour conclure au bien fondé de notre système de retour arrière. Nous savons qu'un système reste toujours bien formé au cours d'un calcul (correction) et que tout système bien formé peut toujours repartir en arrière (complétude) jusqu'à sa racine, mais nous ne savons pas si les états intermédiaires de la trace arrière sont bien dans un passé "acceptable" du calcul. Plus formellement, prenons un système racine r et une trace positive $\sigma : r \rightarrow^* r'$. Que peut on dire du retour arrière correspondant à la trace négative $\gamma^- : r' \rightarrow^* r$? La correction du retour arrière doit être formulée de la manière suivante : le retour arrière est cohérent si γ est *équivalente* à σ. Comme nous l'avons déjà pointé, cette notion d'équivalence ne doit pas être réduite à l'égalité de traces, auquel cas notre retour arrière serait trop contraint ; on ne doit pas non plus mettre trop de traces en relation au risque d'obtenir des états intermédiaires incohérents. L'objet du prochain chapitre est d'établir cette notion d'équivalence qui caractérise le bien fondé du mécanisme retour arrière.

Chapitre 3

Causalité et retour arrière.

Sommaire

Comme nous l'avons argumenté au chapitre précédent, la notion de cohérence syntaxique (Définition 2.1.4) n'est pas suffisante pour assurer la correction du retour arrière ; nous avons besoin d'une formulation qui prenne en compte la notion de passés possibles d'un système. Prenons par exemple le processus simple $p = a.b.0$ et considérons le calcul $p \xrightarrow{a} b.0 \xrightarrow{b} 0$. Il est clair que $b.0$ est le seul passé cohérent du processus final 0 au regard du processus initial ; l'état $a.0$ ne devrait, en revanche, pas être accessible par un retour arrière depuis 0 : l'idée est que si a doit *précéder* b dans la trace positive, alors b^- doit précéder a^- dans la trace opposée. Si on prend maintenant le processus $q = a \parallel b$ et la trace $q \xrightarrow{a} 0 \parallel b \xrightarrow{b} 0 \parallel 0$, on doit pouvoir considérer que le passé $a \parallel 0$ est cohérent au même titre que $0 \parallel b$: intuitivement les actions sur a et sur b peuvent se faire de façon *concurrente* et on aurait pu tout aussi bien considérer la trace *équivalente* $q \xrightarrow{b} a \parallel 0 \xrightarrow{a} 0 \parallel 0$. Suivant cette intuition, nous allons montrer dans ce chapitre que le retour arrière de RCCS se fait bien suivant des chemins équivalents du point de vue de la causalité entre transitions. Une telle équivalence entre traces peut être obtenue en permutant les transitions concurrentes et en écrasant les

transitions successives de signes opposés. Une notion similaire d'équivalence de traces, caractérisée par Lévy[1], peut être définie dans le λ-calcul à l'aide d'étiquettes adaptées. Nous verrons en particulier qu'on peut interpréter le système de mémoire de RCCS comme un étiquetage pour CCS : nous montrons que deux traces co-initiales sont équivalentes si et seulement si elles ont le même ensemble de mémoires. La correction du retour arrière découlera de ce théorème fondamental. Afin de définir notre relation d'équivalence, nous caractérisons dans un premier temps la notion de transitions concurrentes et de transitions causales dans RCCS. Nous nous servirons ensuite de cette notion d'équivalence pour mettre en évidence la correction du retour arrière de RCCS. Nous présenterons, en fin de chapitre, une variante plus légère de la syntaxe de RCCS pour laquelle le retour arrière reste cohérent. Enfin, nous terminerons par une discussion sur l'expressivité de RCCS dans sa forme actuelle.

3.1 Équivalence par permutations.

3.1.1 Algèbres de processus et causalité.

Causalité. L'étude de la causalité dans les algèbres de processus est un secteur de recherche pour le moins développé. Ces études sont motivées par le besoin d'une sémantique de calcul concurrent plus discriminante, dite non entrelacée (*non interleaving semantics*)[2]. À l'inverse de modèles de concurrence tels que les structures d'événements ou les réseaux de Petri, pour lesquels la question de la causalité entre évènements peut être considérée de façon naturelle[3], les algèbres de processus comme CCS ou le π-calcul nécessitent une sémantique enrichie afin de rendre explicite la dépendance causale entre transitions. Cela peut se faire par l'ajout de localités permettant d'identifier les processus impliqués dans une communication[4] où en étiquetant les transitions avec une information caractéristique d'une section de leur preuve dans le LTS[5]. Si on ne considère que les traces positives de RCCS, on obtient une sémantique qui combine ces deux approches.

Localités et concurrence. Dire que des actions sont concurrentes correspond intuitivement à donner un critère prouvant que l'exécution de l'une

[1] [Lévy, 1978, Berry et Lévy, 1979]

[2] [Boudol et Castellani, 1989, Darondeau et Priami, 1989, Degano *et al.*, 1990, Boreale et Sangiorgi, 1998]

[3] [Winskel, 1982, Nielsen et Winskel, 1995, Bruni et Montanari, 2000]

[4] [Castellani, 2001]

[5] [Boudol et Castellani, 1989, Degano et Priami, 1992]

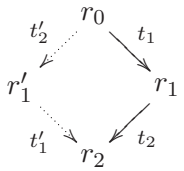
n'interférera pas dans l'exécution de la seconde. Dans les sémantiques de CCS avec localités[6], on formalise cette intuition en associant une adresse à chaque processus s'exécutant en parallèle ; on dit qu'une action entreprise par un processus p_1 est concurrente à celle d'un processus p_2 si les adresses utilisées n'ont pas de préfixe commun. Dans RCCS, les piles mémoires jouent précisément ce rôle de localités : elles dépendent d'une part de la structure initiale des compositions parallèles des processus simples (aspect statique) et d'autre part des futures évolutions du calcul (aspect dynamique). Les approches dynamiques et statiques ont été prouvées équivalentes par Castellani[7] et il n'est pas surprenant de trouver dans cette preuve une structure intermédiaire équipée de mémoires (*occurrence transition system*), qui correspond à la syntaxe de RCCS sans synchronisation. Nous allons à présent voir comment adapter les notions classiques de causalité étudiées dans les approches citées plus haut à RCCS.

3.1.2 Précédence et concurrence de traces.

On définit, tout d'abord, la relation de précédence formelle entre transitions en utilisant la structure des piles mémoires.

Définition 3.1.1 (Précédence) *Soient* $t : r \xrightarrow{\mu:\zeta} r'$ *et* $t' : r' \xrightarrow{\mu':\zeta'} r''$ *deux transitions consécutives. On dit que* t précède *(formellement)* t' *si :*
— *les transitions sont positives et* $\exists m \in \mu, \exists m' \in \mu' : m \sqsubset m'$.
— *les transitions sont négatives et* $\exists m \in \mu, \exists m' \in \mu' : m' \sqsubset m$.

Lemme 3.1.1 *Soient* $t_1 : r_0 \xrightarrow{\mu_1:\alpha_1} r_1$ *et* $t_2 : r_1 \xrightarrow{\mu_2} r_2$ *deux transitions successives de même signe. Si* t_1 *ne précède pas* t_2*, il existe un unique système* r_1' *(à congruence structurelle près) tel que les transitions* $t_2' : r_0 \xrightarrow{\mu_2:\alpha_2} r_1'$ *et* $t_1' : r_1' \xrightarrow{\mu_1:\alpha_1} r_2$ *soient composables.*



Lemme 3.1.1 – Diamant de concurrence.

[6] [Castellani, 2001]
[7] [Castellani, 1995]

La lemme se montre trivialement en notant que si t_1 ne précède pas t_2 alors les transitions agissent sur des ensembles disjoints μ_1 et μ_2 de mémoires. □

On dit que t_1 et t'_2 sont des transitions *concurrente* et que t_2 est le *résidu* de t'_2 après t_1 ; on notera alors $t_2 = (t'_2/t_1)$.

Définition 3.1.2 (Traces équivalentes) *L'équivalence par permutation* $\sim$ *est la plus petite relation d'équivalence satisfaisant :*

$$\begin{array}{lcl} t;(t'/t) & \sim & t';(t/t') \\ t;t^{-} & \sim & \epsilon \\ t^{-};t & \sim & \epsilon \end{array}$$

Comme nous l'avons dit en introduction de ce chapitre, la première règle de cette définition, considérée dans une étude ultérieure de Boudol et Castellani[8], est l'analogue pour CCS de l'équivalence "par permutation" de Berry et Lévy pour le λ-calcul.

3.2 Correction du retour arrière.

3.2.1 Préambule.

Avant de poursuivre avec la preuve de correction du retour arrière, nous souhaitons donner l'intuition que l'équivalence de trace définie plus haut est bien la notion qui permet d'exprimer la cohérence des traces négatives. Considérons l'exemple suivant :

Exemple 3.2.1 *On considère une machine à voter simplifiée :*

$$\mathsf{M}(v) := v.\mathsf{Val} \parallel \mathsf{M}(v)$$

qui attend un vote sur le canal v *avant de démarrer une procédure de validation* Val *susceptible d'échouer (non précisée ici) ; parallèlement, la machine reste en attente de nouveaux votes. Un électeur, numéroté par* i*, est un processus* $\mathsf{E}_i(v) := \bar{v}.\mathsf{Fin}$ *où l'état* Fin *correspond à la phase d'attente d'une éventuelle validation. Le système global est :*

$$\langle\rangle \triangleright \mathsf{E}_1(v) \parallel \ldots \parallel \mathsf{E}_n(v) \parallel \mathsf{M}(v)$$

Considérons à présent un calcul réversible plus grossier que RCCS dans lequel on enlève l'opérateur @ d'instanciation des mémoires dans les règles

[8] [Boudol et Castellani, 1989]

de synchronisation. Avec deux électeurs dans l'exemple 3.2.1 on obtiendrait la trace suivante :

$$\begin{array}{rcl} r & := & \langle 1\rangle \triangleright \mathsf{E}_1(v) \parallel (\langle 12\rangle \triangleright \mathsf{E}_2(v) \parallel \langle 22\rangle \mathsf{M}(v)) \\ & \xrightarrow{1,22:\tau} & \langle \star, \bar{v}\rangle\langle 1\rangle \triangleright \mathsf{Fin} \parallel \big(\langle 12\rangle \triangleright \mathsf{E}_2(v) \parallel (\langle \star, v\rangle\langle 122\rangle \triangleright \mathsf{Val} \parallel \langle 222\rangle \triangleright \mathsf{M}(v))\big) \end{array}$$

À cette étape, l'électeur E_1 a émis son vote et attend confirmation de la machine. Supposons que le deuxième électeur se mette à voter, on obtient :

$$\xrightarrow{12,222:\tau} \langle \star, \bar{v}\rangle\langle 1\rangle \triangleright \mathsf{Fin} \parallel \big(\langle \star, \bar{v}\rangle\langle 12\rangle \triangleright \mathsf{Fin} \parallel (\langle \star, v\rangle\langle 122\rangle \triangleright \mathsf{Val} \parallel \langle \star, v\rangle\langle 222\rangle \triangleright \mathsf{Val})\big)$$

Supposons maintenant que la procédure de validation du deuxième électeur échoue, c'est à dire que le processus d'adresse $\langle \star, v\rangle\langle 222\rangle$ décide d'initier un retour arrière. La transition suivante serait alors dérivable :

$$\xrightarrow{1,222:\tau^-} \langle 1\rangle \triangleright \mathsf{E}_1(v) \parallel \big(\langle \star, \bar{v}\rangle\langle 12\rangle \triangleright \mathsf{Fin} \parallel (\langle \star, v\rangle\langle 122\rangle \triangleright \mathsf{Val} \parallel \langle 222\rangle \triangleright \mathsf{M}(v))\big)$$

Cette transition arrière n'est pas cohérente dans le sens où nous avons autorisé l'échange des partenaires de communication. Si on pousse la trace arrière jusqu'au bout, en faisant échouer la validation du premier vote, on obtient une trace finale qui n'est pas équivalente (aux permutations près) à un retour arrière strict (en annulant les transitions dans l'ordre imposé par la trace positive). On pourrait arguer que les identifiants de votes devraient être spécifiques à chaque électeur, mais cela revient précisément à instancier les mémoires avec un identifiant permettant de retrouver de façon unique le partenaire d'une communication. Le retour arrière (local) d'un processus doit donc être accompagné du retour arrière (global) des processus du contexte ayant interagi le processus. Nous verrons au chapitre 4, que cette propriété s'avère cruciale pour caractériser les effets d'une action irréversible sur le reste du système.

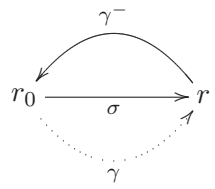
3.2.2 Théorème fondamental de RCCS.

Nous pouvons à présent énoncer la propriété fondamentale des systèmes RCCS, qui garantit la correction du retour arrière.

Théorème 3.2.1 *Deux traces RCCS co-initiales sont co-finales si et seulement si elles sont équivalentes.*

Comme annoncé dans le préambule à cette section, ce théorème entraîne la correction du retour arrière grâce au corollaire suivant :

Corollaire 3.2.1 *Soit une trace positive $\sigma : r_0 \to^* r$. Toute trace γ^- de source r converge vers r_0 et $\gamma \sim \sigma$.*



Corollaire 3.2.1 – Correction du retour arrière : $\gamma \sim \sigma$.

Le corollaire est une conséquence directe de la proposition 2.2.2 et du théorème 3.2.1. □

Nous poursuivons, à présent, par la preuve du théorème 3.2.1.

Lemme 3.2.1 *Deux traces co-initiales et équivalentes sont co-finales.*

Le lemme est une simple conséquence du lemme 3.1.1. □

Nous procédons maintenant à la caractérisation d'un certain nombre de propriétés permettant de déduire une preuve de la deuxième partie théorème 3.2.1.

Lemme 3.2.2 *Soient $t : r \xrightarrow{\mu:\zeta} s$ et $t' : s' \xrightarrow{\mu:\zeta^-} r'$ deux transitions. Pour toutes traces σ et γ, si $\sigma = t;\gamma;t'$ avec γ uniforme, alors $\sigma \sim \gamma$.*

Supposons t positive. Si γ est aussi positive alors le fait que t' soit dérivable après γ implique que les mémoires ajoutées par t restent inchangées par toutes les transitions de γ. On en déduit que t ne précède aucun t'' dans γ. Par le lemme 3.1.1, on peut permuter t avec γ et obtenir une trace $\sigma \sim \gamma; t^{-\prime}; t'$ qui est bien équivalente à γ. Si γ est négative on procède de la même manière en faisant permuter t' avec γ ; on obtient bien $\sigma \sim t; t^-; \gamma \sim \gamma$. Enfin si t est négative alors t' est positive et on procède au même raisonnement en commençant par t'. □

Soit M_σ l'ensemble des mémoires apparaissant dans la trace σ.

Lemme 3.2.3 (Forme polarisée) *Pour toute trace σ, il existe deux traces uniformes σ_0, σ_1 de signe opposés telles que $\sigma \sim \sigma_0; \sigma_1$ et $M_{\sigma_0} \cap M_{\sigma_1} = \emptyset$.*

– On montre tout d'abord qu'on peut mettre σ sous la forme $\sigma_0;\sigma_1$, par induction sur le nombre de paires de transitions successives $\langle t_0;t_1\rangle$ contredisant la polarité. Soit r le but de t_0. Soient μ_0 l'identifiant de t_0 et μ_1 celui de t_1. Si $\mu_0=\mu_1$, alors $t_0;t_1=t_1^-;t_1=\epsilon$ et on obtient une trace avec une paire de mauvaise polarité en moins ; on conclut par hypothèse d'induction. Si $\mu_0\neq\mu_1$ dans ce cas aucune mémoire de μ_0 ne peut être préfixe d'une mémoire de μ_1 et réciproquement car elles se succèdent immédiatement et sont de signe opposé. On peut donc les permuter par le lemme 3.1.1 et on obtient une paire de mauvaise polarité de moins. On conclut de nouveau par hypothèse d'induction.
– On montre à présent que toute trace polarisée $\sigma=\sigma_0;\sigma_1$, est équivalente à une trace $\gamma=\gamma_0;\gamma_1$ telle que $M_{\gamma_0}\cap M_{\gamma_1}=\emptyset$. On procède par induction sur le nombre d'identifiants se répétant dans les deux parties de signe opposés. Notons $\mu\sqsubset\mu'$ en place de : $\exists m\in\mu,\ \exists m'\in\mu' : m\sqsubset m'$. On prend t_0, d'identifiant μ_0, dans σ_0 et t_1, d'identifiant μ_0, dans σ_1, de sorte que pour toute transition d'identifiant μ apparaissant entre t_0 et t_1 on ait $\mu\not\sqsubset\mu_0$ & $\mu_0\not\sqsubset\mu$. On peut toujours choisir de tels t_0 et t_1 car toute trace polarisée de bilan nul pour une pile mémoire donnée contient forcément deux transitions opposées satisfaisant à ces pré-requis ; si la polarité est $+/-$ on prend pour t_0 la transition empilant le dernier élément de la pile et pour t_1 la transition qui dépile ce dernier élément. Pour une trace de polarité $-/+$ on prend les transitions qui dépilent et ré-empilent un élément sur la mémoire minimale. Appelons $\delta_0;\delta_1$ la trace (polarisée) entre t_0 et t_1 et appelons σ'_0 et σ'_1 les traces situées avant t_0 et t_1. Par construction, δ_0 permute avec t_0 et δ_1 avec t_1 en utilisant le lemme 3.1.1. On obtient $\sigma\sim\sigma'_0;\delta_0;\delta_1;\sigma'_1$, qui contient un identifiant de moins contredisant la propriété ; on peut donc conclure par hypothèse d'induction. □

Muni du lemme 3.2.3 nous pouvons maintenant prouver la partie manquante du théorème 3.2.1.

Lemme 3.2.4 *Deux traces co-initiales et co-finales sont équivalentes.*

Soit σ et γ deux traces co-initiales en r et co-finales en s. Nous montrons, par induction sur la longueur de σ et γ, qu'elles sont nécessairement équivalentes. En utilisant le lemme 3.2.3, on polarise dans le même sens σ et γ en (respectivement) $\sigma_0;\sigma_1$ et $\gamma_0;\gamma_1$. Il y a trois cas non symétriques à considérer (donnés figure 3.1). Ou bien $\sigma_0\neq\epsilon$ et $\gamma_0\neq\epsilon$ (1), ou bien $\sigma_0=\epsilon$ et $\gamma_0\neq\epsilon$ (2) ou bien $\sigma=\sigma_1$ et $\gamma=\gamma_1$ (3). Dans le cas (1), soient μ l'identifiant de t_0 et μ' celui de t'_0. Si les transitions sont égales, $r_0=r'_0$ et σ'_0 et γ'_0 sont co-initiales et co-finales. On conclut par hypothèse d'induction. Sinon sup-

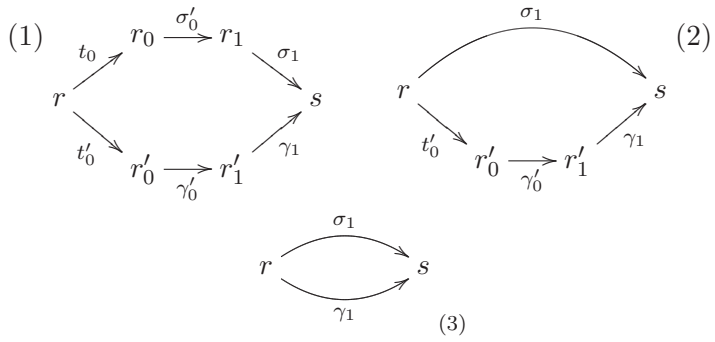


FIG. 3.1 – Les trois cas non symétriques, avec $t_0; \sigma'_0 = \sigma_0$ et $t'_0; \gamma'_0 = \gamma_0$.

posons t_0 et t'_0 positives. Par uniformité de $t_0; \sigma'_0$ et $t'_0; \gamma'_0$, les mémoires de μ et μ' apparaissent en préfixe strict de mémoires dans r_1 et r'_1. Comme σ et γ sont des traces polarisées cette propriété est aussi préservée dans s. On en conclut que t_0 et t'_0 ne peuvent être deux alternatives d'un choix différent sinon les cellules mémoires divergeraient dans leur troisième composant. Afin que les deux traces soient co-finales il faut, de plus, que les mémoires de μ apparaissent comme identifiants de transitions de γ'_0 et que celles de μ' apparaissent comme identifiants de transitions de σ'_0. Si une mémoire de μ apparaît dans γ'_0 alors elle est aussi dans un identifiant μ sans quoi les mémoires divergent en s dans le choix du partenaire. Il en va de même pour les mémoires de μ' apparaissant dans σ'_0. On a donc que μ est l'identifiant d'une transition t' de γ'_0 et μ' est l'identifiant d'une transition t de σ'_0. Appelons δ_0 la trace entre t_0 et t et δ'_0 la trace entre t'_0 et t. Par uniformité de δ_0 et δ'_0, t commute avec δ et t' avec δ' en utilisant le lemme 3.1.1. On obtient une trace équivalente à σ, commençant par $t_0; (t'_0/t_0)$ et de but s et une trace équivalente à γ, commençant par $t'_0; (t_0/t'_0)$ et de but s. On en déduit deux traces co-initiales et co-finales contenant chacune une transition de moins que σ et γ ; on conclut donc par hypothèse d'induction. On montre que le cas (2) est impossible en se servant de l'hypothèse de polarité. En tenant le même raisonnement que précédemment si t'_0 est positive alors les mémoires de μ' apparaissent comme préfixe strict de mémoires de r'_0. Comme ces mémoires ne peuvent être dépilées par γ_1 cette différence sera toujours visible dans s ce qui contredit la co-finalité. De même si t'_0 est négative alors le déficit sera toujours visible dans s et on obtient la même contradiction. Enfin, on montre le cas (3) de façon similaire au cas (1). Appelons t_1 d'identifiant μ_1 et t'_1, d'identifiant μ'_1, les premières transitions de σ_1 et γ_1. Elles ne peuvent

être conflictuelles pour les raisons expliquées plus haut. Si elles sont identiques on conclut par hypothèse d'induction ; si elles sont concurrentes, on en conclut que μ_1 est l'identifiant d'une transition t' dans γ_1' et que μ_1' est l'identifiant d'une transition t dans γ_1. On fait permuter t jusqu'à la rendre consécutive à t_1 et on procède de même pour t' qu'on rend consécutive à t_1'. On en déduit deux nouvelles traces, co-initiales et co-finales, plus petites que σ et γ et on conclut par hypothèse d'induction. □

3.2.3 Syntaxe basée sur des localités.

Dans le souci d'obtenir une preuve simple du théorème 3.2.1, ainsi qu'une syntaxe utilisant le moins d'objets possibles, nous avons fait le choix de travailler dans une sémantique rigide utilisant les mémoires à la fois pour stocker les informations nécessaires au retour arrière et à la fois comme identifiants de communication. L'idée est que la propriété d'unicité des mémoires permet de générer des étiquettes caractéristiques de chaque transition, sans avoir à introduire de nouveaux identifiants de communication. Il est toutefois clair que ce formalisme génère des étiquettes de taille arbitraire, ce qui rend peu aisée la lecture d'exemples. Des travaux récents de Phillips et Ulidowski[9] ont montré qu'il était possible de construire une sémantique réversible, pour une algèbre de processus de type CCS, en modifiant les règles d'application des opérateurs non-statiques du calcul (c'est à dire les opérateurs, comme le + de CCS, dont l'application entraîne une modification de la structure du terme). Afin de rendre réversibles les synchronisations, Phillips et Ulidowski introduisent des clefs de communication fraîches permettant de retrouver les processus partenaires lors d'un retour arrière. Dans RCCS les mémoires jouent elles-même le rôle de clef et, leur unicité étant garantie, il n'est pas nécessaire de formuler d'hypothèse de fraîcheur lorsqu'on les utilise pour identifier des processus communicants. Toutefois, afin d'obtenir un calcul plus léger, nous allons montrer qu'on peut définir une sémantique de RCCS (voir Figure 3.2) utilisant des clefs de communication abstraites générées dynamiquement. Soit Λ un ensemble dénombrable de localités. Dans la sémantique basées sur l'usage de localités, on considère l'axiome suivant :

$$m \triangleright \alpha.p + q \xrightarrow{l:\alpha} \langle l, \alpha, q \rangle \cdot m \triangleright p \text{ if } l \notin Loc(m) \tag{3.1}$$

où $Loc(m) \subseteq \Lambda$ constitue l'ensemble des localités apparaissant dans m. La condition $l \notin Loc(m)$ permet de s'assurer que la localité l n'a pas été

[9] [Phillips et Ulidowski, 2006]

utilisée par le processus au préalable. On s'assure aussi de l'unicité de l dans le contexte du processus, lors de l'application de la règle de contexte :

$$\frac{r \xrightarrow{l:\zeta} r' \quad l \notin Loc(s)}{r \parallel s \xrightarrow{l:\zeta} r' \parallel s} \tag{3.2}$$

où $Loc(s) \subseteq \Lambda$ constitue l'ensemble des localités apparaissant dans les mémoires de s. Cette syntaxe plus souple se situe dans la tradition des algèbres de processus avec localités dynamiques[10]. Par rapport à la sémantique ultérieure, elle à pour avantage principal de simplifier la règle de (dé)synchronisation :

$$\frac{r \xrightarrow{l:\bar{\zeta}} r' \quad s \xrightarrow{l:\zeta} s'}{r \parallel s \xrightarrow{l:\tau} r' \parallel s'} \tag{3.3}$$

Les deux processus r et s utilisent la même localité l pour leurs demi-actions, ce qui constitue une forme d'anticipation de la synchronisation à venir (*early semantics*), qui permet de se passer de l'opérateur d'instanciation @. En intégrant les modifications (3.1), (3.2) et (3.3) on obtient le LTS allégé de la figure 3.2.

$$(\mathsf{act})\ \frac{l \notin Loc(m)}{m \triangleright \alpha.p + q \xrightarrow{l:\alpha} \langle l, \alpha, q\rangle \cdot m \triangleright p} \qquad \frac{}{\langle l, \alpha, q\rangle \cdot m \triangleright p \xrightarrow{l:\alpha^-} m \triangleright \alpha.p + q}\ (\mathsf{act}^-)$$

$$(\mathsf{par})\ \frac{r \xrightarrow{l:\zeta} r' \quad l \notin Loc(s)}{r \parallel s \xrightarrow{l:\zeta} r' \parallel s} \qquad \frac{r \xrightarrow{l:\zeta} r' \quad s \xrightarrow{l:\bar{\zeta}} s'}{r \parallel s \xrightarrow{l:\tau} r' \parallel s'}\ (\mathsf{synch})$$

$$(\mathsf{res})\ \frac{r \xrightarrow{l:\zeta} r' \quad \zeta \neq x, \bar{x}, x^-, \bar{x}^-}{r\backslash x \xrightarrow{l:\zeta} r'\backslash x} \qquad \frac{r \equiv r' \xrightarrow{l:\zeta} s' \equiv s}{r \xrightarrow{l:\zeta} s}\ (\mathsf{equiv})$$

FIG. 3.2 – Système de transition à base de localités.

Pour clore cette section, nous allons montrer que la sémantique à base de localités est équivalente à la sémantique utilisant les mémoires comme identifiants. Soient R_{loc} l'ensemble des termes de la syntaxe légère et R_{mem} les termes utilisant les mémoires. On définit la famille d'applications M_r :

[10] [Boreale et Sangiorgi, 1998, Castellani, 2001]

$R_{loc} \to R_{mem}$, pour $r \in R_{loc}$, de la façon suivante :

$$\begin{array}{lcl} M_r(r' \parallel s) & = & M_r(r') \parallel M_r(s) \\ M_r(r'\backslash x) & = & M_r(r')\backslash x \\ M_r(m \triangleright p) & = & \mathcal{M}_r(m) \triangleright p \end{array}$$

où pour tout $r \in R_{loc}$, $\mathcal{M}_r$ est définie par :

$$\begin{array}{lcll} \mathcal{M}_r(\langle l, \alpha, p\rangle \cdot m) & = & \langle \mathcal{M}_r(m'), \alpha, p\rangle \cdot \mathcal{M}_r(m) & \text{si } \langle l, \bar{\alpha}, q\rangle \cdot m' \lessdot r \\ & = & \langle \star, \alpha, p\rangle \cdot \mathcal{M}_r(m) & \text{sinon.} \\ \mathcal{M}_r(\langle i\rangle \cdot m) & = & \langle i\rangle \cdot \mathcal{M}_r(m) & \\ \mathcal{M}_r(\langle\rangle) & = & \langle\rangle & \end{array}$$

De plus, pour toute transition $r \xrightarrow{l:\zeta} s$, avec $r, s \in R_{loc}$, on pose :

$$\mu_r(l) = \{\mathcal{M}_r(m) \mid \exists \alpha, p : \langle l, \alpha, p\rangle \cdot m \sqsubseteq r\}$$

L'application M_r satisfait la propriété suivante :

Propriété 3.2.1 *Pour tout $r, r' \in R_{loc}$, $r \equiv r' \Rightarrow M_r(r) \equiv M_{r'}(r')$.*

On vérifie aisément la propriété pour les règles de congruence structurelle d'associativité et de commutativité des opérateurs. Dans le cas où on applique la congruence $r = m \triangleright (p \parallel q) \equiv \langle 1\rangle \cdot m \triangleright p \parallel \langle 2\rangle \cdot m \triangleright q$, on a bien :

$$M_r(r) = \mathcal{M}_r(m) \triangleright (p \parallel q) \equiv \langle 1\rangle \cdot \mathcal{M}_r(m) \triangleright p \parallel \langle 2\rangle \cdot \mathcal{M}_r(m) \triangleright q \equiv M_{r'}(r')$$

Pour $r = m \triangleright (p\backslash x) \equiv (m \triangleright p)\backslash x$, la propriété se vérifie aussi simplement. □

On montre maintenant qu'on peut faire correspondre une transition dans R_{mem} à toute transition dans R_{loc} quelque son contexte d'évalutation :

Lemme 3.2.5 *Pour tout $r, s \in R_{loc}, l \in \Lambda$ et tout contexte $C[\cdot]$ tel que $l \notin Loc(C[\cdot])$ on a :*

$$\begin{array}{lcll} r \xrightarrow{l:\alpha} s & \Rightarrow & M_{C[r]}(r) \xrightarrow{\mu_{C[s]}(l):\alpha} M_{C[s]}(s) & (i) \\ r \xrightarrow{l:\alpha^-} s & \Rightarrow & M_{C[r]}(r) \xrightarrow{\mu_{C[r]}(l):\alpha^-} M_{C[s]}(s) & (ii) \end{array}$$

Par induction sur l'arbre de dérivation de $r \xrightarrow{l:\zeta} s$ dans le LTS de la figure 3.2. On ne traite que le cas (i), la preuve du cas (ii) étant symétrique.

– Supposons $\dfrac{}{r = m \triangleright \alpha.p + q \xrightarrow{l:\alpha} \langle l, \alpha, q\rangle \cdot m \triangleright p = s}$.

Par définition $M_{C[r]}(r) = \mathcal{M}_{C[r]}(m) \triangleright \alpha.p + q$. On applique la règle (act) du LTS de la figure 2.1 et on trouve $\mathcal{M}_{C[r]}(m) \triangleright \alpha.p + q \overset{\mathcal{M}_{C[r]}(m):\alpha}{\longrightarrow} \langle \star, \alpha, q \rangle \cdot \mathcal{M}_{C[r]}(m) \triangleright p$. Par ailleurs, $\mathcal{M}_{C[s]}(\langle l, \alpha, q \rangle \cdot m) \triangleright p = \langle \star, \alpha, q \rangle \cdot \mathcal{M}_{C[s]}(m) \triangleright p$ car $l \notin Loc(C[\cdot])$. Par construction $\mathcal{M}_{C[r]}(m) = \mathcal{M}_{C[s]}(m)$ et on a bien $M_{C[r]}(r) \overset{\mu_{C[s]}(l):\alpha}{\longrightarrow} M_{C[s]}(s)$.

– Supposons $\dfrac{r \xrightarrow{l:\alpha} r' \quad l \notin Loc(s)}{r \| s \xrightarrow{l:\alpha} r' \| s}$.

Pour tout contexte $C[\cdot]$, on pose $C_s[\cdot] = C[\,\cdot \parallel s]$. On vérifie que :

$$M_{C[r\|s]}(r \parallel s) = M_{C_s[r]}(r) \parallel M_{C_s[r]}(s)$$

Dans le cas où $\zeta = \alpha$, par hypothèse d'induction on a :

$$M_{C_s[r]}(r) \overset{\mu C_s[r'](l):\alpha}{\longrightarrow} M_{C_s[r']}(r')$$

On notera qu'on peut effectivement appliquer l'hypothèse d'induction grâce à la condition $l \notin Loc(s)$ et à l'hypothèse $l \notin C[\cdot]$. Par application de la règle (par) du LTS de la figure 2.1 on en déduit :

$$M_{C[r\|s]}(r \parallel s) \overset{\mu_{C_s[r']}(l):\alpha}{\longrightarrow} M_{C_s[r']}(r') \parallel M_{C_s[r]}(s)$$

On note que $Loc(r) = Loc(r') \uplus \{l\}$ et que $l \notin Loc(s)$; par conséquent on a $M_{C_s[r]}(s) = M_{C_s[r']}(s)$:

$$M_{C[r\|s]}(r \parallel s) \overset{\mu_{C[r'\|s]}(l):\alpha}{\longrightarrow} M_{C[r'\|s]}(r' \parallel s)$$

– Supposons $\dfrac{r \xrightarrow{l:\bar{\alpha}} r' \quad s \xrightarrow{l:\alpha} s'}{r \| s \xrightarrow{l:\tau} r' \| s'}$.

En appliquant l'hypothèse d'induction sur les prémices de la règle, on obtient :

$$\begin{array}{lll} M_{C_s[r]}(r) & \overset{\mu_{C_s[r']}(l):\bar{\alpha}}{\longrightarrow} & M_{C_s[r']}(r') \\ M_{C_r[s]}(s) & \overset{\mu_{C_r[s']}(l):\alpha}{\longrightarrow} & M_{C_r[s']}(s') \end{array}$$

Posons $m = \mu_{C_s[r']}(l)$ et $m' = \mu_{C_r[s']}(l)$. Sachant que $l \notin Loc(s) \cup Loc(r) \cup Loc(C[\cdot])$ on a aussi $m = \mu_{r'}(l)$ et $m' = \mu_{s'}(l)$. On peut appliquer la règle de synchronisation du LTS de la figure 2.1 pour obtenir :

$$M_{C[r\|s]}(r \parallel s) \overset{m,m':\tau}{\longrightarrow} \left(M_{C_s[r']}(r')\right)_{m'@m} \parallel \left(M_{C_r[s']}(s')\right)_{m@m'}$$

On doit donc vérifier :

$$\left(M_{C[r'\|s]}(r')\right)_{m'@m} \parallel \left(M_{C[r\|s']}(s')\right)_{m@m'} = M_{C[r'\|s']}(r' \parallel s')$$

On se contente de montrer $\left(M_{C[r'\|s](r')}\right)_{m'@m} = M_{C[r'\|s'](r')}$, l'autre égalité $\left(M_{C[r\|s']}(s')\right)_{m@m'} = M_{C[r'\|s']}(s')$ se montrant de manière symétrique. La preuve se fait par induction sur la structure de r'. Dans le cas de base, on a $\left(M_{C[r'\|s]}(m_0 \triangleright p)\right)_{m'@m}$. Supposons $m_0 = \langle l, \alpha, q\rangle \cdot m_1$, puisque l n'apparaît qu'une seule fois dans r' et que $l \notin Loc(s) \cup Loc(C[\cdot])$ on a :

$$\left(M_{C[r'\|s]}(m_0 \triangleright p)\right)_{m'@m} = \left(\langle \star, \alpha, q\rangle \cdot \mathcal{M}_{C[r'\|s]}(m_1) \triangleright p\right)_{m'@m}$$

Par définition, $\mathcal{M}_{C[r'\|s]}(m_1) = m$ et on obtient :

$$\left(\langle \star, \alpha, q\rangle \cdot \mathcal{M}_{C[r'\|s]}(m_1) \triangleright p\right)_{m'@m} = \langle m', \alpha, q\rangle \cdot m \triangleright p = M_{C[r'\|s']}(r')$$

Toujours dans le cas de base, supposons $l \notin Loc(m_0)$. Dans ce cas on a :

$$\left(M_{C[r'\|s]}(m_0 \triangleright p)\right)_{m'@m} = M_{C[r'\|s]}(m_0 \triangleright p)$$

Comme $Loc(s) \uplus \{l\} = Loc(s')$ on trouve $M_{C[r'\|s]}(m_0 \triangleright p) = M_{C[r'\|s']}(m_0 \triangleright p)$. L'étape inductive se fait ensuite aisément : $M_{C[r'\|s]}(r_1 \parallel r_2) = M_{C[r'\|s]}(r_1) \parallel M_{C[r'\|s]}(r_2)$ et par hypothèse d'induction on a $M_{C[r'\|s]}(r_1) = M_{C[r'\|s']}(r_1)$ et $M_{C[r'\|s]}(r_2) = M_{C[r'\|s']}(r_2)$. Par conséquent $M_{C[r'\|s]}(r_1 \parallel r_2) = M_{C[r'\|s']}(r_1 \parallel r_2)$. On procède de même pour $r = r_1\backslash x$.

– Supposons $\dfrac{r \xrightarrow{l:\alpha} r'}{r\backslash x \xrightarrow{l:\alpha} r'\backslash x}$.

Par hypothèse d'induction $M_{C[r]}(r) \xrightarrow{l:\alpha} M_{C[r']}(r')$ et on déduit du LTS de la figure 2.1 que $M_{C[r]}(r)\backslash x \xrightarrow{l:\alpha} M_{C[r']}(r')\backslash x$.

– Enfin, le cas $\dfrac{r \equiv r' \xrightarrow{l:\alpha} s' \equiv s}{r \xrightarrow{l:\alpha} s}$ se traite en utilisant l'hypothèse d'induction et la propriété 3.2.1. □

Nous allons procéder de la même manière pour prouver qu'on peut faire correspondre à toute transition de R_{mem} une transition de R_{loc}. Soit donc l'application $L : R_{mem} \to R_{loc}$ définie par :

$$\begin{array}{lcl} L(r \parallel s) & = & L(r) \parallel L(s) \\ L(r\backslash x) & = & L(r)\backslash x \\ L(m \triangleright p) & = & \mathcal{L}(m) \triangleright p \end{array}$$

avec :

$$\begin{array}{lcl}\mathcal{L}(\langle\star,\alpha,q\rangle\cdot m) & = & \langle\lambda(\{m\}),\alpha,q\rangle\cdot\mathcal{L}(m)\\ \mathcal{L}(\langle m',al,q\rangle\cdot m) & = & \langle\lambda(\{m,m'\}),\alpha,q\rangle\cdot\mathcal{L}(m)\\ \mathcal{L}(\langle i\rangle\cdot m) & = & \langle i\rangle\cdot\mathcal{L}(m)\\ \mathcal{L}(\langle\rangle) & = & \langle\rangle\end{array}$$

où $\lambda(\mu)$ est un application injective qui envoie un ensemble de mémoires μ de R_{mem} vers une localité.

Lemme 3.2.6 *Pour tout $r, s \in R_{mem}$ on a :*

$$r \xrightarrow{\mu:\zeta} s \Rightarrow L(r) \xrightarrow{\lambda(\mu):\alpha} L(s)$$

On notera qu'il n'est pas besoin ici de mentionner le contexte dans lequel est plongé la transition de R_{loc} car l'application L peut renommer les termes de R_{mem} en utilisant exclusivement les informations stockées localement dans les mémoires. La preuve se fait par induction sur l'arbre de dérivation de $r \xrightarrow{\mu:\zeta} s$ dans le LTS donné Figure 2.1. Comme précédemment on se contentera de traiter les transitions positives, les preuves concernant les règles arrières se faisant de manière symétrique.

– Supposons $\frac{}{m\triangleright\alpha.p+q\xrightarrow{m:\alpha}\langle\star,\alpha,q\rangle\cdot m\triangleright p}$.

On a $L(m\triangleright\alpha.p+q) = \mathcal{L}(m)\triangleright\alpha.p+q$ qui se réduit, dans le LTS de la figure 3.2, en $\langle l,\alpha,q\rangle\cdot\mathcal{L}(m)\triangleright p$; comme $l\notin m$ on peut alors poser $\lambda(m)=l$.

– Supposons $\frac{r\xrightarrow{\mu:\alpha}r'}{r\|s\xrightarrow{\mu:\alpha}r'\|s}$.

