Spécification formelle des systèmes mobiles temps réel.

Toufik Messaoud MAAROUK

Equipe Vision et Infographie
Laboratoire LIRE, Université de Mentouri, 25000 Constantine

Notations générales

Dans ce mémoire on va utilisé les notations suivantes :

- – **Substitution** : On note $P\{y_1/x_1, ..., y_n/x_n\}$, ou $P\{y_i/x_i\}_{1\leq i\leq n}$, ou $P\{\widetilde{y}/\widetilde{x}\}$, pour la substitution simultanée de y_i pour toutes les occurrences libres de $x_i (1 \leq i \leq n)$ dans P. σ pour une substitution arbitraire.
  - – **Tuples** : On note par $\widetilde{x}$, le vecteur $\{x_1, ..., x_n\}$de noms et n≥ 1.
  - – **Relations et prédicats** : Etant donnée deux relations $\mathcal{R}$ et $\mathcal{R}'$,
    - ∗ **fermeture réflexive** de $\mathcal{R}$ et on note $r(\mathcal{R})$ la plus petite (au sens de l'inclusion) relation réflexive définie sur E contenant $\mathcal{R}$. Autrement dit $r(\mathcal{R})$ est la relation binaire définie sur l'ensemble E telle que :
      - · $r(\mathcal{R})$ est réflexive
      - · $\mathcal{R} \subseteq r(\mathcal{R})$
      - · Pour toute relation binaire $\mathcal{R}$' réflexive définie sur l'ensemble E, si $\mathcal{R}' \subseteq \mathcal{R}$ alors $\mathcal{R}' \subseteq r(\mathcal{R})$
    - ∗ **fermeture transitive** de $\mathcal{R}$ et on note $t(\mathcal{R})$ la plus petite (au sens de l'inclusion) relation transitive définie sur E contenant $\mathcal{R}$. Autrement dit $t(\mathcal{R})$ est la relation binaire définie sur l'ensemble E telle que :
      - · $t(\mathcal{R})$ est transitive
      - · $\mathcal{R} \subseteq t(\mathcal{R})$
      - · Pour toute relation binaire $\mathcal{R}$' transitive définie sur l'ensemble E, si $\mathcal{R}' \subseteq \mathcal{R}$ alors $\mathcal{R}' \subseteq t(\mathcal{R})$
    - ∗ Nous employons les conventions suivantes : $\mathcal{R}\mathcal{R}'$ c'est la composition des relations $\mathcal{R}$ et $\mathcal{R}'$ tel que $\{(x, y)|\ \exists z, x\mathcal{R}z\mathcal{R}'y\}$;
    - ∗ $\mathcal{R}^{-1}$ pour la relation inverse $\{(y, x)|\ x\mathcal{R}y\}$,
    - ∗ $\mathcal{R}*$ est la fermeture transitive de relation $\mathcal{R}$.

Remerciements

Je remercie tout d'abord Djamel Eddine SAIDOUNI qui m'a fait confiance en acceptant ma candidature pour ce thème. Notre travail en commun a été des plus agréables.

J'aimerais remercier Abdel Madjid ZIDANI, Chawki BATOUCHE, Mohammed BEN-MOHAMMED, de m'avoir fait l'honneur de participer à mon jury.

Mes remerciements les plus sincères s'adressent aux responsables de l'institut d'informatique -Batna- pour m'avoir accueilli au sein de l'institut.

Résumé

Notre sujet d'étude est la programmation de systèmes répartis, temps réel. De tels systèmes comportent de nombreux ordinateurs interconnectés par un réseau. L'étude de ce type de système fait appel a des méthodes formelles permettant de répondre aux exigences au quelles sont soumises ces applications. Dans la littérature, il a été proposé un grand nombre de techniques dotées d'un support mathématique pour raisonner sur la conformité des systèmes informatiques.

La notion importante dans ces systèmes, est le temps, c'est à dire des systèmes dotés d'un comportement qui est contraint par le temps. Un système temps-réel doit interagir correctement avec son environnement non seulement au regard des informations échangées, mais également au regard des instants au quels ces interactions se réalisent.

Dans ce contexte, les sémantiques de vrai parallélisme, comme la sémantique de maximalité, conviennent à être employées lorsqu'on s'abstrait de l'hypothèse de l'atomicité temporelle et structurelle des actions. Le modèle D-LOTOS[1], extension temporelle a l'algèbre de processus LOTOS[2], intégrant à la fois contraintes temporelles et durées des actions.

L'étude exposée dans ce document s'inscrit dans le cadre de la conception de systèmes temps-réel et mobiles, en s'appuyant sur les modèles mobiles, et les méthodes formelles, temps réel. Le modèle proposé (MD-LOTOS)[3], représente un langage de programmation directement utilisable il est implémentable de manière répartie, il s'inspire largement de deux modèles (Join-calcul, D-LOTOS).

Mots-clés Systèmes mobiles, Algèbres de processus, Systèmes temps-réel, Durée d'action, Spécification formelle, LOTOS, Langage D-LOTOS, MD-LOTOS.

[1] Duration LOTOS
[2] Language Of Temporal Ordering Specification
[3] Mobil D-LOTOS

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Context

L'informatique répartie a connue, depuis les dix dernières années, une véritable révolution, avec la génération des réseaux locaux et globaux. Pourtant la programmation distribuée semble n'avoir pas bénéficié de cette révolution, tant le nombre de programmes répartis est encore limité.

Des efforts ont été entrepris, à la fois pour offrir à la programmation répartie des modèles formels, et pour fournir des infrastructures simples d'utilisation. Ainsi les algèbres de processus - on retrouve parmi lesquelles CCS[1][30] et le π-calcul[31] [35]- représentent un cadre naturel pour décrire et analyser les systèmes répartis. Elles fournissent plusieurs descriptions d'un même système à des niveaux d'abstraction différents, et des techniques permettant de montrer leur équivalence. Le π-calcul est une extension de CCS, dans la mesure où les processus peuvent échanger des noms de canaux. Cette possibilité augmente le pouvoir expressif de ce langage, et permet de décrire les systèmes dont la topologie du réseau de communications change dynamiquement. Le prix à payer pour ce gain d'expressivité est la complexité de la vérification et d'analyse de ces systèmes.

De l'autre coté une implémentation distribuée de CCS ou de π-calcul et loin d'être évidente, les deux modèles utilisent la communication par rendez-vous, ce qui donne une synchronisation distribuée, qui n'est pas adéquate à la programmation répartie.

Join-calcul représente un modèle élémentaire de programmation répartie, il hérite la plupart des propriétés formelles de π-calcul. Il utilise la communication asynchrone, ce qui le rend implantable directement. Il offre aussi des primitives pour la migration et les pannes.

Les caractéristiques principales de ces systèmes répartis :

· Mobilité : c'est une notion de base dans un système distribué. Dans le cas de $\pi-$calcul cette notion se traduit par l'existence de noms, qui est inséparable du processus de communication fondamentale dans tout système concurrent. L'existence de noms suggère de plus un espace abstrait de processus connectés, dans lequel les noms représentent les connexions,

[1] Calulus of Communicating Systems

seuls les processus qui partagent des noms sont alors en mesure d'interagir. La structure d'un système change donc d'une manière dynamique, car les liens entre processus sont sans cesse crées et détruits.

· Distribution : la distribution implique le concept de domaine ou de localité, les calculs de processus traitent cette caractéristique d'une façon implicite ou explicite. Le calcul des ambient[6] [7] propose des primitives de domaines explicite, appelés ambient. Le Join-calcul distribué[13] propose une autre manière de traitement tel que la solution chimique locale et la solution distribuée.

Une autre notion crucial dans les applications réparties tels que les protocoles de communication c'est le temps. Ces applications sont de nature des applications temps réel. Ces systèmes sont de plus en plus important dans la société actuelle. L'essor croissant des technologies permet de définir des systèmes informatiques sophistiqués d'une complexité de plus en plus difficile a maîtriser. Ainsi, la conduite automatisée de métro, la supervision d'une centrale nucléaire, sont autant d'exemples de cette informatisation croissante des systèmes complexes.

En effet, le système doit être modélisé à l'aide de langage dit formels. Toutefois, les langages de ce type se révèlent complexes a utiliser. Parmi les algèbres de processus ET-LOTOS[2][21], RT-LOTOS[3][10], D-LOTOS[37], etc. Ces modèles utilisent des contraintes temporelles explicites. Ces contraintes se traduisent par des exigences sur la façon d'observer et/ou de piloter l'environnement.

Dans le cas du langage D-LOTOS ; c'est un modèle basé sur un autre modèle sémantique dit de vrai parallélisme à la place de la sémantique classique d'entrelacement. D-LOTOS intègre à la fois des contraintes temporelles et durées d'actions.

Notre travail consiste à définir un modèle sémantique temps-réel et mobile, basé sur la sémantique de maximalité. Ce modèle doit être capable de spécifier des systèmes répartis, mobile et temps réel.

1.2 Contributions

- la première contribution de ce mémoire, est l'étude détaillée de différents algèbres de processus (CCS, π-calcul, Join-calcul, Ambient) de spécification de systèmes répartis et mobiles. Tout ces modèles sont basé sur la sémantique classique d'entrelacement et ne supportent pas la non-atomicité structurelle des actions, une autre approche exposée dans ce document est la logique de réecriture.

- La seconde contribution consiste a la proposition d'un modèle ou langage (MD-LOTOS[4]) de spécification des systèmes temps-réel et mobiles. Ce langage est basé sur une sémantique de vrai parallélisme entre autre sémantique de maximalité.

[2] Enhanced Timed-LOTOS
[3] Real Time LOTOS
[4] Mobil D-LOTOS

- Une autre contribution est d'offrir une présentation de l'utilisation de ce langage dans un cas réel, et la proposition d'un outil capable de compiler des spécifications écrites dans ce langage.

1.3 Plan du document

Ce document est organisé en cinq chapitres :

Chapitre 1 : Ce chapitre introduit la problématique et le contexte de la description formelle des systèmes temps réel et mobiles.

Chapitre 2 : Ce chapitre expose en détaille les modèles algébriques de spécification des systèmes mobiles, a savoir CCS, LOTOS, $\pi-$calcul, join-calcul, calcul des ambients, et la logique de réécriture, on a étudier les techniques de validations. Enfin une discussion pour mettre en évidence les difficultés de spécification de ces modèles.

Chapitre 3 : Ce chapitre introduit les modèles algébriques de spécification des systèmes temps-réel tels que RT-LOTOS, et D-LOTOS. On termine par une discussion sur les limites de D-LOTOS.

Chapitre 4 : Ce chapitre propose un modèle permettant de spécifier des systèmes temps réel et mobiles, avec une étude de cas, on propose aussi un outil de compilation.

Chapitre 5 : dans ce chapitre nous concluons et présentons les ouvertures possibles.

Chapitre 2

Modèles algébriques de spécification des applications mobiles

Ce chapitre introduit différents modèles algébriques de description formelle pour la spécification des systèmes répartis et mobiles. La première section expose le fomalisme CCS et LOTOS en terme de syntaxe et sémantique opérationnelle.... La seconde expose le modèle $\pi-$calcul, extension de CCS avec mobilité. La troisième expose le Join-calcul : un calcul pour la programmation répartie et mobile, ce modèle s'inspire des deux modèles du parallélisme $\pi-$calcul d'une part et la machine chimique abstraite de Berry et Boudol[5]. La quatrième section expose le calcul des ambients, qui s'inspire du $\pi-$calcul. La cinquième section présente la logique de réécriture une autre approche de spécification des systèmes temps réel et mobiles. Finalement on termine par une conclusion et discussion, pour mettre en évidence les insuffissances de chaque modèle étudier.

2.1 Introduction aux algèbres de processus

2.1.1 CCS : Calculus of Communicating Systems

Le formalisme CCS[30] décrit les systèmes communicants comme des ensembles d'automates non-déterministes (appelés agents ou processus) qui interagissent par le biais de synchronisations. Le comportement d'un agent n'est décrit que partiellement par l'ensemble de ses traces (une trace est une suite de transitions) et la notion de bisimulation propre à cette théorie permet de raffiner la notion d'égalité des agents.

Les actions constituent l'alphabet de base des processus. On suppose un ensemble dénombrable d'actions a, b, c ... chacune ayant un inverse unique $\overline{a}, \overline{b}, \overline{c}$... . Il existe aussi une action particulière τ(tau) considérée comme interne et silencieuse. Le sens intuitif de la no-

tion d'inverse est que l'action et son inverse constituent des actions pouvant se synchroniser si elles proviennent de deux processus communicants.

La syntaxe des processus CCS[30] est la suivante.

$$\begin{array}{ll} \alpha & := a(x) \mid \overline{a}.v \mid \tau \\ P, Q, R & ::= \mathbf{0} \mid a.P \mid P + Q \mid (P|Q) \mid (v\alpha)P \end{array}$$

FIG. 2.1 – *Syntaxe de CCS*

Dans CCS on à trois types d'actions :

$a(x)$: sur le canal a on peut reçevoir n'importe qu'elle valeur.

$\overline{a}.v$: émettre la valeur v sur le canal a.

τ : action interne ou silencieuse.

Un processus P peut être soit le processus inerte $\mathbf{0}$, soit un processus préfixé par une action a, soit une composition parallèle de processus $P|Q$, ou une restriction c'est à dire seul le processus P à accés sur le nom x.