Par hypothèse d'induction, $L(r) \xrightarrow{\lambda(\mu):\alpha} L(r')$. Par cohérence des termes r et s, les mémoires de μ sont uniques donc $\lambda(m)\notin Loc(s)$. On peut donc appliquer la règle (par) du LTS de la figure 3.2 pour conclure.

– Supposons $\frac{r\xrightarrow{m:\bar{a}}r' \quad s\xrightarrow{m':a}s'}{r\|s\xrightarrow{m,m':\tau}r'_{m'@m}\|s'_{m@m'}}$.

Par hypothèse d'induction, $L(r) \xrightarrow{\lambda(m):\bar{\alpha}} L(r')$ et $L(s) \xrightarrow{\lambda(m'):\alpha} L(s')$. Or $\lambda(m)\notin Loc(L(r))$ et $\lambda(m')\notin Loc(L(r))$ par unicité de m'. En faisant un autre choix de localité on obtient $L(r) \xrightarrow{\lambda(m,m'):\bar{\alpha}} L(r')\{\lambda(m,m')/\lambda(m)\}$. Par le même raisonnement on trouve $L(s) \xrightarrow{\lambda(m,m'):\alpha} L(s')\{\lambda(m,m')/\lambda(m')\}$. On peut alors appliquer la règle de synchronisation du LTS de la figure 3.2 pour obtenir :

$$L(r\parallel s) \xrightarrow{\lambda(m,m'):\tau} L(r')\{\lambda(m,m')/\lambda(m)\}\parallel L(s')\{\lambda(m,m')/\lambda(m')\}$$

On montre $L(r')\{\lambda(m,m')/\lambda(m)\} = L(r'_{m'@m})$ par induction sur la structure de r' (le cas de $L(s')\{\lambda(m,m')/\lambda(m)\}$ est symétrique). Si $m \notin r'$ alors on a $L(r')\{\lambda(m,m')/\lambda(m)\} = L(r'_{m'@m}) = L(r')$, en revanche si $r' = \langle\star,\alpha,q\rangle \cdot m \triangleright p$, on a $L(r') = \langle\lambda(m),\alpha,q\rangle \cdot \mathcal{L}(m) \triangleright p$. Par conséquent $L(r')\{\lambda(m,m')/\lambda(m)\} = \langle\lambda(m,m'),\alpha,q\rangle \cdot \mathcal{L}(m) \triangleright p$ qui est bien égal à $L(r'_{m'@m})$. Supposons $r' = r_1 \parallel r_2$, dans ce cas $L(r_1 \parallel r_2) = L(r_1) \parallel L(r_2)$ et on conclut par hypothèse d'induction sur r_1 et r_2. Si $r' = r_1\backslash x$ on applique l'hypothèse d'induction sur r_1.

– Supposons $\frac{r \xrightarrow{\mu:\alpha} r'}{r\backslash x \xrightarrow{\mu:\alpha} r'\backslash x}$.

Par hypothèse d'induction $L(r) \xrightarrow{\lambda(\mu):\alpha} L(s)$. Comme $\alpha \notin \{x, \bar{x}\}$ on peut appliquer la règle (res) du LTS de la figure 3.2 et on a bien $L(r)\backslash x \xrightarrow{\lambda(\mu):\alpha} L(s)\backslash x$.

– Enfin, supposons $\frac{r \equiv r' \xrightarrow{\mu:\alpha} s' \equiv s}{r \xrightarrow{\mu:\alpha} s}$.

Par hypothèse d'induction, $L(r') \xrightarrow{\lambda(\mu):\alpha} L(s')$ et on conclut en utilisant le symétrique de la propriété 3.2.1. □

Théorème 3.2.2 (Équivalence des sémantiques.)

$$\begin{array}{lcl} \forall r,s \in R_{loc} : r \xrightarrow{l:\alpha} s & \Rightarrow & M_r(r) \xrightarrow{\mu_s(l):\alpha} M_s(s) \\ \forall r,s \in R_{loc} : r \xrightarrow{l:\alpha^-} s & \Rightarrow & M_r(r) \xrightarrow{\mu_r(l):\alpha^-} M_s(s) \\ \forall r,s \in R_{mem} : r \xrightarrow{\mu:\zeta} s & \Rightarrow & L(r) \xrightarrow{\lambda(\mu):\alpha} L(s) \end{array}$$

On prouve le théorème 3.2.2 en utilisant les lemmes 3.2.5 et 3.2.6. □

3.3 Discussion.

Au chapitre précédent, nous avons introduit une décoration des processus CCS permettant de définir une sémantique opérationnelle réversible. Nous pouvons à présent exprimer la cohérence du retour arrière qui garantit l'équivalence de toutes les traces négatives issues d'un même processus. Toutefois, nous n'avons pas défini d'équivalence comportementale associée à RCCS. La motivation originelle pour construire une algèbre de processus réversible est le besoin d'un retour arrière générique dans de nombreux processus transactionnels. Dans l'idée de Milner, il nous faudrait donc une

notion de bisimulation permettant de dire si un processus réversible implémentant une transaction donnée est bien conforme à sa spécification. Il serait possible de définir une notion de bisimulation en utilisant comme observations les étiquettes fournies par RCCS, mais on obtiendrait une équivalence qui dépendrait étroitement de la syntaxe choisie, ce qui contredirait l'idée que les mémoires de RCCS ne sont qu'un moyen d'implémenter le retour arrière et non pas des objets observables en eux-même. Il est par ailleurs clair que les bisimulations obtenues seraient plus discriminantes que la bisimulation standard de CCS. Plusieurs travaux cherchant à évaluer le potentiel discriminant de bisimulations enrichies avec des informations de causalité ou de localité ont été menés à ce sujet[11]. On notera, en particulier, les notions de bisimulation avant-arrière (*back and forth bisimulation*)[12] ou préservant l'histoire (*history preserving bisimulation*)[13] qui semblent correspondre à ce qu'on pourrait obtenir avec les observables et la sémantique de RCCS. Nous avons fait le choix d'une approche différente dans laquelle nous voulons garder la notion classique de bisimulation de Milner qui nous paraît correspondre à la propriété naturelle qu'on souhaite connaître pour définir ce que fait réellement un programme concurrent quelque soit la manière dont il est implémenté.

Pour parler des interactions d'un système RCCS avec son environnement il est nécessaire d'introduire une notion de stabilité. En effet, on ne peut pas simplement considérer que le système $\langle\rangle \triangleright a.0$ interagit sur le canal a car cette interaction peut-être à tout moment annulée. Dans le prochain chapitre nous verrons comment définir des interactions stables en ajoutant au calcul la possibilité d'effectuer des actions irréversibles. Nous serons alors à même de définir une notion d'équivalence comportementale pour les systèmes RCCS en restreignant les observations aux seules actions irréversibles.

[11] [Galpin, 2000, Castellani, 2001]
[12] [Nicola *et al.*, 1990]
[13] [Bednarczyk, 1991, Montanari et Pistore, 1997]

Chapitre 4

Systèmes transactionnels.

Sommaire

Au chapitre 1 nous avons vu comment définir la correction d'un processus CCS en fonction d'un LTS de référence. L'idée formalisée par Milner est que le processus CCS correspondant à l'implémentation d'un problème doit être indistingable de sa spécification pour un ensemble d'observations donné. Pour les systèmes RCCS, le problème est donc d'abord de situer ce qu'on est capable d'observer d'une trace de calcul. Comme nous l'avons brièvement suggéré en fin de chapitre précédent, dans l'algèbre réversible que nous avons pour le moment définie, il n'est pas possible de parler d'observations stables, attendu que toutes les interactions d'un système sont potentiellement réversibles. Or une transaction est basée sur l'idée qu'une séquence d'actions précaires, peut être *in fine* validée dès que le consensus est atteint. Pour formaliser des processus transactionnels avec RCCS et répondre à la motivation originelle, nous introduisons dans ce chapitre les *occurrences d'actions irréversibles*. L'idée directrice étant que les transitions classiques d'un système RCCS correspondent à la partie recherche du consensus, tandis que les transitions irréversibles servent à valider les changements d'états obtenus après consensus. Nous introduisons, dans un premier temps les quelques changements syntaxiques et opérationnels nécessaires à la mise en place des

actions irréversibles. Nous verrons ensuite comment formaliser le concept de transaction dans ce formalisme, puis nous énoncerons un théorème transactionnel permettant de définir le comportement observable et pérenne d'un système en fonction du *système de transitions causales* de son processus simple[1]. Nous verrons en conclusion, qu'en se servant de ce théorème on peut définir un mode de programmation pour les systèmes distribués, dit déclaratif, qui simplifie à la fois le travail du programmeur et la vérification des programmes engendrés. Nous terminerons par une brève comparaison avec les approches existantes.

4.1 Validations dans RCCS.

Syntaxe. On considère à présent des processus simples dans lesquels on marque certaines capacités d'actions. Ces dernières sont simplement soulignées dans le processus et seront interprétées comme étant irréversibles dans la sémantique de RCCS. Il s'agit là de la seule interaction entre le programmeur et le mécanisme de retour arrière : préciser quelles actions du processus marquent la conclusion positive de la recherche d'un consensus. On note $p \in P_m$ un processus contenant des capacités marquées comme dans :

$$p := \big(t.0 \parallel \bar{t}.\underline{a} \parallel \bar{t}.\underline{b}\big)\backslash t$$

où les actions entreprises sur a ou b sont déclarées irréversibles. Noter qu'on peut aussi rendre irréversibles des actions internes. On pourra, par exemple, simuler le processus $(\underline{\bar{x}}.p \parallel \underline{x}.q)\backslash x$ par $(\bar{x}.p \parallel x.\underline{\tau}.q)\backslash x$. Ces processus sont en fait équivalents comme nous le verrons par la suite.

Mis à part ces capacités marquées, la syntaxe des systèmes RCCS reste similaire à celle donnée Section 2.1 avec l'ajout d'un type de mémoire particulier $\langle\!|\alpha|\!\rangle \cdot m$, indiquant que la pile mémoire m à été validée par l'action $\underline{\alpha}$; la mémoire sera alors dite *fermée*. La syntaxe modifiée est :

$$\begin{array}{lcll} r, s & ::= & (r \parallel s) \mid r\backslash x & \\ & \mid & m \triangleright p \text{ avec } p \in P_m & \text{Processus} \\ & & & \\ m, m' & ::= & \langle \star, \alpha, p\rangle \cdot m \mid \langle m', a, p\rangle \cdot m \mid \langle 1\rangle \cdot m \mid \langle 2\rangle \cdot m \mid \langle\rangle & \\ & \mid & \langle\!|\alpha|\!\rangle \cdot m & \text{Pile fermée} \end{array}$$

Sémantique opérationnelle. Nous ajoutons à la sémantique opérationnelle donnée figure 2.1 une règle de validation :

$$m \triangleright \underline{\alpha}.p + q \xrightarrow{m:\underline{\alpha}} \langle\!|\alpha|\!\rangle \cdot m \triangleright p \qquad (\mathsf{commit})$$

[1] [Danos et Krivine, 2005]

Noter que nous n'effaçons pas la mémoire fermée par l'action $\underline{\alpha}$, dans un souci de maintenir la cohérence des mémoires. Par ailleurs, une action de validation peut avoir pour effet de bord de clore d'autres piles mémoires. Une opération de ramasse-miettes sur les mémoires fermées ne peut donc se faire uniquement localement[2]. On définit la règle de synchronisation pour les actions de validation complémentaires :

$$\frac{r \xrightarrow{m:\underline{\alpha}} r' \quad s \xrightarrow{m':\underline{\bar{\alpha}}} s'}{r \parallel s \xrightarrow{m,m':\underline{\tau}} r' \parallel s'} \qquad (\mathsf{com-synch})$$

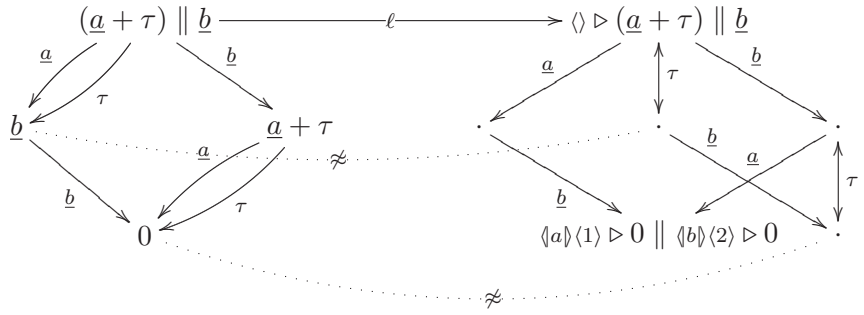
Les transitions de RCCS avec validations sont donc de la forme $t = \langle r, \mu, \zeta, r' \rangle$ avec $\zeta := \alpha \mid \underline{\alpha} \mid \alpha^-$. Le LTS partiellement réversible, est obtenu en ajoutant les règles de validations (commit) et (com-synch) au LTS de la figure 2.1. On utilise A pour désigner l'ensemble des actions positives étiquetant les transitions de ce LTS, A^- pour l'ensemble des actions négatives et $K \subseteq A$ pour celui des actions irréversibles.

Équivalence. On dispose à présent d'un calcul dans lequel des transitions irréversibles se mêlent à des retours arrière. Comme nous l'avons vu en introduction de ce chapitre, une bonne notion d'équivalence se doit de mettre en relation des systèmes RCCS avec leurs correspondants stables. On devrait ainsi pouvoir dire que le système $\langle\rangle \triangleright \underline{a} + b$ est équivalent au processus simple $\underline{a}.0$, l'action sur b étant toujours précaire. Dans cet esprit, seules les actions de validations seront observables et on applique la définition 1.2.1 en prenant K comme ensemble d'observables (et en oubliant l'information donnée par les mémoires dans les transitions RCCS). Étant donné un processus p et un ensemble de validations K, on cherche donc à comprendre quel processus stable (*i.e* unidirectionnel) est bisimilaire à $\ell(p)$. Clairement, si $K = A$ le système obtenu est bisimilaire à p : puisque toute action est irréversible, RCCS se comporte exactement comme CCS. À l'opposé, lorsque $K = \emptyset$ toutes les actions sont réversibles et le système ne progresse jamais ; par conséquent, $\ell(p) \approx 0$. La question clef est donc de caractériser le comportement de $\ell(p)$ lorsque K n'est ni A ni $\emptyset$. Nous répondrons à cette question dans la section suivante ; en attendant, nous pouvons d'ores et déjà écarter une mauvaise intuition, à savoir que se restreindre à l'observation des actions irréversibles dans K, ne suffit pas pour avoir $p \approx \ell(p)$.

Exemple 4.1.1 *Soit $p := (\underline{a} + \tau) \parallel \underline{b}$, le diagramme suivant montre que $p \not\approx \ell(p)$. Il est, en effet, possible de masquer des validations par des actions*

[2]Nous montrons un méthode pour définir un tel ramasse-miettes dans le chapitre de conclusion du mémoire.

réversibles. Ces choix apparaissent comme définitifs dans le LTS, mais sont annulables dans RCCS.



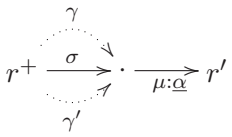
EXEMPLE 4.1.1 – Les systèmes ne sont pas bisimilaires : les choix non validés sont réversibles.

4.2 Équivalence comportementale.

4.2.1 Transactions.

Un système r est dit *initial* s'il n'est source d'aucune transition arrière dérivable ; en particulier, tout système racine est initial et on note r^+ si r est un système initial. Une transition $t = \langle r, \mu, \zeta, r' \rangle$ est une *validation* si $\zeta := \underline{\alpha}$ pour un α donné ; noter que r' n'est pas *a priori* initial. Une trace ne contenant pas de validation est dite *silencieuse*. On définit une trace *transactionnelle*, ou simplement une *transaction*, de la façon suivante :

Définition 4.2.1 *Soit $\sigma; t$ une trace positive avec σ une trace silencieuse t une validation. On dit que $\sigma; t$ est une* transaction (pure) *si et seulement si toute trace équivalente est de la forme $\gamma; t$ où γ est une trace positive et silencieuse.*



DEFINITION 4.2.1 – Trace transactionnelle.

Noter que la définition de transaction ne concerne que les traces vers l'avant, elle s'adapte donc facilement à CCS en oubliant la condition d'initialité du processus source de la trace. Une transaction est une trace terminée par une validation ne permutant avec aucune transition silencieuse la précédant. En d'autres termes toutes les transitions silencieuses (et réversibles) de la transaction sont *nécessaires* à l'obtention de la validation finale ; pour cette

raison on peut dire qu'une transaction est une trace *causale* ou *minimale* pour une validation donnée. Dans une transaction $\sigma; t$, la trace σ correspond à la phase de recherche du consensus et t à sa validation. Dans RCCS, on peut montrer que le but d'une transaction est nécessairement stable :

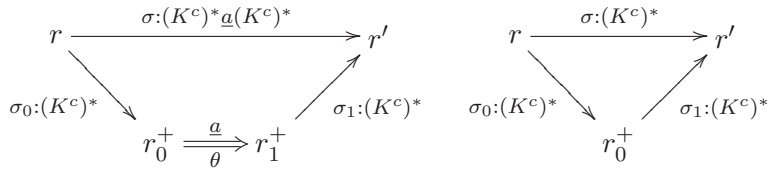
Lemme 4.2.1 *Une transaction est une trace irréversible, i.e de but initial.*

Supposons que σ soit une transaction de source r_0 et de but r non initial. Dans ce cas il existe une transition négative t^- de source r et, par réversibilité, on a $\sigma \sim \sigma; t^-; t$. Les mémoires dépilées par t^- ont été empilées durant la trace σ car r_0 est initial. On note que les piles mémoires de t^- sont nécessairement créées par une même transition t' de σ sans quoi les systèmes ne seraient pas cohérents. La transition σ est donc de la forme $\sigma_0; t'; \sigma_1$, avec t' utilisant les mêmes piles mémoires que t^-. En appliquant le lemme 3.2.2 on obtient $\sigma \sim \sigma_0; t'; \sigma_1; t^-; t \sim \sigma_0; \sigma_1; t$ ce qui contredit le fait que σ soit une transaction. □

Une transaction est donc une trace de la forme $r_0^+ \xrightarrow{(K^c)^*} \xrightarrow{\mu:\underline{\alpha}} r_1^+$ et on utilisera la double flèche $r \stackrel{\underline{\alpha}}{\Longrightarrow} r'$ pour désigner la transaction de source r et de but r', validée par $\underline{\alpha}$. Intuitivement les transactions sont des traces qui auraient une priorité absolue sur l'ordonnanceur : elles ne s'entrelacent avec aucune autre transaction concurrente (ce qui correspond au fait que toutes les actions réversibles sont nécessaires à l'obtention de la validation finale) et sont donc *atomiques*. On définit une *transaction bruitée* simplement comme une trace possédant une unique validation. Les traces de ce type contiennent la phase de validation d'une transaction (responsable de l'observable) dont les transitions se mélangent avec un "bruit" correspondant à des retours arrière et des fragments non validés de transactions concurrentes. La propriété suivante indique qu'on peut toujours lisser une transaction bruitée afin d'extraire la transaction bénéficiant de la validation observée :

Lemme 4.2.2 (Factorisation) *Toute trace σ, contenant au plus une validation, se factorise en une trace équivalente de la forme $\sigma_0; \theta; \sigma_1$ telle que :*
— *σ_0 est négative et de but initial,*
— *θ est une transaction si σ est une transaction bruitée et $\theta = \epsilon$ sinon.*
— *σ_1 positive et de source initiale.*

— *Toute autre factorisation* $\delta_0; \theta'; \delta_1$ *est telle que* $\sigma_i \sim \delta_i$ *et* $\theta \sim \theta'$.



Lemme 4.2.2 – Décomposition de traces RCCS.

La section suivante est dédiée à la mise en place des propriétés permettant d'établir le lemme de factorisation. On pourra toutefois passer directement à la section 4.2.3 dans laquelle nous verrons comment utiliser ce lemme afin de caractériser le comportement de $\ell(p)$ en fonction de celui de p.

4.2.2 Factorisation de traces.

Du fait des validations, on perd la propriété de complétude dans le sens où un retour arrière peut être bloqué par une validation qui empêche de revenir dans un système racine. Toutefois on peut montrer que les traces négatives et co-initiales convergent vers un même système initial :

Lemme 4.2.3 *Le retour arrière est nœtherien et confluent.*

Pour tout système r donné, les mémoires $m \sqsubseteq r$ sont de tailles bornées. Comme chaque transition négative dépile au moins une cellule mémoire, on peut déjà conclure quant à la finitude des traces négatives. Si deux transitions négatives sont co-initiales alors elles sont soit égales, soit elles dépilent des ensembles de mémoires μ et μ'. Du fait de la cohérence des systèmes $\mu \cap \mu' = \emptyset$ et dans ce cas les transitions sont concurrentes ; on peut donc les faire permuter pour obtenir une trace confluente. La confluence locale et la terminaison constituent alors une preuve du lemme. □

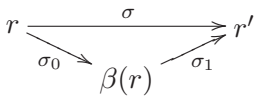
On déduit du lemme ci-dessus qu'on peut associer à chaque système cohérent un unique système initial. Soit donc la fonction $\beta : R \to R$ qui associe à un système r le terme initial $\beta(r)$, obtenu après un retour arrière maximal.

Lemme 4.2.4 *Toute trace* σ *est équivalente à une trace polarisée* $\sigma_0; \sigma_1$ *telle que* σ_0 *est négative et de but initial.*

On met tout d'abord σ en forme polarisée $\gamma_0; \gamma_1$, avec γ_0 négative, en utilisant le lemme 3.2.3. Si le but de γ_0 est initial alors nous avons fini. Sinon

appelons r le but de γ_0. Par réversibilité, la trace $r_0 \xrightarrow{\delta_0^-} \beta(r_0) \xrightarrow{\delta_0} r_0$ est équivalente à ϵ. On obtient donc une trace $\gamma_0; \delta_0^-; \delta_0; \gamma_1$, équivalente à σ, avec $\gamma_0; \delta_0^-$ négative et de but initial et $\delta_0; \gamma_1$ positive. □

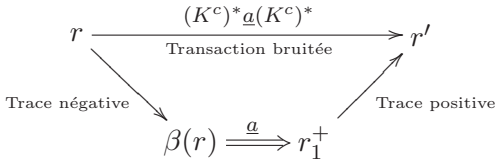
En conjuguant le lemme 4.2.3 et le lemme ci-dessus on peut d'ores et déjà construire la décomposition suivante, avec un unique $\beta(r)$:



avec σ_0 négative et σ_1 positive, de source initiale. Lorsque σ_1 contient une validation, on montre qu'on peut extraire la transaction issue de $\beta(r)$ et dont la validation est α.

Lemme 4.2.5 *Soit σ une trace positive, de source initiale et contenant une unique validation. Il existe une unique transaction θ et une unique trace silencieuse σ' telles que $\sigma \sim \theta; \sigma'$.*

Soit σ_0 la sous-trace de σ précédant la validation $t_{\underline{\alpha}}$. On raisonne par induction sur la taille de σ_0. Si $\sigma_0; t_{\underline{\alpha}}$ est une transaction (ce qui inclut le cas de base où $\sigma_0 = \epsilon$), nous avons fini. Sinon, il existe une transition t et une trace σ_0', plus petite que σ_0, telles que $\sigma_0'; t_{\underline{a}}; t \sim \sigma_0$. On conclut par hypothèse d'induction sur σ_0'. Muni du lemme 4.2.3, nous pouvons compléter la décomposition :



La trace $r_1^+ \xrightarrow{\sigma_1} r'$ étant positive, on peut l'inverser et obtenir une trace négative de but initial. Par le lemme 4.2.3 on en conclut que $r_1^+ = \beta(r')$ et donc que r_1^+ est aussi unique. La décomposition attendue est une trace $r \xrightarrow{\sigma_0} \beta(r) \xRightarrow{\theta} \beta(r') \xrightarrow{\sigma_1} r'$ qui est unique par le théorème 3.2.1. □

4.2.3 Théorème transactionnel.

On définit le *système de transitions causales* (CTS) engendré par un processus p de la façon suivante :

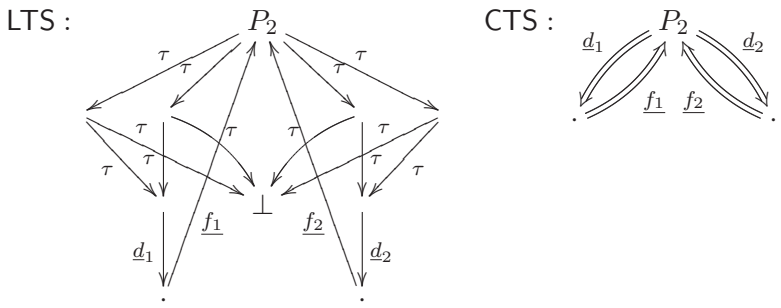
Définition 4.2.2 *On rappelle qu'on note* $p \stackrel{\underline{\alpha}}{\Longrightarrow} p'$ *s'il existe une transaction de* p *vers* p'*, d'observable* $\underline{\alpha}$*. Le quadruplet* $\langle P_m, p, K, \Longrightarrow \rangle$ *est le* système de transitions causales *de* p*, on le note CTS*(p)*.*

Le CTS d'un processus p est obtenu en coupant du système de transitions étiquetées de p les traces qui ne sont pas transactionnelles, *i.e* qui ne satisfont pas à la définition 4.2.1. Le CTS est donc un objet *a priori* plus petit que le LTS complet du processus. Nous montrons ce phénomène de compression dans l'exemple qui suit.

Exemple 4.2.1 *Considérons l'implémentation naïve des philosophes donnée dans l'exemple 1.1.2 (avec* $N = 2$*) :*

$$P_2 := \prod_{i=1}^{2} \big(\mathsf{P}(b_i, b_{i+1}, \underline{d_i}, \underline{f_i}) \parallel b_i\big) \backslash b_i$$

La figure suivante montre le système de transitions engendré par P_2 *ainsi que son CTS.*



Exemple 4.2.1 – LTS et CTS du code des philosophes avec état verrou.

La compression d'un LTS en un système de transitions causales a pour effet de supprimer les états verrous dans lesquels aucun processus ne peut atteindre de validation. Il permet d'extraire l'ensemble des séquences possibles des transactions validables à partir du processus initial. Toutefois, comme on peut le voir dans l'exemple ci-dessus, un processus n'a, en général, pas le même comportement que son CTS. Il peut y avoir des choix partiels et des verrous (état $\bot$ dans l'exemple) qui vont contrarier la recherche du consensus qui correspond, dans notre exemple, à la partie supérieure du LTS de p. Toutefois, comme ces transitions sont silencieuses, elles ne peuvent bloquer une fois interprétées dans RCCS : le CTS de p correspond donc précisément au fonctionnement observable global de $\ell(p)$. Cette propriété est formalisée par le théorème suivant :

Théorème 4.2.1 *Soient $\mathcal{R}(p) := \langle R, \ell(p), A \cup A^-, \rightarrow \rangle$ le LTS engendré par le système racine $\ell(p)$, alors $\mathcal{R}(p) \approx CTS(p)$.*

En d'autres termes, le CTS d'un processus p correspond à la spécification du système $\ell(p)$. Le théorème indique donc qu'il n'est pas nécessaire d'étudier les transaction bruitées de $\mathcal{R}(p)$, pour comprendre son fonctionnement. Il suffit de lisser les transactions et d'étudier uniquement leurs séquences afin d'obtenir le comportement stable du système. La preuve du théorème repose donc sur le lemme 4.2.2, en montrant que la relation $\approx := \{(p, r) \mid p \equiv (\varphi \circ \beta)r\}$ est une bisimulation, avec K comme observables et $\beta(r)$ l'unique système initial dans le passé de r. On prouve les bonnes propriétés de $\approx$, suivant la définition 1.2.1. Soit donc $p \approx r$, on étudie les cas suivants :
– Supposons $r \rightarrow_w^* r'$ et $w \in (K^c)^*$. En utilisant le lemme 4.2.2 on décompose la trace en $r \rightarrow^* \beta(r) \rightarrow^* r'$. Par unicité de cette décomposition on en déduit que $\beta(r') = \beta(r)$ et donc que $p \approx r'$.
– Supposons $r \rightarrow_w^* r'$ et $w \in (K^c)^* \underline{\alpha} (K^c)^*$. On utilise le lemme 4.2.2 et on obtient $r \rightarrow^* \beta(r) \stackrel{\underline{\alpha}}{\Longrightarrow} \beta(r') \rightarrow^* r'$. Par hypothèse, $(\varphi \circ \beta)r \equiv p$, par simulation on obtient $p \stackrel{\underline{\alpha}}{\Longrightarrow} p'$ avec $(\varphi \circ \beta)r' = p'$. On a donc $p' \approx r'$.
– Il nous reste à prouver que $\approx$ est bien symétrique. Supposons $p \stackrel{\underline{\alpha}}{\Longrightarrow} p'$. Par simulation on a $\beta(r) \stackrel{\underline{\alpha}}{\Longrightarrow} r_1^+$ avec $\varphi(r_1) = p'$. En utilisant le lemme 4.2.2 et le fait que la décomposition est unique, on a que $r_1^+ = \beta(r')$ et on a bien $p' \approx r'$. □

4.2.4 Programmation Concurrente Déclarative.

Reprenons les exemples 1.1.1 et 1.1.2 du premier chapitre. Nous avons vu que la présence de choix partiels et de verrous empêchaient les exemples d'être corrects ; il nous a donc fallu modifier les codes initiaux (Exemples 1.2.1 et 1.2.2) afin d'obtenir les bisimulations attendues. À la lumière des nouvelles notions introduites dans ce chapitre reprenons le schéma suivi pour l'exemple des philosophes. Au chapitre 1, nous avons vu que l'implémentation P_N du dîner des philosophes (Exemple 1.1.2) n'était pas bisimilaire à la spécification :

$$\begin{array}{lcl} \mathcal{S}_{philo}(P \cup \{i-1, i, i+1\}, M) & \xrightarrow{d_i} & \mathcal{S}_{philo}(P \cup \{i-1, i+1\}, M \cup \{i\}) \\ \mathcal{S}_{philo}(P, M \cup \{i\}) & \xrightarrow{f_i} & \mathcal{S}_{philo}(P \cup \{i\}, M) \end{array}$$

Dans l'exemple 4.2.1, le LTS engendré par P_2 contient effectivement un état bloqué $\perp$. La méthode standard consiste à ajouter des transitions arrières à

P_N afin de corriger ces défauts mais on obtient alors un code non intuitif et plus gros (voir l'exemple 1.2.2). Cette méthode est donc périlleuse car elle est susceptible d'introduire des erreurs dans le code initial, la modification apportée n'étant en rien générique. À moins d'exhiber une bisimulation, on ne peut alors pas garantir que le retour arrière introduit enlève bien tous les verrous (et les choix partiels) du système d'origine. L'alternative proposée dans ce chapitre est d'utiliser le théorème transactionnel. On extrait le système de transitions causales du processus initial et si celui-ci est conforme à la spécification, on obtient un processus correct par simple relèvement dans RCCS. Ainsi, dans l'exemple 4.2.1, si P_2 n'est pas conforme à $\mathcal{S}_{vote}$, en revanche on a bien $CTS(P_2) \approx \mathcal{S}_{vote}(\{1,2\},\emptyset)$ et on peut en déduire $\mathcal{R}(P_2) \approx \mathcal{S}_{vote}(\{1,2\},\emptyset)$ grâce au théorème 4.2.1. Il est donc possible de générer automatiquement un code correct – ici $\ell(P_2)$ – à partir d'un code – ici P_2 – dont la seule bonne propriété est d'avoir un système de transitions causales bisimilaire à sa spécification. Dans la suite nous ferons référence à cette approche sous le nom de méthode de *déclarative*. Nous pouvons résumer la méthode déclarative de la façon suivante :

1. le programmeur *déclare* comment chaque agent du système peut effectuer une transaction en supposant que les ressources nécessaires sont toujours disponibles.
2. Le *programme déclaratif* est alors constitué des codes de tous les agents déclarés en 1. et de la modélisation des ressources disponibles.
3. On extrait le système de transitions causales du programme déclaratif et on vérifie qu'il est bien conforme à la spécification.
4. Si tel est le cas, on interprète le code dans la machine RCCS et on a la garantie que le système se comportera correctement.

Exemple 4.2.2 (Programmation déclarative.)

1. *Déclaration des transactions :* $\mathsf{V}(t,\underline{v_i}) := \bar{t}.\underline{v_i}$;
2. *Programme déclaratif :* $p := (\prod_{i=1}^{N} \mathsf{V}(t,\underline{v_i}) \parallel \prod_{i}^{m} t.0)\backslash t$;
3. *Vérification :* $CTS(p) \approx \mathcal{S}_{vote}(m,N)$;
4. *Programme correct :* $\ell(p)$.

4.3 Discussion.

Plusieurs modèles à base d'algèbres de processus ont été proposés pour modéliser les systèmes transactionnels. Ces approches visent à compenser les transactions ayant échouées par des processus gérant le comportement

du système en cas de verrous[3]. Ce n'est pas l'optique choisie ici, bien que les compensations se révèlent un mécanisme important dans le cadre des transactions imbriquées (*nested transactions*) pour lesquelles un retour arrière n'est pas toujours possible. Notre approche présente toutefois des similitudes avec les travaux de Bruni, Laneve et Montanari[4] présentant un modèle de transaction dans le formalisme du *join calculus*[5]. Leur approche repose sur l'utilisation de réseaux de Petri, dit Zero-saufs[6] (*Zero-safe nets*), permettant de distinguer les états stables d'une transaction des états intermédiaires susceptibles d'échouer. Ces réseaux de spécification sont ensuite compilés en Join dans un code bas niveau où des processus dédiés se chargent de gérer les messages de blocages émis par les processus dans des états non validés. Le redémarrage des transactions est lui-même basé sur un système de compensation ne présentant pas d'analogie particulière avec nos travaux. Par ailleurs, les systèmes transactionnels obtenus ne peuvent aboutir que dans des états stables, donnés par le réseau de Petri de spécification initial. Dans RCCS cela reviendrait à interdire les transactions bruitées, en se restreignant à n'effectuer que les validations qui rendent initial l'état d'arrivée.

Dans ce chapitre, nous avons montré qu'il est possible de déduire le comportement (bidirectionnel) du relèvement d'un processus p en fonction du simple comportement (vers l'avant) de p. Plus précisément, nous avons montré que les interactions définitives d'un système partiellement réversible $\ell(p)$ coincident avec les interactions qu'on observe sur le système de transitions causales engendré par p (Théorème 4.2.1). Nous avons déduit de cette propriété une méthode déclarative qui consiste à fournir un programme partiellement correct, c'est à dire pour lequel seul le CTS est bisimilaire à la spécification, puis à l'interpréter dans RCCS pour obtenir un comportement asymptotique correct[7]. Il nous reste donc à décrire une méthode efficace permettant de compiler un CTS à partir d'un processus marqué. En effet, la méthode déclarative que nous venons de décrire ne présente d'intérêt que si la procédure d'extraction du CTS est nettement plus rapide que la vérification du code dans lequel on aurait ajouté, à la main, une façon de sortir des verrous. Nous proposerons au prochain chapitre, un algorithme permettant

[3] [Butler *et al.*, 2002, Bocchi *et al.*, 2003, Butler *et al.*, 2004]

[4] [Bruni *et al.*, 2002]

[5] [Fournet et Gonthier, 1996]

[6] [Bruni et Montanari, 2000]

[7] En effet, l'usage d'une sémantique stochastique comme dans [Priami, 1995] ou [Herescu et Palamidessi, 2000] est nécessaire pour implémenter le fonctionnement de RCCS. Par définition les processus réversibles peuvent boucler et l'ajout de probabilités dans le choix des transitions positives ou négatives permettrait de rompre les comportements, trivialement infinis, induits par le retour arrière.

d'extraire le CTS d'un processus donné. Dans le cas des philosophes, nous verrons que le calcul du CTS est largement moins coûteux en temps et en mémoire que la méthode directe consistant à vérifier la correction du code complet, donné dans l'exemple 1.1.2, en utilisant un outil de vérification automatique[8].

[8] [Victor et Moller, 1994]

Chapitre 5

Systèmes de transitions causales.

Sommaire

Dans le chapitre précédent, nous avons vu que le système de transitions causales (CTS) d'un processus p était bisimilaire au système de transitions (partiellement) réversible engendré par $\ell(p)$ (Théorème 4.2.1). Nous en avons déduit une méthode de programmation déclarative qui, étant donnée une spécification *spec*, construit automatiquement un système bisimilaire à partir du relèvement d'un processus simple p tel que $\mathrm{CTS}(p) \approx \mathit{spec}$. Comme nous l'avons argumenté, ceci constitue une alternative potentiellement plus rapide et moins sujette aux erreurs, qu'une méthode directe consistant à modifier p afin de le rendre directement bisimilaire à sa spécification. Nous résumons ces deux alternatives Figure 5.1. Le chemin de gauche correspond à extraire le CTS de p (1), puis à vérifier sa correction vis-à-vis de la spécification (2). Le chemin de droite commence par modifier p afin d'enlever les verrous et les choix partiels ($1'$), puis on vérifie que le processus p' ainsi

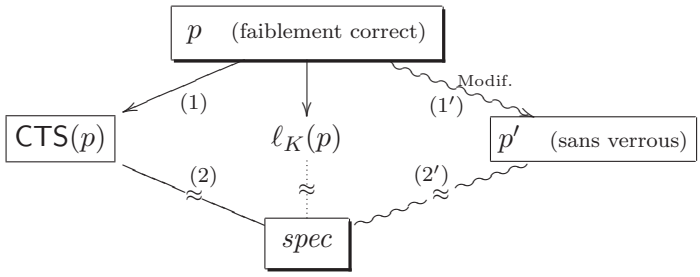


FIG. 5.1 – Programmation concurrente déclarative et méthode standard.

obtenu, est bien bisimilaire à la spécification (2′). Du fait de la taille des systèmes de transitions à considérer (*state explosion problem*), le point (2′) est quasi impossible à obtenir de façon automatique pour des cas non triviaux[1], même si de nombreuses études ont abouti à des techniques permettant de se ramener à des LTS plus petits afin d'accroître les chances d'exhiber des propriétés de correction[2], voir une bisimulation avec un système de référence[3]. À supposer qu'on soit capable d'extraire efficacement le système de transitions causales d'un processus donné, un grand pas serait franchi puisque ce dernier est une abstraction de l'architecture interne des systèmes, qui ne retient que les transactions validables : un CTS est donc aussi une spécification naturelle du système transactionnel attendu et le point (2) consiste donc en général à tester un simple isomorphisme entre le CTS et la spécification. Afin d'accorder du crédit à la méthode indirecte correspondant à la branche gauche de la figure 5.1, il nous faut donc exhiber un algorithme permettant d'extraire le CTS d'un processus CCS donné. Dans une perspective de vérification automatique, un problème d'implémentation se pose : il est impossible de savoir qu'une trace sera finalement transactionnelle tant qu'une validation n'a pas été effectuée. En l'état actuel de la définition, il nous faudrait donc produire toutes les traces positives, puis sélectionner les traces minimales se terminant par une validation pour en extraire les transactions ; le coût d'une telle procédure condamnerait inévitablement la méthode de programmation déclarative. Toutefois, en tirant profit de la structure des processus CCS on pourrait définir une approche inverse consistant à isoler les capacités soulignées, puis à reconstruire l'ensemble des actions permettant de déclencher ces capacités. Ainsi, dans le processus $p := \bar{x}.\underline{a} \mid x.0$, on isolerait d'abord

[1] [Laroussinie et Schnoebelen, 2000]
[2] [Valmari, 1989, Probst et Li, 1990, Godefroid et Wolper, 1991]
[3] [Sangiorgi, 1995, Bianchi *et al.*, 1995, Hirschkoff, 1998, Fisler et Vardi, 2002]

la capacité $\underline{a}$ et on conclurait que l'action $\bar{x}$ est nécessaire pour la déclencher. On obtiendrait donc une première transaction $p \rightarrow^*_{\bar{x}\underline{a}} x.0$. Il faudrait ensuite remarquer que toute synchronisation utilisant $\bar{x}$ déclenche aussi la validation ; d'où la transaction $p \rightarrow^*_{\tau\underline{a}} 0$. Dans ce chapitre, nous présentons une sémantique de CCS basée sur une structure d'évènements qui nous permettra de définir formellement cette procédure. Nous définissons dans un premier temps la notion de structure d'événements puis nous montrerons une méthode inductive permettant de compiler un processus CCS en une structure d'événements dite de flot. Nous décrirons alors un algorithme permettant d'extraire le CTS d'un processus en se servant de structures de flot d'événements comme représentation interne. Les points clefs de l'implémentation réalisée dans un module Ocaml, seront abordés au chapitre suivant.