La signification de la syntaxe des processus est donnée par une relation de transition ternaire $P \xrightarrow{a} Q$, qui signifie "le processus P peut effectuer l'action a et devenir le processus Q". Cette relation est définie par les règles ci-dessous (Figure : 2.2).

$$\text{Act}\ \ \alpha.P \xrightarrow{\alpha} P' \qquad \text{Sum1}\ \ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{Sum2}\ \ \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\text{Par1}\ \ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{Par2}\ \ \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\text{Com1}\ \ \frac{P \xrightarrow{a(x)} P' Q \xrightarrow{\overline{a}v} Q'}{P \mid Q \xrightarrow{\tau} P'\{v/x\} \mid Q} \qquad \text{Com1}\ \ \frac{P \xrightarrow{\overline{a}v} P' Q \xrightarrow{a(x)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q\{v/x\}}$$

$$\text{Res}\ \ \frac{P \xrightarrow{\alpha} P' \alpha \notin \{a, \overline{a}\}}{P + Q \xrightarrow{\alpha} P'}$$

FIG. 2.2 – *Sémantique opérationnelle de CCS*

La sémantique opérationnelle $P \xrightarrow{a} Q$, détermine donc pour tout processus ses comportements ou traces possibles comme s'il s'agissait simplement d'une notation particulière pour un automate non-déterministe dont tous les états atteignables seraient acceptants. Mais à la différence d'un automate fini classique, les comportements d'un processus communicant doivent tenir compte des traces infinies c'est-à-dire ne pas considérer tout comportement infini comme une "erreur" à éviter. Au contraire, ils représentent souvent des programmes serveurs dont le comportement est ; volontairement et naturellement infini puisque c'est alors l'arrêt qui peut-être considéré comme une erreur. Le langage de traces d'un processus communicant est donc en général un ensemble de mots finis et infinis sur l'alphabet des événements.

Mais en plus d'engendrer ses propres comportements de manière autonome, un processus est aussi caractérisé sur sa "connectique" c-à-d son interaction avec l'environnement. Par exemple $(a.P|\overline{a}.P)$ pouvant émettre a, il peut aussi se synchroniser avec un processus de la forme $\overline{a}.Q$. Or cet aspect du comportement n'est pas décrit par le langage de traces qui "oublie" la structure exacte des transitions possibles. MILNER[30] propose une relation d'équivalence entre processus qui soit plus fine que la simple équivalence de traces (égalité des langages de traces). C'est la bisimulation forte en CCS que nous allons expliquer par un exemple avant de la définir formellement.

Soient les processus $A = a.(b.P + c.Q)$ et $B = a.b.P + a.c.Q$. On constate que leurs traces sont les mêmes : trace vide, a, ab, ac, ab suivie d'une trace de P ou ac suivie d'une trace de Q. Mais les comportements de A et B diffèrent de la manière suivante. Le processus obtenu après $A - a- >$ peut réaliser l'une ou l'autre des actions $\{b, c\}$, il peut par exemple se synchroniser indifféremment avec $\overline{b}.R$ ou $\overline{c}.R$. Ce n'est pas le cas du processus obtenu après $B - a- >$ puisque celui-ci n'est pas uniquement déterminé. S'il s'agit de $b.P$, il "bloque" l'action c, c'est-à-dire qu'il ne pourrait se synchroniser avec $\overline{c}.R$ et s'il s'agit de $\overline{c}.Q$ il bloque l'action b. Cette différence de comportement est bien visible sur les systèmes de transition de A et de B (Voir la figure 2.3) :

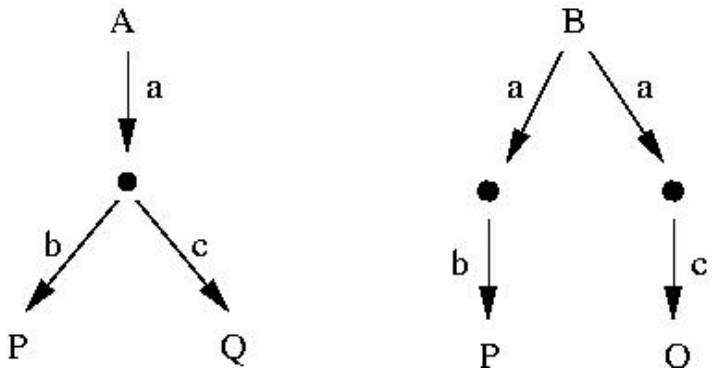


FIG. 2.3 – Comportement 1

Alors que l'ensemble des traces, étant justement un ensemble, n'encode pas les deux occurrences de $B - a- >$ menant à des comportements différents pour B.

Autre exemple de la nécessité d'une notion de bisimulation. Soit **0** un processus qui ne peut effectuer aucune transition, alors du point de vue de la bisimulation on veut aussi distinguer des processus comme dans la figure 2.4 (12).

Car la trace initiale $-a- >$ peut, pour le processus de droite, mener à une situation de blocage ce qui est faux pour le processus de gauche. Or les traces des deux processus sont les mêmes : trace vide, a, ab,ou ab suivie d'une trace de P.

On pourrait penser que la bonne notion de bisimulation est l'équivalence des graphes de transition ce qui reviendrait à dire que deux processus seraient bisimilaires si et seulement si leurs graphes de transitions sont égaux. Mais ce serait alors une relation trop fine car certains

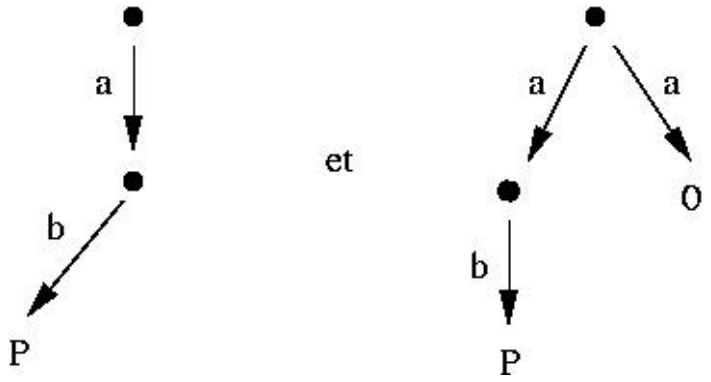


FIG. 2.4 – Comportement 2

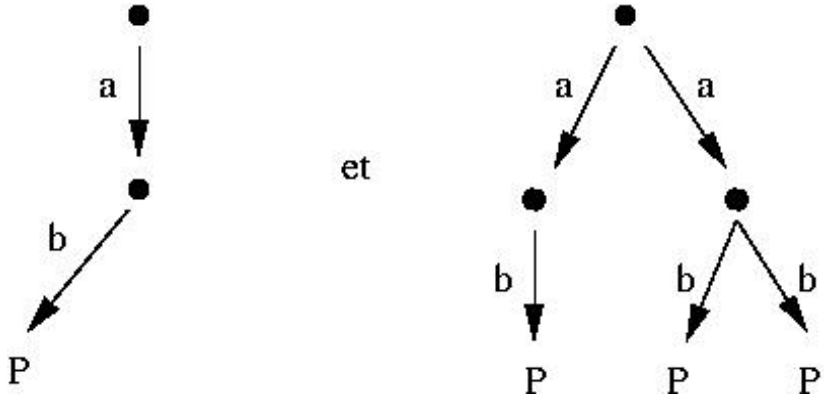


FIG. 2.5 – Comportement 3

graphes de transition ont des symétries et ne sont que le dépliement d'un autre graphe, par exemple dans la figure 2.5, ne diffèrent, ni par leurs traces ni par leurs comportements possibles ; aucun processus extérieur saurait détecter de différence entre ces deux processus. Il sera donc normal de les identifier par la relation de bisimulation. La définition exacte de la bisimulation forte est formulée récursivement le long des transitions.

Définition 2.1 *[30] : Une relation binaire ˜entre processus CCS est une bisimulation forte si, P ˜Q implique que :*

1. Toute transition $P \xrightarrow{a} P\prime$ *implique l'existence d'une transition* $Q \xrightarrow{a} Q\prime$ *telle que* $P\prime \tilde{} Q\prime$

2. Et vice-versa : toute transition $Q \xrightarrow{a} Q\prime$ *implique l'existence d'une transition* $P \xrightarrow{a} P\prime$ *telle que* $P\prime \tilde{} Q\prime$.

On dit que deux processus P et Q sont bisimilaires s'il existe une bisimulation forte $\tilde{}$ pour laquelle $P \tilde{} Q$ et on écrit alors "$P \tilde{} Q$" si la relation générale $\tilde{}$ n'importe pas ou si on peut la déduire du contexte.

Proposition 2.1 *Pour toute bisimulation forte ˜et les processus P,Q,R on a :*
$P + Q \tilde{} Q + P$
$P + (Q + R) \tilde{} (P + Q) + R$
$P + P \tilde{} P$

2.1.2 Algèbre de processus à synchronisation multiple : LOTOS

LOTOS[3][16][15][23] (Language of Temporal Ordering Specification) est une technique de description formelle, promue au rang de norme ISO en 1988.

Il s'appuie sur le langage CCS de MILNER (étendu par un mécanisme de synchronisation multiple hérité de CSP [14] de HOARE) pour la spécification de la partie comportementale ; la partie description des structures de données est inspirée de ACT-ONE , un formalisme de description des types de données abstraits algébriques.

Le concept sous-jacent à LOTOS est que tout système peut être spécifié en exprimant les relations qui existent entre les interactions constituant le comportement observable des composantes du système. En LOTOS, un système est vu comme un processus, qui peut être constitué de sous-processus, un sous-processus étant un processus en lui-même. Une spécification LOTOS décrit ainsi un système par hiérarchie de processus. Un processus représente une entité capable de réaliser des actions internes (non-observable) et d'interagir avec d'autres processus qui forment son environnement.

Les définitions de processus sont exprimées par la spécification d'expressions de comportement qui sont construites à partir d'un ensemble réduit d'opérateurs donnant la possibilité d'exprimer des comportements aussi complexes que l'on désire. Les processus sont en général définis récursivement, et le rendez-vous multidirectionnel constitue le mécanisme de base pour la communication inter-processus. Parmi les opérateurs, ceux de préfixage (;), de choix non-déterministe ([]), de composition parallèle (|[.]|) et d'intériorisation (hide) jouent un rôle fondamental.

Présentation de Basic LOTOS

Basic-LOTOS est un sous ensemble de LOTOS où les processus interagissent entre eux par synchronisation pure, sans échange de valeurs. En Basic-LOTOS les actions sont identiques aux portes de synchronisation des processus. Les principaux opérateurs de Basic-LOTOS sont listés dans le tableau 2.1 :

Syntaxe de Basic- LOTOS

- Soit P l'ensemble des identifiants de processus.
  - Soit $X \in P$.
  - Soit $\mathcal{G}$ l'ensemble des portes définissables (les actions observables en Basic-LOTOS).
  - Soient $g, g_1 ... g_n \in \mathcal{G}$.
  - Soit L un sous-ensemble (pouvant être vide) quelconque de $\mathcal{G}$ noté $L = g_1 ... g_n$.

Opérateur		Notation	Description informelle
Inaction		`stop`	Processus de base n'interagissant pas avec son environnement
Terminaison avec succès		`exit`	Processus qui se termine (action δ) et se transforme en `stop`
Préfixage par une action	non observable	$i; P$	Processus qui réalise l'action i ou g, puis se transforme en P
	observable	$g; P$	
Choix non-déterministe		$P_1 [] P_2$	Processus qui se transforme en P_1 ou en P_2 suivant l'environnement
Composition parallèle	cas général	$P_1 \|[g_1, ...g_n]\| P_2$	P_1 et P_2 s'exécutent en parallèle et se synchronisent sur les portes $g_1, ...g_n$ et δ
	asynchrone	$P_1 \|\|\| P_2$	P_1 et P_2 s'exécutent en parallèle sans se synchroniser (sauf sur δ)
	synchrone	$P_1 \|\| P_2$	P_1 et P_2 s'exécutent en parallèle et se synchronisent sur chaque porte visible
Intériorisation		*hide* $g_1, ...g_n$ *in* P	Les actions $g_1, ..., g_n$ sont cachées à l'environnement de P et deviennent des actions internes
Composition séquentielle		$P_1 \gg P_2$	P_2 est activé dès que P_1 se termine.
Préemption (interruption)		$P_1 [> P$	P_2 peut interrompre P_1 tant que P_1 ne s'est pas terminé

TAB. 2.1 – *Principaux opérateurs de Basic-LOTOS*

- Soit i l'action interne.

La syntaxe formelle du Basic-LOTOS est donnée par la figure 2.6

$$\textbf{Processus } X[g_1, ..., g_n] := P \textbf{ endproc}$$

$$\begin{array}{lcl} P & ::= & \textbf{stop} \mid \textbf{exit} \mid X[L] \mid i;P \mid g;P \\ & \mid & P[]P \mid P|[L]|P \mid \textbf{hide } L \textbf{ in } P \\ & \mid & P >> P \mid P[> P \end{array}$$

FIG. 2.6 – *Syntaxe de Basic-LOTOS*

Sémantique opérationnelle de Basic-LOTOS

La tableau 2.2, page 16 présente les règles d'inférences de la sémantique opérationnelle de Basic-LOTOS. Les notations suivantes seront utilisées :

- δ est l'action de terminaison de Basic-LOTOS.
- $\mathcal{G}^i = \mathcal{G} \cup \{i\}, \mathcal{G}^\delta = \mathcal{G} \cup \{\delta\}, \mathcal{G}^{i,\delta} = \mathcal{G} \cup \{i, \delta\}$
- $P \xrightarrow{g} P'$, signifie que le processus P peut réaliser l'action g et se comporte ensuite comme $P\prime$.

Présentation de full LOTOS

Full LOTOS (ou simplement LOTOS), est Basic-LOTOS étendu avec la possibilité d'échanger des valeurs lors des synchronisation entre processus. Contrairement à Basic-LOTOS, en full LOTOS une action n'est pas simplement une porte de synchronisation, mais une porte plus des données échangées lors de la synchronisation.