5.1 Concepts de base.

5.1.1 Déploiement de processus.

Les structures d'événements sont des objets mathématiques permettant une approche de la concurrence s'abstrayant totalement de l'ordre arbitraire dans lequel les événements d'un calcul concurrent se produisent[4]. Dans cette approche, dite *véritablement concurrente* (*truly concurrent semantics*), la précédence formelle entre événements est représentée de manière explicite. Dans une structure d'événements on ne s'intéresse qu'aux relations fondamentales liant les événements d'un calcul les uns aux autres. Ces relations sont le *conflit* et la *cause* ; les événements n'étant pas en relation sont alors dits *concurrents*. Une structure d'événements est un triplet $\langle E, \leq, \# \rangle$ constitué d'un ensemble E d'événements, d'une relation de causalité $\leq \subseteq E \times E$ et d'une relation de conflit $\# \subseteq E \times E$, qui permet de spécifier quels événements s'excluent mutuellement lors d'un calcul.

On peut faire correspondre à un processus CCS, son déploiement en une structure d'événements[5]. On parle alors de structures d'événements étiquetées, de la forme $\langle E, \leq, \#, \lambda \rangle$, où $\lambda : E \rightarrow A$ est une fonction associant à chaque événement de E, l'action CCS qui lui correspond. Par exemple prenons le processus $p := \bar{a}.b \mid a.c$. On peut lui faire correspondre plusieurs types de structures d'événements représentées, Figure 5.2, avec la notation standard $e \rightarrow e'$ pour noter la causalité et $e \ldots e'$ pour la relation de conflit.

Les deux structures d'événements $\mathcal{E}_{Flow}(p)$ et $\mathcal{E}_{Prime}(p)$ correspondent

[4] [Nielsen *et al.*, 1981, Winskel, 1982, Nielsen et Winskel, 1995]
[5] [Winskel, 1982, Boudol et Castellani, 1989]

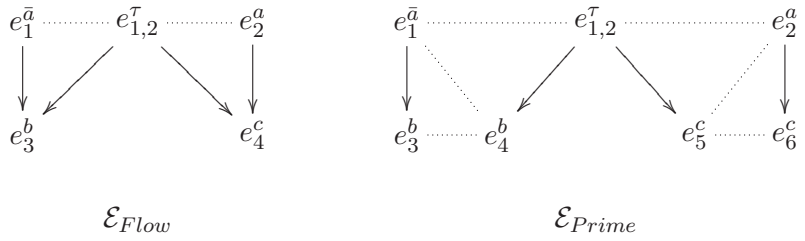


FIG. 5.2 – Structures d'événements de $p := \bar{a}.b \mid a.c$, avec $\lambda(e_i^\alpha) = \alpha$.

chacune à une représentation du processus p. La première est une structure dite de *flot* (*flow event structure*, FES), tandis que la seconde structure, par ailleurs la plus couramment considérée, est dite *première* (*prime event structure*, PES). La contrainte de primalité revient à s'assurer qu'un événement est entièrement caractérisé par son passé. Ainsi dans la structure $\mathcal{E}_{Prime}$, l'événement e_4^b n'a qu'un passé possible $e_{1,2}^\tau$, tandis que dans $\mathcal{E}_{Flow}$ on distingue deux passés contradictoires $e_1^{\bar{a}}$ et $e_{1,2}^\tau$. La construction d'une PES, correspondant à un processus donné, nécessite donc de prendre en cause toutes les synchronisations possibles impliquant un événement, afin d'en dupliquer à chaque fois les successeurs. Pour cette raison, il n'existe pas de méthode simple pour déployer un processus CCS dans une structure première, la contrainte de primalité étant techniquement difficile à préserver par composition.

Les FES, comme $\mathcal{E}_{Flow}$ dans la figure 5.2, sont des structures d'événements particulières qui permettent une interprétation très directe des processus CCS[6]. On peut les voir comme une forme factorisée de PES, dans laquelle on ne duplique pas le futur d'un événement après synchronisation. Ce qu'on gagne en espace de représentation et en simplicité de construction, se perd toutefois en complexité d'analyse car la reconstruction du passé d'un événement ne peut plus être simplement lue en inversant les flèches, comme dans les PES. Il faut pour cela recourir à une notion plus complexe de *configuration* comme nous allons le voir par la suite. Dans la prochaine section, nous définissons formellement les structures de flot, puis nous montrerons comment représenter des processus CCS dans ce formalisme. Nous verrons dans la section 5.3.2 comment utiliser cette structure pour extraire le CTS d'un processus CCS donné.

[6] [Boudol et Castellani, 1989, Boudol et Castellani, 1994, Castellani et Zhang, 1997, van Glabeek et Goltz, 2003]

5.1.2 Structures de flot d'événements.

Comme nous l'avons dit précédemment, les FES sont des structures d'événements aux propriétés algébriques plus relâchées que pour les PES. En particulier, pour ces dernières, la relation de cause est un ordre partiel qui a pour propriété de transmettre la relation de conflit *via* la règle $e\#e' \leq e'' \Rightarrow e\#e''$. On n'exige plus ces propriétés dans les structures de flot pour lesquelles on parle plutôt de relation de *flot* ou de *cause directe*.

Définition 5.1.1 (Structure de flot) *Une* structure de flot d'événements *est un quadruplet* $\langle E, \prec, \#, \lambda \rangle$ *où :*
— E *est un ensemble dénombrable d'événements,*
— $\# \subseteq E \times E$ *est une relation symétrique et irréflexive,*
— $\prec \subseteq E \times E$ *est une relation irréflexive,*
— $\lambda : E \to A \times \mathcal{R} \cup \{\bot\}$ *est la fonction qui associe à chaque événement le couple* (α, ρ) *correspondant à l'action* $\alpha \in A$ *qui lui est associée, ainsi que son identifiant de restriction,* $\rho \in \mathcal{R} \cup \{\bot\}$*. On suppose* $\{\bot\}$ *disjoint de* $\mathcal{R}$ *et on utilisera* $\mathcal{R}_\bot$ *pour dénoter* $\mathcal{R} \cup \{\bot\}$*.*

Par rapport aux structures de flot définies par Boudol et Castellani, nous associons à chaque événement un identifiant de restriction qui permet de gérer explicitement la portée des restrictions. Intuitivement, deux événements e_1 et e_2 seront synchronisables si $\lambda(e_1) = (a, \rho)$ et $\lambda(e_2) = (\bar{a}, \rho)$, c'est à dire si e_1 et e_2 ont des actions complémentaires situées dans le champ d'une même restriction ; le symbole $\bot$ servant à marquer l'absence de restriction. Soient $\pi_1 : A \times \mathcal{R}_\bot \to A$ et $\pi_2 : A \times \mathcal{R}_\bot \to \mathcal{R}_\bot$ les projections usuelles des paires de $\lambda(E)$, on utilisera dans la suite $\lambda_1 : E \to A$ et $\lambda_2 : E \to \mathcal{R}$ en place de $\pi_1 \circ \lambda$ et $\pi_2 \circ \lambda$.

Les configurations d'une structure $\mathcal{E}$, notées $\mathsf{Conf}(\mathcal{E})$, correspondent aux ensembles d'événements pouvant se produire au cours d'un même calcul. Clairement $X \in \mathsf{Conf}(\mathcal{E})$ implique qu'aucun couple d'événements de X n'est dans la relation de conflit. De plus on requiert que la clôture transitive de la relation de flot, restreinte au événements d'une configuration X, soit un ordre partiel strict. En d'autres termes, la relation de flot entre événements d'une même trace exprime aussi la causalité entre événements, elle peut en revanche devenir cyclique si on regarde des événements appartenant à des configurations différentes. Enfin, une configuration doit comporter tous les prédécesseurs compatibles des événements qui la composent : si e est dans une configuration X et si $e' \notin X$ précède e, alors la raison pour laquelle e' a été omis est qu'un autre prédécesseur de e, incompatible avec e', était déjà présent dans X. La définition formelle des configurations est la suivante :

Définition 5.1.2 (Configuration) *Soit $\mathcal{E} := \langle E, \prec, \#, \lambda \rangle$ une FES. Une* configuration $X \in \mathsf{Conf}(\mathcal{E})$*, est un sous-ensemble de E tel que :*
— X est cohérent : $\# \cap (X \times X) = \emptyset$ et pour tout $e \in X$ on a $\lambda_2(e) = \perp$;
— X ne contient pas de cycle : la clôture transitive de $\prec$ restreinte à X est irréflexive ;
— X est clos à gauche (aux conflits près) : pour tout $e \in X$ si $e' \prec e$ et $e' \notin X$ alors il existe $e'' \in X$ tel que $e'' \prec e$ et $e''\#e'$.

Pour tout $e \in X$, on demande de plus que l'ensemble des prédécesseurs de e soit fini. Cette condition étant toujours satisfaite dans les structures d'événements que nous allons considérer, nous la laissons délibérément de côté.

5.2 Une sémantique de flot pour CCS.

5.2.1 Construction inductive.

Nous allons maintenant montrer comment utiliser la structure des processus CCS afin d'en déduire une représentation sous forme de FES. La méthode générale suit le schéma inductif suivant : on dispose de la structure vide correspondant au processus 0, puis on définit le correspondant des opérateurs standard de CCS (préfixe, somme, composition et restriction) dans la sémantique de flot. La composition de structures d'événements est l'opération la plus délicate à définir, mais les propriétés des structures de flot sont précisément faites pour en simplifier la définition[7]. Une fois cette compilation définie, un théorème de représentation permet de certifier que les configurations de $\mathcal{E}(p)$ sont bien isomorphes aux traces issues de p, à équivalence près[8]. On commence par définir les opérateurs sur les structures de flot qui nous permettrons de définir le *déploiement* d'un processus p (voir définition 5.2.1).

— (**Structure vide**) Le correspondant du processus 0 est donné par la structure vide :

$$\mathcal{O} := (\emptyset, \emptyset, \emptyset, \emptyset) \tag{5.1}$$

— (**Préfixe**) Supposons $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle$ avec $e \notin E$ et posons

$$E^{\downarrow} := \{e \in E \mid \forall e' \in E : \neg(e' \prec e)\}$$

[7] Sous certaines conditions, la composition de FES peut être définie en terme de produit catégorique cf. [Castellani et Zhang, 1997] ainsi que [van Glabeek et Goltz, 2003].
[8] [Nielsen *et al.*, 1981, Boudol et Castellani, 1989]

l'opérateur de *préfixe* $\_ \odot \_ : A \times FES \to FES$, est alors défini par :

$$\alpha \odot \mathcal{E} := \langle E \cup \{e\}\,, \prec \cup (\{e\} \times E^{\downarrow}), \#, \lambda \uplus [e \mapsto (\alpha, \bot)]\rangle \qquad (5.2)$$

— (**Choix**) Supposons, $\mathcal{E} = \langle E, \prec, \#, \lambda\rangle$ et $\mathcal{E}' = \langle E', \prec', \#', \lambda'\rangle$ tels que :

$$E \cap E' = \emptyset \;\&\; \lambda_2(E) \cap \lambda'_2(E') = \{\bot\}$$

l'opérateur de *choix* $\_ \oplus \_ : FES \times FES \to FES$, est défini par :

$$\mathcal{E} \oplus \mathcal{E}' := \langle E \cup E', \prec \cup \prec', \# \cup \#' \cup (E \times E'), \lambda \uplus \lambda'\rangle \qquad (5.3)$$

— (**Restriction**) Soient $\mathcal{E} = \langle E, \prec, \#, \lambda\rangle$ et $\rho \notin \lambda_2(E)$. Étant donné un nom de canal $x \in X$, on définit la nouvelle fonction d'étiquetage $\lambda^{x,\rho} : E \to A \times \mathcal{R}$ par :

$$\begin{array}{rcl} \lambda_1^{x,\rho}(e) & = & \lambda_1(e) \\ \lambda_2^{x,\rho}(e) & = & \rho \text{ si } \lambda_1(e) \in \{x, \bar{x}\} \;\&\; \lambda_2(e) = \bot \\ & & \lambda_2(e) \text{ sinon.} \end{array}$$

La *restriction* $\_\backslash\backslash\_ : FES \times (X \times \mathcal{R}) \to FES$ est donnée par :

$$\mathcal{E}\backslash\backslash\langle x, \rho\rangle := \langle E, \prec, \#, \lambda^{x,\rho}\rangle \qquad (5.4)$$

Il nous reste à définir l'opérateur sur les FES qui correspondra à la composition dans CCS. La définition de cette dernière opération étant plus délicate, nous la remettons à plus tard afin de commenter les constructions ci-dessus. Le point (5.1) est assez intuitif : la structure vide correspond simplement à un ensemble d'événements vide. Pour l'opérateur préfixe (5.2), on isole l'ensemble $E^{\downarrow}$ des événements minimaux de la structure, c'est à dire les points n'ayant pas de prédécesseur pour la relation de flot. L'opération de préfixe consiste alors à ajouter un événement frais à E qui sera une cause immédiate de tous les éléments de $E^{\downarrow}$ dans la nouvelle structure. On notera que cette étape diffère légèrement de celle présentée dans [Boudol et Castellani, 1989], où le nouvel événement introduit devient une cause de tous les événements de E, ce qui ne change pas l'espace des configurations de la structure générée, mais induit des relations de flot superflues et quelque peu contre-intuitives. Pour l'opérateur de choix (5.3), on commence par supposer que les structures représentant p et p' ont des ensembles disjoints d'événements et d'identifiants de restriction. On construit ensuite la somme des deux FES en mettant en conflit les éléments de la première avec ceux de la seconde. Enfin, pour l'opérateur de restriction (5.4), on utilise

un ρ frais –*i.e* satisfaisant la propriété $\rho \notin \lambda_2(E)$– et on l'associe à chaque événement dont l'action utilise x, *via* la fonction λ. Afin d'éviter les captures de variables comme dans $(x_\rho \mid (\bar{x}_\rho \backslash x_\rho)) \backslash x_\rho$, on ne fait cette association que si l'événement n'était pas déjà dans le champ d'une restriction, c'est à dire si $\lambda_2(e) = \bot$. Cette approche de la restriction diffère de celle proposée par Boudol et Castellani, qui consiste à utiliser une relation de conflit réflexive afin de rendre les événements restreints auto-conflictuels. Dans le but d'obtenir une représentation finie des structures de flot, nous verrons au chapitre suivant que l'utilisation des identifiants de restriction permet de définir aisément un déploiement partiel des constantes de récursions. Nous procédons maintenant à la définition de l'opération de composition de deux FES[9].

— (**Composition**) On dispose des structures de flot $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle$ et $\mathcal{E}' = \langle E', \prec', \#', \lambda' \rangle$ dont on suppose les identifiants disjoints, c'est à dire satisfaisant :

$$E \cap E' = \emptyset \ \& \ \lambda_2(E) \cap \lambda'_2(E') = \{\bot\}$$

L'ensemble $E_\star$ des événements du produit des structures est obtenu par la construction suivante :

$$E_\star := (E \times \{\star\}) \cup (\{\star\} \times E') \cup (E \times_\star E')$$

où $(e, e') \in (E \times_\star E')$ si $\lambda(e) = (\bar{a}, \rho)$ et $\lambda'(e') = (a, \rho)$. On utilise les projections $\pi_1 : E_\star \to E \cup \{\star\}$ et $\pi_2 : E_\star \to E' \cup \{\star\}$ pour désigner la première et la deuxième composante des événements de $E_\star$. On définit les nouvelles relations de causalité $\prec_\star \subseteq E_\star \times E_\star$ et de conflit $\#_\star \subseteq E_\star \times E_\star$ en conséquence :

$$\prec_\star \ := \ \{(e, e') \mid \pi_1(e) \prec \pi_1(e') \text{ ou } \pi_2(e) \prec' \pi_2(e')\}$$

$$\#_\star \ := \ \left\{ (e, e') \mid \begin{array}{l} \pi_1(e) \# \pi_1(e') \text{ ou } \pi_2(e) \#' \pi_2(e') \text{ ou} \\ e \neq e' \ \& \ \exists i \in \{1, 2\} : \pi_i(e) = \pi_i(e') \neq \star \end{array} \right\}$$

En d'autres termes, deux événements de la nouvelle structure sont en relation si une leurs projections l'étaient déjà dans leurs structures initiales. De plus on ajoute à la relation de conflit tous les événements de synchronisation distincts partageant une même projection (différente de $\star$). On met à jour la nouvelle fonction d'étiquetage $\lambda_\star : E_\star \to A \times \mathcal{R}$ comme suit :

$$\begin{array}{lcl} \lambda_\star(e, \star) & := & \lambda(e) \\ \lambda_\star(\star, e) & := & \lambda'(e) \\ \lambda_\star(e, e') & := & (\tau, \bot) \end{array}$$

[9] [Winskel, 1982, Boudol et Castellani, 1989]

la *composition* $\_ \otimes \_ : FES \times FES \to FES$ est alors donnée par :

$$\mathcal{E} \otimes \mathcal{E}' := \langle E_\star, \prec_\star, \#_\star, \lambda_\star \rangle \tag{5.5}$$

Définition 5.2.1 (Déploiement de processus) *Soit p un processus CCS, muni d'un ensemble de définitions Δ. La structure de flot d'événements $\mathcal{U}_\Delta(p)$ qui lui correspond est définie inductivement par :*

$$\begin{array}{lcl}
\mathcal{U}_\Delta(0) & := & \mathcal{O} \\
\mathcal{U}_\Delta(\mathsf{D}(\tilde{x})) & := & \mathcal{U}_\Delta(p) \textit{ si } (\mathsf{D}(\tilde{x}) := p) \in \Delta \\
\mathcal{U}_\Delta(a.p) & := & a \odot \mathcal{U}_\Delta(p) \\
\mathcal{U}_\Delta(p+q) & := & \mathcal{U}_\Delta(p) \oplus \mathcal{U}_\Delta(q) \\
\mathcal{U}_\Delta(p\backslash x) & := & \mathcal{U}_\Delta(p) \backslash\backslash \langle x, \rho \rangle \quad \textit{pour un } \rho \textit{ frais dans } \mathcal{U}_\Delta(p) \\
\mathcal{U}_\Delta(p \parallel q) & := & \mathcal{U}_\Delta(p) \otimes \mathcal{U}_\Delta(q)
\end{array}$$

Il résulte de cette définition, que le déploiement d'un processus contenant des définitions récursives est un objet infini. Nous laissons la discussion de ce point pour la conclusion de ce chapitre. Concernant la taille des déploiements dans les structures de flot, nous pouvons faire une autre remarque importante. Soient $\mathcal{E}$ et $\mathcal{E}'$ deux FES finies qu'on suppose, pour simplifier, avoir un même nombre d'événements n. Dans le pire des cas, le nombre d'événements du produit $\mathcal{E} \otimes \mathcal{E}'$ croît en $\mathcal{O}(n^2)$ en raison des événements de synchronisation à ajouter à la nouvelle structure. Comme nous l'avons noté, dans les structures premières, tout événement de synchronisation entre e et e' entraîne la duplication des futurs de e et e'. Il résulterait une croissance en $\mathcal{O}(2^n)$ du nombre d'événements à considérer dans le produit des structures.

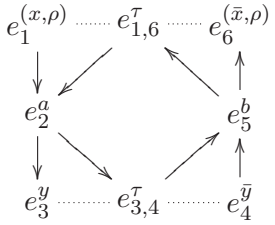
Pour tout processus p, on montre à présent que les configurations de $\mathcal{U}_\Delta(p)$ correspondent précisément aux traces de calcul issues de p, considérées aux permutations près. Pour un processus p donné, on note $\mathcal{T}_\simeq(p)$ l'ensemble des traces issues de p, quotientées par équivalence. On utilise $\leq$ pour dénoter la relation de préfixe sur les traces et on rappelle que $\mathsf{Conf}(\mathcal{E})$ dénote l'ensemble des configurations de la structure d'événements $\mathcal{E}$. On dispose du théorème suivant :

Théorème 5.2.1 ([Boudol et Castellani, 1989]) *Soient p un processus CCS muni d'un ensemble de définitions Δ et $\mathcal{E}_p = \mathcal{U}_\Delta(p)$ son interprétation dans les FES. Les ensembles ordonnés $\big(\mathcal{T}_\simeq(p), \leq\big)$ et $\big(\mathsf{Conf}(\mathcal{E}_p), \subseteq\big)$ sont isomorphes.*

5.2.2 Exemple

Nous donnons à présent un exemple simple de déploiement d'un processus CCS dans sa structure de flot. Par convention, $e_i^{(\alpha,\rho)}$ dénote un événement e_i tel que $\lambda(e_i) = (\alpha, \rho)$. Lorsque $\rho = \bot$ il est simplement omis.

Exemple 5.2.1 ([Boudol et Castellani, 1989]) *La structure de flot du processus* $p = (x.a.y \parallel \bar{y}.b.\bar{x})\backslash x$ *est :*



EXEMPLE 5.2.1 – Structure de flot résultant de $\mathcal{U}(p)$.

On note que la clôture transitive de la relation de flot génère une boucle de dépendance. Toutefois, cette boucle disparaît si on se restreint à l'ensemble des configurations de $\mathcal{U}(p)$, c'est à dire les sous-ensembles de $\mathcal{U}(p)$ répondant aux propriétés de la définition 5.1.2. Ainsi on peut montrer qu'aucune configuration valide ne peut contenir $e_{3,4}$:

Proposition 5.2.1 *Soit p le processus donné dans l'exemple 5.2.1. Aucune trace issue de p n'aboutit à une synchronisation sur les canaux y.*

On prouve la proposition en sélectionnant l'événement $e_{3,4}$ et en tentant de reconstruire une configuration minimale qui le contiendrait. Supposons $X \in \mathsf{Conf}(\mathcal{U}(p))$ et $e_{3,4} \in X$. Comme X doit contenir les causes immédiates de $e_{3,4}$ (aux conflits près), on a nécessairement $e_2 \in X$. Par le même raisonnement, soit $e_1 \in X$, soit $e_{1,6} \in X$, ces deux événements s'excluant mutuellement. Or, e_1 ne peut apparaître dans aucune configuration car $\lambda_2(e_1) \neq \bot$, on doit donc choisir $e_{1,6}$. On doit alors aussi avoir $e_5 \in X$, qui est la seule cause possible de $e_{1,6}$. On obtient de nouveau un choix dans la continuation de la configuration : une première possibilité serait d'avoir $e_{3,4} \in X$, ce qui est interdit attendu que $\prec^+$ ne doit pas boucler dans X et $e_{3,4}$ appartient déjà à X. La seule autre option serait d'avoir $e_4 \in X$, ce qui est aussi interdit car X ne doit pas contenir d'événements en conflit et $e_4 \# e_{3,4}$. Il est donc impossible de construire une configuration contenant $e_{3,4}$ et par le théorème de représentation, on en déduit qu'il n'existe aucune trace aboutissant à une synchronisation entre les canaux y du processus p. □

On peut donc se servir de la structure de flot résultant du déploiement d'un processus pour générer l'ensemble des traces contenant un événement donné. En suivant cette méthode on doit donc pouvoir construire l'ensemble des configurations contenant une validation et en particulier, si on sait reconnaître une configuration correspondant à l'image d'une transaction, être

à même d'extraire le CTS d'un processus. C'est ce que nous proposons de faire dans la prochaine section.

5.3 Extraction de CTS.

5.3.1 Configurations et transactions.

Selon les termes de la définition 4.2.1, une transaction est une trace silencieuse, suivie d'une validation étiquetée dans K qui ne peut permuter avec aucune transition la précédant. Soit $\preceq$ la relation de préfixe sur les traces quotientées par équivalence, définie par :

$$\sigma \preceq \gamma \iff \exists \delta \neq \epsilon : \sigma; \delta \sim \gamma$$

La définition suivante est une reformulation de la définition 4.2.1.

Définition 5.3.1 *Une trace σ terminant par une validation est une transaction si et seulement si toute trace γ telle que $\gamma \preceq \sigma$ ne contient pas de validation.*

Par le théorème 5.2.1, on obtient :

Lemme 5.3.1 *Étant donnée σ une transaction issue d'un processus p, une configuration $X_\sigma \in \mathsf{Conf}(\mathcal{U}(p))$ est l'image de σ si et seulement si :*

$$\forall Y_\gamma \in \mathsf{Conf}(\mathcal{U}(p)) : Y_\gamma \subset X_\sigma \Rightarrow \forall e \in Y_\gamma : \lambda_1(e) \notin K$$

Autrement dit, une configuration X est l'image d'une trace transactionnelle par l'isomorphisme du théorème 5.2.1 si elle est *minimale* pour la validation (on ne peut lui enlever aucun événement silencieux). Soit $\mathsf{Conf}_T(\mathcal{U}(p)) \subseteq \mathsf{Conf}(\mathcal{U}(p))$ l'ensemble des configurations minimales pour une validation donnée. Nous allons maintenant exprimer le correspondant d'un système de transitions causales, engendré par un processus p, dans les structures d'événements. Pour ce faire, on définit le *résidu* de $\mathcal{E}$ par une configuration X dans $\mathsf{Conf}(\mathcal{E})$.

Définition 5.3.2 (Résidu) *Soient $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle$ une FES et X une configuration de $\mathcal{E}$. Posons $X_\# := \{e \in E \mid \exists e' \in X : e' \# e\}$. Le* résidu *de $\mathcal{E}$ par X est $\mathcal{E}|X := \langle E', \prec', \#', \lambda' \rangle$ où :*

$$\begin{array}{rcl} E' & := & E \setminus (X \cup X_\#) \\ \prec' & := & \prec \cap (E' \times E') \\ \#' & := & \# \cap (E' \times E') \\ \lambda' & := & (\lambda \restriction E') \end{array}$$

Le système de transitions étiquetées associé à $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle$ est défini par l'état initial $\mathcal{E}$ et par la relation de transition $\mathcal{E}' \xrightarrow{X} \mathcal{E}''$ si $X \in \mathsf{Conf}(\mathcal{E}')$ et $\mathcal{E}'' = \mathcal{E}'|X$. On peut à présent définir le *système de transitions causales* issu d'une structure d'événements $\mathcal{E}$, en ne considérant que les transitions $\mathcal{E} \xrightarrow{X} \mathcal{E}'$ pour lesquelles $X \in \mathsf{Conf}_T(\mathcal{E})$. Pour un tel X, appelons $k(X) \in X$ l'unique événement satisfaisant $\lambda_1(k(X)) \in K$. On note $\mathcal{E} \stackrel{\underline{\alpha}}{\Longrightarrow} \mathcal{E}'$ si $\mathcal{E} \xrightarrow{X} \mathcal{E}'$ et $\lambda_1(k(X)) = \underline{\alpha}$. Selon le théorème de représentation, nous pouvons définir un système de transitions isomorphe au CTS d'un processus p.

Proposition 5.3.1 *Soit p un processus CCS muni d'un ensemble de définitions Δ. Le système de transitions causales $CTS(p) := \langle P, p, \stackrel{K}{\Longrightarrow} \rangle$ est isomorphe au système de transitions $\langle \mathsf{FES}, \mathcal{U}_\Delta(p), \stackrel{K}{\Longrightarrow} \rangle$.*

Il nous reste donc à donner un algorithme permettant de construire le CTS engendré par l'état initial $\mathcal{U}_\Delta(p)$ et la relation de transition $\Longrightarrow$.

5.3.2 Algorithme.

Le lemme 5.3.1 se contente de donner le correspondant d'une trace transactionnelle dans l'espace des configurations d'une structure de flot. A des fins calculatoires, on caractérise à présent les configurations transactionnelles de façon plus constructive :

Lemme 5.3.2 *Soient p un processus CCS et $\mathcal{E}_p$ son déploiement dans les FES, alors $X \in \mathsf{Conf}_T(\mathcal{E}_p)$ si et seulement si $\forall e \in X : e \prec^* k(X)$.*

On montre d'abord l'implication. Supposons $e \in X$ tel que $\lambda_1(e) \notin K$ et $e \not\prec^+ k(X)$. Dans ce cas, $X \setminus \{e\}$ est une configuration contenant toujours $k(X)$, ce qui contredit l'hypothèse $X \in \mathsf{Conf}_T(\mathcal{E}_p)$. L'autre direction du lemme est immédiate en notant que si $e \prec^+ k(X)$ alors tout sous-ensemble de X contenant $k(X)$ n'est plus clos à gauche pour la relation de flot. □

Le lemme 5.3.2 implique que pour générer les configurations correspondant aux transactions d'un processus p, il suffit de sélectionner les événements dont les actions sont des validations et de construire les configurations minimales les contenant, à la manière de l'exemple 5.2.1, en sélectionnant progressivement les antécédents nécessaires à la clôture de la configuration. Nous procédons maintenant à la description formelle de la méthode de construction du CTS. Soit une structure de flot $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle$. Un événement $e \in E$ est une *validation* si $\lambda_1(e) \in K$ et on note $E_K \subseteq E$, l'ensemble des validations de E. Une configuration $X \in \mathsf{Conf}(\mathcal{E})$ est *silencieuse* si $\forall e \in X : e \notin E_K$. Une validation $e \in E_K$ est *éligible*, s'il existe

une configuration silencieuse X telle que $X \cup \{e\} \in \mathsf{Conf}(\mathcal{E})$; on note alors $e \in E_K^1$. Soit p un processus CCS marqué et Δ l'ensemble de ses définitions récursives, la procédure d'extraction du CTS de p est donnée figure 5.3.

0. $\mathtt{etats.CTS} \leftarrow \{\mathcal{U}_\Delta(p)\}$; $\mathtt{tr.CTS} \leftarrow \emptyset$;
   $\mathcal{S} \leftarrow \{\mathcal{U}_\Delta(p)\}$; //$\mathcal{S}$ contient les FES à analyser.
1. TANT QUE $\mathcal{S} \neq \emptyset$ FAIRE :
2. $\quad \mathcal{S} \leftarrow \mathcal{S} \setminus \{\mathcal{E} = \langle E, \prec, \#, \lambda \rangle\}$;
3. $\quad V \leftarrow E_K^1$;
4. $\quad$ TANT QUE $V \neq \emptyset$ FAIRE :
5. $\quad\quad V \leftarrow V \setminus \{e\}$;
6. $\quad\quad \mathcal{C} \leftarrow min\_conf(\mathcal{E}, e)$; //$\mathcal{C}$ contient l'ensemble des $X \in \mathsf{Conf}_T(\mathcal{E})$ tq. $e \in X$
7. $\quad\quad$ TANT QUE $\mathcal{C} \neq \emptyset$ FAIRE :
8. $\quad\quad\quad \mathcal{C} \leftarrow \mathcal{C} \setminus \{X\}$;
9. $\quad\quad\quad \mathtt{tr.CTS} \leftarrow \mathtt{tr.CTS} \cup \left\{ \mathcal{E} \stackrel{k(X)}{\Longrightarrow} \mathcal{E}|X \right\}$;
10. $\quad\quad\quad$ SI $(\mathcal{E}|X) \notin \mathtt{etats.CTS}$ ALORS $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathcal{E}|X\}$;
11. $\quad \mathtt{etats.CTS} \leftarrow \mathtt{etats.CTS} \cup \{\mathcal{E}\}$;
12. RENDRE CTS ;

FIG. 5.3 – Algorithme d'extraction du CTS de p.

Le CTS qu'on construit au cours de cet algorithme est représenté sous la forme d'une paire $\mathtt{CTS} := \langle \mathtt{etats}, \mathtt{tr} \rangle$ où $\mathtt{etats}$ correspond à l'ensemble des états du CTS désignés par $\mathtt{etats.CTS}$ et $\mathtt{tr} \subseteq \mathtt{etats} \times K \times \mathtt{etats}$ à la relation de transition désignée par $\mathtt{tr.CTS}$. À l'initialisation de l'algorithme, $\mathtt{CTS}$ ne contient que l'état initial et aucune transition ne lui est encore associée. On appelle $\mathcal{S}$ l'ensemble des états du CTS dont on doit extraire les configurations minimales ; $\mathcal{S}$ vaut initialement $\{\mathcal{U}_\Delta(p)\}$. Pour chaque $\mathcal{E} \in \mathcal{S}$, on construit l'ensemble E_K^1 des validations directement accessibles, *i.e* les événements dont les étiquettes sont dans K et qui ne sont précédés d'aucune autre validation. Pour chacune de ces validations e, l'algorithme fait appel à la fonction $min\_conf(\mathcal{E}, e)$ qui construit l'ensemble des configurations minimales contenant e. Pour réaliser cette fonction on procède comme pour l'exemple 5.2.1 : on commence par supposer qu'on a un ensemble X contenant e et on ajoute progressivement les événements qui sont nécessaires pour que X soit une configuration. Lorsqu'un choix dans un ensemble $E = \{e_1, \ldots, e_n\}$ se présente pour la continuation de X, on construit les plus grands sous-ensembles $E_1, \ldots, E_k$ de E tels que $X \cup E_i$

reste compatible avec la définition 5.1.2 et on poursuit avec la construction des configurations incompatibles $X \cup E_i, \ldots, X \cup E_k$. Si aucune continuation pour X n'est possible et que X n'est pas close à gauche, alors on jette X. On obtient au final un ensemble de configurations $\mathcal{C} \subseteq \mathsf{Conf}_T(\mathcal{E})$, tel que $\mathcal{C} = \{X \mid \forall e \in X : e \prec^* k(X)\}$. Par le lemme 5.3.2, cet ensemble correspond précisément aux traces transactionnelles issues de p. On ajoute alors au CTS de p les transactions correspondant à chacune des configurations ainsi construites. On vérifie enfin si le résidu de $\mathcal{E}$ par chacune de ces configurations est déjà présent dans les états du CTS. Si tel n'est pas le cas, on l'ajoute dans l'ensemble $\mathcal{S}$ des états à étudier. Une fois que toutes les validations éligibles de $\mathcal{E}$ ont été traitées, on ajoute $\mathcal{E}$ aux états du CTS et on reprend en 1. tant qu'il reste des résidus dans $\mathcal{S}$.

5.4 Discussion.

Afin de définir une méthode constructive pour extraire un système de transitions causales, nous avons utilisé le déploiement d'un processus dans une structure d'événements. De nombreuses approches antérieures ont exploité des techniques similaires, basées sur des réseaux de Petri, afin de définir des méthodes efficaces de vérification[10]. Dans le cadre des algèbres de processus, l'opération de déploiement est moins immédiate en raison du besoin d'exprimer la composition de processus de façon modulaire. Dans cette optique, les structures de flot offrent le double avantage de se composer de façon relativement simple et de ne pas induire d'explosion dans le nombre d'événements à considérer après composition. Toutefois, les objets obtenus après déploiement sont des structures *a priori* infinies lorsque le processus initial contient des appels à des définitions récursives. Ce dernier point constitue un problème bien connu qui est classiquement traité en travaillant sur un *préfixe* fini de la structure d'événements, obtenu en stoppant le déploiement du processus initial lorsqu'on est certain qu'aucune nouvelle propriété ne pourra être observée[11]. Ce type de coupure nécessite, en principe, que le processus déployé soit à états finis. Nous verrons comment traiter ce point clef, ainsi que plusieurs aspects d'implémentation dans le chapitre suivant.

[10] [McMillan, 1992, Esparza, 1994, Abdulla *et al.*, 2000, Baldan *et al.*, 1999]
[11] [McMillan, 1992, Penczek, 1997, Langerak et Brinksma, 1999, Esparza *et al.*, 2002]

Chapitre 6

Un outil de vérification automatique.

Sommaire

Nous avons vu, au chapitre précédent, qu'il est possible d'extraire le système de transitions causales d'un processus en tirant profit d'une sémantique de CCS basée sur les structures d'événements. Plus précisément, à supposer qu'on cherche à construire le système de transitions causales d'un processus p, nous avons défini un algorithme permettant de générer un système de transitions isomorphe, dont les états sont des structures d'événements et dont les transitions sont étiquetées par des configurations particulières, dites *minimales*. Ces dernières sont caractérisées par la propriété suivante : un événement d'une configuration minimale est, soit l'unique validation de la configuration, soit un prédécesseur de cette validation pour la clôture transitive de la relation de flot (*cf.* Lemme 5.3.2). Générer le CTS de p revient donc à construire pour chaque validation de p, l'ensemble des configurations minimales les contenant. L'intérêt de la méthode décrite est de générer les transactions d'un processus sans passer par des traces qui ne seraient pas transactionnelles où qui ne produiraient pas, *in fine*, de validation. Le pro-

blème qui nous reste à résoudre, afin d'implémenter concrètement l'algorithme de la figure 5.3, est que les structures sur lesquelles notre procédure doit s'exécuter sont des objets potentiellement infinis. Dans ce chapitre nous définissons dans une première partie les *structures de flot récursives* qui correspondent à des préfixes finis de structures de flot. Nous verrons que sous certaines conditions et en utilisant ces structures récursives, l'algorithme du chapitre précédent termine lorsque le CTS du processus qu'on cherche à analyser est fini. Nous étudierons ensuite l'articulation générale de l'implémentation de cet algorithme réalisée dans un module Ocaml, ainsi qu'un banc d'essai permettant de comparer la méthode déclarative et la méthode directe, correspondant respectivement aux chemins de gauche et de droite de la figure 5.1.

6.1 Structures de flot récursives.

6.1.1 Un déploiement borné.

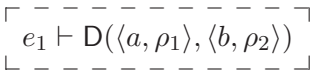
La procédure de déploiement d'un processus en sa structure de flot génère des objets de taille non bornée lorsque les constantes de récursion du processus se déroulent à l'infini. Pour éviter ce phénomène, nous procédons à la définition des structures de flot, dites *récursives*, dans lesquelles les constantes gardées par une action ne sont pas déployées. Les structures de flot récursives (FES_{rec}) sont des FES particulières dont certains événements, que nous appelons *contractions*, sont associés à des appels récursifs. On notera $e \vdash \mathsf{D}(\tilde{x})$ pour dénoter la contraction e associée à $\mathsf{D}(\tilde{x})$. Une FES_{rec} est dite *stable* lorsque toutes ses contractions ont au moins un prédécesseur pour la relation de flot. Si tel n'est pas le cas on construit une nouvelle FES_{rec} en remplaçant chaque contraction par la structure (récursive) qui lui est associée. Ce procédé converge vers une FES_{rec} stable, dans le fragment de CCS à récursions gardées. Dans ce chapitre, nous considérons donc les processus CCS construits sur la syntaxe donnée Figure 6.1. Nous verrons

$$\begin{array}{lcl} p & ::= & \alpha.p_d \mid p+p \mid p\backslash x \mid 0 \\ p_d & ::= & \mathsf{D}(\tilde{x}) \mid (p_d \parallel p_d) \mid p \end{array}$$

FIG. 6.1 – Processus CCS à récursions gardées.

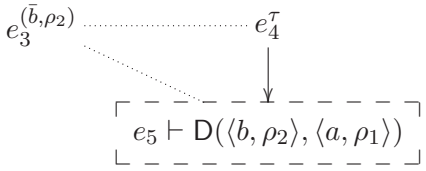
que l'opération de déploiement des contractions est similaire à une opéra-

tion de raffinage (*refinement*) sur des événements minimaux de la structure de flot. Toutefois, une différence notable avec les raffinages traditionnels est que l'instanciation d'une contraction $e \vdash \mathsf{D}(\tilde{x})$, avec $\mathsf{D}(\tilde{x}) := p$, par une structure de flot issue du déploiement de p, peut engendrer de nouvelles interactions avec le contexte e. Autrement dit, le déploiement des contractions n'est pas isolé du reste de la structure, ce qui est plutôt contraire à la logique des raffinages[1]. Avant de passer à la description formelle des FES$_{rec}$, nous allons construire un exemple simple de déploiement borné de processus. Soit $p = (\mathsf{D}(a,b) \parallel a.0)\backslash a\backslash b$ avec $(\mathsf{D}(x,y) := \bar{y} + \tau.\mathsf{D}(y,x)) \in \Delta$. Dans la FES$_{rec}$ correspondant à p, on utilise la contraction e_1 pour obtenir la structure récursive :



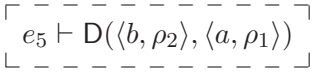
(6.1)

dans laquelle nous avons encadré la contraction. La liste $(\langle a,\rho_1\rangle, \langle b,\rho_2\rangle)$ sera utilisée pour donner un identifiant de restriction correct aux événements issus de l'expansion future de e_1. La FES$_{rec}$ obtenue est instable mais nous verrons qu'elle est en fait équivalente (après expansion) à la structure stable :



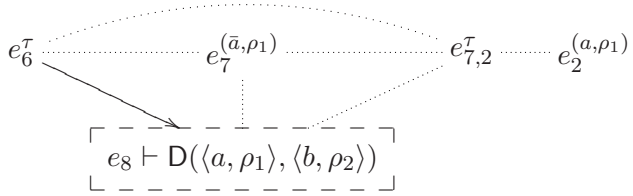
(6.2)

L'expansion de e_1 créé les événements e_3 et e_4, ainsi que la nouvelle contraction e_5. La restriction du canal b dans e_3 est déduite de la liste $(\langle a,\rho_1\rangle, \langle b,\rho_2\rangle)$ associée à e_1. La structure (6.2) est stable et ne peut plus être sujette à expansion. On peut maintenant calculer sur la structure et obtenir le résidu par la configuration $\{e_4^{\tau}\}$ qui correspond à la structure instable :



(6.3)

Cette structure récursive n'est pas isomorphe à (6.1) car l'ordre des paramètres de D diffère. Après expansion de e_5, cette structure devient :



(6.4)

[1] [Aceto, 1992, Darondeau et Degano, 1993, van Glabeek et Goltz, 2003]

On constate que le déploiement de e_5 a créé une nouvelle interaction avec l'événement e_2 qui était jusqu'à présent neutralisé par la restriction ρ_1. La structure stable (6.4) possède deux configurations possibles : $\{e_6\}$ et $\{e_{7,2}\}$. Une transition utilisant la première aboutit à une structure isomorphe à (6.1) et en utilisant la seconde on aboutit au résidu vide. Nous avons donc construit un système de transitions causales fini (avec $K = \{a, b\}$) à 3 états stables (voir figure 6.2), à partir d'un processus dont le déploiement dans les FES est normalement infini.