Les structures de données et les opérations associées sont définies au moyen du langage Act-One qui formalise la spécification de types abstraits de données : il définit les propriétés essentielles des données et les opérations qu'une implémentation correcte du type doit assurer.

Les valeurs en LOTOS peuvent être :

- échangées entre processus lors d'une synchronisation,
- employées dans les prédicats de garde des processus,
- utilisées comme paramètres pour la définition des processus et pour l'instantiation de valeurs,
- associées à l'opérateur de choix généralisé,
- Exportées lors d'une terminaison avec succès d'un processus,

Avec l'ajout des données dans la spécification, les actions deviennent des entités qui adjoignent à une porte trois composantes de base, comme cela est illustré dans le tableau 2.3 page 17. Une action peut :

- offrir une valeur $x(!x)$,
- accepter une valeur $y(?y : type)$,
- incorporer des prédicats qui conditionnent l'acceptation de valeurs.

$exit \xrightarrow{\delta} stop$
$g; P \xrightarrow{g} P \quad (g \in \mathcal{G})$
$i; P \xrightarrow{i} P$
$\dfrac{P \xrightarrow{g} P'}{P[]Q \xrightarrow{g} P'} \quad (g \in \mathcal{G}^{i,\delta})$
$\dfrac{P \xrightarrow{g} P' \quad Q \xrightarrow{g} Q'}{P\|[L]\|Q \xrightarrow{g} P'\|[L]\|Q'} \quad (g \in L \cup \{\delta\})$ $\dfrac{P \xrightarrow{g} P'}{P\|[L]\|Q \xrightarrow{g} P'\|[L]\|Q} \quad (g \in \mathcal{G}^i \diagdown \{\delta\})$
$\dfrac{P \xrightarrow{g} P' \quad (g \in \mathcal{G}^{i,\delta} \diagdown L)}{hide\ L\ in\ P \xrightarrow{g} hide\ L\ in\ P'}$ $\dfrac{P \xrightarrow{g} P' \quad (g \in L)}{hide\ L\ in\ P \xrightarrow{i} hide\ L\ in\ P'}$
$\dfrac{P \xrightarrow{g} P'}{P \gg Q \xrightarrow{g} P' \gg Q} \quad (g \in \mathcal{G}^i)$ $\dfrac{P \xrightarrow{\delta} P'}{P \gg Q \xrightarrow{i} Q}$
$\dfrac{P \xrightarrow{g} P'}{P[> Q \xrightarrow{g} P'[> Q} \quad (g \in \mathcal{G}^i)$ $\dfrac{Q \xrightarrow{g} Q'}{P[> Q \xrightarrow{g} Q'} \quad (g \in \mathcal{G}^{i,\delta})$ $\dfrac{P \xrightarrow{\delta} P'}{P[> Q \xrightarrow{\delta} Q}$
$\dfrac{P_X[g_1/g'_1 ... g_n/g'_n] \xrightarrow{g} Q' \quad X[g_1 ... g_n] = P_X}{X[g_1 ... g_n] \xrightarrow{g} Q'} \quad (g \in \mathcal{G}^{i,\delta})$ $\dfrac{P \xrightarrow{g} P'}{P_\phi \xrightarrow{\phi\{g\}} P'_\phi} \quad (\phi = [g_1/g'_1 ... g_n/g'_n])$ $(g \in \mathcal{G}^{i,\delta})$

TAB. 2.2 – *Sémantique opérationnelle de Basic-LOTOS*

$\mathbf{P_1}$	$\mathbf{P_2}$	**Condition de synchronisation**	**Type d'interaction**	**Résultat**
$g!v_1$	$g!v_2$	$valeur(v_1) = valeur(v_2)$	Concordance des valeurs	Synchronisation
$g!v$	$g?x : T$	$v \in T$	Passage de valeur	Après synchronisation, $x = valeur(v)$
$g?x : T$	$g?y : U$	$T = U$	Choix aléatoire d'une valeur	Après synchronisation, $x = y = v$ avec $v \in T$(ou $v \in U$)

TAB. 2.3 – *Règles de synchronisation en LOTOS avec données*

Le tableau 2.3 illustre les nouvelles règles en prenant l'exemple de deux processus P_1 et P_2 composés par une synchronisation sur la porte g (la fonction valeur renvoie la valeur d'une variable). La règle veut qu'après une synchronisation, les actions impliquées dans cette synchronisation doivent offrir des variables de même type et de même valeur.

La déclaration de processus est étendue avec l'usage de paramètres. Par exemple :

$$\textbf{Processus} \quad X[g_1, ...g_n](x_1 : T_1, x_m : T_m) := P \quad \textbf{endproc}$$

Définit, pour le processus X, m variables $x_1...x_m : T_1...T_m$ qui le paramètrent. Ainsi, ce processus devra être instancié avec une liste de valeurs $v_1...v_m$ affectées à ces paramètres : $X[g_1, ...g_n](v_1...v_m)$

Le tableau 2.4, page 18 liste les principaux opérateurs introduits dans LOTOS pour manipuler les données. Dans ce tableau, x correspond à un nom de variable, $Type$ à un type de données et *pred* à une expression logique (prédicat de garde).

Sémantique opérationnelle de LOTOS

Le tableau 2.7, page 18 présente de quelle manière la sémantique de Basic LOTOS est enrichie pour donner la sémantique opérationnelle de LOTOS.

2.2 π-calcul

π-calcul[31][35] est un calcul de processus parallèles, qui permet de rendre compte de systèmes dont la topologie de communication change dynamiquement. Ce calcul est maintenant considéré comme un possible fondement de la programmation distribuée, au même titre que le λ-calcul[17] est considéré comme le fondement de la programmation fonctionnelle. En particulier, le π-calcul[31][35] permet de rendre compte de la notion de localité et de migration

Opérateur	**Notation**	**Description informelle**
Composition séquentielle	**exit** $(v_1, ...v_m)$ >> **accept** $x_1 : T_1, ...x_m : T_m$ **in** P	Le processus qui a terminé avec success transmit des valeurs $v_1...v_m$ au processus suivant qui peut les consulter à travers les variables $x_1 : T_1, ...x_m : T_m$ respectivement.
Choix généralisé	**choice** $x : Type$ [] $P(x)$	Si P dépend des variables x, l'opérateur offre le choix entre les processus pour toutes les valeurs possibles de $x \in Type$.
Déclaration de variable	**let** $x : Type = v$ **in** P	Réalise l'instanciation de la variable x avec la valeur v dans P.
Prédicat de garde	$[pred]$ –>P	Le processus aura le comportement de P si *pred* est vrai, sinon il devient **stop**.

TAB. 2.4 – *Principaux opérateurs LOTOS avec utilisation de données*

$exit(v_1...v_m) \xrightarrow{\delta!(valeur(v_1)...valeur(v_n))} stop$
$g!v; P \xrightarrow{g : (valeur(v))} P \quad (g \in G)$ $g?y : T; P \xrightarrow{g?(p)}$ **let** $x : T = p$ **in** $P (g \in G)(p \in T)$
$\dfrac{P \xrightarrow{g} P'}{\textbf{let } x : T = v \textbf{ in } P \xrightarrow{g} \textbf{let } x : T = v \textbf{ in } P'} \quad \dfrac{[v/x]P \xrightarrow{g} P'}{\textbf{let } x : T = v \textbf{ in } P \xrightarrow{g} P'}$
$\dfrac{P \xrightarrow{g} P' \quad valeur(v) = true}{[v] -> P \xrightarrow{g} P'}$
$\dfrac{P_X[g_1/g'_1...g_n/g'_n](valeur(v_1)/y_1...valeur(v_n)/y_n) \xrightarrow{g} Q' \quad X[g'_1...g'_n](y_1 : T_1...y_n : T_n) = P_X}{X[g_1...g_n](valeur(v_1)...valeur(v_n)) \xrightarrow{g} Q'}$

FIG. 2.7 – *Sémantique opérationnelle de LOTOS*

au cours de calculs, il permet de plus de décrire des systèmes de processus mobiles, c.à.d. des systèmes dont le nombre de processus ainsi que liens de communication entre processus peuvent varier au cours du temps. π-calcul est principalement utilisé pour établir des preuves d'équivalence entre des modèles de systèmes distribués[31]. Nous présentons ce calcul de systèmes communicants dans lequel non seulement les processus ont la structure dynamique, mais aussi les agents composants le système, peuvent porter l'information qui change ces liens.

Dans cette section on présente le π-calcul dans les deux formes monadique et polyadique : syntaxe, exemples de base, sémantique par réduction et par système de transitions étiquetées, relations bisimulations et théorie algébrique.

Commençons par une série d'exemples, qui ont une signification pratique. Dans le premier exemple, nous le présentons d'abord dans la notion de CCS et nous proposons un diagramme(graphique de flux), qui représente les liens entre les agents. Supposons qu'un agent P veut envoyer la valeur 5 à un agent R, le long d'une liaison nommée a, et que R désire recevoir n'importe quelle valeur le long de cette liaison. Alors le graphique de flux approprié est dans la figure 2.8

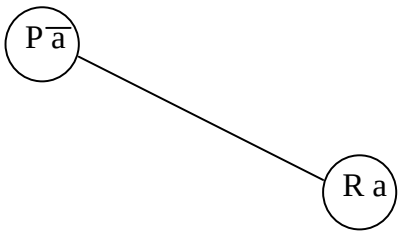


FIG. 2.8 – Exemple 1

Nous pouvons avoir, par exemple, $P \equiv \overline{a}5.P'$ et $R \equiv a(x).R'$. Le préfixe $a(x)$ lie la variable x dans R', en général nous employons des parenthèses pour indiquer la présence d'une variable. Le système dépeint dans le graphique de la figure 2.8 est représenté par l'expression

$$(\overline{a}5.P' \mid a(x).R')/a$$

L'opérateur $/a$ est appelé une restriction, et indique que la liaison a est privée a P et R.

On suppose maintenant que P veut déléguer à un nouvel agent, Q, la tâche de transmettre 5 à R. Nous supposons donc que P est connecté à Q au départ par une liaison b,

Soit le processus $P \equiv \overline{b}a.\overline{b}5.P'$, il envoie le long de la laison b, la liaison a et la valeur 5 pour être transmis le long de a. Et soit $Q \equiv b(y).b(z).\overline{y}z.0$, il reçoit une liaison et une valeur sur b, transmet ensuite la valeur le long de la liaison et se termine. Notez que le nom a n'est pas dans l'expression Q, Q ne possède aucune liaison à R au départ. Le système entier est maintenant :

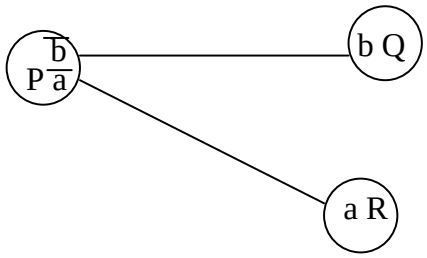


FIG. 2.9 – Exemple 2

$$(\overline{b}a.\overline{b}5.P' \mid b(y).b(z).\overline{y}z.0 \mid a(x).R')/a/b$$

Après deux communications, sur la liaison b, le système devient :

$$(P'|\overline{a}5.0|a(x).R')/a/b$$

Si a n'apparaît pas dans P', la nouvelle configuration du système est donnée par la figure 2.10, indiquant que la liaison a de P s'est déplacée à Q, et Q est devenu $Q' \equiv \overline{a}5.0$.

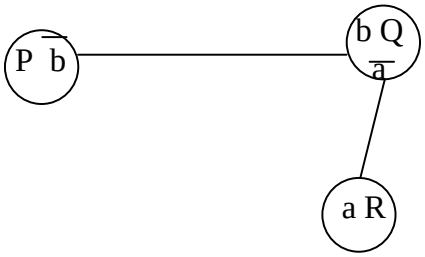


FIG. 2.10 – Exemple 3

Ce formalisme, dans lequel les noms des liaisons apparaissent comme des paramètres dans la communication, va au-delà de CCS. Avec les variables sur des noms de liaison, aussi bien que sur des valeurs de données ordinaires, le calcul deviendrait riche de primitives. Mais toute distinction entre des noms de liaison, des variables et des valeurs de données ordinaires est enlevée, nous les appellerons tous des noms. Il y aura juste deux classes essentielles d'entité : noms et des agents.

2.2.1 π-calcul monadique

Syntaxe

Dans cette section on introduit la syntaxe du π-calcul. L'entité primitive dans le π-calcul est le nom(canal).

· Soit $\mathcal{X}$ l'ensemble des noms parcouru par $x, y, ...$

· Soit $\mathcal{P}$ l'ensemble des identifiants de processus parcouru par $P, Q, ...$

La sysntaxe formelle de π-calcul est donnée par la figure 2.11 :

$P, Q, R ::=$		**Processus**
	$\Sigma\pi_i.P_i$	
	$\mid P\mid Q$	*Définition local*
	$\mid\ !P$	*Composition parallèle*
	$\mid (vn)P$	*Réstriction*

FIG. 2.11 – *Syntaxe de Pi-calcul*

1. La sommation $\sum_{i\in I}\pi_i.P_i$, I ensemble d'index fini.

- Pour l'ensemble vide I=ϕ, la somme devient 0.

- Dans le cas $\pi.P$, le préfixe π représente une action atomique. Ce préfixe(action) peut avoir les formes suivantes :

$x(y)$ Action d'entrée signifie que P peut recevoir n'importe quel nom w, sur le canal x et se transformer ensuite en $P\{w/y\}$.

$\overline{x}.y$ Action de production c'est à dire on peut émettre le nom libre y sur le canal x.

τ Représente l'action silencieuse.