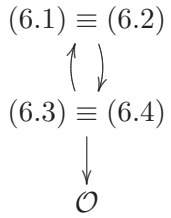


FIG. 6.2 – CTS à états dans les structures récursives.

Nous allons à présent nous employer à décrire cette procédure de façon formelle. Nous définissons, en premier lieu, les structures récursives puis l'opération d'expansion sur les événements de contractions.

Définition 6.1.1 (Structure de flot récursive) *Une* structure de flot récursive *est un 5-uplet, $\mathcal{E}_\Delta := \langle E, \prec, \#, \lambda, \vdash \rangle$, muni d'un ensemble de définitions récursives Δ, tels que :*
— E est un ensemble fini d'événements partitionné en deux sous-ensembles disjoints E_{act} et E_{def} ;
— $\prec \subseteq E_{act} \times E$ est la relation de flot ;
— $\# \subseteq E \times E$ est la relation de conflit ;
— $\lambda : E_{act} \to A \times \mathcal{R}_\perp$, est la fonction d'étiquetage ;
— $\vdash \subseteq E_{def} \times \mathcal{D}$, est la relation d'expansion*, où les éléments de $\mathcal{D}$ sont de la forme $\mathsf{D}(\langle x_1, \rho_1 \rangle, \ldots, \langle x_n, \rho_n \rangle)$, avec $\big(\mathsf{D}(\tilde{x}) := p\big) \in \Delta$ pour un certain processus p.*

La relation d'expansion $e \vdash \mathsf{D}(\langle x_1, \rho_1 \rangle, \ldots, \langle x_n, \rho_n \rangle)$ indique que e est associé à la constante de récursion $\mathsf{D}(\tilde{x})$ dont le $j^{\text{ème}}$ argument est restreint par $\rho_j \in \mathcal{R}_\perp$. Par la suite on utilisera la méta-variable $\tilde{w}$ pour dénoter une liste de couples (paramètre, identifiant) de la forme $(\langle x_1, \rho_1 \rangle, \ldots, \langle x_n, \rho_n \rangle)$ et on notera $\tilde{x}$ pour la liste de paramètres $(x_1, \ldots, x_n)$ sans identifiants de restriction.

6.1.2 Opérations sur les FES_{rec}.

Les opérations sur les FES_{rec} sont similaires aux opérations sur les structures de flot classiques. On traite la récursion en lui associant une contraction e *via* la fonction d'expansion. Noter que λ n'est alors pas défini sur e.
— (**Récursion**) Soit $\big(\mathsf{D}(x_1,\ldots,x_n) := p\big) \in \Delta$. La structure de flot récursive correspondante possède un seul événement, qui est la contraction associée à $\mathsf{D}(\tilde{x})$:

$$\mathcal{C}ont\big(\mathsf{D}(x_1,\ldots,x_n)\big) := \langle \{e\}, \emptyset, \emptyset, \emptyset, \big\{\big(e, \mathsf{D}(\langle x_1,\bot\rangle,\ldots,\langle x_n,\bot\rangle)\big)\big\}\rangle \quad (6.5)$$

Dans une structure récursive $\mathcal{E}_\Delta$, la propagation d'un nouvel identifiant de restriction aux événements de $\mathcal{E}_\Delta$ se fait de façon distincte selon que l'on tombe sur un événement dans E_{act} ou une contraction dans E_{def}. Dans le premier cas on procède comme pour les structures de flot en mettant à jour la fonction d'étiquetage λ. Dans le deuxième cas, en supposant que la restriction porte sur le canal x et que la contraction soit $e \vdash \mathsf{D}(\tilde{w})$, on remplace $\langle x,\bot\rangle \in \tilde{w}$ par $\langle x,\rho\rangle$ où ρ est le nouvel identifiant de restriction. La définition de l'opération de restriction est la suivante :
— (**Restriction**) Soient $\mathcal{E}_\Delta = \langle E, \prec, \#, \lambda, \vdash\rangle$ et $\rho \in \mathcal{R}$ un identifiant de restriction frais. Étant donné un nom de canal $x \in X$, on définit la fonction d'étiquetage $\lambda^{x,\rho} : E \to A \times \mathcal{R}$ comme dans la section 5.2, *i.e* :

$$\begin{array}{rcl} \lambda_1^{x,\rho}(e) & = & \lambda_1(e) \\ \lambda_2^{x,\rho}(e) & = & \rho \text{ si } \lambda_1(e) \in \{x,\bar{x}\} \;\&\; \lambda_2(e) = \bot \\ & & \lambda_2(e) \text{ sinon.} \end{array}$$

La nouvelle relation d'expansion $\vdash^{x,\rho} \subseteq E_{def} \times \mathcal{D}$ est quant à elle donnée par :

$$e \vdash^{x,\rho} \mathsf{D}(\tilde{w}\,\{\langle x,\rho\rangle/\langle x,\bot\rangle\}) \quad \text{si } e \vdash \mathsf{D}(\tilde{w})$$

où $\tilde{w}\,\{\langle x,\rho\rangle/\langle x,\bot\rangle\}$ dénote la substitution du couple $\langle x,\bot\rangle$ par $\langle x,\rho\rangle$ dans $\tilde{w}$.
La *restriction* $\_\backslash\backslash\_ : \mathrm{FES}_{rec} \times (X \times \mathcal{R}) \to \mathrm{FES}_{rec}$ est donnée par :

$$\mathcal{E}_\Delta \backslash\backslash \langle x,\rho\rangle := \langle E, \prec, \#, \lambda^{x,\rho}, \vdash^{x,\rho}\rangle \quad (6.6)$$

Le préfixe, la somme et le produit sur les FES_{rec} sont de simples extensions des opérateurs sur les FES classiques : pour les opérations binaires $\mathcal{E} = \mathcal{E}_1 \oplus \mathcal{E}_2$ et $\mathcal{E} = \mathcal{E}_1 \otimes \mathcal{E}_2$, la relation d'expansion de $\mathcal{E}$ consiste simplement en l'union des relations d'expansions de $\mathcal{E}_1$ et $\mathcal{E}_2$. Pour l'opération $\alpha \odot \mathcal{E}$, la relation d'expansion reste inchangée. Nous pouvons à présent définir le déploiement partiel d'un processus CCS dans les structures de flot récursives :

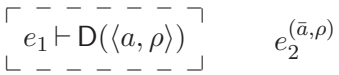
Définition 6.1.2 (Déploiement partiel) *Soit p un processus CCS, muni d'un ensemble de définitions Δ. La structure de flot récursive $\mathcal{U}^{rec}_{\Delta}(p)$ qui lui correspond est définie inductivement par :*

$$\begin{array}{lcl}
\mathcal{U}^{rec}_{\Delta}(0) & := & \mathcal{O} \\
\mathcal{U}^{rec}_{\Delta}(\mathsf{D}(\tilde{x})) & := & \mathcal{C}ont\big(\mathsf{D}(\tilde{x})\big) \\
\mathcal{U}^{rec}_{\Delta}(p\backslash x) & := & \mathcal{U}^{rec}_{\Delta}(p)\lVert\langle x,\rho\rangle \text{ pour un } \rho \text{ frais dans } \mathcal{U}^{rec}_{\Delta}(p). \\
\mathcal{U}^{rec}_{\Delta}(a.p) & := & a \odot \mathcal{U}^{rec}_{\Delta}(p) \\
\mathcal{U}^{rec}_{\Delta}(p+q) & := & \mathcal{U}^{rec}_{\Delta}(p) \oplus \mathcal{U}^{rec}_{\Delta}(q) \\
\mathcal{U}^{rec}_{\Delta}(p \parallel q) & := & \mathcal{U}^{rec}_{\Delta}(p) \otimes \mathcal{U}^{rec}_{\Delta}(q)
\end{array}$$

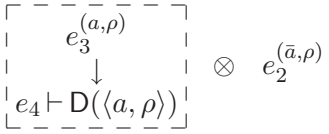
On rappelle qu'une structure récursive $\langle E, \prec, \#, \lambda, \vdash\rangle$ est stable si aucun élément de E_{def} n'est minimal pour la relation de flot, *i.e* :

$$\forall e \in E, \forall e' \in E_{act} : e' \prec e \Rightarrow e \notin E_{def}$$

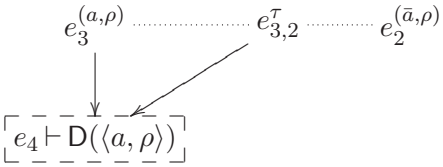
Étant donnée une structure instable, on définit une opération sur les contractions non gardées qui consiste à les remplacer par les structures d'événements correspondant aux expansions des définitions récursives. Cette opération est en fait un simple produit entre la structure d'événements contractée et le contexte de la contraction. À titre d'exemple, considérons le processus $p := (\mathsf{D}(a) \parallel \bar{a})\backslash a$ avec $\mathsf{D}(x) := x.\mathsf{D}(x)$. En utilisant le déploiement partiel $\mathcal{U}^{rec}_{\Delta}(p)$, on construit la structure récursive instable :



On obtient une structure d'événements stable en remplaçant e_1 par le déploiement $\mathcal{U}^{rec}_{\Delta}(a.\mathsf{D}(a))$ (encadré par des tirets dans le schéma) et en le composant avec son contexte e_2, c'est à dire :



ou encore :



Noter que e_3 et e_2 sont bien dans le même champ de restriction ρ, ce qui entraîne la création de l'événement de synchronisation $e_{3,2}$ conformément à

la définition du produit de structures de flot (5.5). La procédure que nous venons de suivre correspond à *l'expansion* de la structure récursive initiale et nous procédons maintenant à sa définition formelle. L'expansion d'une contraction est obtenue par application de la fonction $\xi : E_{def} \to \mathrm{FES}_{rec}$, définie par :

$$\xi(e) := \mathcal{U}^{rec}_{\Delta}(p)\backslash\backslash\tilde{w} \;\text{ si }\; e \vdash \mathsf{D}(\tilde{w}) \;\&\; \big(\mathsf{D}(\tilde{x}) := p\big) \in \Delta$$

en considérant que les ρ_i de $\tilde{w}$ n'apparaissent pas dans $\mathcal{U}^{rec}_{\Delta}(p)$. Par exemple, supposons que $e \vdash \mathsf{D}(\langle x, \bot\rangle, \langle y, \rho\rangle)$ et $\mathsf{D}(x, y) := (x.y.\mathsf{D}(x, y))\backslash x$. On obtient :

$$\mathcal{U}^{rec}_{\Delta}(p) := \quad e_1^{(x,\rho')} \longrightarrow e_2^{(y,\bot)} \longrightarrow \boxed{e_3 \vdash \mathsf{D}(\langle x, \rho'\rangle, \langle y, \bot\rangle)}$$

puis :

$$\xi(e) = \mathcal{U}^{rec}_{\Delta}(p)\backslash\backslash\langle y, \rho\rangle := \quad e_1^{(x,\rho')} \longrightarrow e_2^{(y,\rho)} \longrightarrow \boxed{e_3 \vdash \mathsf{D}(\langle x, \rho'\rangle, \langle y, \rho\rangle)}$$

Nous allons maintenant décrire comment l'expansion d'un événement de E_{def} se compose avec le contexte de ce dernier. Soient $\mathcal{E}_\Delta = \langle E, \prec, \#, \lambda, \vdash\rangle$ une structure de flot récursive et $E^{\downarrow}_{def} \subseteq E_{def}$ l'ensemble des contractions n'ayant pas de prédécesseur pour la relation de flot. Pour tout $e \in E^{\downarrow}_{def}$, on utilise :

$$\mathcal{E}_\Delta - e := \langle E', \prec', \#', \lambda', \vdash'\rangle$$

pour dénoter la structure résiduelle telle que $E' := E - \{e\}$ et $\prec' := \prec \cap (E' \times E')$, $\#' := \# \cap (E' \times E')$, $\lambda' := (\lambda \restriction E')$ et $\vdash' := \vdash \cap (E' \times \mathcal{D})$. On définit alors l'opération d'expansion sur les structures de flot de la façon suivante :

Définition 6.1.3 (Expansion) *Soient $\mathcal{E}_\Delta$ et $\mathcal{E}'_\Delta$ deux FES$_{rec}$, $\mathcal{E}'_\Delta$ est une* expansion *de $\mathcal{E}_\Delta$, notée $\mathcal{E}_\Delta \ll \mathcal{E}'_\Delta$, si :*

$$\exists e \in E^{\downarrow}_{def} : \mathcal{E}'_\Delta = (\mathcal{E}_\Delta - e) \otimes \xi(e)$$

Nous allons voir à présent que la relation d'expansion permet d'obtenir une forme normale pour les FES$_{rec}$ dans le fragment de CCS à récursions gardées (voir Figure 6.1).

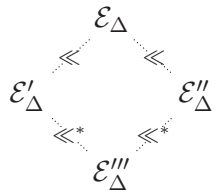
Proposition 6.1.1 *Soit $\mathcal{E}_\Delta$ une FES$_{rec}$ à récursions gardées. Il existe une unique $\mathcal{E}'_\Delta$ telle que $\mathcal{E}_\Delta \ll^* \mathcal{E}'_\Delta \not\ll$.*

On prouve la proposition en montrant la terminaison et la confluence locale.

Lemme 6.1.1 (Terminaison) *Soient $\mathcal{E}_\Delta$ une FES$_{rec}$, munie d'un ensemble de définitions gardées et $\ll^*$ la clôture réflexive et transitive de $\ll$. Il existe une structure stable $\mathcal{E}'_\Delta$ telle que $\mathcal{E}_\Delta \ll^* \mathcal{E}'_\Delta$.*

Supposons que $\mathcal{E}_\Delta$ ne soit pas stable. On montre le lemme par induction sur le nombre de contractions non gardées de $\mathcal{E}_\Delta$. Soit $e \vdash \mathsf{D}(\tilde{w})$ l'une de ces contractions et supposons $\mathsf{D}(\tilde{x}) := \alpha.p \in \Delta$. On procède à l'expansion $\xi(e) = \mathcal{U}^{rec}_\Delta(\alpha.p)\backslash\backslash\tilde{w}$ qui vaut $\big(\alpha \odot \mathcal{U}^{rec}_\Delta(p)\big)\backslash\backslash\tilde{w}$ selon la définition 6.1.2. Par construction, l'opérateur de préfixe ne produit que des structures stables et la stabilité est préservée par l'opérateur de restriction. Il nous reste a effectuer le produit $\xi(e) \otimes (\mathcal{E}_\Delta - e)$ qui possède bien une contraction non gardée de moins que $\mathcal{E}_\Delta$. On conclut par hypothèse d'induction. □

Lemme 6.1.2 (Confluence locale) *Le processus d'expansion d'une FES$_{rec}$ est localement confluant, i.e on a le diagramme suivant :*



LEMME 6.1.2 – Confluence locale de l'expansion.

– On montre tout d'abord que $\otimes$ est une opération associative et commutative. Soient $\mathcal{E}_1, \mathcal{E}_2$ et $\mathcal{E}_3$ des structures de flot. Soient $\mathcal{E}_{1(23)} := \mathcal{E}_1 \otimes (\mathcal{E}_2 \otimes \mathcal{E}_3)$ et $\mathcal{E}_{(12)3} := (\mathcal{E}_1 \otimes \mathcal{E}_2) \otimes \mathcal{E}_3$. Par construction, ces structures ont le même ensemble d'événements $E_{123} := \bigcup_{i\in\{1,2,3\}} E_i \cup (E_1 \times_* E_2) \cup (E_1 \times_* E_3) \cup (E_2 \times_* E_3)$. On pose $\mathcal{E}_i = \langle E_i, \prec_i, \#_i, \lambda_i, \vdash_i \rangle$. Si $e \prec_{1(23)} e'$ alors il existe une projection $e_i \in E_i$ de e et une projection $e'_i \in E_i$ de e' tel que $e_i \prec_i e'_i$, ce qui entraîne aussi $e' \prec_{(12)3} e'$. De même, supposons $e \#_{1(23)} e'$. Ou bien deux des projections de e et e' étaient en conflit dans E_i et on en déduit que $e \#_{(12)3} e'$, ou bien e et e' coïncident sur une projection et on a aussi $e \#_{(12)3} e'$. Toujours par construction, $\lambda_{1(23)}$ et $\lambda_{(12)3}$ coïncident sur les événements de synchronisation car $\lambda_{1(23)}(e) = \lambda_{(12)3}(e) = (\tau, \bot)$. Sur tous les autres événements on a $\lambda_{1(23)} = \lambda_1 \circ (\lambda_2 \circ \lambda_3)$ et $\lambda_{(12)3} = (\lambda_1 \circ \lambda_2) \circ \lambda_3$. Les E_i étant supposés disjoints, la composition des fonctions d'étiquetage est associative et $\lambda_{1(23)} = \lambda_{(12)3}$. Enfin, les relations d'expansion $\vdash_{1(23)} = \vdash_1 \cup (\vdash_2 \cup \vdash_3)$ et $\vdash_{(12)3} = (\vdash_1 \cup \vdash_2) \cup \vdash_3$ coïncident aussi trivialement. On conclut donc que le produit sur les FES$_{rec}$ est bien associatif, la commutativité étant quant à elle immédiate.
– Supposons à présent que $\mathcal{E}'_\Delta$ et $\mathcal{E}''_\Delta$ résultent des expansions de $e_1 \in E^{\downarrow}_{def}$

et $e_2 \in E^{\downarrow}_{def}$. Par définition on a :

$$\begin{array}{rcl} \mathcal{E}'_\Delta & = & (\mathcal{E}_\Delta - e_1) \otimes \xi(e_1) \\ \mathcal{E}''_\Delta & = & (\mathcal{E}_\Delta - e_2) \otimes \xi(e_2) \end{array}$$

La structure $\mathcal{E}'_\Delta$ ne peut être stable car elle contient toujours la contraction e_2 qui était non gardée dans $\mathcal{E}_\Delta$. Il en va de même pour $\mathcal{E}''_\Delta$ qui contient toujours e_1. On procède donc aux expansions de $\mathcal{E}'_\Delta$ et $\mathcal{E}''_\Delta$ et on obtient :

$$\begin{array}{rcl} \mathcal{E}'_\Delta & \ll & \big((\mathcal{E}_\Delta - e_1) \otimes \xi(e_1) - e_2\big) \otimes \xi(e_2) \\ \mathcal{E}''_\Delta & \ll & \big((\mathcal{E}_\Delta - e_2) \otimes \xi(e_2) - e_1\big) \otimes \xi(e_1) \end{array}$$

Les contractions e_1 et e_2 n'intervenant pas dans le produit on peut les retirer en même temps et on a :

$$\begin{array}{rcl} \mathcal{E}'_\Delta & \ll & \big((\mathcal{E}_\Delta - e_1 - e_2) \otimes \xi(e_1)\big) \otimes \xi(e_2) \\ \mathcal{E}''_\Delta & \ll & \big((\mathcal{E}_\Delta - e_2 - e_1) \otimes \xi(e_2)\big) \otimes \xi(e_1) \end{array}$$

on en déduit que les structures sont égales par associativité et commutativité du produit de structures. □

À tout déploiement partiel d'un processus p, à récursions gardées, on peut maintenant associer une FES_{rec} stable construite par une série maximale d'expansions effectuées à partir de $\mathcal{U}^{rec}(p)$. On note $\mathsf{fn}(\mathcal{E}_\Delta)$ la forme normale de $\mathcal{E}_\Delta$ ainsi obtenue et on définit la relation d'équivalence $\equiv$ sur les FES_{rec} par :

$$\mathcal{E}_\Delta \equiv \mathcal{E}'_\Delta \iff \mathsf{fn}(\mathcal{E}_\Delta) = \mathsf{fn}(\mathcal{E}'_\Delta) \tag{6.7}$$

où l'égalité entre structures d'événements est considérée à isomorphisme près. Nous terminons cette section pas la définition du système de transitions associé aux FES_{rec}. Soit $\mathbb{E}$ l'ensemble de FES_{rec} stables. La relation de transition entre structures récursives est la plus petite relation satisfaisant :

$$\frac{\mathcal{E}_\Delta \in \mathbb{E} \quad X \in \mathsf{Conf}(\mathcal{E}_\Delta)}{\mathcal{E}_\Delta \xrightarrow{X} (\mathcal{E}_\Delta | X)}\ (1) \qquad \frac{\mathcal{E}_\Delta \equiv \mathcal{E}''_\Delta \xrightarrow{X} \mathcal{E}'''_\Delta \equiv \mathcal{E}'_\Delta}{\mathcal{E}_\Delta \xrightarrow{X} \mathcal{E}'_\Delta}\ (2)$$

On notera que $\mathcal{E}'_\Delta := \mathcal{E}_\Delta | X$, le résidu de la structure $\mathcal{E}_\Delta$ par la configuration X, n'est pas nécessairement stable. Toutefois, si Δ est à récursions gardées on sait que $\mathcal{E}'_\Delta$ est équivalente à une FES_{rec} stable et on peut alors utiliser la règle (2). Comme pour les FES, la relation de transition causale, nécessaire à la construction du CTS, est définie par :

$$\mathcal{E}_\Delta \stackrel{k(X)}{\Longrightarrow} \mathcal{E}'_\Delta \iff \mathcal{E}_\Delta \xrightarrow{X} \mathcal{E}'_\Delta \ \&\ X \in \mathsf{Conf}_T(\mathcal{E}_\Delta) \tag{6.8}$$

Nous allons maintenant montrer que système de transitions sur les FES_{rec} coïncide bien avec le LTS de CCS. Posons $\mathcal{E}_\Delta \xrightarrow{\alpha} \mathcal{E}'_\Delta$ si $\mathcal{E}_\Delta \xrightarrow{\{e\}} (\mathcal{E}_\Delta | \{e\})$ avec $\lambda(e) = \alpha$ et $\mathcal{E}'_\Delta = (\mathcal{E}_\Delta | \{e\})$. On a le théorème suivant :

Théorème 6.1.1 *Les systèmes de transitions engendrés par un processus p et son déploiement $\mathcal{U}^{rec}_\Delta(p)$ sont fortement bisimilaires.*

Nous allons montrer que la relation $\sim := \{\langle \mathcal{E}_\Delta, p\rangle \mid \mathcal{E}_\Delta \equiv \mathcal{U}_\Delta(p)\}$ est une bisimulation. On commence par montrer un lemme de simulation :

Lemme 6.1.3 *Pour tout processus p, on a le diagramme suivant :*

$$\begin{array}{ccc} p & \xrightarrow{\quad\alpha\quad} & q \\ \downarrow & & \vdots \\ \sim & & \sim \\ \vdots & & \vdots \\ \mathcal{U}^{rec}_\Delta(p) & \overset{\alpha}{\cdots\cdots\succ} & \mathcal{U}^{rec}_\Delta(q) \end{array}$$

Lemme 6.1.3 – Simulation.

On prouve le lemme par induction sur l'arbre de dérivation de $p \xrightarrow{\alpha} q$ dans le LTS de CCS (Figure 1.2).

– Dans le cas de base on suppose $\frac{}{\alpha.p+q \xrightarrow{\alpha} p}$.
Par construction, la structure $\mathcal{U}^{rec}_\Delta(\alpha.p+q)$ possède un événement minimal e_α tel que $\lambda(e_\alpha) = (\alpha, \perp)$. Par conséquent $\{e_\alpha\}$ est bien une configuration de $\mathcal{U}^{rec}_\Delta(\alpha.p+q)$ et on en déduit $\mathcal{U}^{rec}_\Delta(\alpha.p+q) \xrightarrow{\alpha} \mathcal{U}^{rec}_\Delta(\alpha.p+q)|\{e_\alpha\}$ avec, par application de la définition 5.3.2 :

$$\mathcal{U}^{rec}_\Delta(\alpha.p+q)|\{e_\alpha\} = \mathcal{U}^{rec}_\Delta(\alpha.p+q) - \big(\{e_\alpha\} \cup \{e \in \mathcal{U}^{rec}_\Delta(\alpha.p+q) \mid e \# e_\alpha\}\big)$$

Toujours par construction, $\mathcal{U}^{rec}_\Delta(\alpha.p+q) = \mathcal{U}^{rec}_\Delta(\alpha.p) \oplus \mathcal{U}^{rec}_\Delta(q)$ et les événements de $\mathcal{U}^{rec}_\Delta(q)$ sont en conflit avec ceux de $\mathcal{U}^{rec}_\Delta(\alpha.p)$ et en particulier avec e_α. Ils sont donc tous supprimés dans le résidu et on obtient $\mathcal{U}^{rec}_\Delta(\alpha.p+q)|\{e_\alpha\} = \mathcal{U}^{rec}_\Delta(\alpha.p)|\{e_\alpha\} = \alpha \odot \mathcal{U}^{rec}_\Delta(p)|\{e_\alpha\}$. Comme aucun événement de $\mathcal{U}^{rec}_\Delta(p)$ ne peut être en conflit avec e_α, ils sont donc toujours présents dans le résidu et on obtient $\mathcal{U}^{rec}_\Delta(\alpha.p+q)|\{e_\alpha\} = \mathcal{U}^{rec}_\Delta(p)$ qui est bien en relation avec p.

– Supposons $\frac{p \xrightarrow{\alpha} p'}{p \| q \xrightarrow{\alpha} p' \| q}$.
Par Hypothèse d'induction, $\mathcal{U}^{rec}_\Delta(p) \xrightarrow{\alpha} \mathcal{U}^{rec}_\Delta(p') = \mathcal{U}^{rec}_\Delta(p)|\{e_\alpha\}$; on doit montrer :

$$\mathcal{U}^{rec}_\Delta(p) \otimes \mathcal{U}^{rec}_\Delta(q) \xrightarrow{\alpha} \mathcal{U}^{rec}_\Delta(p') \otimes \mathcal{U}^{rec}_\Delta(q)$$

Dans $(\mathcal{U}_\Delta^{rec}(p)|\{e_\alpha\}) \otimes \mathcal{U}_\Delta^{rec}(q)$ on peut faire commuter l'opération de résidu avec l'application du produit $\otimes$. En effet, les événements en conflits avec e_α sont soit une synchronisation utilisant e_α, auquel cas ils sont bien supprimés dans les deux structures, soit déjà en conflit dans $\mathcal{U}_\Delta^{rec}(p)$ et ils sont supprimés aussi bien dans le résidu de $\mathcal{U}_\Delta^{rec}(p)$ ou de $\mathcal{U}_\Delta^{rec}(p) \otimes \mathcal{U}_\Delta^{rec}(q)$. On obtient $(\mathcal{U}_\Delta^{rec}(p) \otimes \mathcal{U}_\Delta^{rec}(q))|\{e_\alpha\} = \mathcal{U}_\Delta^{rec}(p)|\{e_\alpha\} \otimes \mathcal{U}_\Delta^{rec}(q) = \mathcal{U}_\Delta^{rec}(p' \parallel q)$.

– Supposons $\frac{p \xrightarrow{\alpha} p'}{p\backslash x \xrightarrow{\alpha} p'\backslash x}$.
Par hypothèse d'induction $\mathcal{U}_\Delta^{rec}(p) \xrightarrow{\alpha} \mathcal{U}_\Delta^{rec}(p') = \mathcal{U}_\Delta^{rec}(p)|\{e_\alpha\}$. D'après la condition de bord $\alpha \notin \{x, \bar{x}\}$ et donc $\lambda(e_\alpha) = \langle \alpha, \bot \rangle$ avec $\alpha \neq x$. On en déduit $\mathcal{U}_\Delta^{rec}(p) \backslash\backslash \langle x, \rho \rangle \xrightarrow{\alpha} \mathcal{U}_\Delta^{rec}(p') \backslash \langle x, \rho \rangle$.

– Supposons $\frac{p \xrightarrow{\bar{a}} p' \quad q \xrightarrow{a} q'}{p \| q \xrightarrow{\tau} p' \| q'}$.
Par hypothèse d'induction $\mathcal{U}_\Delta^{rec}(p) \xrightarrow{\bar{a}} \mathcal{U}_\Delta^{rec}(p')$ et $\mathcal{U}_\Delta^{rec}(q) \xrightarrow{a} \mathcal{U}_\Delta^{rec}(q')$. Soit $e_{\bar{a}}$ et e_a les événements impliqués dans les transitions. Le produit $\mathcal{U}_\Delta^{rec}(p) \otimes \mathcal{U}_\Delta^{rec}(q)$ aboutit à la création d'un événement $(e_{\bar{a}}, e_a)$ qui est en conflit avec $e_{\bar{a}}$ et e_a. Les deux événements disparaissent donc dans le résidu et on obtient bien $\mathcal{U}_\Delta^{rec}(p \parallel q) \xrightarrow{\tau} \mathcal{U}_\Delta^{rec}(p' \parallel q')$.

– Enfin, supposons $\frac{\mathsf{D}(\tilde{x}) := p \xrightarrow{\alpha} p'}{\mathsf{D}(\tilde{x}) \xrightarrow{\alpha} p'}$, qui est le cas intéressant pour la règle (equiv).
Toujours par hypothèse d'induction, $\mathcal{U}_\Delta^{rec}(p) \xrightarrow{\alpha} \mathcal{U}_\Delta^{rec}(p')$. On note ensuite que $\mathcal{U}_\Delta^{rec}(\mathsf{D}(\tilde{x}))$ est équivalent à son expansion $\mathcal{U}_\Delta^{rec}(p) \otimes \mathcal{O}$ dans un contexte vide et on obtient $\mathcal{U}_\Delta^{rec}(p) \otimes \mathcal{O} \xrightarrow{\alpha} \mathcal{U}_\Delta^{rec}(p')$. □

Le système de transitions sur les FES_{rec} n'étant pas construit sur un format compositionnel (SOS), la deuxième partie de la preuve du théorème 6.1.1 est plus délicate car on ne peut plus raisonner par simple induction. Nous allons donc établir quelques propriétés de passage au contexte du système de transition sur les FES_{rec} qui vont nous permettre de simplifier la preuve de la deuxième partie du théorème. En premier lieu, on note que l'application de l'opérateur de restriction sur les FES_{rec} peut toujours être retardée.

Propriété 6.1.1 *Pour toute configuration X telle que $\forall e \in X : \lambda_1(e) \notin \{x, \bar{x}\}$, $(\mathcal{E}_\Delta|X) \backslash\backslash \langle x, \rho \rangle = (\mathcal{E}_\Delta \backslash\backslash \langle x, \rho \rangle)|X$.*

Il suffit de noter que la construction du résidu $\mathcal{E}_\Delta|X$ est indépendante des identifiants de restriction des événements de $\mathcal{E}_\Delta$. □

On vérifie ensuite que la congruence structurelle pour CCS est correctement traduite dans les FES_{rec}.

Propriété 6.1.2 *Si $p \equiv q$ alors $\mathcal{U}^{rec}_{\Delta}(p) \equiv \mathcal{U}^{rec}_{\Delta}(q)$.*

Supposons $p\backslash x \parallel q \equiv (p \parallel q)\backslash x$ et posons $\mathcal{E}_{\Delta} := \mathcal{U}^{rec}_{\Delta}(p\backslash x \parallel q)$. On a $\mathcal{E}_{\Delta} = (\mathcal{U}^{rec}_{\Delta}(p)\!\!\setminus\!\!\setminus\langle x, \rho\rangle) \otimes \mathcal{U}^{rec}_{\Delta}(q)$. Comme x n'apparaît pas libre dans q, tout événement e de $\mathcal{U}^{rec}_{\Delta}(q)$ est tel que $\lambda_1(e) \in \{x, \bar{x}\} \Rightarrow \lambda_2(e) \neq \perp$. On peut donc sans risque de capture faire commuter l'opérateur de restriction et on obtient $\mathcal{E}_{\Delta} = (\mathcal{U}^{rec}_{\Delta}(p) \otimes \mathcal{U}^{rec}_{\Delta}(q))\!\!\setminus\!\!\setminus\langle x, \rho\rangle = \mathcal{U}^{rec}_{\Delta}((p \parallel q)\backslash x)$. Supposons $p = \mathsf{D}(\tilde{x}) \equiv q$. Par expansion, on a $\mathcal{U}^{rec}_{\Delta}(\mathsf{D}(\tilde{w})) \equiv \mathcal{U}^{rec}_{\Delta}(q)\!\!\setminus\!\!\setminus\tilde{w}$ avec $\tilde{w} = (\langle x_1, \perp\rangle, \ldots, \langle x_n, \perp\rangle)$. On a donc $\mathcal{U}^{rec}_{\Delta}(q)\!\!\setminus\!\!\setminus\tilde{w} = \mathcal{U}^{rec}_{\Delta}(q)$. Il est enfin clair que le déploiement d'un processus est invariant pour les autres règles de congruence structurelle. □

Lemme 6.1.4 *Pour tout déploiement $\mathcal{U}^{rec}_{\Delta}(p)$, on a le diagramme suivant :*

$$\begin{array}{ccc} \mathcal{U}^{rec}_{\Delta}(p) & \xrightarrow{\alpha} & \mathcal{U}^{rec}_{\Delta}(q) \\ \downarrow \sim & & \sim \vdots \\ p & \xrightarrow[\alpha]{} & q \end{array}$$

Lemme 6.1.4 – Symétrie de la relation de simulation.

Pour étudier les différents cas possibles on se ramène à une forme homogène de p (voir Définition 2.1.1), c'est à dire qu'on "normalise" un processus p en une forme générique $(\prod_{i\in I} \sum_{j\in J_i} \alpha_{ij}.p_{ij})\backslash\tilde{x}$ qui lui est structurellement congruente. Les propriétés 6.1.1 et 6.1.2 nous permettent alors de ne considérer que des transitions de la forme $\mathcal{U}^{rec}_{\Delta}((\prod_{i\in I} \sum_{j\in J_i} \alpha_{ij}.p_{ij})\backslash\tilde{x}) \xrightarrow{\{e_\alpha\}} \mathcal{E}_{\Delta}$. On a :

$$\mathcal{U}^{rec}_{\Delta}(p) = \big(\bigotimes_{i\in I} \bigoplus_{j\in J_i} \alpha_{ij} \odot \mathcal{U}^{rec}_{\Delta}(p_{ij})\big)\!\!\setminus\!\!\setminus\langle x_1, \rho_1\rangle \ldots \!\!\setminus\!\!\setminus\langle x_n, \rho_n\rangle$$

Appelons e_{ij} les événements correspondant aux α_{ij}. Nécessairement, e_α est soit l'un de ces e_{ij} soit un événement de synchronisation de la forme (e_{kl}, e_{mn}). En effet, l'opération $\alpha_{ij} \odot \mathcal{U}^{rec}_{\Delta}(p_{ij})$ aboutit à un processus dont le seul événement minimal est e_{ij}. L'ensemble des événements minimaux de $\mathcal{U}^{rec}_{\Delta}(p)$ est alors composé des événements minimaux des $\mathcal{U}^{rec}_{\Delta}(p_{ij})$ et de toutes les synchronisations possibles entre ces événements. On cherche à évaluer :

$$\mathcal{E}_{\Delta} = \big(\bigotimes_{i\in I} \bigoplus_{j\in J_i} \alpha_{ij} \odot \mathcal{U}^{rec}_{\Delta}(p_{ij})\big)\!\!\setminus\!\!\setminus\tilde{w} \mid \{e_\alpha\}$$

Dans le cas le plus difficile e_α est une synchronisation et on peut poser $e_\alpha := (e_{kl}, e_{mn})$. Afin de construire le résidu de $\mathcal{U}^{rec}_\Delta(p)$ par $\{(e_{kl}, e_{mn})\}$ on doit enlever de $\mathcal{U}^{rec}_\Delta(p)$ l'ensemble $\{(e_{kl}, e_{mn})\}_\#$ des événements incompatibles avec (e_{kl}, e_{mn}). Appelons E_{ij} l'ensemble des événements de $\mathcal{U}^{rec}_\Delta(p_{ij})$, on a :

$$\{(e_{kl}, e_{mn})\}_\# = \big\{e \mid \forall j \neq l, \forall j' \neq n : \pi_i(e) \in E_{kj} \cup E_{mj'}\big\}$$

où $\forall i : \pi_i(e_{ij}, \star) = e_{ij}$ et $\pi_i(e_{ij}, e_{kl}) = e_{ij}$ si $i = 1$ et e_{kl} si $i = 2$. On construit le résidu :

$$\mathcal{E}_\Delta = \big(\mathcal{U}^{rec}_\Delta(p) - (e_{kl}, e_{mn})\big) \setminus \{(e_{kl}, e_{mn})\}_\#$$

On trouve :

$$\begin{aligned} \mathcal{E}_\Delta &= \big(\bigotimes_{i\in I\setminus\{k,m\}} \bigoplus_{j\in J_i} \alpha_{ij} \odot \mathcal{U}^{rec}_\Delta(p_{ij})\big) \otimes \mathcal{U}^{rec}_\Delta(p_{kl}) \otimes \mathcal{U}^{rec}_\Delta(p_{mn}) \lVert\!\rVert \tilde{w} \\ &= \mathcal{U}^{rec}_\Delta\big((\prod_{i\in I\setminus\{k,m\}} \sum_{j\in J_i} \alpha_{ij}.p_{ij} \parallel p_{kl} \parallel p_{mn}) \setminus \tilde{x}\big) \end{aligned}$$

Cette transition correspond bien à la transition dérivable dans le LTS de CCS :

$$(\prod_{i\in I} \sum_{j\in J_i} \alpha_{ij}.p_{ij}) \setminus \tilde{x} \xrightarrow{\tau} (\prod_{i\in I\setminus\{k,m\}} \sum_{j\in J_i} \alpha_{ij}.p_{ij} \parallel p_{kl} \parallel p_{mn}) \setminus \tilde{x}$$

Le cas où e_α n'est pas une synchronisation est traité de manière similaire. On a donc $\mathcal{U}^{rec}_\Delta(p) \xrightarrow{\alpha} \mathcal{E}_\Delta \Rightarrow p \xrightarrow{\alpha} p'$ & $\mathcal{U}^{rec}_\Delta(p') = \mathcal{E}_\Delta$ ce qui permet de conclure quant au lemme 6.1.4. La relation $\sim := \langle\langle \mathcal{E}_\Delta, p\rangle \mid \mathcal{U}^{rec}_\Delta(p) \equiv \mathcal{E}_\Delta\rangle$ est une simulation (Lemme 6.1.3) dont nous venons de montrer la symétrie : c'est donc bien une bisimulation. □

6.1.3 Algorithme sur les FES$_{rec}$.

On rappelle que l'ensemble des validations éligibles E^1_K de $\mathcal{E}$ est donné par les validations $e \in E_K$, pour lesquelles on peut trouver un ensemble silencieux X tel que $X \cup \{e\} \in \mathsf{Conf}(\mathcal{E})$. Pour chaque état $\mathcal{E}$ du CTS en construction, l'algorithme que nous avons présenté au chapitre précédent (figure 5.3), cherche à construire toutes les configurations minimales contenant une validation dans E^1_K. Dans le cadre des FES$_{rec}$, chaque état $\mathcal{E}_\Delta$ du CTS est une structure partiellement déployée, il faut donc avoir la garantie que tous les événements de E^1_K sont visibles dans $\mathsf{fn}(\mathcal{E}_\Delta)$ sans quoi on oublierait des validations éligibles cachées par un appel récursif. Nous allons donc nous restreindre aux processus CCS dans lesquels les définitions récursives sont précédées d'une validation, ce qui est un cas particulier de processus à récursions gardées (voir Figure 6.1). Nous pouvons alors être sûr que toutes les

validations éligibles sont immédiatement visibles dans les FES_{rec} obtenues par déploiement partiel des processus issus de cette syntaxe. En conservant les notations de la section 5.3.2, nous donnons, figure 6.3, l'algorithme permettant d'extraire le CTS d'un processus p en utilisant les structures de flot récursives de taille bornée au lieu des FES potentiellement infinies.