Dans les deux cas on appel x sujet et y l'objet de l'action.

2. La composition $P|Q$: Les deux processus sont activés simultanément, donc indépendamment, mais ils peuvent communiquer.

3. L'opérateur de duplication $!P$: donne plusieurs copies parallèles de P, $P|P|...$

4. La restriction $(vx)P$: seul le processus P a accès sur le nom x.

Définitions de base

- Les noms libres $\text{fn}(P)$ de P sont les noms qui figurent dans P, et ils sont pas liés par un préfixe d'entrée ou par une restriction. Et en les appellent noms liés $\text{bn}(P)$ dans les autres cas.

$$\begin{aligned} \text{bn}(x(y)) &= \{y\}, \qquad \text{fn}(x(y)) = \{x\}. \\ \text{bn}(\overline{x}y) &= \phi, \qquad\ \ \text{fn}(\overline{x}y) = \{x, y\}. \end{aligned}$$

Exemples

- Soit le processus : $\overline{x}y.0 \mid x(u).\overline{u}v.0 \mid \overline{x}z.0$

  Ce transforme en : $\overline{x}y \mid x(u).\overline{u}v \mid \overline{x}z$ ce processus est de la forme $P|Q|R$. Une des deux communications (mais pas les deux) peut se faire a travers le canal x, P envoi y a Q, ou R envoi z a Q. Donc on obtient :

$$0|\overline{y}v|\overline{x}z \quad \text{ou} \quad \overline{x}y|\overline{z}v|0$$

- Soit le processus $(vx)(\overline{x}y \mid x(u).\overline{u}v) \mid \overline{x}z$

  Dans ce cas le nom x qui est libre dans R est différent de x qui est lié dans P et Q, donc on a une seule communication :

$$0|\overline{y}v|\overline{x}z$$

2.2.2 π-calcul polyadique

Afin de faciliter la manipulation des termes du π-calcul, on va utiliser une forme moins primitive pour la communication sur un canal. On va permettre le passage simultané de plusieurs arguments sur un canal. La syntaxe est donc la même que celle de la forme monadique cependant les changements suivants sont a considérer :

· Le processus $x(\widetilde{y}).P$, attend un tuple $\widetilde{z}$ soit transmis sur le canal x, puis il continu le processus P avec la substitution de tuple $\widetilde{y}$ par le tuple $\widetilde{z}$.

· Le processus $\overline{x}\widetilde{y}.P$, envoie le tuple $\widetilde{y}$ sur le canal x, et ensuite continu le processus P.

2.2.3 Sémantique opérationnelle du π-calcul

Traditionnellement, la sémantique d'une algèbre de processus est donnée, en terme de système de transitions étiquetées, décrivant les évolutions possibles des processus. Milner offre dans [31] une directive pour la définition des systèmes de réduction dans des algèbres de processus, où les axiomes pour la relation de congruence structurelle sont présentés avant le système de réduction.

La sémantique opérationnelle de π-calcul, est représentée par des systèmes de transitions étiquetées(STE) [35], ou un système réduit avec une relation de congruence structurelle ($\equiv$)[31], c'est la plus petite relation de congruence sur les processus.

Sémantique de réduction

Définition 2.2 *(congruence structurelle)[31] : Dans un système de réduction, on définit une congruence structurelle notée $\equiv$ telle que les opérateures + et | sont associative et commutative. Cette congruence c'est la plus petite relation de congruence sur $\mathcal{P}$, qui vérifier les règles suivantes :*

1. Les agents(processus) sont identifiés, s'ils diffèrent seulement par un changement de noms liés.

2. $(\mathcal{X}/\equiv, +, 0)$*, est un monoïde symétrique.*

3. $(\mathcal{P}/\equiv, +, 0)$*, est un monoïde symétrique.*

4. $!P \equiv P|!P$

5. $(vx)0 \equiv 0, (vx)(vy)P \equiv (vy)(vx)P$

6. si $x \in fn(P)$ *alors* $(vx)(P|Q) \equiv (vx)P|(vx)Q$

Règles de réduction Une relation de réduction $P \longrightarrow P'$ sur un processus P. Signifie que P peut ce transformer en P' par une étape de calcul simple. Donc toute étape de calcul, c'est une interaction entre deux termes. La première réduction c'est la communication :

$$\textbf{COMM} \ :(... + x(y).P \mid ... + \overline{x}z.Q \ \rightarrow \ P\{z/y\} \mid Q \ )$$

Elle représente une communication entre deux processus atomiques $\pi.P$ qui sont complémentaires. Les (...) peuvent être **0**, ou d'autres communications.

COMM c'est un axiome pour la réduction, les autres réductions sont des règles d'inférence.

$$\textbf{PAR} : \quad \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q} \qquad \textbf{RES} : \frac{P \rightarrow P'}{(vx)P \rightarrow (vx)P'}$$

$$\textbf{STRUCT} : \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

Remarque : on ne peut pas réduire dans les cas suivants

1. Si on a un préfixe, par exemple :

$$u(v).(x(y) \mid \overline{x}z)$$

L'opération de préfixage est prioritaire par apport à l'opération de communication.

2. Si on a une réplication.

Si on a $P \longrightarrow P'$ alors, au lieu de la réduction, $!P \longrightarrow !P'$, qui est équivalente à la réduction illimitée de copies de P, nous pouvons toujours réduire de la manière suivante :

$$!P \equiv P|P|...|P|!P \longrightarrow^n P'|P'|...|P'|!P'$$

Ainsi (après n réductions) réduisant autant de copies de P que nous exigeons.

Sémantique de transitions étiquetées

Action : la transition dans le π-calcul est de la forme :

$$P \xrightarrow{a} Q$$

Intuitivement, cette transition représente une transformation de processus P vers le processus Q en consommant l'action a. Dans le π-calcul il y a quatre forme de l'action a :

1. L'action silencieuse τ, comme dans CCS, signifie que P se transforme en Q , avec aucune interaction avec l'environnement. Les actions silencieuses peuvent être naturellement le résultat des agents de la forme τ.P, mais aussi des communications dans un agent.

2. Une action de production libre $\overline{x}y$, la transition $P \xrightarrow{\overline{x}y} Q$ implique que P peut émettre le nom libre y sur le canal x . Les actions de production libres résultent de la forme de préfixe de production $\overline{x}y.P$.

3. Une action d'entrée $x(y)$. Intuitivement, $P \xrightarrow{x(y)} Q$ signifie que P peut recevoir n'importe quel nom w, sur le canal x et se transformer ensuite en $Q\{w/y\}$. Ici y représente une référence a la place où le nom reçu ira. Les actions d'entrée résultent de la forme de préfixe d'entrée $x(y).P$.

4. Une action de production liée $\overline{x}(y)$. Intuitivement, $P \xrightarrow{\overline{x}(y).} Q$ signifie que P émet un nom privé sur le canal x , et (y) est une référence où ce nom privé arrive. Comme dans l'action d'entrée ci-dessus, y est inclus entre parenthèses pour souligner que c'est une référence et ne représente pas de nom libre.

La relation de transition c'est la plus petite relation qui satisfait les règles du tableau 2.5, page 25 :

Communication de noms libres

On considère les règles de CCS pour les communications internes suivantes :

$$\frac{-}{\overline{a}v.P \xrightarrow{\overline{a}v} P} \qquad \frac{-}{a(x).P \xrightarrow{av} P\{v/x\}} \qquad \frac{P \xrightarrow{\overline{a}v} P' \quad Q \xrightarrow{av} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

Dans la deuxième règle, la variable x est substituée à une valeur v, en déduisant une action de la form $a(x).P$, la règle admet une substitution à n'importe quelle valeur et donc l'agent $a(x).P$ peut se combiner avec n'importe quelle transition de production dans la règle de communication. MILNER appel ce type de substitution "substitution précoce", puisque les variables sont substituer au moment de la déduction de la transition d'entrée.

Dans le π-calcul MILNER adapte un autre schéma de substitution appelé substitution tardive[35], où les actions d'entrée contiennent les objets liés qui deviennent substituer seulement en déduisant une communication interne. La raison est que cela admettra une notion d'équivalence pour laquelle la théorie algébrique apparaît quelque peu plus simple [35]. Le schéma d'instantiation tardive, dans le π-calcul est représenté par les règles OUTPUT-ACT, INPUT-ACT et COM dans le tableau 2.5.

Cependant, des objets liés nécessitent un traitement prudent : un objet lié est essentiellement une référence aux emplacements dans un agent, et il est important que telles références soient maintenues dans toutes les règles d'actions. La règle problématique dans cet égard est une des règles de CCS pour la composition parallèle :

TAU-ACT : $\dfrac{-}{\tau.P \xrightarrow{\tau} P}$	OUTPUT-ACT : $\dfrac{-}{\overline{x}y.P \xrightarrow{\overline{x}y} P}$
INPUT-ACT : $\dfrac{-}{x(z).P \xrightarrow{x(w)} P\{w/z\}}$ $w \notin \text{fn}((z)P)$	
SUM : $\dfrac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$	MATCH : $\dfrac{P \xrightarrow{\alpha} P'}{[x=x]P \xrightarrow{\alpha} P'}$
IDE : $\dfrac{P\{\widetilde{y}/\widetilde{x}\} \xrightarrow{\alpha} P'}{A(\widetilde{y}) \xrightarrow{\alpha} P'}$ $A(\widetilde{x}) \stackrel{def}{=} P$	
PAR : $\dfrac{P \xrightarrow{\alpha} P'}{P\|Q \xrightarrow{\alpha} P'\|Q}$ $\text{bn}(\alpha)\cap\text{fn}(Q) = \phi$	
COM : $\dfrac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P\|Q \xrightarrow{\tau} P'\|Q'\{y/z\}}$	CLOSE : $\dfrac{P \xrightarrow{\overline{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P\|Q \xrightarrow{\tau} (vw)(P'\|Q')}$
RES : $\dfrac{P \xrightarrow{\alpha} P'}{(vy)P \xrightarrow{\alpha} (vy)P'}$ $y \notin \text{n}(\alpha)$	OPEN : $\dfrac{P \xrightarrow{\overline{x}y} P'}{(vy)P \xrightarrow{\overline{x}(y)} P'\{w/y\}}$ $y \neq x$ $w \notin \text{fn}((y)P')$

TAB. 2.5 – Sémantique opérationnelle de pi-calcul

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad (1)$$

La règle correspondante dans π-calcul est PAR dans la tableau 2.5, la seule différence entre les deux règles est la condition $\text{bn}(\alpha)\cap\text{fn}(Q) = \phi$. Pour voir l'importance de cette condition, considèrent la transition $P \xrightarrow{x(z)} P'$. Ici z une référence aux emplacements dans P', l'intuition est que dans une communication suivante un nom sera reçu et substitué a z dans P'. Mais si z figure aussi libre dans Q, donc dans la conclusion de (1) l'objet lié z se référera aux emplacements complémentaires dans Q. La communication suivante substituera alors pas seulement le z dans P', mais aussi z qui est libre dans Q. Par exemple, de INPUT-ACT, (1) et COM nous pouvons tirer la transition incorrecte suivante :

$$(x(z).P|Q) \mid \overline{x}y.R \xrightarrow{\tau} (P|Q)\{y/z\} \mid R \qquad (2)$$

Cette transition est incorrecte, puisque le nom libre z dans Q n'est pas le même nom lié z dans $x(z).P$. Pour cette raison la syntaxe de calcul exige dans la règle PAR la condition $\text{bn}(\alpha)\cap\text{fn}(Q) = \phi$,c'est à dire que (1) peut être appliqué quand un nom lié dans a ne figure pas libre dans Q. Cela explique aussi pourquoi INPUT-ACT ne peut pas être simplifiée au suivant :

$$\frac{-}{x(z).P \xrightarrow{x(z)} P}$$

Avec cette règle la condition dans PAR empêcherait toutes les transitions d'entrée de la forme $x(z).P \mid \overline{z}y.Q$. Le changement de nom lié dans INPUT-ACT est inoffensif puisque des noms liés représentent des références aux places dans un agent. Clairement, si w ne figure pas libre dans P, donc w se réfère aux mêmes places dans $P\{w/z\}$ que se réfère z dans P. Au lieu de la communication incorrect (2) nous pouvons maintenant déduire :

$$(x(z).P|Q) \mid \overline{x}y.R \xrightarrow{\tau} (P\{w/z\}|Q)\{y/w\} \mid R$$

La condition de la règle INPUT-ACT assure que $w = z$ ou $w \notin \text{fn}(P)$, et la condition de la règle PAR assure $w \notin \text{fn}(P)$. Donc l'agent peut être simplifié à

$$(P\{y/z\}|Q)|R$$

Qui est le résultat attendu de la communication.

Extrusion de nom.

On va introduire cet aspect a travers un exemple :

Soit le processus $P \equiv \overline{y}x.P'$ et $Q \equiv y(z).Q'$ et un processus R, avec l'existence d'un canal x privé entre P et R, donc on peut avoir le graphique dans la figure 2.12 :

Le système complet est : $(x)(\overline{y}x.P'|R)|y(z).Q\prime$

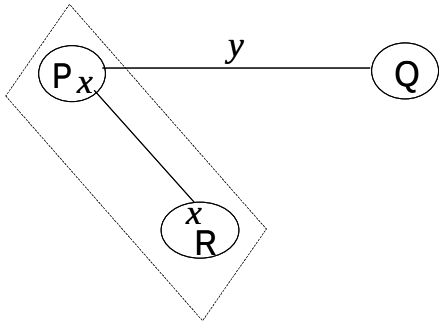


FIG. 2.12 – Extrusion de nom

A partir de cette configuration on peut avoir la transition suivante :

$$(x)(\overline{y}x.P'|R)|y(z).Q\prime \xrightarrow{\tau} (x)(P'|R|Q\prime\{x/z\})$$

Quand le nom ou canal x est exporté a Q sa portée est étendue. On dit que P exporte la portée du canal privé x. On peut schématiser le résultat de la transition par la figure 2.13 :

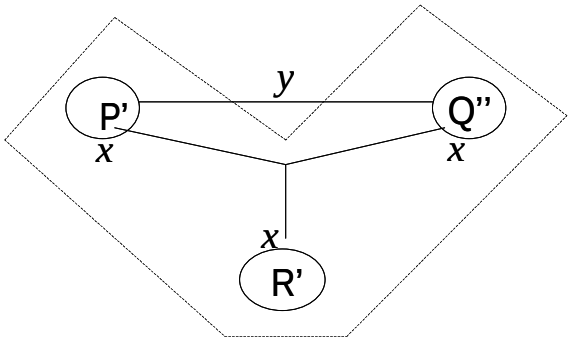


FIG. 2.13 – Exemple de portée étendue

2.2.4 Equivalences

bisimilarité forte et équivalence[35]

Définition 2.3 *(simulation, bisimulation, bisimilarité) : Une relation binaire $\mathcal{S}$ sur les agents est une simulation(forte) s'il satisfait le suivant :*

1. *Si $P \xrightarrow{a} P'$ et a action libre, alors $Q \xrightarrow{a} Q'$, et P' $\mathcal{S}$ Q'.*
2. *Si $P \xrightarrow{x(y)} P'$ et $y \notin n(P,Q)$, alors , $Q \xrightarrow{x(y)} Q'$ et pour tout w, $P'\{w/y\}\mathcal{S}Q'\{w/y\}$.*
3. *Si $P \xrightarrow{\overline{x}(y)} P'$ et $y \notin n(P,Q)$, alors , $Q \xrightarrow{\overline{x}(y)} Q'$ et $P'\mathcal{S}Q'$.*

La relation $\mathcal{S}$ est une bisimulation(forte) si $\mathcal{S}$ et son inverse sont des simulations. La relation $\dot{\sim}$, bisimilarité(forte), sur les agents est définie par $P \dot{\sim} Q$ si et seulement si existe

une bisimulation $\mathcal{S}$ tel que P $\mathcal{S}$ Q.

Bisimilarité tardive et bisimilarité précoce

Soit la bisimulation obtenue en commutant les quantificateurs dans la clause 2 dans la définition 2.3 de la simulation :

2'. Si $P \xrightarrow{x(y)} P'$ et $y \notin \mathrm{n}(P,Q)$, et pour tout w il y Q' telle que, $Q \xrightarrow{x(y)} Q'$ et, $P'\{w/y\}\mathcal{S}Q'\{w/y\}$.

Cette nouvelle équivalence est notée $\dot{\sim}'$. La relation $\dot{\sim}'$ est strictement plus faible que $\dot{\sim}$, plus d'agents sont équivalents quand la clause 2' est adoptée. La raison est que la clause 2 nécessite qu'il y ait celui simulant la transition d'entrée qui est équipotent pour toutes les instances de l'objet. Au contraire, la clause 2' nécessite seulement que pour chaque instance de l'objet existent une transition de simulation (et ces transitions de simulation peut être différentes pour des instances différentes). La relation $\dot{\sim}'$ est appelée bisimilarité précoce et $\dot{\sim}$ bisimilarité tardive.

Par exemple, considèrent les agents suivants

$$\begin{aligned} P &= x(u).R + x(u).0 \\ Q &= P + x(u).[u = z]R \end{aligned}$$

$P \dot{\sim}' Q$ toujours vérifiée mais $P \dot{\sim} Q$ ne soient pas vrai en général.

Lois algébriques pour la bisimilarité

Dans[35] les auteurs, développent une théorie de lois algébriques pour la relation $\dot{\sim}$. Toutes les preuves des théorèmes cités ici peuvent se retrouvés dans [35].

Théorème 1 :

(a) $\dot{\sim}$ c'est une relation d'équivalence.

(b) Si $P \dot{\sim} Q$ alors $a.P \dot{\sim} a.Q$ a action libre.
$P + R \dot{\sim} Q + R$,
$[x = y]P \dot{\sim} [x = y]Q$,
$P|R \dot{\sim} Q|R$,
$(vw)P \dot{\sim} (vw)Q$.

(c) Si pour tout $v \in \mathrm{fn}(P,Q,y)$, $P\{v/y\} \dot{\sim} Q\{v/y\}$ alors $x(y).P \dot{\sim} x(y).Q$.

Ce théorème établit que la bisimilarité est prèsque une congruence. Bien que $x(y).P \dot{\sim} x(y).Q$ n'est pas de la forme $P \dot{\sim} Q$, qu'il suit de la supposition plus forte que P et Q est bisimilaire pour tous les instances y.

Théorème 2 :

(a) $P + 0 \dot{\sim} P$

(b) $P + P \dot{\sim} P$

(c) $P_1 + P_2 \dot{\sim} P_2 + P_1$
(d) $P_1 + (P_2 + P_3) \dot{\sim} (P_1 + P_2) + P_3$

Théorème 3 : if $A(\widetilde{x}) \stackrel{def}{=} P$ alors $A(\widetilde{y}) \dot{\sim} P\{\widetilde{y}/\widetilde{x}\}$

Théorème 4 :

(a) $(vy)P \dot{\sim} P$ si $y \notin \text{fn}(P)$
(b) $(vy)(vz)P \dot{\sim} (vz)(vy)P$
(c) $(vy)(P + Q) \dot{\sim} (vy)P + (vy)Q$
(d) $(vy)a.P \dot{\sim} a.(vy)P$ si $y \notin \text{n}(a)$
(e) $(vy)a.P \dot{\sim} 0$ si y est le sujet de a

Dans (a) les restrictions peuvent être enlevées. Et dans (b) la restriction est commutative. (C) implique la distribution de la restriction sur l'addition. Notez que dans (d) et (e) est immédiatement applicable quand y est l'objet dans a. Si y est un objet lié, donc une a-conversion fera une application de (d). Mais si y est un objet libre ($a = \overline{x}y$) alors la restriction ne peut pas être propagée par l'opérateur préfixe.

Théorème 5 :

(a) $P|0 \dot{\sim} P$
(b) $P_1|P_2 \dot{\sim} P_2 + P_1$
(c) $(vy)P_1|P_2 \dot{\sim} (vy)(P_2 + P_1)$ si $y \notin \text{fn}(P_2)$
(d) $(P_1|P_2)|P_3 \dot{\sim} P_1|(P_2|P_3)$

(a), (b) et (d) affirment que 0 est une unité pour la composition parallèle et que la composition parallèle est commutative et associative. (c) est l'extension de portée : une restriction peut sans risque s'étendre sa portée à l'agent qui ne contient pas les occurrences libres du nom sur le quelle est faite la restriction.

Théorème 6 :

$$(y)(P_1|P_2) \dot{\sim} (y)P_1|(y)P_2 \quad \text{si} \quad y \notin \text{fn}(P_1) \cap \text{fn}(P_2)$$

2.2.5 Applications

Dans cette section, on présente quelques exemples[31][35] pour illustrer l'utilisation de π-calcul

Passage de liaison

Soit le graphique dans la figure 2.14 :

L'agent P est lié a l'agent R par le canal x, et désir envoyer ce canal à Q a travers le canal y. Q prét de le recevoir. Ainsi P peut être $\overline{y}x.P'$ et Q peut être $y(z).Q'$, dans ce cas la transition est :

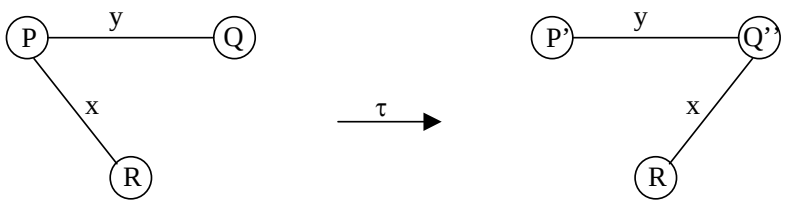


FIG. 2.14 – Exemple 1 : passage de liaison

$$\overline{y}x.P'|y(z).Q'|R \xrightarrow{\tau} P'|Q'\{x/z\}|R$$

Donc Q" dans la figure 2.14 est $Q'\{x/z\}$. Cette figure illustre le cas dans lequel $x \notin \text{fn}(Q)$, signifiant que Q ne possède aucune liaison x avant la transition. Mais la transition c'est la même si $x \in \text{fn}(Q)$, il n'y a aucune raison que Q ne doit pas recevoir une liaison qu'il possède déjà. La figure illustre aussi le cas dans lequel $x \notin \text{fn}(P')$, signifiant que P' n'a aucune x-liaison après la transition, mais de nouveau cette condition n'affecte pas la transition.

La situation n'est pas différente quand la liaison y entre P et Q est privée. Dans ce cas le graphe de flux approprié est dans la figure 2.15 :

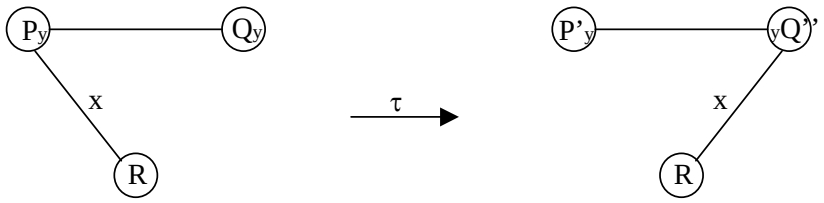


FIG. 2.15 – Exemple 2 : passage de liaison

La liaison y (privée) est représentée par une restriction, donc la transition est maintenant

$$(vy)(\overline{y}x.P'|y(z).Q')|R \xrightarrow{\tau} (vy)(P'|Q'\{x/z\})|R$$

Téléphone portable

La figure 2.16 représnte le graphe flux de l'application.

Le CENTRE est en contact permanent avec deux stations BASE, chacune dans une partie différente du pays. Un CAR avec un téléphone portable est aux mouvements dans le pays, il doit être toujours en contact avec une BASE. Si CAR devient plus loin de sa courante BASE de contact, alors une procédure de remise est introduite et en conséquence CAR abandonne le contact avec la courante BASE et met un nouveau contact avec une autre BASE.

Le graphe montre le système dans l'état où CAR est dans un contact avec BASE_1, il peut être écrit

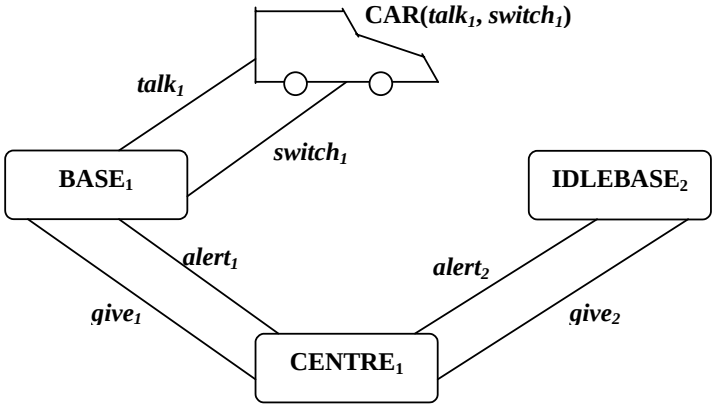


FIG. 2.16 – Graphe de l'application : Téléphone portable

$$\mathbf{SYSTEME_1} \stackrel{def}{=} (v\ talk_i, switch_i, give_i, alert_i\colon i = 1, 2)$$
$$(\mathbf{CAR}(talk_1, switch_1) \mid \mathbf{BASE_1} \mid \mathbf{IDLEBASE_2} \mid \mathbf{CENTRE_1})$$

En ce qui concerne les composants :

CAR est paramétrique sur le canal *talk* et le canal *switch*. Sur *talk* il peut parler à plusieurs reprises, mais a tout moment il peut recevoir le long de son canal *switch* deux nouveaux canaux qu'il doit alors commencer à les utiliser

$$\mathbf{CAR}(talk, switch) \stackrel{def}{=} talk.\mathbf{CAR}(talk_1, switch_1)$$
$$+\ switch(talk\prime switch\prime).\ \mathbf{CAR}(talk\prime, switch\prime)$$

La **BASE** peut communiquer à plusieurs reprises avec le **CAR**, mais à tout moment elle peut recevoir le long de son canal *give* deux nouveaux canaux qu'elle doit communiquer au **CAR**, et devenir ensuite inoccupé lui-même, nous définissons :

$$\mathbf{BASE}(t,s,g,a) = t.\ \mathbf{BASE}(t,s,g,a)$$
$$+\ g(t's').\overline{s}\ t's'.\ \mathbf{IDELBASE}(t,s,g,a)$$

D'autre part un **IDLEBASE**, peut être communiquer sur son canal alert pour devenir actif

$$\mathbf{IDLEBASE}(t, s, g, a) \stackrel{def}{=} a.\ \mathbf{BASE}(t, s, g, a)$$

Nous définissons l'abréviation

$$\mathbf{BASE_i} \stackrel{def}{=} \mathbf{BASE}(talki, switchi, givei, alerti) \qquad (i = 1, 2)$$

Et une abréviation similaire, ainsi, par exemple,

$$\mathbf{BASE}_i \equiv talk_i.\mathbf{BASE}_i + give_i(t\prime s\prime).\overline{switch_i}t\prime s\prime.\mathbf{IDLEBASE}_i$$

$$\mathbf{IDLEBASE}_i \equiv \ alert_i.\mathbf{BASE}_i$$

Finalement le **CENTRE**, qui sait initialement que **CAR** est en contact avec $\mathbf{BASE_1}$, peut décider de transmettre les canaux $talk_2, switch_2$ au **CAR** via **BASE** et alerter $\mathbf{BASE_2}$ de ce fait, donc nous définissons

$$\mathbf{CENTRE}_2 \stackrel{def}{=} \overline{give}_1 talk_1 switch_1.alert_1.\mathbf{CENTRE}_1$$
$$\mathbf{CENTRE}_1 \stackrel{def}{=} \overline{give}_2 talk_2 switch_2.alert_2.\mathbf{CENTRE}_2$$

2.3 Join-Calcul

L'objectif de cette section est d'étudier un langage de programmation élémentaire nommé **Join-calcul**[9][13], dans lequel chaque étape du calcul correspond naturellement à l'envoi de plus un message entre machine. Un calcul est un langage réduit à sa plus simple expression afin de modélisé précisément certains aspects du langage, en faisant abstraction de tout le reste. Le join-calcul est un modèle de spécification formelle des applications asynchrones, distribuées, avec la notion de mobilité.