0. $\mathcal{E}_\Delta \leftarrow \mathcal{U}^{rec}_\Delta(p)$;
   $\texttt{etats.CTS} \leftarrow \{\mathcal{E}_\Delta\}$; $\texttt{tr.CTS} \leftarrow \emptyset$; $\mathcal{S} \leftarrow \{\mathcal{E}_\Delta\}$; //$\mathcal{S}$ contient les FES à analyser.
1. TANT QUE $\mathcal{S} \neq \emptyset$ FAIRE :
2. $\quad \mathcal{S} \leftarrow \mathcal{S} \setminus \{\mathcal{E}_\Delta = \langle E, \prec, \#, \lambda\rangle\}$;
3. $\quad V \leftarrow E^1_K$;
4. $\quad$ TANT QUE $V \neq \emptyset$ FAIRE :
5. $\quad\quad V \leftarrow V \setminus \{e\}$;
6. $\quad\quad \mathcal{C} \leftarrow min\_conf(\mathcal{E}_\Delta, e)$; //$\mathcal{C}$ contient l'ensemble des $X \in \mathsf{Conf}_T(\mathcal{E}_\Delta)$ tq. $e \in X$
7. $\quad\quad$ TANT QUE $\mathcal{C} \neq \emptyset$ FAIRE :
8. $\quad\quad\quad \mathcal{C} \leftarrow \mathcal{C} \setminus \{X\}$;
9. $\quad\quad\quad \mathcal{E}'_\Delta \leftarrow \mathsf{fn}(\mathcal{E}_\Delta | X)$;
10. $\quad\quad\quad \texttt{tr.CTS} \leftarrow \texttt{tr.CTS} \cup \left\{ \mathcal{E}_\Delta \stackrel{k(X)}{\Longrightarrow} \mathcal{E}'_\Delta \right\}$;
11. $\quad\quad\quad$ SI $\mathcal{E}'_\Delta \notin \texttt{etats.CTS}$ ALORS $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathcal{E}'_\Delta\}$;
12. $\quad \texttt{etats.CTS} \leftarrow \texttt{etats.CTS} \cup \{\mathcal{E}_\Delta\}$;
13. RENDRE CTS ;

FIG. 6.3 – Extraction du CTS sur les structures de flot récursives.

Proposition 6.1.2 *Soit p un processus CCS muni d'un ensemble de définitions gardées Δ. L'algorithme de la figure 6.3 termine pour p si $CTS(p)$ est lui-même fini.*

On commence par montrer l'invariant suivant :

Propriété 6.1.3 *Si $\mathcal{E}_\Delta$ est un état ajouté dans* etats.CTS*, alors $\mathcal{E}_\Delta$ est stable.*

La propriété est vraie dans la phase d'initialisation car $\mathcal{U}^{rec}_\Delta(p)$ est le déploiement d'un processus dans lequel toutes les définitions récursives sont gardées. Le point 11 est le seul autre endroit où un nouvel état est ajouté au CTS et cet état à été auparavant normalisé en 9. □

On passe à présent à la preuve du lemme. La boucle TANT QUE en 1. correspond au seul point de l'algorithme susceptible de diverger, car $\mathcal{S}$ peut croître en 11. Toutefois si le CTS du processus p est fini, l'algorithme ne produira qu'un nombre fini de résidus distincts à l'étape 9 car chaque résidu est le préfixe fini d'une FES isomorphe à un état du CTS de p (par le théorème 5.2.1). On montre donc le lemme par induction lexicographique sur le nombre d'états du CTS restant à construire et sur la taille de $\mathcal{S}$. À chaque passage dans la boucle, on note qu'on fait décroître $\mathcal{S}$ en 2. En utilisant la propriété 6.1.3 au point 11., on sait que $\mathcal{E}'_\Delta \notin$ `etats.CTS` est suffisant pour être sûr qu'aucune structure équivalente à $\mathcal{E}_\Delta$ n'a été considérée au préalable. Donc le nombre d'états distincts du CTS décroît et on peut conclure par hypothèse d'induction. Si aucun nouvel état n'est détecté, alors c'est la taille de $\mathcal{S}$ qui décroît et on peut de nouveau conclure par hypothèse d'induction. □

6.2 Module Causal : un outil de vérification.

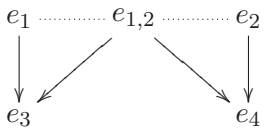
Dans la section précédente nous avons défini un algorithme permettant d'extraire le CTS d'un processus CCS. Pour ce faire, nous avons introduit un type de structure de flot permettant de construire les configurations isomorphes aux transactions du CTS, tout en conservant des structures de données de taille bornée. Nous avons aussi vu que l'algorithme terminait lorsque le CTS à construire était fini, sous la condition de ne considérer que des processus dans lesquels les appels récursifs sont gardés par des validations. Nous abordons, dans cette section, une étude plus qualitative de l'algorithme à travers une implémentation réalisée dans un module du langage Ocaml[2]. Nous décrivons, en premier lieu, la méthode choisie pour représenter la relation de conflit dans le module Causal, nous illustrerons ensuite l'usage du module à travers la construction du CTS du code partiellement correct du dîner des philosophes donné dans l'exemple 1.1.2. Nous comparerons le temps d'extraction du CTS de ce code avec celui nécessaire pour la vérification du code des philosophes avec retour arrière explicite donné dans l'exemple 1.2.2.

6.2.1 Représentation du conflit.

Les structures de flot offrent l'avantage d'un nombre relativement restreint d'événements pour représenter un processus donné. De plus, comme

[2]Le langage Objective Caml est librement disponible sur `http://caml.inria.fr`.

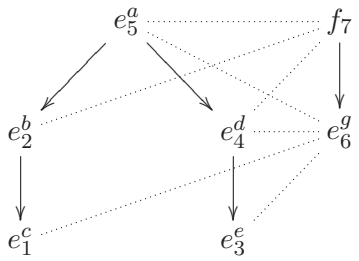
nous l'avons déjà pointé, le produit de FES n'entraîne qu'une croissance polynomiale du nombre d'événements à considérer, tandis que cette croissance devient exponentielle dans les structures premières (PES). Toutefois la représentation de la relation de conflit dans les FES est potentiellement plus coûteuse que dans les PES. En effet, dans ces dernières le conflit entre deux événements est toujours transmis par la relation de cause. Ainsi $e\#e'$ et $e < e''$ implique nécessairement $e'\#e''$. Il en résulte une représentation minimale du conflit qui consiste à ne stocker que les paires (e, e') dont le conflit ne provient pas des antécédents de e et e'. Dans les FES la même simplification ne s'applique pas car le conflit n'est, en général, pas hérité par les descendants de deux actions incompatibles comme dans la structure :



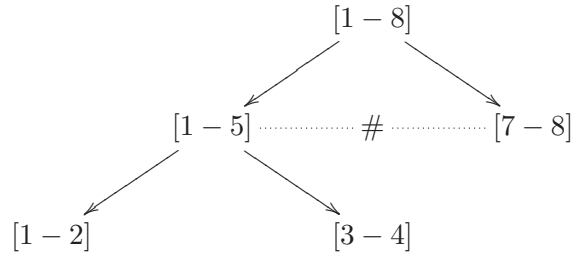
Toutefois, on peut noter qu'il est inutile de stocker les paires $(e_1, e_{1,2})$ ainsi que $(e_{1,2}, e_2)$ dans la représentation interne de la relation de conflit. En effet, l'incompatibilité de e_1 et e_2 avec $e_{1,2}$ peut se déduire directement en notant que $e_{1,2}$ partage une projection commune avec les deux autres événements. On peut donc se contenter de donner une représentation compacte du conflit induit par l'opérateur de choix.

Dans la représentation des structures de flot du module Causal, chaque événement est caractérisé par un entier, ou un couple d'entier pour les synchronisations. Pour chaque structure d'événements $\mathcal{E}$, on note $n(\mathcal{E})$ le plus grand entier utilisé dans $\mathcal{E}$ et $m(\mathcal{E})$ le plus petit. Dans la suite, on considère que pour effectuer une opération de la forme $\mathcal{E}_1 \oplus \mathcal{E}_2$ ou $\mathcal{E}_1 \otimes \mathcal{E}_2$, on a au préalable décalé les identifiants de $\mathcal{E}_2$ de $n(\mathcal{E}_1) + 1$ afin d'être sûr que les ensembles d'événements des deux structures sont bien disjoints. Considérons l'opération $\mathcal{E}_1 \oplus \mathcal{E}_2$. Au lieu de stocker un ensemble de paires d'événements, il suffit de dire que tous les événements de $\mathcal{E}_1$ sont en conflit avec ceux de $\mathcal{E}_2$. On le note par $[m(\mathcal{E}_1) - n(\mathcal{E}_1)] \# [m(\mathcal{E}_2) - n(\mathcal{E}_2)]$, où $[m - n]$ désigne l'intervalle des événements compris entre m et n (noter que par construction on a $n(\mathcal{E}_1) = m(\mathcal{E}_2) - 1$). Par exemple, considérons le processus

$p := a.(b.c \parallel d.e) + f.g$ dont la structure de flot est :



On construit parallèlement l'arbre suivant dont on pourra tirer la relation de conflit :



L'arbre de conflit d'une structure $\mathcal{E}$ est noté $\mathcal{A}_\#(\mathcal{E})$. Nous montrons à présent comment obtenir de tels arbres lors de la construction inductive des structures de flot. On note $\mathcal{A}_\#(\mathcal{E}) := \langle [m(\mathcal{E}) - n(\mathcal{E})], fg, fd, rel \rangle$ pour l'arbre de conflit de la structure $\mathcal{E}$, dans lequel fg et fd sont respectivement les fils gauche et droit de $\mathcal{A}_\#$ et où $rel = \#$ si les fils sont en conflit et $rel = \emptyset$ sinon. On a :

$$\begin{array}{lcl}
\mathcal{A}_\#(\mathcal{O}) & := & \langle [0-0], \emptyset, \emptyset, \emptyset \rangle \\
\mathcal{A}_\#(\mathsf{D}(\tilde{x})) & := & \langle [1-1], \emptyset, \emptyset, \emptyset \rangle \\
\mathcal{A}_\#(\alpha \odot \mathcal{E}) & := & \langle [m(\mathcal{E}) - (n(\mathcal{E}) + 1)], fg, fd, rel \rangle \\
\mathcal{A}_\#(\mathcal{E} \oplus \mathcal{E}') & := & \langle [m(\mathcal{E}) - n(\mathcal{E}')], \mathcal{A}_\#(\mathcal{E}), \mathcal{A}_\#(\mathcal{E}'), \# \rangle \\
\mathcal{A}_\#(\mathcal{E} \otimes \mathcal{E}') & := & \langle [m(\mathcal{E}) - n(\mathcal{E}')], \mathcal{A}_\#(\mathcal{E}), \mathcal{A}_\#(\mathcal{E}'), \emptyset \rangle \\
\mathcal{A}_\#(\mathcal{E} \backslash\backslash \langle \alpha, \rho \rangle) & := & \mathcal{A}_\#(\mathcal{E})
\end{array}$$

Dans le module Causal, une FES_{rec} est donc donnée par $\langle E, \prec, \mathcal{A}_\#, \lambda, \vdash \rangle$. Posons $\mathcal{A}_\# < \mathcal{A}'_\#$ lorsque $\mathcal{A}'_\#$ est un sous-arbre de $\mathcal{A}_\#$ et notons $e \in \mathcal{A}_\#$ pour :

$$\mathcal{A}_\# = \langle I, fg, fd, rel \rangle \quad \text{et} \quad \left\{ \begin{array}{l} e \in I \\ \text{ou } e = (i,j) \ \& \ (i \in I \text{ ou } j \in I) \end{array} \right.$$

On retrouve la relation de conflit usuelle en posant :

$$e \# e' \iff \left\{ \begin{array}{l} e = (i,j) \ \& \ e' = (i',j') \ \& \ (i' \in \{i,j\} \vee j' \in \{i,j\}) \\ \text{ou } e \in \mathcal{A}'_\#, e' \in \mathcal{A}''_\# \ \& \ \exists I : \langle I, \mathcal{A}'_\#, \mathcal{A}''_\#, \# \rangle \leq \mathcal{A}_\# \end{array} \right.$$

Noter que par construction, dans un arbre de conflit $\mathcal{A}_{\#} = \langle I, fg, fd, rel \rangle$, si $e \in I$ et $e \in fg$ (resp. $e \in fd$) alors $e \notin fd$ (resp. $e \notin fg$). La recherche des événements en conflit avec un événement donné s'obtient par un simple parcours en profondeur de l'arbre.

6.2.2 Banc d'essai.

Dans la figure 5.1, nous avons décrit deux chemins permettant d'avoir la certitude qu'un programme est bien conforme à sa spécification. Dans la méthode directe, on construit un programme *complet* qui ne contient aucun verrous ni choix partiel et on tente d'exhiber une bisimulation entre son LTS et le LTS de référence. Dans la méthode déclarative, on construit un programme *partiel* dont le CTS doit être bisimilaire à la spécification. Si tel est le cas, par le théorème 4.2.1, le code RCCS du programme partiel devient automatiquement équivalent au code complet. Nous allons à présent comparer ces deux approches dans l'exemple du dîner des philosophes (voir section 1.1.3). Nous prenons comme système de spécification, le LTS donné dans la section 1.2.2 :

$$\begin{array}{lcl} \mathcal{S}_{phil}(P \cup \{i-1, i, i+1\}, M) & \xrightarrow{d_i} & \mathcal{S}_{phil}(P \cup \{i-1, i+1\}, M \cup \{i\}) \\ \mathcal{S}_{phil}(P, M \cup \{i\}) & \xrightarrow{f_i} & \mathcal{S}_{phil}(P \cup \{i\}, M) \end{array}$$

Nous avions donné deux implémentations du problème (exemples 1.1.2 et 1.2.2), paramétrées par un ensemble de définitions Δ, toutes deux ayant pour état initial :

$$P_{\Delta} := \left(\prod_{i=1}^{N} \big(\mathsf{P}(b_i, b_{i+1}, \underline{d_i}, \underline{f_i}) + \mathsf{P}(b_{i+1}, b_i, \underline{d_i}, \underline{f_i}) \big) \parallel b_i.0 \right) \backslash \tilde{b}$$

où, pour le code complet, Δ valait :

$$\Delta_{comp} = \left\{ \begin{array}{lcl} \mathsf{P}(b_1, b_2, d, f) & := & \bar{b}_1.\big(\mathsf{Q}(b_1, b_2, d, f) + \tau.(\mathsf{P}(b_1, b_2, d, f) \parallel b_1)\big) \\ \mathsf{Q}(b_1, b_2, d, f) & := & \bar{b}_2.\big(\mathsf{M}(b_1, b_2, d, f) + \tau.(\mathsf{P}(b_1, b_2, d, f) \parallel b_1 \parallel b_2)\big) \\ \mathsf{M}(b_1, b_2, d, f) & := & f.\big(\mathsf{P}(b_1, b_2, d, f) \parallel b_1 \parallel b_2\big) \end{array} \right.$$

et où, pour le code partiel, Δ valait :

$$\Delta_{part} := \left\{ \begin{array}{lcl} \mathsf{P}(b_1, b_2, d, f) & := & \bar{b}_1.\bar{b}_2.d.\mathsf{M}(b_1, b_2, d, f) \\ \mathsf{M}(b_1, b_2, d, f) & := & f.\big(\mathsf{P}(b_1, b_2, d, f) \parallel b_1.0 \parallel b_2.0\big) \end{array} \right.$$

Appelons p_{comp} le code utilisant Δ_{comp} et p_{part} le code utilisant Δ_{part}. Noter que l'algorithme d'extraction de CTS s'applique bien à p_{part} car les

définitions de Δ_{part} sont gardées par des validations (ce qui n'est pas le cas dans Δ_{comp}). Nous utilisons le *Mobility Workbench* (MWB)[3], comme outil de référence pour effectuer nos tests de bisimulation. La méthode directe consiste à fournir $\mathcal{S}_{phil}$ et p_{comp} à cet outil et à lui demander d'extraire une relation de bisimulation automatiquement. La deuxième méthode consiste à extraire d'abord le CTS de p_{part} par l'intermédiaire du module Causal, puis à tester la bisimulation entre CTS(p_{part}) et $\mathcal{S}_{phil}$. Nous montrons, figures 6.4 et 6.5, les temps de calculs nécessaires à la réalisation de chacune de ces méthodes[4]. Afin d'avoir un ordre de grandeur du nombre d'états à mettre en relation pour construire les bisimulations, nous avons ajouté aux graphiques la courbe correspondant aux nombres d'états du LTS de spécification. On peut, en effet, montrer que le nombre d'états distincts de la spécification pour n philosophes est donné par une suite de Fibonacci[5]. On retiendra donc que le nombre d'états de $\mathcal{S}_{phil}$ croît de manière exponentielle. On s'aperçoit qu'il est impossible de construire automatiquement une bisimulation entre le code complet et la spécification pour plus de 5 philosophes (environs 160 états dans le LTS de $\mathcal{S}_{phil}$).

La méthode indirecte permet d'extraire le CTS de p_{part} et de construire une bisimulation, pour un système comportant jusqu'à 19 philosophes (près de 15.000 états dans le LTS de $\mathcal{S}_{phil}$).

Concernant ces mesures, il convient de faire une remarque importante : le temps de calcul de la méthode indirecte est essentiellement constitué du temps nécessaire à l'extraction du CTS. En effet, le CTS obtenu est lui-même une spécification du système car il ne contient plus que des transitions visibles. Construire une bisimulation entre le CTS de p_{part} et $\mathcal{S}_{phil}$ revient à

[3][Victor et Moller, 1994]

[4]Les tests ont été réalisés à l'aide d'un Intel Pentium 4 CPU 3.20GHz avec 1GB de RAM.

[5]Soit $P(n)$ le nombre de configurations possibles pour n philosophes. On commence par noter que $P(1) = 1$ (le philosophe pense car il n'a qu'une seule baguette) et $P(2) = 3$ (personne ne mange ou seul un des deux philosophes mange). Posons $P_0(n)$ (resp. $P_1(n)$) le nombre de configurations de n philosophes telles que le $n^{\text{ème}}$ philosophe pense (resp. mange). Trivialement $P(n) = P_0(n) + P_1(n)$ (i). On remarque ensuite que $P_0(n)$ est donné par $P(n-1)$ plus le nombre de configurations à $n-1$ philosophes (normalement interdites) dans lesquelles les philosophes $n-2$ et 1 (*modulo* $n-1$) mangent. Ce nombre de configurations est égal à $P_1(n-2)$ (on fusionne les philosophes $n-2$ et 1) et on a $P_0(n) = P(n-1) + P_1(n-2)$ (ii). On note ensuite que $P_1(n)$ correspond aussi au nombre de configurations de $n-1$ philosophes tels que les philosophes $n-1$ et 1 (modulo $n-1$) pensent. En fusionnant ces deux philosophes on se ramène à $P_0(n-2)$ et par conséquent $P_1(n) = P_0(n-2)$ (iii). Par $(i), (ii)$ et (iii) on trouve $P(n) = P(n-1) + P(n-2)$ qui est bien une suite de Fibonacci, initialisée à $P(1) = 1$ et $P(2) = 3$. □

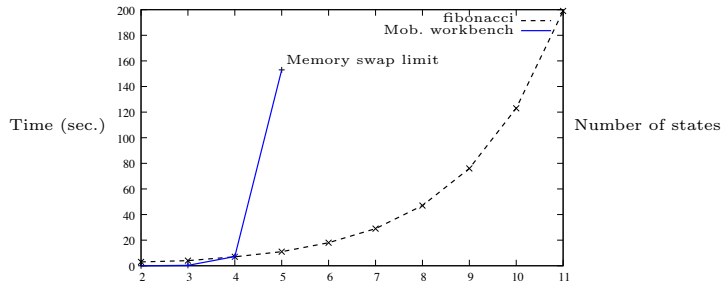


FIG. 6.4 – Test de bisimulation pour le code complet p_{comp}.

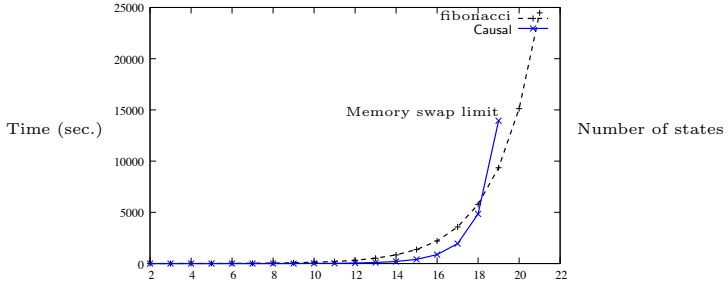


FIG. 6.5 – Méthode indirecte utilisant le CTS de p_{part}.

tester la bisimilarité de deux systèmes de transitions isomorphes, ce qui est négligeable devant le temps d'extraction du CTS[6].

6.3 Discussion.

Le critère de correction, traditionnel dans la communauté des algèbres de processus, qui consiste à exhiber une bisimulation entre le LTS d'un programme et sa spécification, est souvent considéré comme trop complexe pour être traité par des outils de vérification automatique[7]. Pour ces derniers, on utilise plutôt des formules de logiques temporelles où modales pour définir les propriétés de correction qu'on attend d'un programme et on vérifie que ces formules sont vraies dans tous les états du LTS engendré[8]. La bisimulation avec un système de référence peut être vue comme un critère de correction ultime puisque dans la logique de Henessy-Milner, deux systèmes sont bisi-

[6]Par exemple, MWB met $0.4s$ pour construire une bisimulation entre $\mathcal{S}_{phil}$ et le CTS de p_{part} pour 10 philosophes, construit par Causal en $13s$ CPU.

[7][Laroussinie et Schnoebelen, 2000, Fisler et Vardi, 2002]

[8][Clarke *et al.*, 1986, Hennessy et Milner, 1985]

milaires si et seulement si ils satisfont le même ensemble de formules[9]. Au delà des techniques traditionnelles pour réduire le nombre d'états à considérer lorsqu'on cherche à construire une bisimulation, nous avons montré qu'il est possible de tirer parti du système de retour arrière intégré à RCCS pour simplifier à la fois la conception et la preuve de systèmes transactionnels. En particulier, pour le fragment de CCS à récursions gardées, nous avons construit un module Ocaml permettant d'extraire le CTS d'un processus de façon automatique. Les tests effectués, pour l'exemple du dîner des philosophes, montrent que la combinaison extraction de CTS, bisimulation et relèvement dans RCCS (voir figure 5.1) est suffisamment digeste pour construire des systèmes et prouver leur correction avec un nombre d'états laissant loin derrière une approche directe basée sur un programme intégrant une gestion des verrous explicite. Une remarque s'impose toutefois : étant donnés une spécification $\mathcal{S}$ et un programme partiel p (*i.e* tel que $\mathrm{CTS}(p) \approx \mathcal{S}$) et un programme complet p' (*i.e* tel que $p' \approx \mathcal{S}$), nous savons que $\ell(p) \approx p'$ (par le théorème 4.2.1 et par transitivité de la bisimulation). Si $\ell(p)$ et p' sont équivalents, nous ne pouvons pas affirmer qu'ils auront la même efficacité et il est même facile de construire un programme p' dont le retour arrière serait plus efficace que celui de $\ell(p)$[10]. L'intérêt de la méthode réside donc principalement dans la "mécanisabilité" de la preuve de correction $\ell(p)$, bien qu'il existe certaines classes de systèmes transactionnels pour lesquels le retour arrière devient arbitrairement difficile à programmer[11]. Une comparaison plus quantitative de $\ell(p)$ et p', qui sont *a priori* des processus divergents, requerrait de se placer dans un système de transitions probabiliste hors du cadre, traditionnellement non-déterministe, des algèbres processus.

Un autre point sujet à discussion est de ne considérer que des constantes de récursions gardées par des validations. De notre point de vue, cette restriction est en fait raisonnable pour les systèmes transactionnels car elle revient à dire que toute trace infinie est due à un nombre infini de transactions successives. Le comportement divergeant d'un processus transactionnel peut être dû au fait que le processus a besoin de revenir en arrière pour tenter une nouvelle transaction, auquel cas, la divergence serait silencieuse. Dans les processus partiellement corrects, on a précisément supprimé le besoin de coder ce genre de comportement et une trace infinie provient alors d'une boucle décrivant un système transactionnel itératif comme celui des

[9] [Hennessy et Milner, 1985, Stirling, 2001]

[10] Dans un cas limite, il suffit de prendre un programme ne nécessitant aucun retour arrière.

[11] [Danos *et al.*, 2006b]

philosophes.

Chapitre 7

Théorème transactionnel généralisé.

Sommaire

Dans les chapitres 2 et 3, nous avons mis au point une algèbre de processus équipée d'un mécanisme distribué de retour arrière basé sur un système de mémoire. Nous avons ensuite montré que la correction du retour arrière provenait d'une propriété fondamentale de RCCS (Théorème 3.2.1) : cette dernière permettant de considérer deux traces co-initiales et co-finales comme une seule et même trace, quotientée par écrasements de transitions opposées et permutations de transitions concurrentes. Nous avons alors introduit, au chapitre 4, des actions irréversibles permettant de bloquer le retour arrière de séquences de transitions appelées transactions. Nous avons vu dans le lemme 4.2.2 qu'on pouvait décomposer, de façon unique, une trace RCCS quelconque de façon à exhiber la transaction (possiblement vide) qui correspond aux actions réellement exécutées par le système ; l'unicité de la décomposition reposant sur la propriété fondamentale de RCCS décrite plus haut. Le théorème transactionnel, qui montre l'équivalence entre le système de transition causal d'un processus p et son relèvement $\ell(p)$, découle directement de cette factorisation (Théorème 4.2.1). Enfin nous avons

vu qu'on pouvait utiliser ce théorème transactionnel, pour définir un mode de programmation déclaratif, qui permet de simplifier à la fois la tâche de programmation et la preuve des programmes construits, en tirant parti du retour arrière intégré à RCCS. Nous résumons les différentes étapes que nous venons de décrire dans la figure 7.1 où $\mathcal{I}$ et $\mathcal{R}$ désignent respectivement l'ensemble des traces irréversibles et réversibles de RCCS.

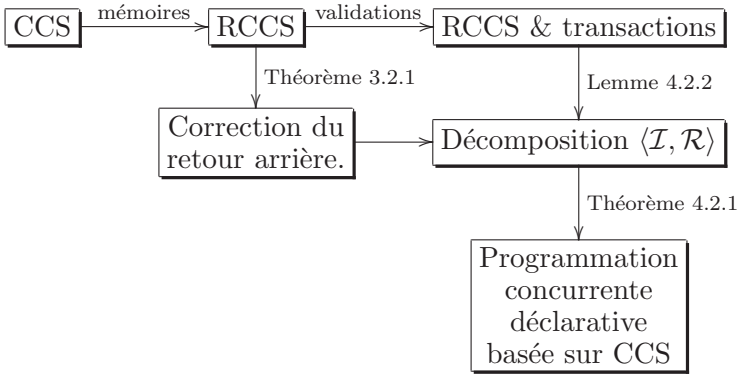


FIG. 7.1 – Programmation déclarative avec RCCS.

Dans ce chapitre, nous allons montrer que la construction d'un calcul réversible au dessus d'un langage concurrent donné peut être formulée de façon générale, en s'abstrayant des constructions syntaxiques du langage choisi. Plus précisément, nous montrons qu'à partir d'une catégorie $\mathbf{C}$ des calculs d'un langage C, la conception d'un calcul réversible, disons RC, implique la construction sous-jacente d'une catégorie des "Histoires" de $\mathbf{C}$. Le fait que les traces de calculs de RC correspondent aux transactions de C (ce qui fournit un théorème transactionnel) est alors capturé par un théorème d'équivalence de catégories. Le plan de ce chapitre est le suivant : nous définissons, en premier lieu, la notion de système de factorisation sur une catégorie $\mathbf{C}$. Nous illustrerons un tel système en montrant que les sous-catégories $\mathcal{R}$ et $\mathcal{I}$ des traces réversibles et des traces transactionnelles forment un système de factorisation $\langle \mathcal{I}, \mathcal{R} \rangle$ sur la catégorie des traces de CCS. Nous montrerons ensuite comment construire la catégorie des "histoires réversibles" $h_\star(\mathbf{C}, \mathcal{R})$ (voir définition 7.2.2) qui est le pendant abstrait de l'ajout d'un système de mémoires au dessus du langage C. Nous verrons alors que le système de factorisation entraîne l'équivalence des catégories $\mathcal{I}$ et $h_\star(\mathbf{C}, \mathcal{R})$, ce qui revient à un théorème transactionnel généralisé à tout langage concurrent pour lequel on sait définir un système de factorisation $\langle \mathcal{I}, \mathcal{R} \rangle$. Nous conclurons ce

chapitre par une discussion sur les difficultés de conception d'un langage concurrent, similaire à RCCS, traitées ici par l'abstraction catégorique.

7.1 Système de factorisation.

7.1.1 Définitions

On commence par s'abstraire des langages concurrents en considérant une simple catégorie $\mathbf{C}$ dont les objets $p, q \ldots$ peuvent être vus comme les termes du langage et les morphismes $f, g \ldots, \varphi, \ldots$ comme des traces de calcul quotientées par équivalence. On utilisera la notation $f \in \mathsf{Mor}_{\mathbf{C}}$ pour désigner un morphisme f de $\mathbf{C}$ et $\varphi \in \mathsf{Iso}_{\mathbf{C}}$ pour dénoter les isomorphismes de $\mathbf{C}$.

Exemple 7.1.1 *La catégorie* **CCS** *correspondant aux calculs de CCS est donnée par :*
— des objets : processus $p \in P$;
— des morphismes : $p \xrightarrow{f} q$ où f représente les réductions de p à q, considérées à équivalence par permutations près et où les diagrammes commutatifs de la forme :

$$\begin{array}{ccc} p_1 & \xrightarrow{h} & p_2 \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ q_1 & \xrightarrow[i]{} & q_2 \end{array}$$

indiquent que les traces correspondant à $i \circ f$ et $g \circ h$ sont équivalentes par permutations de transitions concurrentes. On notera que les isomorphismes $p \xrightarrow{\varphi} q$ de $\mathbf{C}$ peuvent être vus comme l'application d'une règle de congruence structurelle entre p et q.

Étant donnée une catégorie $\mathbf{C}$ et deux flèches $f, g \in \mathsf{Mor}_{\mathbf{C}}$, on note $f \perp g$ si étant donné h, i deux morphismes arbitraires de $\mathbf{C}$ tels que $g \circ h = i \circ f$, il existe un unique $j : q_1 \to q_2$ satisfaisant le diagramme commutatif suivant :

$$\begin{array}{ccc} p_1 & \xrightarrow{h} & p_2 \\ {\scriptstyle f}\downarrow & \overset{j}{\nearrow} & \downarrow{\scriptstyle g} \\ q_1 & \xrightarrow[i]{} & q_2 \end{array}$$

Nous allons nous servir de la relation d'orthogonalité $\perp$ afin de séparer les flèches irréversibles des flèches réversibles. En attendant on peut d'ores et déjà définir les opérations de clôtures usuelles pour $\perp$: étant donné un

ensemble $\mathcal{F}$ de flèches de $\mathbf{C}$ on note $\mathcal{F}^{\perp} := \{g \in \mathsf{Mor}_{\mathbf{C}} \mid \forall f \in \mathcal{F} : f \perp g\}$. De façon similaire, on note $\mathcal{F}^{\top} := \{f \in \mathsf{Mor}_{\mathbf{C}} \mid \forall g \in \mathcal{F} : f \perp g\}$.

Propriété 7.1.1 *Les propriétés suivantes sont des conséquences immédiates de l'opération de clôture :*

$$\mathcal{F}^{\perp\top\perp} = \mathcal{F}^{\perp} \tag{7.1}$$

$$\mathcal{F}^{\top\perp\top} = \mathcal{F}^{\top} \tag{7.2}$$

$$\mathcal{F} \subseteq \mathcal{F}' \Rightarrow \mathcal{F}'^{\perp} \subseteq \mathcal{F}^{\perp} \tag{7.3}$$

$$\mathcal{F} \subseteq \mathcal{F}' \Rightarrow \mathcal{F}'^{\top} \subseteq \mathcal{F}^{\top} \tag{7.4}$$

Muni d'une telle opération, on définit un système de préfactorisation de la façon suivante :

Définition 7.1.1 ([Freyd et Kelly, 1972]) *Deux classes de flèches $\langle \mathcal{I}, \mathcal{R} \rangle$ d'une catégorie $\mathbf{C}$ forment un* système de préfactorisation *si $\mathcal{I}^{\perp} = \mathcal{R}$ et $\mathcal{R}^{\top} = \mathcal{I}$.*

Notons que, par les propriétés 7.1 et 7.2, on peut définir un système de préfactorisation pour n'importe quel sous ensemble de flèches $\mathcal{F}$ de $\mathbf{C}$ en posant $\langle \mathcal{I}, \mathcal{R} \rangle = \langle \mathcal{F}^{\top}, \mathcal{F}^{\top\perp} \rangle$ ou $\langle \mathcal{I}, \mathcal{R} \rangle = \langle \mathcal{F}^{\perp\top}, \mathcal{F}^{\perp} \rangle$. Nous pouvons donner une série de propriétés bien connues des systèmes de préfactorisation :

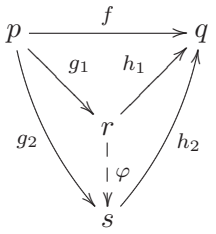
Propriété 7.1.2 *Soit $\langle \mathcal{I}, \mathcal{R} \rangle$ un système de préfactorisation sur $\mathbf{C}$. Alors :*
(i) $\mathcal{I} \cap \mathcal{R} = \mathsf{Iso}_{\mathbf{C}}$;
(ii) $\mathcal{I}$ et $\mathcal{R}$ sont closes par composition ;
(iii) si $f_2 \circ f_1 \in \mathcal{I}$ et $f_1 \in \mathcal{I}$ alors $f_2 \in \mathcal{I}$;
(iv) si $g_2 \circ g_1 \in \mathcal{R}$ et $g_2 \in \mathcal{R}$ alors $g_1 \in \mathcal{R}$.

Notons que les points (i) et (ii) impliquent, en particulier, que $\mathcal{I}$ et $\mathcal{R}$ sont des sous-catégories de $\mathbf{C}$ – elles contiennent bien l'identité et sont closes par composition.

Définition 7.1.2 (Factorisation) *Un système $\langle \mathcal{I}, \mathcal{R} \rangle$ de préfactorisation sur $\mathbf{C}$ est un* système de factorisation *si toute flèche f de $\mathbf{C}$ peut s'écrire $f = h \circ g$ avec $g \in \mathcal{I}$ et $h \in \mathcal{R}$.*

Clairement, $\langle \mathbf{C}, \mathsf{Iso}_{\mathbf{C}} \rangle$ et $\langle \mathsf{Iso}_{\mathbf{C}}, \mathbf{C} \rangle$ sont des systèmes de factorisation. Un autre exemple bien connu est la factorisation $\langle \mathcal{E}, \mathcal{M} \rangle$ sur la catégorie des ensembles **Set**, où $\mathcal{E}$ est la classe des morphismes surjectifs et $\mathcal{M}$ celle des injections. Pour souligner le parallèle avec RCCS, nous allons montrer, dans la section suivante, que l'ensemble des transactions et des traces silencieuses de CCS forment un système de factorisation. En attendant, nous concluons cette section avec une propriété standard des systèmes de factorisation.

Lemme 7.1.1 *Tout système de factorisation $\langle \mathcal{I}, \mathcal{R} \rangle$ est unique (à isomorphismes près), i.e pour toute flèche f de $\mathbf{C}$ et tout $g_1, g_2 \in \mathcal{I}$ et $h_1, h_2 \in \mathcal{R}$ tels que $f = h_1 \circ g_1 = h_2 \circ g_2$, il existe un unique isomorphisme φ tel que $\varphi \circ g_1 = g_2$ et $h_2 \circ \varphi = h_1$.*



LEMME 7.1.1 – Unicité de la factorisation.

7.1.2 Exemple : un système de factorisation pour CCS.

Afin de mettre en relation les outils théoriques que nous venons de mettre en place avec la méthode suivie pour RCCS, nous allons montrer le lien étroit qui existe entre le lemme 4.2.2 permettant de décomposer des traces RCCS et l'existence d'un système de factorisation sur la catégorie **CCS**. Le lemme 4.2.2 s'applique ici dans le cas particulier de traces RCCS positives correspondant au simple relèvement de traces CCS[1]. Pour tout $f \in \mathsf{Mor}_{\mathbf{CCS}}$, on pose $f \in \mathcal{I}$ si f est de but initial et $f \in \mathcal{R}$ si f ne contient aucune validation. On remarquera que la trace vide est à la fois dans $\mathcal{I}$ et dans $\mathcal{R}$. On note $p \Longrightarrow q$ pour les flèches de $\mathcal{I}$ et $p \rightarrow q$ pour les flèches de $\mathcal{R}$. Le lemme suivant est le correspondant du lemme 4.2.2.

Lemme 7.1.2 *Tout morphisme arbitraire f de* **CCS** *peut s'écrire de façon unique (à isomorphisme près) $f = g \circ f'$ avec $f' \in \mathcal{I}$ et $g \in \mathcal{R}$.*

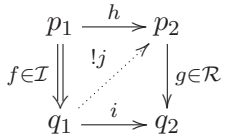
On montre la décomposition par induction sur le nombre de validations dans f. Si $f \in \mathcal{R}$ alors on prend $f' \in \mathsf{Iso}_{\mathbf{C}}$. Sinon, on pose $f = f_2 \circ f_1$ avec f_2 contenant une unique validation. Par hypothèse de récurrence on a $f_1 = g \circ f'$ avec $g \in \mathcal{R}$ et $f' \in \mathcal{I}$. Par construction, $f_2 \circ g$ est une trace ne contenant qu'une seule validation. Nous renvoyons au lemme 4.2.2 pour la décomposition unique de $f_2 \circ g$, dans le cas particulier d'une trace RCCS positive. On obtient la décomposition $f_2 \circ g = g' \circ f''$ avec $f'' \in \mathcal{I}$ et $g' \in \mathcal{R}$ et la factorisation de f est donnée par $g' \circ (f'' \circ f')$. □

[1] On peut donc voir ici RCCS restreint aux transitions positives comme une décoration permettant d'identifier syntaxiquement les transitions concurrentes.

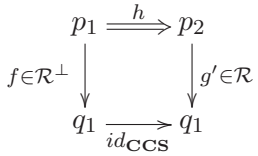
La décomposition donnée par le lemme ci-dessus permet d'obtenir un système de factorisation sur **CCS** :

Lemme 7.1.3 *Le couple $\langle \mathcal{I}, \mathcal{R} \rangle$ est un système de factorisation sur la catégorie* **CCS**.