Le join-calcul s'inspire largement de deux modèles bien connus du parallélisme, les calculs des processus comme CCS ou le $\pi-$calcul d'une part, et la machine chimique abstraite de Berry et Boudol d'autre part[5]. Il correspond au noyau d'un langage de programmation directement utilisable, en particulier, il est implémenté de manière répartie. Il préserve la plupart des propriétés formelles du π-calcul[13].

2.3.1 Machine chimique abstraite

Dans [5][4] Berry et Boudol proposent un modèle nomé : machine abstraite chimique (CHAM), la sémantique chimique utilise un ensemble de règles chimiques, sur des multi-ensembles de molécules(termes). Nous employons les règles chimiques qui fonctionnent sur les multi-ensemble des termes (solutions chimiques) :

- – Les règles structurelles ($\rightleftharpoons$) sont réversibles ; ils représentent des réarrangements syntaxtiques de termes dans la solution.
  – Les règles de réduction ($\rightarrow$) consomment quelques termes spécifiques dans la solution chimique, les remplaçant par d'autres termes, ils correspondent aux pas de calcul de base.

Pour illustrer l'approche chimique, nous prenons l'exemple de passage d'une valeur suivant :

$$\textbf{Str-join} \quad P|P' \rightleftharpoons P, P'$$

$$\textbf{Red} \quad \overline{x} < \widetilde{v} > |x < \widetilde{y} > .P \rightarrow P\{\widetilde{v}/\widetilde{y}\}$$

Str-join : chaque molécule de la forme $P|P'$ peut se partitioner en deux petite molécules P et P'. Et inversement chaque paire de molécules P et P' peut devenir une seule molécule $P|P'$.

Red : c'est une règle de réduction qui consomme la seule molécule qui contient deux processus en composition parallèle, et elle produit une seule molécule.

Par exemple on a les étapes chimique suivante

$$\begin{aligned}\{\overline{x} < 1 > |P|x < u > .Q\} &\rightharpoonup\!\rightharpoonup \{\overline{x} < 1 >, P, x < u > .Q\} \\ &\leftharpoonup \{P, \overline{x} < 1 > |x < u > .Q\} \\ &\rightarrow \{P, Q\{1/u\}\} \\ &\rightharpoonup \{P|Q\{1/u\}\}\end{aligned}$$

2.3.2 La machine chimique réflexive (RCHAM)

Présentation

Les seules valeurs dans le join-calcul sont les noms de canaux, ou adresses. De ce point de vue, le join-calcul est une variante de π-calcul, dont la mobilité de nom.

Le modèle opère sur deux multi-ensemble ($\mathcal{D} \vdash \mathcal{P}$). La molécule $\mathcal{P}$ représente des processus en cours d'exécution parallèle, $\mathcal{D}$ c'est l'ensemble des règles réactives.

Soit une version simplifiée d'un serveur d'impression sur un réseau local. Chaque station de travail sur le réseau peut envoyer des requêtes d'impression au serveur. D'autre part, chaque imprimante peut signaler sa présence au serveur, qui répartit la charge entre les imprimantes disponibles. Nous modélisons l'interface du serveur par deux noms, *imprimer* pour les requêtes d'impression, et *accepter* pour les imprimantes disponibles. Ainsi le processus

$$imprimer < 1 > |imprimer < 2 > |accepter < laser >$$

Se compose de trois message, et décrit un état du système où deux impressions des fichiers 1 et 2 sont en attente, tandis qu'une imprimante *laser* est disponible. L'ordre des messages est sans importance, formellement la composition parallèle est associative et commutative.

Une règle de réaction notée D ou $J \rhd P$, consomme un ensemble de messages de la forme décrite dans le filtre J, et déclenche l'exécution d'une copie du processus P dans lequel les paramètres formels de J sont remplacés par les arguments transmis dans ces messages. La même règle peut être utilisée a plusieurs reprises, tant qu'il y a des messages à consommer.

Par exemple nous définissons le comportement du serveur par une règle de réaction notée D, qui décrit comment les messages envoyés sur les noms *imprimer* et *accepter* sont traités.

$$D \stackrel{def}{=} accepter < imprimante > | imprimer < fichier > \triangleright\, imprimante < fichier >$$

Cette règle consomme deux messages, l'un sur *accepter*, l'autre sur *imprimer*, et déclenche l'impression en envoyant le fichier à l'imprimante. Nous pouvons regrouper la définition du serveur d'impression est sont état courant dans un seul processus.

$$P = \textbf{def}\ D \textbf{in}\ imprimer < 1 > | imprimer < 2 > | accepter < laser >$$

Suivant notre explication informelle de la règle D, le message $accepter < laser >$ et en particulier le choix entre 1 et 2 n'est pas déterminé avant la réduction.

$D \vdash accepter < laser > | imprimer < 1 >, imprimer < 2 >$

$\rightarrow D \vdash laser < 1 >, imprimer < 2 >$

Le modèle est réflexif, cela signifie que les réactions peuvent être se créer dynamiquement.

On peut aussi écrire

$\vdash \textbf{def}\ D\ \textbf{in}\ accepter < laser > | imprimer < 1 > | imprimer < 2 >$

$\quad \rightharpoonup D \vdash accepter < laser > | imprimer < 1 > | imprimer < 2 >$

$\quad \rightharpoonup\rightharpoonup D \vdash accepter < laser >, imprimer < 1 >, imprimer < 2 >$

Syntaxe

Nous supposons un ensemble infini ($\mathcal{N}$) de noms du ports (les ports sont appelés aussi des canaux). Nous utilisons des variables minuscules x, y, foo,... pour les éléments de noms de $\mathcal{N}$.

Il y a trois sortes d'expressions, des processus, des définitions et des join-patterns(filtres), (voir figure 2.17).

Un processus P peut être soit l'envoi asynchrone d'un message $x < v_1, ..., v_n >$, soit la définition local de nouveaux noms **def** D **in** P, soit une composition parallèle de processus $P|Q$, ou le processus inerte **0**. Une définition D peut être soit la définition vide $\top$, soit une composition de définition D, D' connectées par l'opérateur $\wedge$, soit une règle de réaction $J \triangleright P$, consommant les messages correspondant au filtre J pour exécuter P. Les filtres J modélisent la réception et la synchronisation de messages : un filtre de message $x < \widetilde{y} >$ attend qu'un message $\widetilde{y}$ sur le canal x soit présent, le filtre $J|J'$ attend que les filtres J et J' soient satisfaits pour être satisfait.

Les noms de ports définis sont récursivement liés dans toute la définition def D in P, c'est-à-dire dans le processus principal P et dans les processus gardés sous la définition D. On note les variables reçues rv[J], variables définies dv [J] et dv [D], et variables libre fv[D] et fv[P] sont formellement définies pour le calcul dans la figure 2.18.

$P, Q, R ::=$		**Processus**
	$x < v_1, ..., v_n >$	*Message asynchrone*
	$\mid$ **def** D **in** P	*Définition local*
	$\mid\ P \mid Q$	*Composition parallèle*
	$\mid$ **0**	*Processus inerte*
$D ::=$		**Définition**
	$J \rhd P$	*Règle réactive*
	$\mid\ D \wedge D\prime$	*Composition*
	$\mid\ \mathsf{T}$	*Définition vide*
$J ::=$		**Join-patterns**
	$x < y_1, ..., y_n >$	*Patterns message*
	$\mid\ J \mid J\prime$	*Pattern de join*

FIG. 2.17 – *Syntaxe de Join-calcul*

$$
\begin{aligned}
\mathrm{fv}[x < v_1, ..., v_n >] &\stackrel{def}{=} \{x, v_1, ..., v_n\} \\
\mathrm{fv}[\textbf{def } D \textbf{ in } P] &\stackrel{def}{=} (\mathrm{fv}[P] \cup \mathrm{fv}[D]) \backslash \mathrm{dv}[D] \\
\mathrm{fv}[P|P'] &\stackrel{def}{=} \mathrm{fv}[P] \cup \mathrm{fv}[P'] \\
\mathrm{fv}[\mathbf{0}] &\stackrel{def}{=} \phi \\
\\
\mathrm{fv}[J \rhd P] &\stackrel{def}{=} \mathrm{dv}[J] \cup (\mathrm{fv}[P] \backslash \mathrm{rv}[J]) \\
\mathrm{fv}[D \wedge D\prime] &\stackrel{def}{=} \mathrm{fv}[D] \cup \mathrm{fv}[D\prime] \\
\mathrm{fv}[T] &\stackrel{def}{=} \phi \\
\\
\mathrm{dv}[J \rhd P] &\stackrel{def}{=} \mathrm{dv}[J] \\
\mathrm{dv}[D \wedge D'] &\stackrel{def}{=} \mathrm{dv}[D] \cup \mathrm{dv}[D'] \\
\mathrm{dv}[T] &\stackrel{def}{=} \phi
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{dv}[x < y_1, ..., y_n >] &\stackrel{def}{=} \{x\} & \qquad \mathrm{dv}[J|J'] &\stackrel{def}{=} \mathrm{dv}[J] \uplus \mathrm{dv}[J'] \\
\mathrm{rv}[x < y_1, ..., y_n >] &\stackrel{def}{=} \{y_1, ..., y_n\} & \qquad \mathrm{rv}[J|J'] &\stackrel{def}{=} \mathrm{rv}[J] \uplus \mathrm{rv}[J']
\end{aligned}
$$

FIG. 2.18 – *Les portées dans Join-calcul*

str-join	$\vdash P_1 \mid P_2$	$\rightleftharpoons$	$\vdash P_1, P_2$
str-null	$\vdash 0$	$\rightleftharpoons$	$\vdash$
str-and	$D_1 \wedge D_2 \vdash$	$\rightleftharpoons$	$D_1, D_2 \vdash$
str-nodef	$T \vdash$	$\rightleftharpoons$	$\vdash$
str-def	$\vdash$**def** D **in** P	$\rightleftharpoons$	$D\sigma_{dv} \vdash P\sigma_{dv}$

Red	$J \rhd P \vdash J\sigma_{rv}$	$\rightarrow$	$J \rhd P \vdash P\sigma_{rv}$

Condition pour la substitution

str-def σ_{dv} instanciation des variables de port dv[D] à des noms distincts, frais : $dom(\sigma_{dv}) \cap \text{fv}[\mathcal{S}] = \phi$ où $\mathcal{S}$ est la solution initiale,

red σ_{rv} Substitue les noms transmis aux variables distinctes reçues rv [J]

FIG. 2.19 – *La machine chimique réflexive*

Sémantique opérationnelle

Solution réflexive : la solution chimique comporte deux parties de molécules de multi-ensemble ($\mathcal{D} \vdash \mathcal{P}$). La molécule $\mathcal{P}$ représente des processus en cours d'exécution parallèle, $\mathcal{D}$ c'est l'ensemble des règles réactives. La définition active D représente des règles réactives définies des réductions possibles pour les processus, les processus P, représentent l'état du calcul, il peut introduire de nouveaux noms et règles de réactions, ce qui signifie la machine réflexive, les règles chimiques pour la RCHAM sont définies dans la figure 2.19, page 36.

La sémantique donnée dans la figure 2.19, c'est une collection de règles chimique applicables sur un fragment de solution réflexive.

Les quatre premières règles structurelles avec les deux opérateurs | et $\wedge$, sont associative et commutative, avec des unités 0 et $\top$. La réduction simple **Red** illustre l'utilisation des règles active $J \rhd P$ apparue dans la partie gauche de la solution chimique. La règle réaction $J \rhd P$ est inchangeable a gauche, et peut être employée plus tard pour effectuer des nouveaux pas de réduction.

Exemple 2.1 *le processus le plus simple est de la forme $x < y >$; il envoi un nom y sur un autre nom x. Dans l'exemple, nous supposons l'existence de valeurs de base, comme des entiers, des chaines,...etc. Par exemple, supposons un service d'impression a été défini sur le nom print, nous écrivions print $< 3 >$. Le programme prends la forme :*

$$\textbf{def} \quad print < x > \rhd ... \textbf{in} \quad print < 3 >$$

Pour imprimer plusieurs entiers dans l'ordre, nous aurions besoin d'une imprimante pour envoyer les messages sur l'achèvement. Pour cela, l'imprimante doit retourner un canal k.

$$\textbf{def} \quad print < x, k > \triangleright ... k <> ... \textbf{in def } k <> \triangleright print < 4, k' > \textbf{ in} \quad print < 3, k >$$

2.3.3 Equivalences

L'objectif principal est d'employer ces équivalences pour rapprocher des programmes distribués écrits dans le join-calcul ou dans ses extensions, ces équivalences permettent d'énoncer des relations intéressantes et de les prouver. Ainsi une bonne équivalence doit être facile à interpréter, ne pas identifier de processus évidemment différents, ni séparer de processus intuitivement équivalents dans un environnement réparti, et si possible offrir des techniques de preuves suffisantes pour établir ces équations.