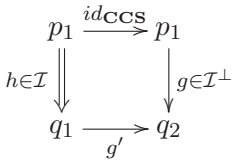
– On commence par montrer que $\mathcal{I} \subseteq \mathcal{R}^{\top}$ et $\mathcal{I}^{\top} \subseteq \mathcal{R}$. On doit exhiber pour cela l'unique j qui complète le diagramme suivant :



Soit $g_h \circ f_h$ l'unique décomposition de h et $g_i \circ f_i$ celle de i. D'après le diagramme ci-dessus, on a $g \circ h = g_i \circ (f_i f) = (g g_h) \circ f_h$, avec $g_i, g_h \in \mathcal{R}$ et $f_i, f_h \in \mathcal{I}$. On en déduit que $g_i \circ (f_i f)$ et $(g g_h) \circ f_h$ sont deux factorisations de $g \circ h$ et donc par le lemme 7.1.2, il existe $\varphi \in \mathsf{Iso}_{\mathbf{CCS}}$ tel que $f_i f \circ \varphi = f_h$ et $j = g_h \circ \varphi \circ (f_i f)$ est le morphisme recherché dont l'unicité provient de l'unicité des décompositions utilisées.
– On montre ensuite que $\mathcal{R}^{\top} \subseteq \mathcal{I}$. Supposons $f \in \mathcal{R}^{\perp}$ et $f \notin \mathcal{I}$. Puisque f n'est pas transactionnelle on peut la factoriser en $f = g' \circ h$ avec $h \in \mathcal{I}$ et $g' \in \mathcal{R}$ tel que $g' \notin \mathsf{Iso}_{\mathbf{CCS}}$. On a le diagramme commutatif suivant :



Par définition de $\mathcal{R}^{\perp}$, on doit trouver un morphisme j tel que $g' \circ j = id_{\mathbf{CCS}}$ ce qui est impossible étant donné que $g \notin \mathsf{Iso}_{\mathbf{CCS}}$.
– On montre enfin que $\mathcal{I}^{\perp} \subseteq \mathcal{R}$. On suppose $g \in \mathcal{I}^{\perp}$ et $g \notin \mathcal{R}$. Par application du lemme 7.1.2 on a $g = g' \circ h$ avec $h \in \mathcal{I}$ et $h \notin \mathsf{Iso}_{\mathbf{CCS}}$. On a le diagramme commutatif suivant :



Par définition de $\mathcal{I}^{\perp}$ il nous faudrait exhiber un morphisme j tel que $j \circ h = id_{\mathbf{CCS}}$ ce qui est impossible attendu que $h \notin \mathsf{Iso}_{\mathbf{CCS}}$.
– En utilisant les lemmes 7.1.1 et 7.1.3, on conclut que $\langle \mathcal{I}, \mathcal{R} \rangle$ est bien un

système de factorisation. □

7.2 Ajout des histoires.

7.2.1 Catégorie des histoires.

Comme nous l'avons vu dans la section 7.1.2, le lemme 4.2.2 consiste en fait à exhiber un système de factorisation sur la catégorie des calculs de CCS. Nous allons voir, à présent, qu'un calcul qui respecte la réversibilité des traces de $\mathcal{R}$ et l'irréversibilité de celles de $\mathcal{I}$ peut être représenté abstraitement en assumant simplement que le couple $\langle \mathcal{I}, \mathcal{R} \rangle$ est un système de factorisation.

Définition 7.2.1 (Catégorie des histoires) *On suppose $\langle \mathcal{I}, \mathcal{R} \rangle$ un système de factorisation sur $\mathbf{C}$. Soit $h(\mathbf{C}, \mathcal{R})$ la catégorie définie par :*

- *objets : les flèches de $\mathcal{R}$;*
- *flèches : les couples de flèches $\langle f, f' \rangle$, avec f et f' deux flèches de $\mathbf{C}$ avec $f' \in \mathcal{I}$, satisfaisant le diagramme commutatif suivant :*

$$\begin{array}{ccc} p_1 & \xrightarrow{f'} & p_2 \\ {\scriptstyle g_1}\downarrow & & \downarrow{\scriptstyle g_2} \\ q_1 & \xrightarrow[f]{} & q_2 \end{array}$$

On notera que pour le moment nous n'avons introduit de catégorie pour les calculs réversibles. En effet, $h(\mathbf{C}, \mathcal{R})$ correspond simplement à la mise en place du cadre catégorique qui nous permettra de définir plus tard la catégorie $h_\star(\mathbf{C}, \mathcal{R})$ des histoires réversibles (voir définition 7.2.2)[2]. Par ailleurs, la construction de $h(\mathbf{C}, \mathcal{R})$ à partir de $\mathbf{C}$ n'est *a priori* pas fonctorielle. En effet, la construction intuitive qui associe un objet p de $\mathbf{C}$ à un objet $p \to p$ de $h(\mathbf{C}, \mathcal{R})$ et une flèche $p \xrightarrow{f} q$ de $\mathbf{C}$ au diagramme commutatif :

$$\begin{array}{ccc} p & \longrightarrow & p \\ \downarrow & & {\scriptstyle f}\downarrow \\ p & \xrightarrow[f]{} & q \end{array}$$

ne permet pas de composer l'image de deux morphismes composables de $\mathbf{C}$ car les flèches de $h(\mathbf{C}, \mathcal{R})$ partent nécessairement d'un objet "à mémoires vides" de la forme $p \to p$.

[2]Dans l'intuition RCCS, on peut voir $h(\mathbf{C}, \mathcal{R})$ comme le pendant de RCCS restreint aux transitions positives.

Propriété 7.2.1 *Soit* $p_1 \xrightarrow{g_1} q_1$ *un objet de* $h(\mathbf{C}, \mathcal{R})$ *et* f *un morphisme quelconque de* $\mathbf{C}$.

(i) Il existe un unique objet g_2 *de* $h(\mathbf{C}, \mathcal{R})$ *(à isomorphismes près) et une unique flèche* $p_1 \xrightarrow{f'} p_2$ *de* $\mathcal{I}$ *tels que* $\langle f, f' \rangle$ *soit une flèche de* $h(\mathbf{C}, \mathcal{R})$ *;*

(ii) de plus, si $f \in \mathcal{R}$ *alors* $f' \in \mathsf{Iso}_{\mathbf{C}}$.

(i) On considère la factorisation $\langle \mathcal{I}, \mathcal{R} \rangle$ de $f \circ g_1$. Par le lemme 7.1.1 cette décomposition est unique à isomorphisme près et correspond à $g_2 \circ f'$.
(ii) Si $f \in \mathcal{R}$ alors $f \circ g_1 \in \mathcal{R}$ par composition. On en déduit que $g_2 \circ f'$ est aussi dans $\mathcal{R}$ et par la propriété 7.1.2, $f \in \mathcal{R} \cap \mathcal{I}$ et donc $f \in \mathsf{Iso}_{\mathbf{C}}$. □

On rappelle que $\mathcal{I}$ peut être vu à la fois comme un type de flèches de $\mathbf{C}$ et comme une catégorie (Propriété 7.1.2). On dispose d'un foncteur évident

$$M : h(\mathbf{C}, \mathcal{R}) \to \mathcal{I} \tag{7.5}$$

qui envoie un objet $p_1 \xrightarrow{g_1} q_1$ de $h(\mathbf{C}, \mathcal{R})$ sur un objet p_1 de $\mathcal{I}$ et les flèches $\langle f, f' \rangle \in \mathsf{Mor}_{h(\mathbf{C},\mathcal{R})}$ sur $f' \in \mathsf{Mor}_{\mathcal{I}}$. Si on revient à notre intuition RCCS, le foncteur M envoie une transaction bruitée (positive) sur la transaction pure qui lui correspond. On notera que conformément au lemme 4.2.2, si une trace f ne contient pas de validation, il est clair que $M\langle f, f' \rangle = f' \in \mathsf{Iso}_{\mathbf{C}}$ par la propriété 7.2.1.

On peut, par ailleurs, définir un foncteur (plein et fidèle)

$$N : \mathcal{I} \to h(\mathbf{C}, \mathcal{R}) \tag{7.6}$$

qui envoie les objets $p_1 \in \mathcal{I}$ sur l'identité $p_1 \to p_1 \in h(\mathbf{C}, \mathcal{R})$ et les morphismes $f \in \mathsf{Mor}_{\mathcal{I}}$ vers une flèche $\langle f, f \rangle \in \mathsf{Mor}_{h(\mathbf{C},\mathcal{R})}$. Toujours dans l'intuition RCCS, ce foncteur revient à relever des transactions CCS vers une trace RCCS dont l'origine et le but ont des mémoires vides. On identifie donc dans l'abstraction les mémoires bloquées avec les mémoires vides. Pour obtenir la même propriété dans RCCS il nous faut utiliser un système de ramasse-miettes de mémoires bloquées dont la définition sera donnée dans la partie exploratoire de la conclusion de ce mémoire.

Lemme 7.2.1 *Le foncteur* N *est adjoint à gauche de* M.

On exhibe l'unité $\eta : Id_{\mathcal{I}} \to MN$ et la co-unité $\epsilon : NM \to Id_{h(\mathbf{C},\mathcal{R})}$ de l'adjonction. L'unité est triviale puisque $MN = Id_{\mathcal{I}}$. Pour la co-unité on

prend $\epsilon_g = \langle g, id \rangle$ correspondant au diagramme commutatif :

$$\begin{array}{ccc} p & \longrightarrow & p \\ \downarrow & & \downarrow g \\ p & \xrightarrow[g]{} & q \end{array}$$

qui a bien la propriété $MN(g) \xrightarrow{\epsilon_g} g$. On vérifie ensuite qu'on a bien les triangles suivants (pour lesquels on doit montrer l'existence et l'unicité des morphismes en pointillés) :

$$\begin{array}{ccc} x & \xrightarrow{\eta_x} & MN(x) \\ & \searrow^{f} & \vdots\, M(g) \\ & & M(y) \end{array} \qquad \begin{array}{ccc} N(x) & & \\ \vdots\, N(f) & \searrow^{g} & \\ NM(y) & \xrightarrow[\epsilon_y]{} & y \end{array}$$

Dans le premier triangle, pour tout $f \in \mathsf{Mor}_{\mathcal{I}}$ de la forme $f : p_1 \to q_1$, l'unique $g \in \mathsf{Mor}_{h(\mathbf{C},\mathcal{R})}$ tel que $M(g) = f$ et $g : N(p_1) \to q_1$ est $g = \langle f, f \rangle$. Dans le deuxième triangle, pour tout $g \in \mathsf{Mor}_{h(\mathbf{C},\mathcal{R})}$ de la forme :

$$\begin{array}{ccc} p_1 & \xrightarrow{f'} & p_2 \\ \downarrow & & \downarrow h \\ p_1 & \xrightarrow[i]{} & q_2 \end{array}$$

on a $NM(h) = p_2 \xrightarrow{id} p_2$ et $p_1 \xrightarrow{f} M(h) = p_2$; on en déduit que $f = f'$ est l'unique morphisme permettant de clore le triangle. □

On notera que l'adjonction n'entraîne pas l'équivalence des catégories $\mathcal{I}$ et $h(\mathbf{C}, \mathcal{R})$ attendu que la co-unité ϵ n'est pas un isomorphisme naturel. En effet, il est clair qu'ajouter des mémoires au dessus d'un langage concurrent (ici, la simple définition de la catégorie des histoires) ne suffit pas à le rendre réversible. Il reste à modifier la sémantique opérationnelle en ajoutant les règles de transitions arrières basées sur la structure de ces mémoires.

7.2.2 Catégorie des histoires réversibles.

Soit $\mathbf{C}[\mathcal{R}^{-1}]$ la *catégorie des fractions*[3] qu'on peut obtenir en ajoutant à $\mathsf{Mor}_{\mathbf{C}}$ les inverses formels de chaque morphisme de $\mathcal{R}$. Cette catégorie peut être construite à l'aide d'une *propriété universelle*, à savoir l'existence d'un foncteur :

$$R : \mathbf{C} \to \mathbf{C}[\mathcal{R}^{-1}] \tag{7.7}$$

[3]Voir aussi [Gabriel et Zisman, 1967].

qui envoie chaque flèche de $\mathcal{R}$ vers un isomorphisme et qui est l'identité sur les objets de $\mathbf{C}$. L'universalité est obtenue en posant que pour tout foncteur $F : \mathbf{C} \to \mathbf{D}$, qui envoie une flèche de $\mathcal{R}$ vers un isomorphisme et préserve les objets de $\mathbf{C}$, il existe un foncteur $G : \mathbf{C}[\mathcal{R}^{-1}] \to \mathbf{D}$ faisant commuter le diagramme :

$$\begin{array}{ccc} \mathbf{C} & \xrightarrow{R} & \mathbf{C}[\mathcal{R}^{-1}] \\ & \searrow^{F} & \downarrow{\scriptstyle G} \\ & & \mathbf{D} \end{array}$$

Définition 7.2.2 (Catégorie des histoires réversibles) *Soit $\langle \mathcal{I}, \mathcal{R} \rangle$ un système de factorisation sur une catégorie $\mathbf{C}$ et soit R le foncteur canonique de $\mathbf{C}$ vers la catégorie des fractions. La* catégorie des histoires réversibles *$h_\star(\mathbf{C}, \mathcal{R})$ est définie par :*

- *objets : flèches $g \in \mathcal{R}$;*
- *flèches : des diagrammes commutatifs de la forme*

$$\begin{array}{ccc} p_1 & \xrightarrow{f} & p_2 \\ {\scriptstyle g_1}\downarrow & & \downarrow{\scriptstyle g_2} \\ q_1 & \xrightarrow[f_\star]{} & q_2 \end{array}$$

où $f \in \mathcal{I}$ et $f_\star \in \mathbf{C}[\mathcal{R}^{-1}]$ telles que $f_\star \circ R(g_1) = R(g_2 \circ f)$.

Si la construction de $h(\mathbf{C}, \mathcal{R})$ à partir de $\mathbf{C}$ n'était pas fonctorielle, on peut définir en revanche un foncteur évident :

$$\Psi : h(\mathbf{C}, \mathcal{R}) \to h_\star(\mathbf{C}, \mathcal{R}) \tag{7.8}$$

qui préserve les objets de $h(\mathbf{C}, \mathcal{R})$ et envoie $\langle f, f' \rangle \in \mathsf{Mor}_{h(\mathbf{C},\mathcal{R})}$ vers les diagrammes de $h_\star(\mathbf{C}, \mathcal{R})$ en appliquant R à f :

$$\begin{array}{ccc} p_1 & \xrightarrow{f'} & p_2 \\ {\scriptstyle g_1}\downarrow & & \downarrow{\scriptstyle g_2} \\ q_1 & \xrightarrow[f]{} & q_2 \end{array} \quad \overset{\Psi}{\longmapsto} \quad \begin{array}{ccc} p_1 & \xrightarrow{f'} & p_2 \\ {\scriptstyle g_1}\downarrow & & \downarrow{\scriptstyle g_2} \\ q_1 & \xrightarrow[Rf]{} & q_2 \end{array}$$

On peut adapter les définitions des foncteurs M et N définis en (7.5) et (7.6) à la catégorie des histoires réversibles. Soit donc

$$M_\star : h_\star(\mathbf{C}, \mathcal{R}) \to \mathcal{I} \tag{7.9}$$

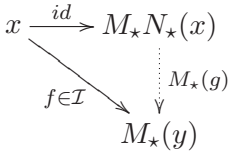
le foncteur qui associe les flèches $\langle f_\star, f' \rangle \in \mathsf{Mor}_{h_\star(\mathbf{C},\mathcal{R})}$ à $f' \in \mathsf{Mor}_{\mathcal{I}}$ et les objets $p_1 \xrightarrow{g_1} q_1$ de $h_\star(\mathbf{C}, \mathcal{R})$ à $p_1 \in \mathcal{I}$ et le foncteur

$$N_\star : \mathcal{I} \to h_\star(\mathbf{C}, \mathcal{R}) \tag{7.10}$$

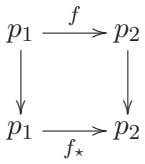
définit par $N_\star = \Psi N$.

Théorème 7.2.1 (Théorème transactionnel généralisé) *Les foncteurs $M_\star$ et $N_\star$ définissent une équivalence de catégories entre $\mathcal{I}$ et $h_\star(\mathbf{C}, \mathcal{R})$.*

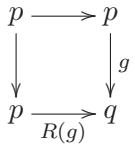
– L'unité $\eta : Id_{\mathcal{I}} \to M_\star N_\star$ reste triviale attendu que $M_\star N_\star = Id_{\mathcal{I}}$. On vérifie l'unicité de $g \in \mathsf{Mor}_{h_\star(\mathbf{C},\mathcal{R})}$ dans le diagramme commutatif suivant :



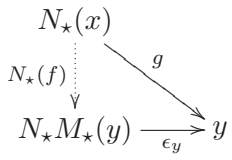
Nécessairement, g est un diagramme de la forme :



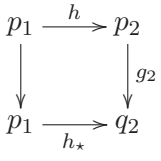
avec $f_\star \circ R(id) = R(id \circ f)$. Comme $f \in \mathcal{I}$ on a $Rf = f$, ce qui donne $f = f_\star$ et qui entraîne bien l'unicité de g.
– Pour la co-unité $\epsilon : N_\star M_\star \to Id_{h_\star(\mathbf{C},\mathcal{R})}$ on prend pour ϵ_g le diagramme commutatif :



On vérifie l'unicité de $f \in \mathsf{Mor}_{\mathcal{I}}$ dans le diagramme commutatif suivant :



Pour tout $g \in \mathsf{Mor}_{h_\star(\mathbf{C},\mathcal{R})}$ de la forme :



l'unique flèche $f \in \mathsf{Mor}_{\mathcal{I}}$ telle que $f : p_1 \to p_2$ est nécessairement $f = h$, par unicité de la décomposition de $g_2 \circ h$, ce qui clôt le diagramme ci-dessus.
– η est clairement un isomorphisme naturel ; on note de même que pour tout $g \in \mathcal{R}$, ϵ_g est bien inversible (par construction de $h(\mathbf{C}, \mathcal{R})$, g est envoyé vers un isomorphisme) et par conséquent, les catégories $\mathcal{I}$ et $h_\star(\mathbf{C}, \mathcal{R})$ sont équivalentes. □

7.3 Discussion

Dans ce chapitre, nous avons exposé les concepts théoriques sous-jacents à la construction d'un théorème transactionnel pour un calcul concurrent donné. La figure 7.1 initiale peut être remplacée par un nouveau schéma (voir Figure 7.2) indépendant du formalisme choisi : à partir de la catégorie $\mathbf{C}$ des traces de calculs du langage donné et d'un système de factorisation entre les traces irréversibles et réversibles de $\mathbf{C}$, on construit la catégorie des histoires de $\mathbf{C}$ et on obtient un théorème transactionnel pour le langage de départ.

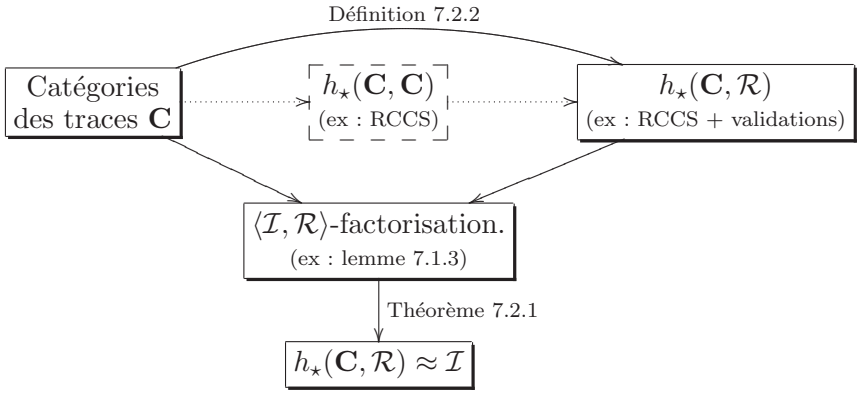


FIG. 7.2 – Programmation déclarative généralisée.

Si on compare le schéma de la figure 7.2 avec le schéma initial, on s'aperçoit que la partie correction du retour arrière est masquée par la construction de la catégorie $h_\star(\mathbf{C}, \mathcal{R})$ *via* le foncteur Ψ défini en (7.8). En effet un diagramme commutatif $\langle f, f' \rangle \in \mathsf{Mor}_{h(\mathbf{C},\mathcal{R})}$ est envoyé vers un autre diagramme commutatif $\langle Rf, f' \rangle$ par Ψ. Dans un calcul réversible, cela implique que Rf est une trace réversible telle que $Rf(Rf)^{-1} = id_{\mathbf{C}[\mathcal{R}^{-1}]}$. Si on pose

$Rf = \sigma$ et $(Rf)^{-1} = \delta^-$ cela revient à vérifier que δ et σ sont bien des traces équivalentes, ce qui correspond précisément au théorème fondamental de RCCS (Théorème 3.2.1). En résumé, l'abstraction proposée indique que pour construire un calcul réversible $h_\star(\mathbf{C}, \mathcal{R})$, il nous faut exhiber un système de factorisation $\langle \mathcal{I}, \mathcal{R} \rangle$ et implémenter *syntaxiquement* un retour arrière qui permet d'inverser les flèches de $\mathcal{R}$. Le théorème transactionnel correspondant au langage choisi est alors une conséquence du théorème 7.2.1.

On peut, par ailleurs, se demander si la construction d'un langage réversible est aussi généralisable. Une approche récente[4], vise à définir les conditions syntaxiques nécessaires à l'obtention d'un langage réversible en montrant comment prévenir l'altération des termes au cours du calcul. L'idée étant de donner abstraitement une méthode pour transformer les règles du LTS en des opérateurs statiques[5] tout en préservant la structure compositionnelle de la sémantique (format SOS). Toujours dans l'optique de s'abstraire du langage concurrent choisi, des travaux ultérieurs pourraient viser à fusionner cette approche avec le cadre catégorique de ce chapitre afin d'obtenir un formalisme qui prenne à la fois en compte les problèmes syntaxiques liés à l'implémentation du retour arrière et les concepts mathématiques sous-jacents à l'utilisation du langage pour la modélisation de systèmes transactionnels.

Ce chapitre vient clore la partie la plus formelle de cette thèse et nous consacrerons les chapitres suivants à l'exploration, moins formelle, de différentes applications ou extensions de RCCS.

[4] [Phillips et Ulidowski, 2006]

[5] Ce qui correspond dans le cas de RCCS à ajouter des mémoires aux termes.

Chapitre 8

Biologie moléculaire formelle.

Sommaire

Dans ce chapitre nous nous proposons d'illustrer l'application de notre formalisme à la modélisation de processus biologiques, domaine dans lequel une algèbre de processus réversible comme RCCS et la technique de programmation déclarative qui lui est associée prennent une dimension toute particulière. Après avoir présenté un modèle pour la biologie des systèmes basé sur le π-calcul, nous montrerons qu'il est possible d'obtenir un langage plus conforme au modèle biologique, en se servant de l'interprétation réversible fournie par RCCS, permettant de représenter un assemblage arbitraire d'agents biologiques doté d'une mesure de robustesse. Nous verrons, en particulier, comment en déduire l'influence qu'un réactant peut avoir sur la formation des complexes du système dans lequel il est plongé.

8.1 Un langage pour la cellule.

Représentation des données. On appelle *biologie (moléculaire) formelle* les approches sémantiques, visant à modéliser, simuler et analyser

des processus biologiques : autrement dit, la recherche d'un langage pour la cellule. Déjà prégnant depuis de nombreuses décennies, le besoin d'un formalisme permettant de représenter des interactions biologiques à un niveau abstrait est devenu quasiment vital au cours de ces dernières années. De nouveaux outils de mesure à l'échelle moléculaire, tels que les puces à ADN, produisent des quantités gigantesques de données plus rapidement qu'on est réellement en mesure de les analyser. Plusieurs notations graphiques, comme la base de donnée KEGG[1] ont été mises au point afin de pallier le manque de formalisme. Toutefois l'accès à la dynamique des réactants demande, dans la plupart des cas, un travail de reconstruction non automatisable basé sur des publications scientifiques et des comptes rendus d'expériences. Bien entendu, la question de mêler représentation formelle et dynamique moléculaire n'est pas restée ignorée et la biologie formelle s'est développée, en marge de ces avancées technologiques, en proposant différents types de langages. Les sémantiques opérationnelles les plus fréquemment utilisées par les biologistes sont basées sur des systèmes d'équations différentielles dont les coefficients sont des paramètres ajustables dans le but de coller au plus près des observations réalisées *in vivo*. Ces approches souffrent toutefois d'un inconvénient majeur : la découverte d'un nouveau paramètre entrant en jeu dans l'expérience observée entraîne souvent une redistribution complète des coefficients précédents. Bien qu'ils permettent de donner une description assez fine des évolutions potentielles d'un système, les modèles à base d'équations différentielles sont donc relativement arbitraires et principalement quantitatifs.

Quand la sémantique vient en aide. Utiliser un langage abstrait, avec une sémantique opérationnelle bien définie, constituerait un gain certain en permettant de découpler la dynamique qualitative, qui s'adapte assez facilement aux nouvelles données (séquence des événements observés, causalité, accessibilité), de la dynamique quantitative obtenue en fixant les coefficients régentant les transitions du modèle. C'est sur ces points précis que l'informatique théorique peut apporter sa contribution et, en particulier, les modèles de concurrence qui se prêtent bien aux systèmes biologiques constitués de réactants en constante compétition. Dans ce but, plusieurs outils issus de la théorie de concurrence, comme les réseaux de Petri[2], les automates hybrides[3] ou les *state charts*[4], ont été appliqués à des modèles biologiques. Si

[1] Kyoto Encyclopedia of Genes and Genomes, Bioformatics Center, Institute for Chemical Research, Kyoto University, `www.genome.jp/kegg/`

[2] [Hofestädt et Thelen, 1998, Matsuno *et al.*, 2000]

[3] [Ghosh et Tomlin, 2001]

[4] [Kam *et al.*, 2001]

ces modèles permettent bien de *décrire* des interactions bio-moléculaires, ils ne sont pas à proprement parler un *langage* biologique dans le sens où il n'existe pas véritablement de correspondance entre les termes syntaxiques de ces calculs et les objets biologiques étudiés. Cette correspondance est pourtant une propriété intéressante : on pourrait imaginer, en supposant que toutes les primitives du langage soient des événements biologiques plausibles, effectuer des allers-retours entre le modèle et l'éprouvette en ajoutant de nouvelles primitives au langage chaque fois que les précédentes seraient insuffisantes pour décrire un événement biologique observé. Récemment, Regev et Shapiro[5] ainsi qu'un certain nombre d'autres auteurs[6] ont mis en avant l'idée qu'une algèbre de processus comme le π-calcul[7] pourrait être le langage recherché. L'idée directrice est la suivante : on considère que les molécules dans une solution sont représentées par des processus formels et que les communications de ces derniers s'interprètent comme des liaisons entre les réactants biologiques qu'ils représentent. Suivant l'approche décrite plus haut, consistant à ajouter au langage les primitives correspondant à des observations biologiques n'ayant pas de correspondant formel, divers extensions ou adaptations du π-calcul ont été proposées, notamment afin de représenter les compartiments biologiques qui jouent un rôle fondamental dans la régulation des interactions[8].

Réversibilité et biologie. Toutefois, on peut se demander si le formalisme de départ de ces approches est réellement adéquat. Les modèles basés sur le π-calcul ont la possibilité de créer de nouveaux noms de canaux et d'exporter ces noms à des processus distants afin de créer des communications privées. Ce mécanisme de passage de nom est très expressif et sans réelle limitation de son usage, il entraîne la perte de la correspondance processus-molécules, pourtant l'argument fondamental en faveur de l'usage d'une algèbre de processus comme langage pour la biologie formelle. Si on regarde de plus près les modèles proposés, la création et le passage de noms sont rendus nécessaires afin de rendre compte du fait que la plupart des complexations de molécules sont réversibles. Prenons par exemple une protéine A qui va se lier à une protéine B pour former le complexe $A : B$, qui lui-même peut se décomplexer en A et B. Dans les modèles basés sur le π-calcul on donne la capacité à A de créer un nouveau nom, disons x, qu'il envoie à B lors de la complexation sur un canal public. Si le complexe

[5] [Regev *et al.*, 2001, Priami *et al.*, 2001, Regev et Shapiro, 2002]
[6] [Baldi *et al.*, 2002, Danos et Laneve, 2003, Blossey *et al.*, 2006]
[7] [Milner *et al.*, 1992]
[8] [Cardelli et Gordon, 2000, Cardelli, 2004, Priami et Quaglia, 2004]

$A : B$ doit se briser, on effectuera une communication sur le canal x connus des seuls A et B afin de qu'ils se mettent tous les deux d'accord pour se séparer. Outre le fait que ce type de mécanisme rend les modèles arbitraires, peu lisibles et plus difficiles à analyser, on est en fait en train de *simuler* les réactions biologiques et non plus en train de définir un *modèle* de ces dernières. Sachant que la réversibilité des interactions biologiques fait office de loi et non pas d'exception dans la plupart des cas, il est naturel de considérer qu'un langage pour la biologie devrait être intrinsèquement réversible (ce qui n'exclut pas de maintenir un mécanisme de passage de noms si on y trouve une justification biologique). Plus encore, on peut aussi se demander *pourquoi* la plupart des complexations biologiques sont réversibles. Une explication peut se trouver tout simplement dans ce qui a motivé l'élaboration de RCCS. Dans une solution bio-chimique, plusieurs milliers de réactants sont en compétition pour l'acquisition de ressources (sites de liaisons, nutriments), on peut donc dire que ces systèmes sont hautement transactionnels. Pour autant, aucune de ces transactions ne semble se bloquer dans un état incohérent, comme un gène à demi-activé ou un complexe à moitié formé. Si on exclut l'usage du passage de noms privés entre molécules, une hypothèse demeure : ces complexes incohérents se produisent mais ils sont instables et le biologiste n'observe que les complexes stables que la cellule peut utiliser comme briques constitutives pour implémenter la machinerie moléculaire dont elle a besoin.

Plan. Dans ce chapitre nous introduisons en premier lieu la syntaxe et la sémantique opérationnelle du π-calcul. Nous illustrerons ensuite la perte de correspondance "processus = molécule", mentionnée plus haut, à travers un exemple simple de système de régulation transcriptionnelle. Nous verrons ensuite qu'une algèbre de processus réversible comme RCCS permet de modéliser ce type de système, tout en conservant la correspondance recherchée. De plus, nous verrons que la structure des mémoires permet d'associer, à tout système RCCS, un graphe donnant la structure des complexes formés ainsi qu'un mesure abstraite de leurs robustesses. Nous verrons qu'il est possible de tirer profit de ces informations afin de déduire le rôle possiblement inhibiteur ou activateur d'un agent vis à vis d'un substrat donné. Nous conclurons le chapitre par une discussion sur l'usage d'une algèbre de processus réversible, pour la biologie formelle, basée sur un calcul plus expressif que CCS.

8.2 Biologie des systèmes et π-calcul.

8.2.1 Syntaxe du π-calcul.

Comme CCS, le π-calcul est une algèbre dont les termes sont des processus communiquant sur des canaux nommés. Le pouvoir expressif du π-calcul provient de la possibilité de créer et d'échanger des noms de canaux lors des communications. La syntaxe complète du π-calcul est donnée Figure 8.1. Afin de conserver les notations introduites pour CCS, on utilise ici une version du π-calcul dans laquelle la récursion est traitée au moyen de constantes de récursion. On suppose donc que chaque processus est accompagné d'un ensemble de définitions $\Delta = \{\mathsf{D}_i(\tilde{x}_i) := p_i\}$.

$$\begin{array}{lrcl} \text{Capacité} & \pi & := & \bar{x}y \mid x(y) \mid \tau \\ \text{Processus} & p & := & \pi.p \mid p + p \mid p \parallel p \mid \mathsf{D}(\tilde{x}) \mid p\backslash x \mid 0 \end{array}$$

FIG. 8.1 – Syntaxe du π-calcul.

La capacité $\bar{x}y$ dénote l'envoi d'un nom y sur le canal x ; son complémentaire est la capacité $x(z)$ qui permet la réception d'un nom sur le canal x. La variable z est alors instanciée, dans la continuation de la capacité, par le nom reçu. Dans le π-calcul, le préfixe de réception $x(y).p$ est donc un lieur pour y dans p, comme indiqué dans la construction de l'ensemble $\mathsf{fn}(p)$ des noms libres dans le processus p.

$$\begin{array}{lcl} \mathsf{fn}(0) & := & \emptyset \\ \mathsf{fn}(\mathsf{D}(\tilde{x})) & := & \bigcup_i x_i \in \tilde{x} \\ \mathsf{fn}(\bar{x}y.p) & := & \{x, y\} \cup \mathsf{fn}(p) \\ \mathsf{fn}(x(y).p) & := & \{x\} \cup \mathsf{fn}(p) - \{y\} \\ \mathsf{fn}(p + q) & := & \mathsf{fn}(p) \cup \mathsf{fn}(q) \\ \mathsf{fn}(p \parallel q) & := & \mathsf{fn}(p) \cup \mathsf{fn}(q) \\ \mathsf{fn}(p\backslash x) & := & \mathsf{fn}(p) - \{x\} \end{array}$$

8.2.2 Sémantique opérationnelle.

La sémantique opérationnelle du π-calcul se distingue quelque peu de celle de CCS (voir Section 1.1.2) car elle permet de procéder à l'échange d'un nom de canal entre deux processus. La sémantique opérationnelle est traditionnellement donnée sous forme de LTS, mais nous nous contenterons

d'un simple système de réductions (donné Figure 8.2), suffisant pour les besoins de ce chapitre, dont le règle centrale est la communication :

$$\bar{x}y.p + q \parallel x(z).p' + q' \to p \parallel p'\{y/z\} \tag{8.1}$$

Où $p'\{y/z\}$ dénote la substitution de z par y dans le processus p'. On se ramène à cette règle en manipulant les processus à congruence structurelle près donnée par :

$$\begin{array}{lcll} p \parallel q & \equiv & q \parallel p & \text{PAR-COM} \\ (p \parallel q) \parallel q' & \equiv & p \parallel (q \parallel q') & \text{PAR-ASSO} \\ p + q & \equiv & q + p & \text{SUM-COM} \\ (p + q) + q' & \equiv & p + (q + q') & \text{SUM-ASSO} \\ p\backslash x\backslash y & \equiv & p\backslash x\backslash y & \text{RES-COM} \\ (p\backslash x) \parallel q & \equiv & (p \parallel q)\backslash x \quad \text{si } x \notin \mathsf{fn}(q). & \text{SCOPE-EXT} \end{array}$$

La création d'un nouveau nom ainsi que sa diffusion peut s'illustrer de la manière suivante : soit $p = (\bar{x}y.p')\backslash y \parallel x(z).\bar{z}t.q$ un π-processus composé d'un sous terme $\bar{x}y.p'$ capable d'émettre le nom y sur le canal x. On peut faire "passer" y au membre droit de p en appliquant la règle SCOPE-EXT :

$$p \equiv p_1 = (\bar{x}y.p \parallel x(z).\bar{z}t.q)\backslash y$$

en supposant que $y \notin \mathsf{fn}(q)$ (ce qu'on peut toujours obtenir par α-conversion) puis en effectuant la réduction $p_1 \to (p' \parallel \bar{y}t.q\{y/z\})\backslash y$.

$$\frac{}{(\bar{x}y.p + q) \mid (x(z).p' + q') \to p \mid p'\{y/z\}}\ (\text{com})$$

$$\frac{}{\tau.p \to p}\ (\text{tau})$$

$$\frac{p \to p'}{p \mid q \to p' \mid q'}\ (\text{par-left}) \qquad \frac{p \to p'}{p + q \to p'}\ (\text{sum-left})$$

$$\frac{p \equiv p' \to q' \equiv q}{p \to q}\ (\text{equiv}) \qquad \frac{p \to p'}{p\backslash x \to p'\backslash x}\ (\text{res})$$

FIG. 8.2 – Sémantique de réductions pour le π-calcul.

8.2.3 Application à la biologie des systèmes.

Modélisation. Nous montrons maintenant comment définir le modèle π-calcul d'un système simple de régulation de la production d'une protéine au niveau de l'ADN d'une cellule[9]. Dans cet exemple, on considère un brin d'ADN contenant un gène codant pour la production d'une protéine B. Le brin d'ADN possède des sites de liaisons sur lesquels des protéines peuvent se connecter, de manière réversible, afin d'*activer* ou d'*inhiber* la transcription du gène, causée par la liaison sur le brin d'une machine de transcription T. Le système qu'on souhaite modéliser est donné figure 8.3. Il est composé du brin d'ADN avec ses sites de liaisons, d'une protéine inhibitrice I, d'une protéine activatrice A et d'un système de transcription T. Lorsque le complexe T

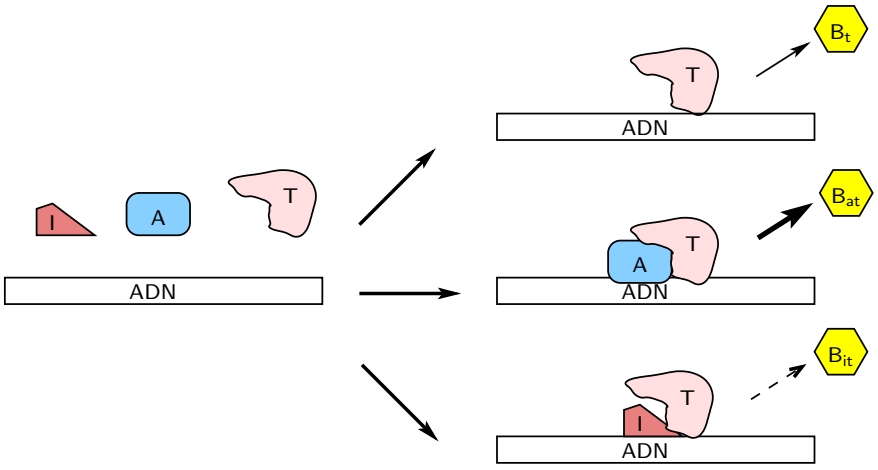


FIG. 8.3 – Système de régulation transcriptionnelle.

se lie au brin d'ADN il produit toujours la protéine B, mais le taux de production, symbolisé par une flèche plus ou moins épaisse dans la figure, varie grandement selon que T a pu se lier à l'activateur ou non. On ne peut pas représenter cette différence dans un π-calcul simple, il faudra pour cela avoir recours à un modèle plus quantitatif qui, en plus d'exprimer la possibilité d'une réaction, traduira ces variations en ajoutant des taux aux capacités standard du calcul. On se contente donc de donner le processus π qui permet de représenter de manière qualitative ce système de régulation.

[9][Francois et Hakim, 2004]

$$
\begin{array}{ll}
\mathsf{ADN}(a,i,t) & := a(x).\mathsf{ADN_a}(a,i,t,x) + t(x).\mathsf{ADN_t}(a,i,t,x) + i(x).\mathsf{ADN_i}(a,i,t,x) \\
\mathsf{T}(t) & := \big(\bar{t}u.\bar{u}.\mathsf{T}(t)\big)\backslash u \\
\mathsf{A}(a) & := \big(\bar{a}u.(\tau.0 + \bar{u}.\mathsf{A}(a))\big)\backslash u \\
\mathsf{I}(i) & := \big(\bar{\imath}u.(\tau.0 + \bar{u}.\mathsf{I}(i))\big)\backslash u
\end{array}
$$

Dans l'état initial, le brin d'ADN possède trois types de sites de liaisons : $a(x)$ pour la connexion d'une protéine activatrice, $t(x)$ pour la machine de transcription et $i(x)$ pour l'inhibiteur. Noter que dans les trois cas, les connexions sont accompagnées d'un nom x qui permettra de gérer les déconnexions. Le schéma des trois réactants T, A et I sont similaires : les protéines commencent par se connecter au brin d'ADN en transmettant un nom frais u, qu'elles peuvent ensuite utiliser pour annoncer une décomplexation. Les protéines activatrices et inhibitrices sont aussi susceptibles de se dégrader après un certain temps symbolisé par τ. En fonction de l'influence des réactants T, A et I, le brin d'ADN peut se trouver dans un des 5 états suivants :

$$
\begin{array}{lcl}
\mathsf{ADN_a}(a,i,t,u_a) & := & t(x).\mathsf{ADN_{at}}(a,i,t,u_a,x) + u_a().\mathsf{ADN}(a,i,t) \\
\mathsf{ADN_i}(a,i,t,u_i) & := & t(x).\mathsf{ADN_{it}}(a,i,t,u_i,x) + u_i().\mathsf{ADN}(a,i,t) \\
\mathsf{ADN_t}(a,i,t,u_t) & := & \tau.(\mathsf{B_t} \parallel u_t().\mathsf{ADN_t}(a,i,t)) + u_t().\mathsf{ADN}(a,i,t) \\
\mathsf{ADN_{at}}(a,i,t,u_a,u_t) & := & \tau.(\mathsf{B_{at}} \parallel u_t().\mathsf{ADN}(a,i,t)) \\
 & & + u_a().\mathsf{ADN_t}(a,i,t,u_t) + u_t().\mathsf{ADN_a}(a,i,t,u_a) \\
\mathsf{ADN_{it}}(a,i,t,u_i,u_t) & := & \tau.(\mathsf{B_{it}} \parallel u_t().\mathsf{ADN}(a,i,t)) \\
 & & + u_i().\mathsf{ADN_t}(a,i,t,u_t) + u_t().\mathsf{ADN_i}(a,i,t,u_i)
\end{array}
$$

Dans les états de la forme $\mathsf{ADN_{xt}}$ où $\mathsf{x} \in \{\mathsf{i}, \mathsf{a}, \epsilon\}$, c'est à dire correspondant au brin d'ADN lié à la machine de transcription, le système est capable de produire une des variantes de B symbolisant des concentrations différentes de la même protéine avec, intuitivement, $[\mathsf{B_{it}}] \ll [\mathsf{B_t}] < [\mathsf{B_{at}}]$. L'état initial du système est donné par :

$$reg := \mathsf{ADN}(a,i,t) \parallel \mathsf{A}(a) \parallel \mathsf{I}(i) \parallel \mathsf{T}(t) \tag{8.2}$$

Commentaires. Un première critique qu'on peut adresser au modèle est qu'il rompt la correspondance recherchée entre primitives du calcul et événements biologiques. Nous allons voir que la rupture de cette correspondance ne permet pas d'être "offensif" vis à vis du modèle biologique et donc de suggérer des approches potentiellement enrichissantes. En premier lieu, le mécanisme de décomplexation des protéines est encodé par un mélange de passage de nom frais et de récursion qui n'est pas réellement motivé par un correspondant biologique. On peut arguer que, dans les modèles π-calcul, la

liaison entre deux agents est caractérisée non pas par une simple communication mais par le partage d'un nom privé. Toutefois nous avons vu que l'usage de la récursion entraînait une explosion dans l'entrelacement des transitions qui pouvait nuire considérablement à l'analyse des systèmes. En plus de renoncer à la correspondance intuitive entre communication et liaison, on risque de rendre l'analyse automatique de gros systèmes plus délicate.