Deux programmes P et Q sont équivalents si seulement si ont exactement les mêmes propriétés ; en particulier si c'est toujours possible de remplacer un par l'autre dans le système, sans affecté le comportement du système. Inversement, si P et Q ne sont pas équivalents, si c'est toujours possible de remplacer l'un par l'autre dans le système avec un changement de comportement.

Sémantique de réduction

Dans cette section, nous rappelons des notions standard dans la sémantique basée sur la réduction.

Système de réduction abstrait

Définition 2.4 *un système de réduction abstrait (ARS) est un triple ($\mathcal{P}$, $\rightarrow$, $\downarrow_x$), où $\mathcal{P}$ un ensemble de termes, $\rightarrow \subseteq \mathcal{P} \times \mathcal{P}$ une relation sur les termes, et $\downarrow_x$ une famille de prédicats sur les termes. Ces prédicats d'observation $\downarrow_x$ sont destinés à détecter le résultat du calcul (succès, la convergence, ou le deadlock).*

Soit $P, Q \in \mathcal{P}$ deux processus ; P est équivalent a Q s'ils passent le même teste ($\forall x.P \downarrow_x$ ssi $Q \downarrow_x$), P est plus petit que Q si P passe moins de testes que Q ($\forall x.P \downarrow_x$ implique $Q \downarrow_x$).

Définition 2.5 *Le prédicat d'observation $\downarrow_x$ est appelé aussi barbe forte sur x, est destiné à détecter si un processus envoi sur un nom libre x*

$$P \downarrow_x \stackrel{def}{=} \exists P', P \equiv P' | x <> .$$

Barbes (barbs) sont des messages en forme simple c'est à dire ils portent rien, donc des signaux.

Sémantique faible On dit qu'une sémantique est faible, si elle est définie sur des termes qui ont des séquences de réduction $\rightarrow^*$ au lieu de la seule réduction $\rightarrow$.

Définition 2.6 *le prédicat $\Downarrow_x$ - connu comme barbe faible sur x- détecte si un processus peut satisfaire le prédicat d'observation de base $P \downarrow_x$, probablement après l'exécution d'une séquence de réductions internes.*

$$P \Downarrow_x \stackrel{def}{=} \exists P', P \rightarrow^* P' \downarrow_x$$

En général, des barbes faibles peuvent apparaître comme le résultat de quelques pas de calculs. Par exemple, le processus $x <> \oplus y <> \oplus 0$ a deux barbes $\Downarrow_x$ et $\Downarrow_y$

Contexte d'évaluation : c'est simplement un processus join-calcul avec un trou $[.]s$. Si P un processus quelconque, $C[P]s$ c'est le processus obtenu par le remplacement de trou dans $C[.]s$ par P.

Définition 2.7 *soit P un processus join-calcul et x un nom de channel.*

$P \downarrow_x$ ssi $P = C[x < \widetilde{v} >]$ pour le tuple $\widetilde{v}$ de noms et le contexte d'évaluation $C[.]s$ Cela ne capture pas x , tel que $x \notin \mathcal{S}$.

$$P \Downarrow_x \text{ ssi } P \rightarrow^* P' \text{ tel que } P' \downarrow_x$$

Equivalence de teste équitable (Fair testing)

Définition 2.8 *le prédicat fair-must $\Downarrow_{\Box x}$détecte si un processus conserve toujours la possibilité d'envoyer sur x :*

$$P \Downarrow_{\Box x} \stackrel{def}{=} \forall P', \text{si } \ P \rightarrow^* P', \text{alors } \exists P'', P' \rightarrow^* P'' \downarrow x$$

Définition 2.9 *le préodre de teste équitable $\subseteq_{fair}$ et l'équivalence de teste équitable $\approx_{fair}$ sont la plus grande précongruence et congruence, respectivement, qui préserve le prédicat de teste équitable $\Downarrow_{\Box x}$:*

$P \subseteq_{fair} Q \stackrel{def}{=} \forall C \in \varepsilon, x \in \mathcal{N}$, $C[P]s \Downarrow_{\Box x}$ implique $C[Q]s \Downarrow_{\Box x}$.

$P \approx_{fair} Q \stackrel{def}{=} \forall C \in \varepsilon, x \in \mathcal{N}$, $C[P]s \Downarrow_{\Box x}$ ssi $C[Q]s \Downarrow_{\Box x}$.

Congruence barbue

La notion de bisimulation est devenue une sémantique standard dans la théorie de concurrence.

Définition 2.10 *(Simulation, bisimulation) : soit $(\mathcal{P}, \xrightarrow{a})$ un STE, et $\mathcal{R}$ une relation sur les processus. $\mathcal{R}$ est une simulation forte, si pour tout les processus P, Q, pour chaque étiquette a, si $P \xrightarrow{a} P'$, alors pour Q' on a $Q \xrightarrow{a} Q'$, et P' $\mathcal{R}Q'$. $\mathcal{R}$ c'est une bisimulation forte si $\mathcal{R}$ et $\mathcal{R}^{-1}$ sont des simulations fortes.*

Définition 2.11 *(bisimilarité barbue) : soit $(\mathcal{P}, \rightarrow, \downarrow_x)$, un ARS et $\mathcal{R}$ une relation sur les processus. $\mathcal{R}$ est une simulation barbue si c'est une simulation (basée sur réduction) qui raffine les barbes : pour tout $\Downarrow_x$, si P $\mathcal{R}$ Q et $P \Downarrow_x$, alors $Q \Downarrow_x$. $\mathcal{R}$ est bisimulation barbue si $\mathcal{R}$ et $\mathcal{R}^{-1}$ sont des simulations barbues.*

Bisimilarité barbue est la plus grande bisimulation barbue.

Comme l'équivalence à réduction, bisimilarité barbue peut être utilisée pour définir des équivalences comportementales sur des processus.

Simulations couplées

Définition 2.12 *une paire de relations $\leq, \geq$ sont une simulation couplée si $\leq$ et $\geq^{-1}$ sont deux simulation qui satisfait le diagramme d'accouplement dans la figure 2.20 :*

FIG. 2.20 – Diagramme d'accouplement 1

Une relation $\mathcal{R}$ est une équivalence couplée et barbue s'il y a une paire de simulations couplées $(\leq, \geq)$ qui préserve les barbes $\Downarrow_x$ et tel que $\mathcal{R} = \leq \cap \geq$. L'union de toutes les équivalences barbues couplées est appelée la similitude barbue couplée et est notée π ; c'est aussi l'équivalence couplée obtenue des plus grandes simulations de couples. Dans le cas où $\leq = \geq$, les diagrammes d'accouplement sont toujours vérifiés.

Par exemple on a le diagramme de la figure 2.21 :

Où les relations pointillées soulignent l'exigence d'accouplement sur la simulation entre $P \oplus Q$ et $P \oplus (Q \oplus R)$.

2.3.4 Le calcul ouvert

Les équivalences définies dans la section précédente, fournissent une sémantique du join-calcul peu explicite. En effet, l'interaction entre un processus et son environnement n'est pas apparente ; elle est révélée par l'application de contextes particuliers, suivie de réductions internes. Pour obtenir des techniques de preuves plus directes, en particulier pour éviter

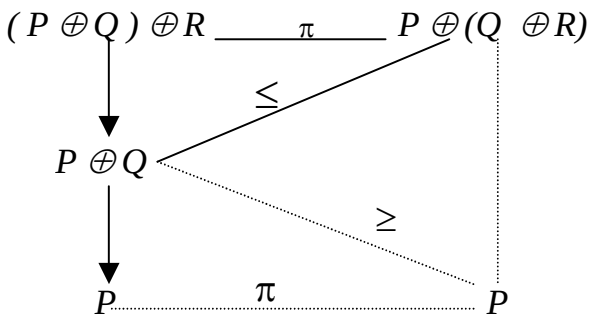


FIG. 2.21 – Diagramme d'accouplement 2

la quantification sur tout les contextes dans la définition des équivalences, la sémantique a été raffiner par l'ajout des interactions primitives avec l'environnement. Cela permet de générer des transitions étiquetées, dont l'enchaînement décrit entièrement le comportement observable des processus.

L'interaction entre un processus et son environnement se décompose en deux actions complémentaires : un processus peut émettre des messages vers le contexte, ce qui fournit au contexte de nouveaux noms, et réciproquement le contexte peut utiliser les noms qu'il a reçu pour émettre des messages vers le processus.

Syntaxe ouverte

Le processus ouvert $A, B, ... \in \mathcal{A}$ est défini par la grammaire suivante, les occurrences D, P, J sont les mêmes que dans la syntaxe de join-calcul définie précédemment.

$A ::=$		**Processus ouvert**
	$x < v_1, ..., v_n >$	message
	$\mid \mathbf{def}_s\ D\ \mathbf{in}\ P$	Définition ouverte
	$\mid A \mid A'$	Composition parallèle
	$\mid \mathbf{0}$	Processus nul

Le processus ouvert A, c'est comme un processus régulier dans join-calcul, à l'exception que sa définition dans un contexte contient des noms exportés(extruded), la définition ouverte $\mathbf{def}_s\ D\ \mathbf{in}\ P$ expose le sous-ensembles S des variables définis par D et visible dans l'environnement.

L'interface d'un processus ouvert A comporte deux ensembles de noms disjoints, noms libre fv$[A]$ utilisés dans les messages sortant, et les noms exportés xv$[A]$, utilisées par l'environnement pour envoyer les messages entrant. Ces noms servent de supports exclusifs pour l'interaction avec l'environnement.

A partir de cette sémantique ouverte, nous appliquons la définition habituelle de la bisimulation étiquetée faible : deux processus sont bisimilaires si lorsque l'un d'eux produit

une transition, l'autre peut produire la même transition éventuellement suivie ou précédée de réduction internes, et que les deux processus résultants restent bisimilaires.

Définition 2.13 *les noms dans les termes sont : soit des noms reçus, noms défini-local, noms exportés, et les noms libres.*

Chimie ouverte On considère le processus : $\mathbf{def}_\phi \;\; x <>\triangleright\, y <> \mathbf{in}\; z < x >$, l'interface ne contient pas des noms exportés, et contient deux noms libres y, z. Le message $z < x >$ peut être consommé par l'environnement, en exportant x :

Cela donne la transition

$$\mathbf{def}_\phi \;\; x <>\triangleright\, y <> \mathbf{in}\; z < x > \overset{\{x\}\overline{z}<x>}{\rightarrow} \mathbf{def}_{\{x\}}\; x <>\triangleright\, y <> \mathbf{in}\; \mathbf{0}$$

Donc x devient connue dans l'environnement, elle est plus local, c.a.d l'environnement peut émettre sur x, mais l'environnement ne peut pas modifier ou étendre sa définition. La transition d'intrusion est activée :

$$\mathbf{def}_{\{x\}}\; x <>\; y <> \mathbf{in}\; \mathbf{0} \overset{x<>}{\rightarrow} \mathbf{def}_{\{x\}}\; x <>\triangleright\, y <> \mathbf{in}\; x <>$$

Maintenant le processus résultat peut introduire quelques messages sur x, et indépendamment il peut faire les transitions suivantes :

$$\mathbf{def}_{\{x\}}\; x <>\triangleright\, y <> \mathbf{in}\; x <> \;\rightarrow \mathbf{def}_{\{x\}}\; x <>\triangleright\, y <> \mathbf{in}\; y <>$$
$$\overset{\{\}y<>}{\rightarrow} \mathbf{def}_{\{x\}}\; x <>\; y <> \mathbf{in}\; \mathbf{0}$$

Les noms visibles sont les noms libres fv[A] et les noms exportés xv[A]. les noms locaux(noms reçus rv[J] et les noms définis dv[J]) sont liés dans join-patters,

Les règles de portée sont donner dans la figure 2.22, page 42 :

Définition 2.14 *Une solution chimique ouverte $\mathcal{S}$ c'est une triple ($\mathcal{D}$, $\mathcal{S}$, $\mathcal{A}$), notée $\mathcal{D} \vdash_{\mathcal{S}} \mathcal{A}$, où :*

- – *$\mathcal{A}$ c'est un multi-ensemble de processus ouvert avec des ensembles de noms exportés disjoints*
  – *$\mathcal{D}$ c'est un multi-ensemble de définitions tel que dv[$\mathcal{D}$]$\cap$xv[A]=ϕ*
  – *$\mathcal{S}$ c'est un sous-ensemble de dv[$\mathcal{D}$]*

Les règles chimiques pour le calcul ouvert sont présentées dans la figure 2.23, page 43.

Les règles chimiques pour RCHAM ouverte, donnée dans la figure 2.23 ; définie une famille de transitions entre la solution ouverte $\rightharpoonup$,$\leftharpoondown$, $\rightarrow$,$\overset{a}{\rightarrow}$; a représente les étiquettes $x < v_1, ..., v_p >$ et $S\overline{x} < v_1, ..., v_p >$ pour tout les sous-ensembles S de $\{v_1, ..., v_p\}$.

Les six premières règles sont les règles de RCHAM : une petite différence c'est l'utilisation des noms exportés, la règle **Red** inchangeable.