Nous pouvons, par ailleurs, émettre une critique plus subtile mais aussi plus importante : afin de pouvoir différencier les trois états de l'ADN dans lesquels la transcription de B peut démarrer, nous avons codé une partie du passé du système dans les noms des constantes de récursion. Pour respecter les observations données par les biologistes on doit alors indiquer *à la main* que le taux de production de B est plus fort dans l'état $\mathsf{ADN_{at}}$ que dans $\mathsf{ADN_t}$ ou $\mathsf{ADN_{it}}$. Dans une variante plus quantitative de π-calcul[10], cela revient à attribuer un taux de production différent dans chacun des cas. Si un biologiste arrivait avec une expérience infirmant ces taux, par exemple en décrétant que $\mathsf{ADN_{at}}$ produit moins de B que $\mathsf{ADN_t}$, on pourrait adapter le modèle en changeant simplement les coefficients sans modifier le système d'aucune autre manière. Pourtant, si la nouvelle expérience est vraie, le modèle biologique est bien *qualitativement* différent car la protéine A ne favorise plus la production de B. Le modèle π ne permet donc pas de rendre compte, de manière structurelle, de l'influence de A sur la production de B, cette dernière étant totalement déterminée par les coefficients qu'on ajoute au modèle. On peut remarquer qu'ajouter une possibilité de liaison entre A et T, comme suggéré par la figure 8.3, ne change pas la nécessité d'introduire des coefficients pour distinguer les différents taux de production possibles de B. Sachant que le gène codant pour B n'est exprimé que si la machine T est parvenue à se lier à l'ADN, on peut supposer que la protéine A est un obstacle à la déconnexion de T et qu'au contraire la protéine I favorise les chances de déconnexion. Dans la section suivante nous allons voir qu'en se servant de RCCS comme langage de modélisation, on peut aboutir à une représentation de notre exemple sans utiliser de mécanisme de passage de nom et en mettant en lumière de rôle activateur de A (et le rôle inhibiteur de I), sans faire appel à des coefficients *ad hoc* et en se servant de la causalité entre les liaisons induite par la structure des mémoires pour mesurer abstraitement la robustesse des complexes.

[10] [Priami *et al.*, 2001, Blossey *et al.*, 2006]

8.3 Biologie formelle avec RCCS.

8.3.1 Les systèmes RCCS vus comme des graphes.

Modèle. Afin de simplifier les notations, on considère la syntaxe souple de RCCS, donnée dans la section 3.2.3, dans laquelle des adresses $l \in \mathcal{L}$ remplacent les mémoires comme identifiants de communications. Dans la correspondance qu'on souhaite maintenir avec les modèles biologiques, on associe la trace d'une connexion dans les mémoires des processus avec un liaison biologique. On reprend l'exemple de la figure 8.3 en commençant par fixer certaines hypothèses devant être observables dans le modèle :

1. La transcription peut démarrer lorsque T est fixé sur le brin d'ADN.
2. La protéine A renforce les chances de produire du B lorsqu'elle est connectée sur l'ADN.
3. I appauvrit les chances de produire du B lorsqu'elle est sur le brin d'ADN.
4. Les connexions de A et I sont concurrentes à la connexion de T.

Pour satisfaire ces conditions, nous proposons la modélisation suivante du brin d'ADN :

$$\mathsf{ADN}(a, i, t_i, t) := (a.0 + i.0) \parallel t.(t_i \parallel \underline{\tau}.\mathsf{B})$$

Dans notre modèle les connexions sur a et i sont conflictuelles et toutes deux concurrentes à une connexion sur t. Suite à cette dernière, la transcription de B peut immédiatement débuter indépendamment de l'état des sites a et i. Pour répondre à la troisième contrainte, nous avons supposé l'existence d'un deuxième site de liaison t_i qui sera partagé par l'inhibiteur et la machine transcriptionnelle. La connexion de T à t_i n'est pas nécessaire pour faire débuter la transcription mais ajoutera de la robustesse au complexe formé par T et le brin d'ADN. La production de B est réalisée au bout d'un certain temps marqué par l'action interne et irréversible $\underline{\tau}$. On modélise les autres réactants du système par :

$$\begin{array}{lcl} \mathsf{A}(a, t_a) & := & \bar{a}.\bar{t}_a.0 \\ \mathsf{I}(i, t_i) & := & \bar{\imath}.\bar{t}_i.0 \\ \mathsf{T}(t, t_a, t_i) & := & \bar{t}.(t_a.0 \parallel \bar{t}_i.0) \end{array}$$

La protéine activatrice possède maintenant un rôle explicite : après connexion à l'ADN sur le site a, elle présente un nouveau site t_a qui permettra de renforcer la stabilité de T sur le brin. Dans l'état le plus stable, T sera directement

lié à l'ADN par les sites t et t_i, ainsi qu'à l'activateur *via* le site t_a. L'état initial du modèle est donné par le processus CCS :

$$reg' := \boxed{\mathsf{A}(a,t_a)} \parallel \boxed{\mathsf{I}(i,t_i)} \parallel \boxed{\mathsf{T}(t,t_a,t_i)} \parallel \boxed{\mathsf{ADN}(a,i,t_i,t)} \tag{8.3}$$

dans lequel nous avons encadré les sous processus correspondant à des réactants distincts. On peut simuler (qualitativement) le modèle en relevant l'état initial (8.3) dans RCCS :

$$\boxed{\langle 1\rangle \triangleright \mathsf{A}(a,t_a)} \parallel \boxed{\langle 12\rangle \triangleright \mathsf{I}(i,t_i)} \parallel \boxed{\langle 122\rangle \triangleright \mathsf{T}(t,t_a,t_i)} \parallel \boxed{\langle 222\rangle \triangleright \mathsf{ADN}(a,i,t_i,t)}$$

Graphes de connexions. Afin d'associer chaque processus à une molécule, on note @R l'adresse initial du réactant R dans $\ell(reg')$. On a :

$$@\mathsf{A} = 1 \quad @\mathsf{I} = 12 \quad @\mathsf{T} = 122 \quad @\mathsf{ADN} = 222$$

Cette notation nous permet de suivre l'évolution des réactants au cours d'un calcul. On note $m@\mathsf{R}$ lorsque m est de la forme $m' \cdot \langle @\mathsf{R}\rangle$. On dira que r est un composant du réactant R si toutes ses mémoires débutent par l'adresse de R, c'est à dire $\forall m \sqsubseteq r : m@\mathsf{R}$. Par ailleurs on peut se servir de ces adresses pour observer les différentes complexations qui peuvent avoir lieu au cours d'un calcul. Ainsi, l'application de la règle de communication :

$$\frac{r \xrightarrow{l:\bar{a}} r' \quad s \xrightarrow{l:a} s'}{r \parallel s \xrightarrow{l:\tau} r' \parallel s'}$$

donne naissance à une liaison entre les réactants R_1 et R_2 si les mémoires, disons m_1 et m_2, contenant la localité l sont telles que $m_1@\mathsf{R}_1$ & $m_2@\mathsf{R}_2$. On notera $\mathsf{R}_1 \overset{l}{\bowtie} \mathsf{R}_2$ lorsque une communication identifiée par l entraîne une liaison entre R_1 et R_2 selon les termes définis ci-dessus.

Définition 8.3.1 (Graphe de connexion) *On associe a tout système r, un* graphe de connexion $\mathcal{G}(r) := (V, E)$ *défini par :*

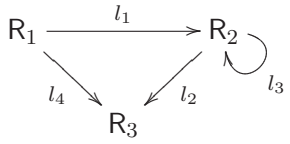
- *V est l'ensemble des réactants de r.*
- *$E \subseteq V \times \mathsf{Loc} \times V$ est l'ensemble des arêtes étiquetées, pour lesquelles $(\mathsf{R}_1 \xrightarrow{l} \mathsf{R}_2) \in E$ si $\mathsf{R}_1 \overset{l}{\bowtie} \mathsf{R}_2$.*

Exemple 8.3.1 *On considère le système suivant :*

$$r \quad := \quad \boxed{\langle l_4,u\rangle\langle l_1,\bar{x}\rangle\langle @\mathsf{R}_1\rangle \triangleright 0} \parallel \boxed{\langle l_4,\bar{u}\rangle\langle l_2,y\rangle\langle @\mathsf{R}_3\rangle \triangleright 0}$$

$$\parallel \boxed{\langle l_3,z\rangle\langle 1\rangle\langle l_1,x\rangle\langle @\mathsf{R}_2\rangle \triangleright 0} \parallel \boxed{\langle l_3,\bar{z}\rangle\langle l_2,\bar{y}\rangle\langle 2\rangle\langle l_1,x\rangle\langle @\mathsf{R}_2\rangle \triangleright 0}$$

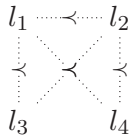
Le graphe $\mathcal{G}(r)$ associé à r est :



Mesure de robustesse. Soit $\langle l, \alpha, p\rangle \cdot m$ une mémoire apparaissant dans un système r donné. Pour tout $l' \in \mathcal{L}oc(m)$ on pose $l > l'$ pour indiquer que l' est plus profond que l dans une mémoire de r. On peut associer à tout graphe de connexion (V, E) une relation de *repliement* $\prec \subseteq E \times E$ définie par :

$$(\mathsf{R}_1 \xrightarrow{l} \mathsf{R}_2) \prec (\mathsf{R}_3 \xrightarrow{l'} \mathsf{R}_4) \text{ si } l < l' \tag{8.4}$$

Intuitivement cette relation indique que la complexation entre R_1 et R_2 ne pourra être brisée sans rupture préalable de l'arête entre R_3 et R_4. Dans le graphe de l'exemple précédent on obtient les dépendances entre arêtes suivantes :



dont on déduit, par exemple, que la liaison l_1 est renforcée par toutes les autres liaisons du système.

Définition 8.3.2 (Graphe de repliements) *On appelle* graphe de repliements *d'un système r une structure $\mathcal{F}(r) := \langle \mathcal{G}(r), \prec\rangle$ où $\prec$ correspond à la relation de repliement sur les arêtes de G.*

Il est donc possible d'évaluer, sans donner de coefficients concrets, l'influence (positive, négative, ou neutre) d'un réactants sur la solidité d'un complexe donné. Dans la section suivante, nous revenons à notre exemple de régulation transcriptionnelle dans lequel nous allons considérer que la production de B est proportionnelle à la robustesse du lien entre T et l'ADN.

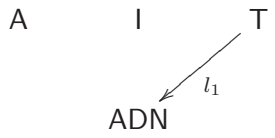
8.3.2 Analyse qualitative.

Étudions à présent les divers états dans lesquels la transcription de B peut débuter. La transcription simple, qui correspond au cas où T est le seul réactant connecté à l'ADN se produit à partir de l'état suivant :

$$\boxed{\langle @\mathsf{A}\rangle \triangleright \mathsf{A}(a, t_a)} \parallel \boxed{\langle @\mathsf{I}\rangle \triangleright \mathsf{I}(i, t_i)} \parallel \boxed{\langle l_1, \bar{t}\rangle\langle @\mathsf{T}\rangle \triangleright (t_a.0 \parallel \bar{t}_i.0)}$$

$$\parallel \boxed{\langle 1\rangle\langle @\mathsf{ADN}\rangle \triangleright (a.0 + i.0) \parallel \langle l_1, t\rangle\langle 2\rangle\langle @\mathsf{ADN}\rangle \triangleright (t_i \parallel \underline{\tau}.\mathsf{B})}$$

Le graphe de repliement de ce système est :

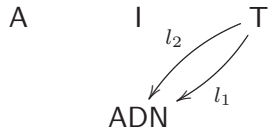


On peut noter que le complexe constitué de T et du brin d'ADN est pour le moment relativement faible. Bien que la transcription puisse débuter "immédiatement" en effectuant l'action de validation $\underline{\tau}$, on peut supposer que ce mécanisme prenne un certain temps durant lequel une décomplexation de T est possible, stoppant le mécanisme de transcription. Toutefois, en l'absence d'inhibiteur, le site de connexion t_i de l'ADN est libre et peut être utilisé par T pour affermir sa prise sur le brin en attendant que la transcription démarre. On obtient alors l'état suivant :

$$\boxed{\langle @\mathsf{A}\rangle \triangleright \mathsf{A}(a, t_a)} \parallel \boxed{\langle @\mathsf{I}\rangle \triangleright \mathsf{I}(i, t_i)} \parallel \boxed{\langle 1\rangle\langle l_1, \bar{t}\rangle\langle @\mathsf{T}\rangle \triangleright t_a.0 \parallel \langle l_2, \bar{t}_i\rangle\langle 2\rangle\langle l_1, \bar{t}\rangle\langle @\mathsf{T}\rangle \triangleright 0}$$

$$\parallel \boxed{\langle 1\rangle\langle @\mathsf{ADN}\rangle \triangleright (a.0 + i.0) \parallel \langle l_2, t_i\rangle\langle 1\rangle\langle l_1, t\rangle\langle 2\rangle\langle @\mathsf{ADN}\rangle \triangleright 0 \parallel \langle 2\rangle\langle l_1, t\rangle\langle 2\rangle\langle @\mathsf{ADN}\rangle \triangleright \underline{\tau}.\mathsf{B}}$$

dont le graphe de repliement est :



avec $l_1 \prec l_2$. Il est maintenant plus difficile d'empêcher la transaction de se conclure, la liaison critique l_1 étant protégée par une la liaison l_2. Si on prend l'analogie d'un jeu entre un joueur J_1 cherchant à produire du B et un autre J_2 cherchant à l'empêcher : tant que la transcription n'a pas commencé, J_2 a toujours une série de coups gagnants puisque toutes les connexions sont réversibles. En revanche, J_1 est maintenant en meilleure posture puisque une décomplexation de T nécessite maintenant de rompre la connexion l_2 en premier, puis la connexion l_1. Suivant cette logique on peut "lire" directement l'influence positive de A sur la transcription en notant que la connexion de la protéine activatrice permet d'offrir une prise supplémentaire à T *via* le site

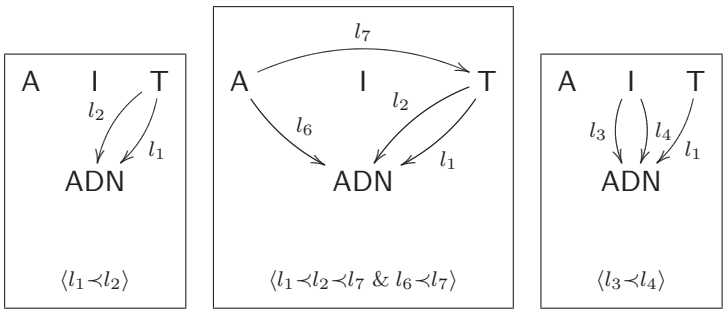


FIG. 8.4 – Les différents états de la transcription.

t_a. Nous donnons, Figure 8.4, les graphes de repliements qui résument les informations qu'on peut extraire des trois états de transcription possibles.

Afin d'évaluer les chances de succès de la transcription on mesure dans chacun des cas la robustesse du lien l_1 qui permet de déclencher la production de B. On obtient les graphes de repliements de la figure 8.4 dans lesquels on peut évaluer la résistance du complexe critique entre T et l'ADN. Muni du graphe de repliement d'un complexe, on peut se donner une approximation quantitative de la robustesse d'un lien donné. Cette forme de mesure considère que deux liens concurrents peuvent se rompre simultanément (ce qui est certainement une approximation très large). La robustesse d'un lien l est donnée par la longueur de la plus longue chaîne décroissante de repliements aboutissant à l. Par exemple, la robustesse d'un complexe formé des liens, l_1, l_2 et l_3, dont le repliement est donné par $l_1 \prec l_2, l_1 \prec l_3$ est identique à celle du repliement $l_1 \prec l_2$ et vaut 1. Dans les graphes de la figure 8.4, la robustesse du lien l_1 (qu'on suppose proportionnelle à la production de B), notée $\rho(l_1)$, est :

- Transcription simple : $\rho(l_1) = 1$;
- Transcription activée : $\rho(l_1) = 3$;
- Transcription inhibée : $\rho(l_1) = 0$

On peut ensuite comparer ces résultats avec les données numériques tirées d'un modèle biologique. Pour donner un ordre de grandeur, dans l'exemple qui a motivé notre modèle[11], les taux de production de B indiqués en fonction des différents états de réplication sont : $[B]_{\mathsf{t}} = 0.37$ (transcription simple), $[B]_{\mathsf{at}} = .89$ (activée) et $[B]_{\mathsf{it}} = 0.027$ (inhibée) ce qui est

[11] [Francois et Hakim, 2004]

relativement conforme avec les approximations données plus haut.

8.4 Discussion.

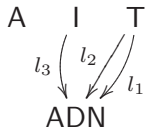
Nous avons vu qu'une approche possible de la recherche d'un modèle pour la biologie des systèmes revient à partir d'un langage minimal dont toutes les primitives ont un correspondant biologique plausible. Le fait que la plupart des interactions biologiques soient réversibles a donc motivé l'usage d'une algèbre de processus dont les primitives sont elles-même réversibles. Disposer d'un retour arrière intégré au calcul permet alors d'entretenir la correspondance avec les événements biologiques, sans introduire de mécanisme s'apparentant à de la simulation. L'intérêt de préserver cette correspondance avec le monde biologique n'est pas "philosophique" car elle permet de formuler des hypothèses significatives sur la structure des complexes formés à partir de relevés expérimentaux. Ainsi, nous avons pu "observer" qualitativement la différence entre une transcription activée et une transcription inhibée en mesurant la robustesse du lien critique, dans le complexe correspondant. Toutefois la mesure de robustesse que nous avons introduite est sans doute trop grossière et des travaux ultérieurs pourraient permettre de définir un mesure plus fine, prenant en compte les probabilité de rupture de chacun des liens en fonction, par exemple, de leur nature chimique.

Nous noterons que des travaux ultérieurs, basés sur des modèles π-calcul, ont été consacrés à l'étude des relations de causalités entre événements de modèles biologiques[12] présentant des similarités avec l'approche présentée dans ce chapitre. Nous avons déjà souligné le fait que l'usage du π-calcul pouvait entraîner une perte de correspondance entre biologie et formalisme, on peut donc craindre que des analyses de causalités sur ce type de modèle entraîne des résultats induits par des transitions purement artefacts. Par ailleurs, l'information déduite de ces analyses concerne exclusivement les dépendances entre transitions tandis que la construction du graphe de repliements d'un système nous donne une information sur les dépendances entre agents pouvant être constitués de plusieurs sous-processus mis en parallèle, comme dans le cas du modèle de notre brin d'ADN.

Concernant la correspondance que nous avons tâché d'entretenir avec les événements biologiques nous pouvons apporter un nouvel éclairage sur une raison possible de la présence de complexations réversibles en biologie. Si on considère le modèle que nous avons proposé, nous avons négligé de nombreux

[12] [Baldi *et al.*, 2002]

complexes intermédiaires non significatifs. Par exemple le complexe :



qui est équivalent a la simple complexation entre T et l'ADN du point de vue de la robustesse de T. D'un point de vue biologique, ces deux complexes ne sont *a priori* pas les mêmes sachant que le complexe ci-dessus mobilise une protéine I qui pourrait être utilisée ailleurs. Du point de vue de I, il s'agit donc d'un verrou car il n'y a plus moyen d'inhiber T qui s'est totalement connectée au brin. Une autre approche intéressante serait donc de tenter de lier les taux élevés de décomplexations avec la présence de consensus distribués entraînant une forte possibilité d'échec dans les transactions entreprises.

Comme nous l'avons dit, grâce à un mécanisme de retour arrière il n'est pas nécessaire d'utiliser le passage de noms privés afin de modéliser les décomplexations d'un système. Dans le modèle présenté, RCCS est donc suffisamment expressif pour exprimer tous les événements du système. Il est toutefois clair que dans un modèle plus complexe, faisant intervenir des membranes où des mécanismes dans lesquels le passage de noms est raisonnable (les phosphorylations sont des exemples biologiques typiques, dans lesquels un mécanisme s'apparentant à du passage de nom se produit), l'usage d'un formalisme réversible plus riche que RCCS sera nécessaire. À ce titre, la construction générique définie au chapitre précédent pourrait s'avérer être un outil efficace. Par ailleurs, nous verrons concrètement au chapitre suivant comment adapter le système de mémoires de RCCS afin d'obtenir une sémantique de réductions du π-calcul réversible.

Conclusion et travaux futurs.

Un regard en arrière.

En guise de conclusion, nous nous proposons de dresser un bilan des principaux concepts que nous avons développés. Dans le chapitre introductif nous formulions la question suivante :

> *"Peut-on définir un langage permettant de faire revenir en arrière le calcul d'un ensemble de processus (ici, des termes de CCS) sans affaiblir le degré de distribution de ce calcul (1) et en préservant la cohérence des états du système (2) ?"*

Nous sommes maintenant en mesure de répondre à cette question de manière relativement complète. En premier lieu, nous avons mis en place un système de piles mémoires permettant de rendre réversibles les règles de communications de CCS. Afin de satisfaire au pré-requis de ne pas endommager le degré de distribution implicite du calcul, les mémoires d'un système suivent la structure arborescente des processus lors des duplications, ce qui constitue une réponse au point (1) de notre problématique. Nous avons ensuite établi que le mécanisme de retour arrière de RCCS est correct et maximal, dans le sens où il offre le plus grand degré de flexibilité possible dans le choix des actions négatives à entreprendre pour annuler une trace, tout en maintenant des états de calculs cohérents. Plus précisément, nous avons montré que toute trace γ, annulant une trace σ, revient en arrière en suivant un chemin nécessairement équivalent au retour arrière strict consistant à annuler les transitions dans l'ordre fixé par σ (Théorème 3.2.1). Cette dernière propriété permettant de répondre positivement au deuxième point de notre problématique.

Une fois le formalisme réversible mis en place nous avons étudié les ajustements théoriques nécessaires pour en faire un langage adapté à la programmation de systèmes transactionnels. Nous nous sommes donc placés dans une sémantique partiellement réversible en ajoutant la possibilité d'effectuer des actions de validation. Dans cette forme plus riche de la sémantique opéra-

tionnelle, les transitions classiques ne marquent pas de réel progrès de calcul tant qu'elles n'ont pas été validées par une action irréversible. On peut se représenter la véritable évolution d'un calcul en considérant que tous les états connectés par des transitions réversibles forment un même macro-état (qui est donc un ensemble d'états fortement connexes) ; les transitions entre macro-états correspondant aux transitions irréversibles. Appelons *comportement macro* ce système de transition partiellement réversible. Étant donné un processus CCS (avec actions marquées), nous nous sommes attachés à déterminer le comportement macro de celui-ci une fois interprété dans RCCS. Nous avons alors montré que le système de transition macro d'un processus interprété dans RCCS était bisimilaire au système de transitions causales (CTS) de ce même processus dans la sémantique standard de CCS (Théorème 4.2.1). Ce théorème transactionnel montre qu'il n'est pas nécessaire de se placer dans le la sémantique réversible pour déduire le comportement macro d'un système : il suffit d'extraire le CTS du processus d'origine. Partant de ce constat, nous avons défendu l'idée que ce théorème transactionnel permet de définir une méthode de programmation concurrente, que nous avons qualifiée de déclarative, consistant à générer un programme correct à partir de la simple mise en concurrence des processus transactionnels du système, sans avoir à intégrer de mécanisme gestion des verrous pouvant être causés par cette compétition non arbitrée. En effet, si chacune des transactions en compétition a été correctement déclarée, le CTS du processus constitué de l'ensemble de ces déclarations correspond à l'exécution *atomique* de chacune d'entre elles dans un ordre arbitraire et coïncide avec la spécification du système. Par le théorème transactionnel, l'interprétation RCCS du programme déclaratif est alors mécaniquement correcte. On peut donc dire qu'un programme CCS est correct, dans le sens de la programmation concurrente déclarative, si son CTS l'est dans le sens classique défini par Milner[13]. Le gain escompté de la méthode proposée réside dans le fait que le code déclaratif est intuitivement plus concis et plus intuitif qu'un code complet qui doit anticiper les verrous induits par un entrelacement imprévu des sous-transactions du système, conduisant au blocage mutuel de processus en compétition pour une même ressource.

Après avoir mis en places les fondements théoriques de la programmation concurrente déclarative, nous avons cherché à évaluer dans quelle mesure les programmes déclaratifs pourraient être certifiés automatiquement (voir Figure 5.1). Pour ce faire nous avons mis au point un algorithme (voir Figure 5.3) permettant d'extraire le CTS d'un processus en utilisant son

[13] *i.e* la bisimilarité avec un système de référence

déploiement dans une structure d'événements. L'algorithme repose sur un théorème de représentation[14] qui garantit la correspondance entre les traces d'un processus CCS (quotientées par équivalence) et les configurations de son déploiement. Afin de pouvoir utiliser cette algorithme dans le cas de processus récursifs, nous avons développé une notion de structures d'événements récursives permettant de borner le déploiement d'un processus tout en conservant une information suffisante pour en construire le CTS (Théorème 6.1.1). Nous avons alors montré que l'algorithme, une fois adapté aux structures récursives (voir Figure 6.3), termine lorsque le CTS du processus considéré est à états finis. Nous avons implémenté l'algorithme d'extraction de CTS dans un module du langage Ocaml[15], afin de pouvoir évaluer le temps nécessaire à la vérification automatique d'un code par la méthode déclarative. Le banc d'essai que nous avons mis au point consiste à comparer le temps nécessaire à la vérification d'une implémentation complète (sans verrou) du dîner des philosophes avec un outil permettant de construire des bisimulations[16] et le temps nécessaire à l'extraction du CTS d'un code déclaratif (contenant des verrous) et de sa vérification à l'aide du même outil. Les résultats du banc d'essai montrent un rendement significativement meilleur de la méthode déclarative qui permet de construire des systèmes allant jusqu'à 19 philosophes, pour une spécification de près de 15.000 états, contre seulement 5 philosophes et une spécification de 160 états dans le cas de la méthode standard (voir Figures 6.4 et 6.5).

Nous avons ensuite cherché à formaliser de manière plus abstraite les étapes de construction de RCCS. L'idée étant de voir sous quelles conditions il est possible de faire de la programmation concurrente déclarative pour un langage quelconque (par exemple, le π-calcul ou les réseaux de Petri). Nous l'avons dit, la première étape consiste à formaliser syntaxiquement un langage satisfaisant aux deux points de la problématique rappelée au début de cette conclusion. Dans un deuxième temps on doit montrer que le macro comportement du système réversible correspond bien au CTS du processus déclaratif. Nous avons vu que cette deuxième étape pouvait être décrite en s'abstrayant totalement du langage choisi en se plaçant dans un cadre catégorique dans lequel le langage d'origine est représenté par une catégorie des traces de calculs. La transformation de ce langage en un calcul réversible est alors représentée par la construction d'une structure que nous avons appelée catégorie des histoires réversibles. Nous avons alors montré que la sous-catégorie des traces irréversibles, qui correspond à un CTS dans

[14] [Winskel, 1982, Boudol et Castellani, 1994]

[15] Module Causal, `http://moscova.inria.fr/~krivine`

[16] The Mobility Workbench,`http://www.it.uu.se/research/group/mobility/mwb`

l'intuition RCCS, et la catégorie des histoires réversibles étaient bien équivalentes (Théorème 7.2.1). Le point important est que la preuve de ce théorème transactionnel généralisé ne repose que sur l'existence d'un système de factorisation, entre morphismes irréversibles et réversibles, sur la catégorie des traces de calculs. Nous avons donc ramené le problème de la création d'un langage concurrent pour la programmation déclarative à l'existence d'une unique factorisation, permettant de décomposer toute trace de calcul en une trace irréversible suivi d'une trace réversible. Dans le cas particulier de CCS, nous avons vu, *a posteriori*, que le lemme 4.2.2 à la base de la preuve du théorème transactionnel pour RCCS, était bien le pendant de cette propriété de factorisation.

Par la suite, nous nous sommes intéressés à l'usage d'un calcul de type RCCS pour la modélisation de processus biologiques. Nous avons commencé par une présentation succincte d'un modèle π-calcul de transaction biologique en s'inspirant d'un système de régulation transcriptionnelle. Nous avons ensuite argué que l'usage (non régulé) du passage de noms privés du π-calcul relevait plus de la simulation de systèmes biologiques que de la définition d'un véritable modèle de ces derniers. En particulier, nous avons défendu l'idée qu'un bon langage pour la biologie des systèmes devrait permettre une analyse qualitative capable de renseigner sur le rôle des divers agents représentés avant même d'utiliser une sémantique plus quantitative du langage (à base de transitions probabilistes par exemple). Partant de ce principe et de l'observation que la plupart des interactions biologiques sont réversibles, nous avons montré que les modèles déclaratifs interprétés dans RCCS permettent d'éviter l'usage contre-intuitif du passage de noms et dans un même temps de fournir une information, lisible statiquement, sur le rôle accélérateur ou inhibiteur de chacun des agents dans les diverses transactions du systèmes. Plus formellement, nous avons défini une notion de graphe de repliements qui est une forme simple de structure d'événements correspondant aux différentes complexations du système. Muni d'un tel graphe, il est alors possible d'évaluer grossièrement la robustesse d'un complexe et le rôle de chacune des liaisons dans la formation de celui-ci. Si ce complexe permet la validation d'une transaction (dans notre exemple, la production d'une nouvelle protéine) on peut alors conclure que tous les agents jouant une rôle de consolidation du complexe favorisent la transaction.

En plus des pistes étudiées dans les sections de discussion des chapitres antérieurs, il existe de nombreuses extensions de RCCS pouvant faire l'objet d'investigations futures. Nous avons déjà mentionné l'intérêt de l'élaboration d'une sémantique probabiliste de RCCS qui permettrait d'évaluer le temps nécessaire à la réalisation d'une transaction tout en brisant les

boucles infinies liées au retour arrière. Un travail ultérieur pourrait donc viser à l'adaptation à RCCS des sémantiques probabilistes existantes[17]. Parmi les autres continuations envisagées de nos travaux, l'implémentation d'une plateforme capable d'interpréter des processus simples dans une sémantique réversible constitue un axe de recherche particulièrement fertile. Pour clore ces travaux, nous nous proposons d'explorer quelques points clefs en relation avec la réalisation d'un langage dédié à la programmation concurrente déclarative. Nous nous attacherons à donner une description la plus rigoureuse possible des extensions envisagées, en laissant l'étude de leurs propriétés formelles à des travaux futurs. Nous présenterons, dans un premier temps, une modification possible de la sémantique de RCCS incorporant un système de ramasse-miettes (*garbage collector*) de mémoires validées. Nous explorerons ensuite la question de l'expressivité de CCS en étudiant les ajustements syntaxiques nécessaires à l'adaptation de RCCS au passage de noms.

Un ramasse-miettes de mémoires.

Comme nous l'avons fait remarquer, il n'est pas possible de procéder à l'effacement des mémoires de façon purement locale par la simple règle :

$$m \triangleright \underline{\alpha}.p + q \xrightarrow{l:\underline{\alpha}} \langle\rangle \triangleright p$$

En effet, si l'on considère le système $\langle l, a\rangle \cdot m \triangleright \underline{\alpha}.p \parallel m' \cdot \langle l, \bar{a}\rangle \cdot m'' \triangleright q$, l'effacement de la mémoire du processus de gauche, suite à la validation $\underline{\alpha}$, doit être accompagné de l'effacement du préfixe $\langle l, \bar{a}\rangle \cdot m''$ de la mémoire du processus de droite. Si on poursuit le raisonnement, il est clair que l'effacement d'une section de pile mémoire doit déclencher une série d'effacements en cascade aboutissant à un système cohérent et entièrement débarrassé des mémoires inutiles. Dans cette section, nous allons voir comment effacer les mémoires ayant été directement ou indirectement validées par une transition irréversible, par l'intermédiaire d'une légère modification de la sémantique opérationnelle de RCCS (nous considérerons ici la syntaxe souple de RCCS présentée dans la section 3.2.3).

Gestion des identifiants mémoire.

Suivant les notations de la section 3.2.3, on rappelle qu'on dispose d'un ensemble $\mathcal{L}$ d'identifiants et que $Loc(m) \subseteq \mathcal{L}$ (resp. $Loc(r) \subseteq \mathcal{L}$) désigne l'ensemble des identifiants de communications de la mémoire m (resp. des mémoires du processus r).

[17] [Priami, 1995, Herescu et Palamidessi, 2000, Deng *et al.*, 2005]

Afin d'éviter la redondance de mémoires, liée à l'application de la règle de congruence $m \rhd p \parallel q \equiv \langle 1 \rangle \cdot m \rhd p \parallel \langle 2 \rangle \cdot m \rhd q$ (dist) et peu propice à un travail d'implémentation, nous allons faire hériter la mémoire du processus père à un seul des deux fils. Nous supprimons donc la règle (dist) pour la "simuler", dans la sémantique opérationnelle (donnée Figure 8.5), par la transition réversible :

$$m \rhd p \parallel q \xrightarrow{l:\tau} \langle l, \mathbf{1} \rangle \cdot m \rhd p \parallel \langle l, \mathbf{2} \rangle \rhd q \tag{8.5}$$

Le processus monitoré par $\langle l, \mathbf{2} \rangle$ peut être vu comme le fils du processus originel de continuation $\langle l, \mathbf{1} \rangle \cdot m \rhd p$. L'identifiant l permet de rendre réversible la règle (8.5) de manière non ambiguë :

$$\langle l, \mathbf{1} \rangle \cdot m \rhd p \parallel \langle l, \mathbf{2} \rangle \rhd q \xrightarrow{l:\tau^-} m \rhd p \parallel q \tag{8.6}$$

On remarquera que l'ancienne règle de congruence est remplacée par des axiomes et non pas par une règle de la forme :

$$m \rhd p \parallel q \equiv \langle l, \mathbf{1} \rangle \cdot m \rhd p \parallel \langle l, \mathbf{2} \rangle \rhd q$$

Il s'agit là d'une simple commodité permettant de s'assurer de l'unicité de l'identifiant l vis à vis du contexte (voir la sémantique opérationnelle donnée Figure 8.5). Les mémoires de RCCS avec ramasse-miettes ($\text{RCCS}_{\mathsf{gc}}$) sont de la forme :

$$m := \langle l, \alpha, p \rangle \cdot m \mid \langle l, \mathbf{1} \rangle \cdot m \mid \langle l, \mathbf{2} \rangle \mid \langle\rangle \tag{8.7}$$

Nous allons voir, dans la section suivante, que le symbole de mémoire bloquée $\langle\!|\alpha|\!\rangle$ n'est plus nécessaire. En échange, les processus simples sont monitorés par un couple formé d'une mémoire m et d'un ensemble *corbeille* $\mathcal{C} \subseteq \mathcal{L}$ permettant de stocker les identifiants de communications en passe d'être nettoyés. Les systèmes sont de la forme :

$$r := \langle \mathcal{C}, m \rangle \rhd p \mid (r \parallel s) \mid r \backslash x \mid \mathbf{0} \tag{8.8}$$

On notera que nous utilisons le symbole $\mathbf{0}$ pour dénoter le système nul. On pourra obtenir ce dernier lorsqu'un processus bloqué vers l'avant aura totalement été nettoyé de ses mémoires. On considère les règles de congruence intuitives :

$$\begin{aligned} \langle \emptyset, \langle\rangle \rangle \rhd 0 &\equiv \mathbf{0} \\ r \parallel \mathbf{0} &\equiv r \end{aligned}$$

qui viennent s'ajouter aux règles standard pour RCCS – moins la règle (dist).

Sémantique opérationnelle.

Pour définir l'opération de ramasse-miettes, nous introduisons trois nouveaux axiomes. Le premier permet de vider une pile mémoire après validation :

$$\langle \mathcal{C}, m\rangle \triangleright \underline{\alpha}.p + q \xrightarrow{\epsilon:\underline{\alpha}} \langle \mathcal{C} \cup \mathcal{L}oc(m), \langle\rangle\rangle \triangleright p \qquad \text{(commit)}$$

On note que la structure de pile n'est pas maintenue dans la corbeille $\mathcal{C}$ qui n'a pas besoin de prendre en compte la causalité entre les communications pour effectuer les opérations de nettoyage des mémoires inutiles. Par ailleurs, on introduit l'identifiant $\epsilon \in \mathcal{L}$ pour dénoter l'identifiant vide. On définit ensuite la règle permettant d'informer le contexte de l'inutilité d'un identifiant :

$$\langle \{l\} \cup \mathcal{C}, m\rangle \triangleright p \xrightarrow{l:\overline{\mathsf{gc}}} \langle \mathcal{C}, m\rangle \triangleright p \qquad \text{(gc-out)}$$

et enfin la règle complémentaire permettant de mettre à la corbeille un préfixe qu'on sait être bloqué :

$$\langle \mathcal{C}, m \cdot \langle l, -\rangle \cdot m'\rangle \triangleright p \xrightarrow{l:\mathsf{gc}} \langle \mathcal{C} \cup Loc(m'), m\rangle \triangleright p \qquad \text{(gc-in)}$$

Cette règle doit se lire en complément de la règle (gc-out) : lorsqu'un identifiant l est supprimé *via* la règle (gc-out), on peut appliquer la règle (gc-in) sur tout préfixe mémoire de la forme $\langle l, -\rangle \cdot m'$. On notera que la suppression de la règle de congruence (dist) permet de s'assurer qu'après application des règles (gc-out) et (gc-in) sur un identifiant l, ce dernier est effectivement effacé du système[18]. Le mécanisme de ramasse-miettes étant décentralisé, il est tout à fait possible qu'un identifiant l se retrouve dans deux corbeilles en même temps : on pose donc $\overline{\overline{\mathsf{gc}}} = \overline{\mathsf{gc}}$ pour indiquer que la suppression d'un identifiant l *via* la règle (gc-out) est synchronisable avec tout autre règle de ramasse-miettes sur l. On notera, en revanche, que la règle (gc-in) n'est pas son propre complément afin d'éviter qu'un système ne nettoie des piles mémoires encore utiles.

Les transitions de $\text{RCCS}_{\mathsf{gc}}$ sont des quadruplets de la forme $\langle r, l, \zeta, s\rangle$ où ζ est une action RCCS standard (positive ou négative) ou une action de nettoyage gc. Le système de transitions pour $\text{RCCS}_{\mathsf{gc}}$ est donné Figure 8.5.

On notera que la règle de passage au contexte (par) prend un rôle particulièrement important dans le bon fonctionnement du mécanisme de ramasse-miettes. C'est en effet elle qui permet de s'assurer qu'une règle de gc sur un identifiant l est nécessairement réalisée conjointement avec le gc complémentaire. Par exemple, considérons le système $r = \langle \emptyset, \langle\rangle\rangle \triangleright \bar{x}.\underline{a}.0 \parallel x.0$ et le

[18] Les mémoires n'étant pas dupliquées, on a la garantie qu'un identifiant de communication apparaît au plus deux fois dans le système.

$$\langle \mathcal{C}, m\rangle \rhd \alpha.p + q \xrightarrow{l:\alpha} \langle \mathcal{C}, \langle l, \alpha, q\rangle \cdot m\rangle \rhd p \quad \text{(act)}$$
$$\langle \mathcal{C}, \langle l, \alpha, q\rangle \cdot m\rangle \rhd p \xrightarrow{l:\alpha^-} \langle \mathcal{C}, m\rangle \rhd \alpha.p + q \quad (\text{act}^-)$$
$$\langle \mathcal{C}, m\rangle \rhd \underline{\alpha}.p + q \xrightarrow{\epsilon:\underline{\alpha}} \langle \mathcal{C} \cup Loc(m), \langle\rangle\rangle \rhd p \quad \text{(commit)}$$
$$\langle \{l\} \cup \mathcal{C}, m\rangle \rhd p \xrightarrow{l:\overline{\mathsf{gc}}} \langle \mathcal{C}, m\rangle \rhd p \quad (\text{gc}-\text{out})$$
$$\langle \mathcal{C}, m \cdot \langle l, -\rangle \cdot m'\rangle \rhd p \xrightarrow{l:\mathsf{gc}} \langle \mathcal{C} \cup Loc(m'), m\rangle \rhd p \quad (\text{gc}-\text{in})$$
$$\langle \mathcal{C}, m\rangle \rhd (p \parallel q) \xrightarrow{l:\tau} \langle \mathcal{C}, \langle l, \mathbf{1}\rangle \cdot m\rangle \rhd p \parallel \langle \emptyset, \langle l, \mathbf{2}\rangle\rangle \rhd q \quad \text{(fork)}$$
$$\langle \mathcal{C}, \langle l, \mathbf{1}\rangle \cdot m\rangle \rhd p \parallel \langle \emptyset, \langle l, \mathbf{2}\rangle\rangle \rhd q \xrightarrow{l:\tau^-} \langle \mathcal{C}, m\rangle \rhd (p \parallel q) \quad (\text{fork}^-)$$

$$\frac{r \xrightarrow{l:\zeta} r' \quad l \notin Loc(s)}{r \mid s \xrightarrow{l:\zeta} r' \mid s} \text{ (par)} \qquad \frac{r \xrightarrow{l:\zeta} r' \quad s \xrightarrow{l:\bar{\zeta}} s'}{r \mid s \xrightarrow{l:\tau} r' \mid s'} \text{ (syn)}$$

$$\frac{r \xrightarrow{l:\zeta} r' \quad \zeta \neq x, \bar{x}, x^-, \bar{x}^-}{r\backslash x \xrightarrow{l:\zeta} r'\backslash x} \text{ (res)} \qquad \frac{r \equiv r' \xrightarrow{l:\zeta} s' \equiv s}{r \xrightarrow{l:\zeta} s} \text{ (equiv)}$$

FIG. 8.5 – Système de transitions pour $\text{RCCS}_{\mathsf{gc}}$.

calcul suivant :

$$\begin{array}{rcl} r & \xrightarrow{l_1:\tau} & \langle \emptyset, \langle l_1, \mathbf{1}\rangle\rangle \rhd \bar{x}.\underline{a}.0 \parallel \langle \emptyset, \langle l_1, \mathbf{2}\rangle\rangle \rhd x.0 \\ & \xrightarrow{l_2:\tau} & \langle \emptyset, \langle l_2, \bar{x}\rangle\langle l_1, \mathbf{1}\rangle\rangle \rhd \underline{a}.0 \parallel \langle \emptyset, \langle l_2, x\rangle\langle l_1, \mathbf{2}\rangle\rangle \rhd 0 \\ & \xrightarrow{\epsilon:\underline{a}} & r_1 = \langle \{l_1, l_2\}, \langle\rangle\rangle \rhd 0 \parallel \langle \emptyset, \langle l_2, x\rangle\langle l_1, \mathbf{2}\rangle\rangle \rhd 0 \end{array}$$

À cette étape, du fait de la règle (par), le processus de gauche ne peut pas effacer l_1 (ou l_2) sans se synchroniser avec le processus de droite. On obtient :

$$r_1 \xrightarrow{l_1:\tau} r_2 = \langle \{l_2\}, \langle\rangle\rangle \rhd 0 \parallel \langle \emptyset, \langle l_2, x\rangle\rangle \rhd 0$$

On note ici que le processus de droite ne peut pas revenir sur la communication l_2, toujours du fait de la restriction liée à la règle (par), car $l_2 \in Loc\big(\langle \{l_2\}, \langle\rangle\rangle \rhd 0\big)$. Le calcul se termine par :

$$\begin{array}{rcl} r_2 & \xrightarrow{l_2:\tau} & \langle \emptyset, \langle\rangle\rangle \rhd 0 \parallel \langle \emptyset, \langle\rangle\rangle \rhd 0 \\ & \equiv & \mathbf{0} \end{array}$$

Discussion

Nous avons défini une sémantique opérationnelle de RCCS mieux adaptée à une optique d'implémentation. En premier lieu nous avons supprimé la

redondance des piles mémoires due à la règle de congruence (dist) qui se justifiait essentiellement pour des raisons d'uniformité du calcul[19]. En tirant profit de cette modification nous avons proposé un système de ramasse-miettes de mémoires dont les actions peuvent se dérouler concurremment au cours standard de l'exécution du système. Même si cela reste à démontrer formellement, il est clair que les transitions du gc peuvent commuter avec les transitions standard (dans un cas limite on peut même ne jamais toucher aux identifiants des corbeilles). L'opération de nettoyage des mémoires n'influence donc pas le déroulement des transactions en cours. Elle est, en revanche, clairement nécessaire à toute tentative d'implémentation permettant de programmer des systèmes dans lesquels des transactions peuvent s'effectuer de manière itérative, afin de vider les piles mémoires correspondant aux transactions validées.

Une extension de $\text{RCCS}_{\mathsf{gc}}$ intéressante serait de conditionner certaines actions d'un processus par un test de la vacuité de la pile mémoire qui lui est associée. De la même manière que le retour arrière intégré permet de ne pas avoir à programmer certains comportement, le fait de pouvoir vérifier localement que la transaction à laquelle un processus a participé a été validée permettrait de ne pas avoir à implémenter certaines mises à jour qui seraient alors automatiquement gérées. Afin de motiver une telle extension, nous donnons un exemple simple d'application. Notons [gc] pour indiquer une garde ne pouvant être levée que par application de la règle de congruence :

$$\langle \mathcal{C}, \langle\rangle \rangle \triangleright [\mathsf{gc}]p \equiv \langle \mathcal{C}, \langle\rangle \rangle \triangleright p \tag{8.9}$$

On note que la règle ci-dessus requiert que la mémoire du processus soit vide. On considère un système transactionnel composé de trois processus :

$$\begin{array}{lcl} \mathsf{A}(a, x, y, ok_x, ok_y) & := & \bar{x}.\bar{y}.\underline{a}.(\overline{\underline{ok}}_x.0 \parallel \overline{\underline{ok}}_y.0 \parallel \mathsf{CONT_A}) \\ \mathsf{B}(x, ok_x) & := & x.\underline{ok}_x.\mathsf{CONT_B} \\ \mathsf{C}(y, ok_y) & := & y.\underline{ok}_y.\mathsf{CONT_C} \end{array}$$

La transaction à réaliser se déroule de la manière suivante : en premier lieu, le processus A doit contacter les processus B et C par l'intermédiaire des canaux x et y. Une fois le consensus établi, A peut valider la transaction sur le canal $\underline{a}$ et poursuivre son travail *via* la continuation $\mathsf{CONT_A}$. Les processus

[19] On peut noter que l'asymétrie du résidu du processus $m \triangleright (p \parallel q)$ après application de (fork) est similaire à la gestion des duplications de processus UNIX par un appel à la primitive `fork()`. Dans $\text{RCCS}_{\mathsf{gc}}$ on peut considérer que la continuation héritant de la mémoire est le père du processus nouvellement créé. Conformément à l'intuition système, le processus qui deviendra la continuation du père est choisi de façon non-déterministe si on considère la commutativité de $p \parallel q$ dans $m \triangleright (p \parallel q)$.

B et C sont alors en attente d'une confirmation sur les canaux ok_x et ok_y avant de pouvoir entamer leurs continuations respectives. On peut noter que nous avons ici explicitement programmé les deux phases de validation d'un consensus distribué (*two phase commit protocol*) : la validation du processus *leader* (canal $\underline{a}$) puis la mise à jour des processus secondaires (canaux ok_x et ok_y). Dans RCCS la deuxième phase de validation est toutefois moins décisive que la première puisqu'après la première validation aucun des deux processus secondaires ne peut revenir sur son choix ; en d'autres termes, ils ont été *implicitement* validés. Les validations ok_x et ok_y ne constituent donc pas un moyen d'empêcher les processus d'annuler leurs votes, mais agissent comme des gardes conditionnant l'exécution des continuations de B et C au succès de la transaction entreprise par A. L'usage de la garde $[\mathsf{gc}]$ permet de refléter ce fait tout en allégeant la tâche du programmeur :

$$\begin{array}{lcl} \mathsf{A}'(a,x,y) & := & \bar{x}.\bar{y}.\underline{a}.\mathsf{CONT}_\mathsf{A} \\ \mathsf{B}'(x) & := & x.[\mathsf{gc}]\mathsf{CONT}_\mathsf{B} \\ \mathsf{C}'(y) & := & y.[\mathsf{gc}]\mathsf{CONT}_\mathsf{C} \end{array}$$

L'usage de ce type de garde dans un processus constitue une forme de coupure du système de transition causal engendré. Une telle modification de la sémantique de $\mathrm{RCCS}_{\mathsf{gc}}$ devrait donc être précédée d'une étude théorique de l'impacte de ces coupures sur le théorème transactionnel (Théorèmes 4.2.1 et 7.2.1).

Un π-calcul réversible.

De nombreux systèmes transactionnels font intervenir des processus *serveurs*, capables de traiter des requêtes simultanées venant de processus *clients* distincts. Le fonctionnement typique d'un serveur est d'attendre en permanence des requêtes sur un canal public (connu de tous les clients potentiels), puis de traiter ces requêtes et d'envoyer la réponse adéquate au client d'origine. Le problème qui se pose est alors de s'assurer que la réponse du serveur sera bien reçue par le client initial. Lorsqu'on utilise un calcul de type CCS, on doit considérer que chaque client C_i communique avec le serveur S par l'intermédiaire d'un canal privé a_i sur lequel il pourra attendre la réponse de sa requête. Ce système implique que le serveur (dans l'état initial) doit connaître un nombre arbitrairement grand de canaux privés ; il est alors de la forme :

$$\mathsf{S}(a_1,\ldots,a_n) := \sum_{i=1}^{n} a_i.(\mathsf{Traiter}_i \parallel \mathsf{S}(a_1,\ldots,a_n))$$

Clairement, cette “astuce” n'est guère raisonnable si on se place dans une optique de programmation. En utilisant le π-calcul, qui possède des primitives d'envoi et de réception de noms, on peut programmer ce type de comportement en se servant d'un unique canal de requête public, disons pub :

$$\mathsf{S}(pub) := pub(x).(\mathsf{Traiter}(x) \parallel S(pub))$$

Le serveur reçoit un nom frais venant d'un client et il pourra utiliser ce nom à la fin de la procédure de traitement. Pour un client, le protocole à suivre pour émettre une requête au serveur est alors simplement de générer un nouveau nom, de l'envoyer au serveur sur le canal public, puis d'attendre une réponse sur ce canal :

$$\mathsf{C}(pub) := (\overline{pub}\, x.x(rep).\mathsf{Cont})\backslash x$$

On peut se demander si un tel mécanisme de passage de noms s'adapte bien à la technique de retour arrière utilisée pour RCCS. Le cas échéant on pourrait alors utiliser le théorème transactionnel généralisé défini au chapitre 7, afin d'obtenir une méthode de programmation déclarative pour le π-calcul. Dans cette section nous allons montrer comment réaliser une telle adaptation pour la sémantique de réductions du π-calcul utilisée dans la section 8.2. Nous discuterons brièvement de la correction du retour arrière obtenu ainsi que du problème de l'extraction de CTS.

Substitutions retardées.

L'adaptation du mécanisme de retour arrière à un langage de la famille du π-calcul se heurte à un problème fondamental : les substitutions, à la base du mécanisme de passage de noms, sont des opérations intrinsèquement irréversibles. Par exemple, considérons le processus $p = \bar{x}z.0 \parallel x(y).q$ et la réduction $p \to p' = 0 \parallel q\,\{z/y\}$. On remarque que prendre $q = \bar{z}.0$ ou $q = \bar{y}.0$ conduit au même état d'arrivée p'. Définir une sémantique réversible pour le π-calcul implique pourtant de distinguer ces deux états d'arrivée afin de préserver le déterminisme du retour arrière. La solution que nous proposons est de ne pas effectuer les substitutions directement sur les termes mais de les conserver dans les mémoires pour déduire le nom correct d'un canal lorsqu'on tente une communication. Un canal possédera alors un nom initial qui restera inchangé au cours du calcul et un nom instancié qui dépend des substitutions en attente dans la mémoire du processus. Prenons, par exemple, le processus $p = \langle 1\rangle \triangleright \bar{x}y.\bar{y}.0 \parallel \langle 2\rangle \triangleright x(z).z().0$. Après une synchronisation sur le canal x on obtient :

$$p \to p' = \langle l_1, \bar{x}y\rangle\langle 1\rangle \triangleright \bar{y}.0 \parallel \langle l_1, x[y/z]\rangle\langle 2\rangle \triangleright z()$$

On note que la variable z n'a pas été instanciée dans le processus monitoré par la mémoire $\langle 1, x[y/z]\rangle\langle 2\rangle$. En revanche on a stocké la substitution $[y/z]$ à effectuer pour connaître le nom instancié de z. Nous verrons qu'on peut utiliser cette information pour déduire la réduction :

$$p' \rightarrow \langle l_2, \bar{y}\rangle\langle l_1, \bar{x}y\rangle\langle 1\rangle \triangleright 0 \parallel \langle l_2, z[\,]\rangle\langle l_1, x[y/z]\rangle\langle 2\rangle \triangleright 0$$

Les mémoires des processus du π-calcul réversible sont de la forme :

$$m := \langle l, \bar{x}y, p\rangle \cdot m \mid \langle l, x[z/y], p\rangle \cdot m \mid \langle 1\rangle \cdot m \mid \langle 2\rangle \cdot m \mid \langle\!|l|\!\rangle \cdot m \mid \langle\rangle \tag{8.10}$$

où $\langle\!|l|\!\rangle \cdot m$ dénote une mémoire bloquée par une validation identifiée par l. Soit X l'ensemble des noms de canaux, on associe à chaque nom $x \in X$ dans un processus monitoré par m, un nom instancié calculé en utilisant la *fonction d'historique*, $h_m : X \rightarrow X$, définie de la manière suivante :

$$\begin{array}{lcll}
h_{\langle l, y[z/x], p\rangle \cdot m}(x) & = & z & \\
h_{\langle\rangle}(x) & = & x & \\
h_{\langle\!|l|\!\rangle \cdot m}(x) & = & x & \\
h_{\langle l, y[z/t], p\rangle \cdot m}(x) & = & h_m(x) & \text{si } t \neq x \\
h_{\langle l, \bar{y}z, p\rangle \cdot m}(x) & = & h_m(x) & \\
h_{\langle 1\rangle \cdot m}(x) & = & h_m(x) & \\
h_{\langle 2\rangle \cdot m}(x) & = & h_m(x) &
\end{array}$$

On notera que dans le cas d'une validation, on ne cherche pas de substitution s'appliquant à x car une validation entraîne déjà l'application de toutes les substitutions en attente dans la pile (voir Figure 8.6). On étend la fonction d'historique à tout un processus monitoré par m en substituant les noms libres de p par leurs noms instanciés. Pour tout processus p tel que $\mathsf{fn}(p) = \{x_1, \ldots, x_n\}$, on pose :

$$h_m(p) = p\,\{h_m(x_1)/x_1, \ldots, h_m(x_n)/x_n\} \tag{8.11}$$

Sémantique réversible.

La sémantique opérationnelle du π-calcul réversible (voir Figure 8.6) suit, dans les grandes lignes, le schéma appliqué à RCCS. C'est une symétrisation de la sémantique opérationnelle donnée Figure 8.2 dans laquelle on déduit le nom instancié des canaux au moment de la communication. On notera que la question d'une gestion plus explicite des substitutions dans un langage de type π-calcul a fait l'objet de plusieurs études antérieures[20]. En particulier,

[20] [Ferrari *et al.*, 1996, Hirschkoff, 1998, Gardner et Wischik, 2000]

le mécanisme de gestion des substitutions que nous proposons, présente de fortes similarités avec le π_ξ-calcul, une sémantique du π-calcul avec substitutions explicites[21], dans laquelle un environnement accompagne l'exécution des processus et permet d'obtenir un système de transition similaire à celui de CCS en enregistrant les substitutions à effectuer. Toutefois, à la différence du π_ξ-calcul, dans le π-calcul réversible l'environnement permettant de déduire les noms instancié des processus est explicitement distribué.

$$\frac{h_m(x) = h_{m'}(u) \qquad y' = h_m(y)}{(m \triangleright \bar{x}y.p + q) \parallel (m' \triangleright u(v).p' + q') \overset{l}{\leftrightarrow} (\langle l, \bar{x}y, q\rangle.m \triangleright p) \parallel (\langle l, u[y'/v], q\rangle.m' \triangleright p')}\ (\mathsf{com})$$

$$\frac{h_m(x) = h_{m'}(u) \qquad p'' = p'\{h_m(y)/v\}}{(m \triangleright \underline{\bar{x}}y.p + q) \parallel (m' \triangleright \underline{u}(v).p' + q') \overset{l}{\rightarrow} \langle\!|l|\!\rangle.m \triangleright h_m(p) \parallel \langle\!|l|\!\rangle.m' \triangleright h_{m'}(p'')}\ (\underline{\mathsf{com}})$$

$$(\mathsf{sil})\ \frac{}{m \triangleright \tau.p + q \overset{l}{\leftrightarrow} \langle l, \tau, q\rangle.m \triangleright p} \qquad \frac{}{m \triangleright \underline{\tau}.p + q \overset{l}{\rightarrow} \langle\!|l|\!\rangle.m \triangleright h_m(p)}\ (\underline{\mathsf{sil}})$$

$$\frac{r \overset{l}{\leftrightarrow} r' \;\&\; l \notin Loc(s)}{r \parallel s \overset{l}{\leftrightarrow} r' \parallel s}\ (\mathsf{par}) \qquad \frac{r \overset{l}{\leftrightarrow} r'}{r\backslash x \overset{l}{\leftrightarrow} r'\backslash x}\ (\mathsf{res}) \qquad \frac{r \equiv r' \overset{l}{\leftrightarrow} s' \equiv s}{r \overset{l}{\leftrightarrow} s}\ (\mathsf{equiv})$$

FIG. 8.6 – Sémantique de réductions du Rπ-calcul.

Dans la règle (com), les prémices indiquent qu'une communication peut avoir lieu lorsque les canaux de communication des deux processus sont unifiables *via* leurs historiques respectifs. On notera que, contrairement à la sémantique de réductions traditionnelle du π-calcul (voir Figure 8.2), aucune substitution n'est effectuée au moment de la synchronisation. Elles sont effectuées à la volée par le système de mémoires qui interprète chaque nom de variable en fonction des substitutions mémorisées. Les règles de validations se lisent intuitivement ; on notera que lors d'une validation, on effectue les substitutions en attente dans les processus simples par l'opération $h_m(p)$ définie en (8.11). La règle de congruence structurelle utilisée dans la règle (equiv) permet quant à elle de considérer les systèmes *modulo* associativité et commutativité de la composition parallèle (et les règles standard pour la

[21] [Ferrari *et al.*, 1996]

restriction), plus les règles spécifiques aux systèmes réversibles :

$$\begin{array}{lcl} m \rhd p \parallel q & \equiv & \langle 1 \rangle \cdot m \rhd p \parallel \langle 2 \rangle \cdot m \rhd q \\ m \rhd p \backslash x & \equiv & (m \rhd p) \backslash x \text{ si } x \notin m \end{array}$$

Comme pour RCCS on notera, pour finir, que le calcul réversible défini n'est qu'une décoration d'un processus du π-calcul classique. En effet, tout système r se projette sur un processus $\varphi(r)$ du π-calcul de la manière suivante :

$$\begin{array}{lcl} \varphi(m \rhd p) & := & h_m(p) \\ \varphi(r \parallel s) & := & \varphi(r) \parallel \varphi(s) \\ \varphi(r \backslash x) & := & \varphi(r) \backslash x \end{array}$$

Discussion.

La question de la correction du retour arrière que nous venons de définir se résume à prouver l'équivalent du théorème 3.2.1 pour le π-calcul réversible. Nous ne donnons pas la preuve d'une telle propriété dans ce chapitre exploratoire, mais nous pouvons noter que la définition de la causalité entre réductions du π-calcul coïncide avec la causalité structurelle de CCS[22]. En d'autres termes, nous conjecturons que les propriétés liées à la causalité (ou à la concurrence) que nous avons montrées pour CCS vont se montrer de manière similaires pour la sémantique de réductions du π-calcul. Le théorème transactionnel nécessite quant à lui d'exhiber un système de factorisation entre les traces irréversibles et réversibles du calcul. Là encore, les réductions causales du π-calcul correspondant avec celles de CCS, nous pouvons conjecturer que les réductions irréversibles du Rπ-calcul vont correspondre précisément avec les transitions causales du π-calcul simple. La simplicité de la définition de la relation de causalité pour les réductions du π-calcul disparaît toutefois dès qu'on considère des sémantiques plus compositionnelles basées sur des LTS. En effet, définir un LTS réversible pour le π-calcul nécessite de prendre en compte une nouvelle forme de dépendance qui n'est pas induite par la structure des processus. Pour se donner un idée du problème, considérons le processus $p = \bar{x}y.0 \backslash y \parallel u(v).v()$ donnant lieu aux transitions :

$$p \xrightarrow{\bar{x}y} 0 \parallel u(v).v() \xrightarrow{uy} 0 \parallel y()$$

En considérant la seule causalité structurelle, on déduirait que les deux transitions sont indépendantes. Un système de retour arrière autorisera donc l'annulation de l'une indépendamment de l'autre, ce qui serait une erreur

[22] [Boreale et Sangiorgi, 1998, Degano et Priami, 1999]

car le processus de droite ne peut recevoir le nom y qu'*après* envoi, ou *extrusion*, de celui-ci au contexte lors de la transition précédente. Comme nous l'avons dit plus haut cette causalité liée au mécanisme d'extrusion disparaît si on se restreint aux réductions. Ainsi les réductions issues du processus $p' = \bar{x}y.0 \backslash y \parallel x(z).\bar{u}z.\bar{z} \parallel u(v).v()$:

$$p' \rightarrow (0 \parallel \bar{u}y.\bar{y}) \backslash y \parallel u(v).v() \rightarrow (0 \parallel 0 \parallel \bar{y} \parallel y()) \backslash y \rightarrow 0$$

sont nécessairement entreprises (et annulées) dans l'ordre donné par la précédence structurelle classique. Un travail futur pourrait donc consister à définir une sémantique compositionnelle pour le π-calcul réversible en forçant le retour arrière à respecter cette notion de précédence non-structurelle.

La difficulté de définir une sémantique compositionnelle respectant la causalité rend la question de l'extraction automatique du CTS d'un processus délicate. En effet, l'algorithme que nous avons défini au chapitre 6 consiste à déployer partiellement un processus dans sa structure d'événement (récursive), ce qui nécessiterait la mise au point de structures d'événements pour le π-calcul algorithmiquement exploitables[23]. Afin de réaliser de telles structures d'événements, une piste intéressante que nous comptons explorer serait d'utiliser la notion d'historique pour associer à chaque événement de la structure, la capacité π qui lui correspond. Comme l'historique d'un événement dépend du calcul mené précédemment, on obtiendrait une structure dont les étiquettes d'événements dépendraient de la configuration (minimale) dans laquelle ils se trouvent.

[23] Voir à ce sujet [Varacca et Yoshida, 2006].

Bibliographie

[Abdulla *et al.*, 2000] Parosh Aziz Abdulla, S. Purushothaman Iyer, et Aletta Nylén. Unfoldings of unbounded petri nets. Dans *Proceedings of CAV'00 : Computer Aided Verification*, volume 1855 of *LNCS*, pages 495–507, 2000.

[Aceto, 1992] Luca Aceto. *Action Refinement in Process Algebra*, volume 3 of *Distinguished Dissertations in Computer Science*. Cambridge University Press, 1992.

[Amadio et Prasad, 1994] Roberto Amadio et Sanjiva Prasad. Localities and failures. Dans *Proceedings of FST-TCS'94*, volume 880 of *LNCS*, pages 205–216, 1994.

[Baldan *et al.*, 1999] Paolo Baldan, Andrea Corradini, et Ugo Montanari. Unfolding and event structure semantics for graph grammars. Dans *Proceedings of FOSSAC'99*, volume 1578 of *LNCS*, pages 367–386, 1999.

[Baldi *et al.*, 2002] C. Baldi, Pierpaolo Degano, et Corrado Priami. Causal π-calculus for biochemical modeling. Dans *Proceedings of the AI*IA Workshop on BioInformatics 2002*, pages 69–72, 2002.

[Bednarczyk, 1991] Marek A. Bednarczyk. Hereditary history preserving bisimulations or what is the power of the future perfect in program logics. Rapport Technique, ICS PAS, 1991.

[Bergstra *et al.*, 1994] Jan A. Bergstra, Alban Ponse, et J.J van Wamel. Process algebra with backtracking. Dans *REX School/Symposium*, 1994.

[Berry et Lévy, 1979] Gérard Berry et Jean-Jacques Lévy. Minimal and optimal computation of recursive programs. *JACM*, 26 :148–175, 1979.

[Bianchi *et al.*, 1995] Alessandro Bianchi, Stefano Coluccini, Pierpaolo Degano, et Corrado Priami. An efficient verifier of truly concurrent properties. Dans *Proceedings of Parallel Computing Technologies*, volume 964 of *LNCS*, pages 36–50, 1995.

[Blossey *et al.*, 2006] Ralf Blossey, Luca Cardelli, et Andrew Phillips. A compositional approach to the stochastic dynamics of gene networks. *TCSB*, 3939 :99–122, January 2006.

[Bocchi *et al.*, 2003] Laura Bocchi, Cosimo Laneve, et Gianluigi Zavattaro. A calculus for long running transactions. Dans *Proceedings of FMOODS'03, sixth IFIP international conference*, volume 2884 of *LNCS*, pages 124–138, 2003.

[Boreale et Sangiorgi, 1998] Michele Boreale et Davide Sangiorgi. A fully abstract semantics for causality in the π-calculus. *Acta Inf.*, 35 :353–400, 1998.

[Boudol et Castellani, 1989] Gérard Boudol et Ilaria Castellani. Permutation of transitions : An event structure semantics for CCS and SCCS. Dans *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 411–427, 1989.

[Boudol et Castellani, 1994] Gérard Boudol et Ilaria Castellani. Flow models of distributed computations : three equivalent semantics for CCS. *Inf. Comput.*, 114(2) :247–314, 1994.

[Bougé, 1988] Luc Bougé. On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes. *Acta Inf.*, 25(2) :179–201, 1988.

[Bruni *et al.*, 2002] Roberto Bruni, Cosimo Laneve, et Ugo Montanari. Orchestrating transactions in the join calculus. Dans *Proceedings of CONCUR'02, conference on concurrency theory*, 2002.

[Bruni et Montanari, 2000] Roberto Bruni et Ugo Montanari. Zero-safe nets : Comparing the collective and individual token approaches. *Information and Computation*, 156(1) :46–89, 2000.

[Butler *et al.*, 2002] Michael Butler, Carla Ferreira, Peter Henderson, Mandy Chessell, Catherine Griffin, et David Vines. Extending the concept of transaction compensation. *IBM Systems Journal*, 41(4) :743–758, 2002.

[Butler *et al.*, 2004] Michael Butler, C.A.R Hoare, et Carla Ferreira. Trace semantics for long–running transactions. Dans *25 years communicating sequential processes 2004*, volume 2949 of *LNCS*, pages 87–104, 2004.

[Cardelli, 2004] Luca Cardelli. Brane calculi - interactions of biological membranes. Dans *Computational Methods in Systems Biology*, pages 257–278. 2004.

[Cardelli et Gordon, 2000] Luca Cardelli et Andrew Gordon. Mobile ambients. *Theoretical Computer Science*, 24 :177–213, 2000.

[Castellani, 1995] Ilaria Castellani. Observing distribution in processes : Static and dynamic localities. *International Journal of Foundations of Computer Science*, 6(4) :353–393, 1995.

[Castellani, 2001] Ilaria Castellani. *Handbook of process algebra*, Chapitre 15, pages 945–1045. Elsevier, 2001.

[Castellani et Zhang, 1997] Ilaria Castellani et Guo-Qiang Zhang. Parallel product of event structures. *Theoretical Computer Science*, 179(1-2) :203–215, 1997.

[Clarke *et al.*, 1986] Edmund Melson Clarke, E. Allen Emerson, et A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, 1986.

[Danos et Krivine, 2003] Vincent Danos et Jean Krivine. Formal molecular biology done in CCS-R. Dans *Bio-Concur'03, satellite workshop of CONCUR'03*, ENTCS, 2003.

[Danos et Krivine, 2004] Vincent Danos et Jean Krivine. Reversible communicating systems. Dans *Proceedings of CONCUR'04 : 15th International Conference on Concurrency Theory*, volume 3170 of *LNCS*, pages 292–307, 2004.

[Danos et Krivine, 2005] Vincent Danos et Jean Krivine. Transactions in RCCS. Dans *Proceedings of CONCUR'05 : 16th International Conference on Concurrency Theory*, volume 3653 of *LNCS*, 2005.

[Danos *et al.*, 2006a] Vincent Danos, Jean Krivine, et Pawel Sobociński. General reversibility. Dans *Proceedings of EXPRESS'06*, ENTCS, 2006. To appear.

[Danos *et al.*, 2006b] Vincent Danos, Jean Krivine, et Fabien Tarissan. Self assembling trees. Dans *Proceedings SOS'06*, ENTCS, 2006. To appear.

[Danos et Laneve, 2003] Vincent Danos et Cosimo Laneve. Core formal molecular biology. Dans *Proceedings of the 12th European Symposium on Programming (ESOP'03, Warsaw, Poland)*, volume 2618 of *LNCS*, pages 302–318, 2003.

[Darondeau et Degano, 1993] Philippe Darondeau et Pierpaolo Degano. Refinement of actions in event structures and causal trees. *Theoretical Computer Science*, 118 :21–48, 1993.

[Darondeau et Priami, 1989] Philippe Darondeau et Degano Priami. Causal trees. Dans *Proceedings of ICALP'89*, volume 372 of *LNCS*, pages 234–248, 1989.

[Degano *et al.*, 1990] Pierpaolo Degano, Rocco De Nicola, et Ugo Montanari. A partial ordering semantics for CCS. *Theoretical Computer Science*, 75 :223–262, 1990.

[Degano et Priami, 1992] Pierpaolo Degano et Corrado Priami. Proved trees. Dans *Automata, Languages and Programming*, volume 623 of *LNCS*, pages 629–640, 1992.

[Degano et Priami, 1999] Pierpaolo Degano et Corrado Priami. Non-interleaving semantics for mobile processes. *Theoretical Computer Science*, 216(1–2) :237–270, 1999.

[Deng *et al.*, 2005] Yuxin Deng, Catuscia Palamidessi, et Jun Pang. Compositional reasoning for probabilistic finite-state behaviors. Dans *Processes, Terms and Cycles : Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday*, volume 3838 of *LNCS*, pages 309–337, 2005.

[Esparza, 1994] Javier Esparza. Model checking using net unfolding. *Science of Computer Programing*, 23(2) :151–195, 1994.

[Esparza *et al.*, 2002] Javier Esparza, Stefan Römer, et Walter Vogler. An improvement of McMillan's unfolding algorithm. *Form. Methods Syst. Des.*, 20(3) :285–310, 2002.

[Ferrari *et al.*, 1996] Gianluigi Ferrari, Ugo Montanari, et Paola Quaglia. A π-calculus with explicit substitutions. *Theoretical Computer Science*, 168(1) :53–103, 1996.

[Fisler et Vardi, 2002] Kathi Fisler et Moshe Y. Vardi. Bisimulation and model checking. *Formal Methods in Systems Design*, 2002.

[Fournet et Gonthier, 1996] Cédric Fournet et Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. Dans *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, 1996.

[Francois et Hakim, 2004] Paul Francois et Vincent Hakim. Design of genetic networks with specified functions by evolution in silico. Dans *PNAS*, volume 101, pages 580–585, 2004.

[Freyd et Kelly, 1972] Peter J. Freyd et G. Max Kelly. Categories of continuous functors, i. *Journal of Pure and Applied Algebra*, 2 :169–191, 1972.

[Gabriel et Zisman, 1967] Pierre Gabriel et M. Zisman. *Calculus of Fractions and Homotopy Theory*. Springer, 1967.

[Galpin, 2000] Vashti Galpin. A comparison of bisimulation-based semantic equivalences for non interleaving behaviour over CCS processes. *South African Computer Journal*, 26 :13–21, 2000.

[Gardner et Wischik, 2000] Philippa Gardner et Lucian Wischik. Explicit fusions. Dans *MFCS 2000*, volume 1893 of *LNCS*, pages 373–382, 2000.

[Ghosh et Tomlin, 2001] Ronojoy Ghosh et Claire J. Tomlin. Lateral inhibition through delta-notch signaling : A piecewise affine hybrid model ? Dans *Proceedings of HSCC 2001*, volume 2034 of *LNCS*, pages 232–246, 2001.

[Godefroid et Wolper, 1991] Patrice Godefroid et Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. Dans *Proceedings of CAV'91 : Computer-aided verification*, volume 575 of *LNCS*, pages 332–342, 1991.

[Hennessy et Milner, 1985] Matthew Hennessy et Robin Milner. Algebraic laws for nondeterminsm and concurrency. *Journal of the ACM*, 32(1) :137–161, 1985.

[Herescu et Palamidessi, 2000] Oltea Mihaela Herescu et Catuscia Palamidessi. Probabilistic asynchronous π-calculus. Dans *Proceeding of FOSSAC'00*, volume 1784 of *LNCS*, pages 146–160, 2000.

[Hirschkoff, 1998] Danièle Hirschkoff. Automatically proving up-to bisimulation. Dans *Proceedings of MFCS'98*, volume 18 of *ENTCS*, 1998.

[Hoare, 1985] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Hofestädt et Thelen, 1998] R. Hofestädt et S. Thelen. Quantitative modeling of biochemical networks. *In Silico Biology*, 1, 1998.

[Janowski, 1994] Tomasz Janowski. Stepwise transformations for fault-tolerant design of CCS processes. Dans *Proceedings of* 7^{th} *International conference on Formal Description Techniques*, pages 505–520, 1994.

[Kam *et al.*, 2001] Naaman Kam, Irun R. Cohen, et David Harel. The immune system as a reactive system : modeling T-cell activation with statecharts. Dans *Proceedings of the IEEE Symposium on Human-centric Computing*, pages 15–22, September 2001.

[Krivine, 2006] Jean Krivine. A verification algorithm for declarative concurrent programming. Rapport Technique, INRIA-Rocquencourt, 2006.

[Langerak et Brinksma, 1999] Rom Langerak et Ed Brinksma. A complete finite prefix for process algebra. Dans *Proceedings of CAV '99 : Computer Aided Verification*, volume 1633 of *LNCS*, pages 184–195, 1999.

[Laroussinie et Schnoebelen, 2000] François Laroussinie et Philippe Schnoebelen. The state-explosion problem from trace to bisimulation equivalence. Dans *Proceedings of FoSSaCS'00 : Int. Conf. on Foundations of Software Science and Computation Structures*, volume 1784 of *LNCS*, pages 192–207, 2000.

[Lévy, 1978] Jean-Jacques Lévy. *Réduction optimales en λ-calcul*. PhD thesis, Université Paris 7, 1978.

[Matsuno *et al.*, 2000] H. Matsuno, A. Doi, M. Nagasaki, et S. Miyano. Hybrid Petri Net representation of gene regulatory networks. Dans *Proceedings of the Pacific Symposium on Biocomputing (2000)*, pages 338–349, 2000.

[McMillan, 1992] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. Dans *Proceedings of CAV'92 : Computer-aided verification*, volume 663 of *LNCS*, pages 164–177, 1992.

[Milner, 1989] Robin Milner. *Communication and Concurrency*. International Series on Computer Science. Prentice Hall, 1989.

[Milner *et al.*, 1992] Robin Milner, Joachim Parrow, et David Walker. A calculus of mobile process (i and ii). *Inf. and Comp.*, 100 :1–77, 1992.

[Montanari et Pistore, 1997] U. Montanari et M. Pistore. Minimal transition systems for history-preserving bisimulation. Dans *Proceedings of STACS'97*, LNCS 1200, pages 413–425, 1997.

[Nestmann et Pierce, 1996] Uwe Nestmann et Benjamin C. Pierce. Decoding choice encodings. Dans *CONCUR'96 : Proceedings of the 7th International Conference on Concurrency Theory*, pages 179–194, London, UK, 1996.

[Nicola *et al.*, 1990] Rocco De Nicola, Ugo Montanari, et Frits Vaandrager. Back and forth bisimulations. Dans *Proceedings of CONCUR'90*, volume 458 of *LNCS*, pages 152–165, 1990.

[Nielsen *et al.*, 1981] Mogens Nielsen, Gordon Plotkin, et Glynn Winskel. Petri nets, event structures and domains. *Theoretical Computer Science*, 13 :85–108, 1981.

[Nielsen et Winskel, 1995] Mogens Nielsen et Glynn Winskel. *Handbook of Logic and the Foundations of Computer Science*, volume 4, Chapitre Models For Concurrency, pages 1–148. Oxford University Press, 1995.

[Palamidessi et Herescu, 2005] Catuscia Palamidessi et Oltea Mihaela Herescu. A randomized encoding of the π-calculus with mixed choice. *Theoritical Computer Science*, 335(2-3) :373–404, 2005.

[Penczek, 1997] Wojciech Penczek. Model-checking for a subclass of event structure. Dans *Proceedings of TACAS'97*, pages 145–164, 1997.

[Phillips et Ulidowski, 2006] Iain Phillips et Irek Ulidowski. Reversing algebraic process calculi. Dans *Proceedings of FOSSAC'06*, volume 3921 of *LNCS*, pages 246–260, 2006.

[Prasad, 1987] K.V.S Prasad. *Combinators and bisimulation proofs for restartable systems*. PhD thesis, University of Edinburgh, 1987.

[Priami, 1995] Corrado Priami. Stochastic π-calculus. *The Computer Journal*, 38(6) :578–589, 1995.

[Priami et Quaglia, 2004] Corrado Priami et Paola Quaglia. Beta-binders for biological interactions. Dans *Proceedings of CMSB'04*, Lecture Notes in Bioinformatics, 2004.

[Priami *et al.*, 2001] Corrado Priami, Aviv Regev, William Silverman, et Ehud Shapiro. Application of stochastic name-passing calculus to representation and simulation of molecular processes. *Infomation Processing Letters*, 80(1) :25–31, 2001.

[Probst et Li, 1990] David K. Probst et Hon F. Li. Using partial-order semantics to avoid the state explosion problem in asynchronous systems. Dans *Proceedings of CAV'90 : Computer aided verification*, volume 531 of *LNCS*, pages 146–155, 1990.

[Regev et Shapiro, 2002] Aviv Regev et Ehud Shapiro. Cells as computation. *Nature*, 419, September 2002.

[Regev *et al.*, 2001] Aviv Regev, William Silverman, et Ehud Shapiro. Representation and simulation of biochemical processes using the π-calculus process algebra. Dans R. B. Altman, A. K. Dunker, L. Hunter, et T. E. Klein, éditeurs, *Pacific Symposium on Biocomputing*, volume 6, pages 459–470. World Scientific Press, 2001.

[Riely et Hennessy, 2001] James Riely et Matthew Hennessy. Distributed processes and location failures. *Theoretical Computer Science*, 226 :693–735, 2001.

[Sangiorgi, 1995] Davide Sangiorgi. On the bisimulation proof method. Dans *Proceedings of MFCS'95*, volume 969 of *LNCS*, 1995.

[Stirling, 2001] Colin Stirling. *Modal and Temporal Properties of Processes*, volume 269 of *Texts in Computer Science*. Springer, 2001.

[Valmari, 1989] Antti Valmari. Eliminating redundant interleavings during concurrent program verification. Dans *PARLE'89 : Parallel Architectures and Languages Europe*, volume 366 of *LNCS*, pages 89–103, 1989.

[van Glabeek et Goltz, 2003] Rob van Glabeek et Ursula Goltz. Well-behaved flow event structures for parallel composition and action refinement. *Theoretical Computer Science*, 311(1-3) :463–478, 2003.

[Varacca et Yoshida, 2006] Daniele Varacca et Nobuko Yoshida. Typed event structures and the π-calculus. Dans *Proceedings of MFPS XXII*, 2006. To appear.

[Victor et Moller, 1994] Björn Victor et Faron Moller. The Mobility Workbench — a tool for the π-calculus. Dans *Proceedings of CAV'94 : Computer-Aided Verification*, volume 818 of *LNCS*, pages 428–440, 1994.

[Winskel, 1982] Glynn Winskel. Event structure semantics for CCS and related languages. Dans *Proceedings of 9th ICALP*, volume 140, pages 561–576, 1982.

Index des définitions