Les deux dernière règles : interaction avec le contexte. La règle d'extrusion **Ext** consomme les messages envoyés sur les noms libres, ces messages peuvent contenir des noms définis inconnus à l'environnement. La règle d'intrusion **Int** autorise seulement l'intrusion de messages sur les noms exportés.

$$
\begin{array}{rcl}
\mathrm{xv}[x < v_1, ..., v_n >] & \stackrel{def}{=} & \phi \\
\mathrm{xv}[\mathbf{def}_S\ D\ \mathbf{in}\ A] & \stackrel{def}{=} & S \uplus \mathrm{xv}[A] \\
\mathrm{xv}[A|A'] & \stackrel{def}{=} & \mathrm{xv}[A] \uplus xv[A'] \\
\mathrm{xv}[\mathbf{0}] & \stackrel{def}{=} & \phi \\
 & & \\
\mathrm{fv}[x < v_1, ..., v_n >] & \stackrel{def}{=} & \{x, v_1, ..., v_n\} \\
\mathrm{fv}[\mathbf{def}_S\ D\ \mathbf{in}\ A] & \stackrel{def}{=} & (\mathrm{fv}[A] \cup (\mathrm{fv}[D] \backslash \mathrm{xv}[A])) \backslash \mathrm{dv}[D] \\
\mathrm{fv}[A|A'] & \stackrel{def}{=} & (\mathrm{fv}[A] \backslash \mathrm{xv}[A']) \cup (\mathrm{fv}[A'] \backslash \mathrm{xv}[A]) \\
\mathrm{fv}[\mathbf{0}] & \stackrel{def}{=} & \phi \\
 & & \\
\mathrm{fv}[J \rhd P] & \stackrel{def}{=} & \mathrm{dv}[J] \cup (\mathrm{fv}[P] \backslash \mathrm{rv}[J]) \\
\mathrm{fv}[D \wedge D'] & \stackrel{def}{=} & \mathrm{fv}[D] \cup \mathrm{fv}[D'] \\
\mathrm{fv}[\mathsf{T}] & \stackrel{def}{=} & \phi \\
 & & \\
\mathrm{dv}[J \rhd P] & \stackrel{def}{=} & \mathrm{dv}[J] \\
\mathrm{dv}[D \wedge D'] & \stackrel{def}{=} & \mathrm{dv}[D] \cup \mathrm{dv}[D'] \\
\mathrm{dv}[\mathsf{T}] & \stackrel{def}{=} & \phi \\
\mathrm{rv}[J],\ \mathrm{dv}[J], \text{et } \mathrm{dv}[D] \text{ sont définis précédemment ; } \mathrm{fv}[P] \stackrel{def}{=} \mathrm{fv}[A] & &
\end{array}
$$

FIG. 2.22 – *Les portées dans join-calcul ouvert*

2.3.5 Localité, migration et pannes

Calcul avec location

Le join-calcul est un modèle élémentaire adapté à la programmation répartie, mais la location des processus et des définitions est implicite. Plus précisément, le modèle est implémenter dans un environnement à plusieurs machines, quelle que soit la répartition des ressources à l'exécution, a condition que toutes les machines participantes puissent s'échanger des messages asynchrones.

Dans un programme distribué, il est parfois nécessaire de contrôler la localité, en particulier parce que celle-ci détermine la résistance aux pannes. Ainsi, chaque groupe de processus et de définition peut migrer d'une machine à l'autre, ou s'arrêter. En termes de langage de programmation, cela correspond à un modèle très expressif d'agents mobiles.

Dans join-calcul la répartition des ressources est organisée par emplacements. Intuitivement, un emplacement réside sur un site particulier, et peut contenir des processus et des règles de réductions. Un emplacement peut se déplacer d'un site à l'autre, ainsi un emplacement peut représenter un agent mobile. Un emplacement peut également contenir des

str-join	$\vdash_S A\|B$	$\rightleftharpoons$	$\vdash_S A, B$
str-null	$\vdash_S 0$	$\rightleftharpoons$	$\vdash_S$
str-and	$D \wedge D' \vdash$	$\rightleftharpoons$	$D, D' \vdash_S$
str-nodef	$\intercal \vdash_S$	$\rightleftharpoons$	$\vdash_S$
str-def	$\vdash_S \mathbf{def}_{S'}\ D\ \mathbf{in}\ A$	$\rightleftharpoons$	$D\sigma \vdash_{S \uplus S'} A\sigma$

Red	$J \rhd P \vdash_S J\sigma$	$\rightarrow$	$J \rhd P \vdash_S P\sigma$
Ext	$\vdash_S x < v_1, ..., v_p >$	$\overset{S'\overline{x}<v_1,...,v_p>}{\rightarrow}$	$\vdash_{S \uplus S'}$
Int	$\vdash_{S \cup \{x\}}$	$\overset{x<v_1,...,v_p>}{\rightarrow}$	$\vdash_{S \uplus \{x\}} x < v_1, , v_p >$

Condition sur la solution $\mathcal{S} = \mathcal{D} \vdash_S \mathcal{A}$

str-def	σ remplace $\mathrm{dv}[D] \backslash S\prime$ avec des noms distinct frais.
Red	ρ noms de substitution pour $\mathrm{rv}[J]$
Ext	$x \in \mathrm{fv}[\mathcal{S}]$, $S\prime = \{v_1, ..., v_p\} \cap (\mathrm{dv}[\mathcal{D}] \setminus \mathrm{S})$
Int	$\{v_1, ..., v_p\} \cap \mathrm{dv}[\mathcal{D}] \subseteq S$

FIG. 2.23 – *La machine chimique réflexive ouverte*

sous-emplacements, ce qui donne une structure hiérarchique au calcul et permet de modéliser les sites comme des emplacements particuliers.

Les noms d'emplacements sont des valeurs de première classe, tout comme les noms de canaux. Ces noms peuvent être communiqués à d'autres processus éventuellement dans d'autres places, ce qui permet de programmer la gestion des places tout en contrôlant la migration par la portée lexical. Chaque emplacement contrôle ses propres mouvements relativement à son sur-emplacement, en désignant son nouvel emplacement. Ce mécanisme permet de simplifier l'analyse des programmes lorsque certains emplacements sont immobiles, et fournit l'ébauche d'un mécanisme plus élaboré de sécurité.

Solution distribuée (Plusieurs machines chimiques)

Afin de modéliser la présence de plusieurs sites de calcul, la machine abstraite chimique réflexive a été raffiner, chaque emplacement contient une machine abstraite, et l'ajout d'une règle de communication asynchrone entre emplacements : Ainsi, l'état du calcul est maintenant représenté par une famille de paires de multi-ensembles $\{\mathcal{D}_i,\ \mathcal{P}_i\}$ qui contiennent respectivement les règles de réactions et les processus en cours d'exécution dans chaque em-

placement.

Chacune des machines chimiques évolue localement comme précédemment : certains messages locaux sont consommés par des règles locales et remplacés par de nouveaux processus, tandis que les processus peuvent introduire de nouveaux noms avec leurs règles de réactions.

Par ailleurs, une règle supplémentaire décrit la communication globale : lorsqu'un message est émis dans un emplacement et que ce message est défini dans un autre emplacement, une étape de calcul transporte ce message de l'emplacement émetteur vers l'emplacement récepteur. Cette étape est muette, elle ne dépend pas du message lui-même ou de l'emplacement émetteur, mais uniquement de l'emplacement récepteur, elle n'affecte qu'un message à la fois. Par la suite, ce message pourra être consommé localement, par la règle **Red**, peut être avec d'autres messages.

Informellement, ce nouveau mécanisme de calcul reflète le *routage* des messages d'un point à l'autre du réseau.

$$\textbf{COMM} \quad \vdash x<\widetilde{v}> \,||\, J \rhd P \vdash \;\rightarrow\; \vdash \,||\, \quad J \rhd P \vdash x<\widetilde{v}> \quad (x \in \mathrm{dv}[J])$$

Passage de message distant Soit l'exemple du serveur d'impression : nous distinguons trois machines : la machine du serveur s, une imprimante *laser* p qui contacte le serveur, et la machine d'un utilisateur u sur laquelle une requête d'impression est en attente. Nous conservons la même définition pour le serveur

$$D \stackrel{def}{=} accepte{<}imprimante{>}|imprime{<}fichier{>} \rhd imprimante{<}fichier{>}$$

Nous avons la série de réductions suivante :

	$D\vdash_s$	$\|\|laser<f>\rhd P \vdash_p accepte<laser>$	$\|\| \vdash_u imprime<1>$
$\stackrel{com}{\rightarrow}$	$D\vdash_s imprime<1>$	$\|\|laser<f>\rhd P \vdash_p accepte<laser>$	$\|\| \vdash_u$
$\stackrel{com}{\rightarrow}$	$D\vdash_s accepte<laser>$,$imprime<1>$	$\|\|laser<f>\rhd P \vdash_p$	$\|\| \vdash_u$
$\stackrel{red}{\rightarrow}$	$D\vdash_s laser<1>$	$\|\|laser<f>\rhd P \vdash_p$	$\|\| \vdash_u$
$\stackrel{com}{\rightarrow}$	$D\vdash_s$	$\|\|laser<f>\rhd P \vdash_p laser<1>$	$\|\| \vdash_u$

La première étape transmet le message $imprime < 1 >$ de la machine de l'utilisateur à la machine du serveur, cette réduction a lieu parce que le nom *imprime* est uniquement défini sur la machine du serveur, en particulier on peut imaginer que l'information statique *imprime* est défini dans s est propagée en même temps que le nom *imprime*. De la même manière, la deuxième étape du calcul transmet le message $accepte < laser >$ de l'imprimante au serveur. Ensuite, les deux messages maintenant sur le serveur sont conjointement reçus, cette étape représente une synchronisation locale au serveur, et la seule étape si l'on oublie

la localisation. Elle déclenche un nouveau message sur le serveur, adressé à l'imprimante. A nouveau, ce message est d'abord transmis à cette machine, puis traité localement.

Cet exemple permet aussi d'illustrer la portée statique globale des noms dans le join-calcul réparti, ainsi que le mécanisme d'extension de portée par la communication. En supposant que le nom *laser* est initialement local à la machine p, une étape de réduction préliminaire sur cette machine peut être

$$\vdash_p \textbf{def } laser < f >\triangleright P \textbf{ in } accepte < laser > \overset{Str-def}{=} laser < f >\triangleright P \vdash_p accepte < laser >$$

Ensuite, la deuxième réduction **Comm** dans la série ci-dessus étend effectivement la portée de *laser* à la machine du serveur d'impression - il n'est plus possible d'effectuer une étape **Str-def** pour restreindre sa portée à une seule machine. Ainsi, le serveur d'impression devient capable d'envoyer des messages à l'imprimante qu'il ne connaissait pas auparavant.

L'arbre de location Les différents emplacements qui apparaissent au cours du calcul sont structurés en arbre, nous commençons par justifier ce choix. En effet, une alternative serait de considérer un modèle d'emplacements indépendants, dynamiquement liés à la machine sur laquelle ils s'exécutent. Ce modèle pose cependant certains problèmes.

Dans le cas ou plusieurs emplacements changent de place au cours du calcul, par exemple, il est souhaitable que la configuration finale ne dépende pas de l'ordre des migrations. L'objectif d'une migration est le plus souvent de se retrouver au même endroit qu'un autre emplacement. Or, cette liaison entre emplacements n'est pas assurée en cas de migrations multiples : Par exemple, si un programme crée un agent mobile pour se rendre sur un serveur, tandis que le serveur change de machine, il faut programmer explicitement l'agent pour suivre le serveur. De plus, la migration du serveur et celle de l'agent ne sont pas atomiques, il est ainsi possible que le serveur se déplace avant l'agent, puisque l'ancienne machine du serveur tombe en panne.

En revanche, une structure d'emplacements imbriqués permet de refléter de manière stable la proximité de certaines parties du calcul, ainsi que l'atomicité de leur migration. Par ailleurs, cette structure permet de modéliser facilement des systèmes à objets répartis avec des sous-objets. Techniquement enfin, les machines peuvent elles-mêmes être modélisées par des emplacements immobiles, ce qui permet de décrire des configurations distribuées complexes par des hiérarchies d'emplacements.

Soit un ensembles de noms de location noté par $\mathcal{X}$, les lettres $a, b, ... \in \mathcal{X}$, sont utilisées pour les noms de location, et $\varphi, \psi ... \in \mathcal{X}^*$, sont utilisées pour des chaînes finis, pour les noms de location. Comme les noms des ports, les noms de locations peuvent être créer localement, envoyer et recevoir des message.

La nouvelle syntaxe avec location, est définie en rajoutant le constructeur :

$$D \overset{def}{=} ...|a[D' : P]$$

Informalement, la définition $a[D\prime : P]$correspond à la solution local $\{D'\} \vdash_{\varphi a} \{P\}$.

La sémantique de ce nouveau constructeur est, la création d'une sous-location de la location courante, qui contient initialement l'unique définition D et l'unique processus P en cours d'exécution. Plus précisément, on a la nouvelle règle structurelle :

$$\textbf{Str-loc} \quad a[D : P] \vdash_{\varphi} \quad \rightleftharpoons \quad \vdash_{\varphi} ||\{D\} \vdash_{\varphi a} \{P\}$$

Telque, il n'y a pas de solution de la forme $\vdash_{\varphi a \phi}$ dans le contexte chimique pour tout φ, ψ dans $\mathcal{X}^*$.

Migration

On rajoute dans la syntaxe des processus une nouvelles primitives pour la migration :

$$P \stackrel{def}{=} ...|go < b, k >$$

avec une nouvelle règle chimique de réduction :

$$\textbf{Go} \ a[D : P|go < b, k >] \vdash_{\varphi} || \ \vdash_{\varphi b} \rightarrow \vdash_{\varphi} ||a[D : P|k <>] \vdash_{\varphi b}$$

La location a déplace de sa position courante φa dans l'arbre vers une nouvelle position φba juste sous la location nommée b. La solution destination $\vdash_{\varphi b}$ est définie par son nom b. Une fois a est arrivée, la continuation $k <>$ peut déclencher d'autres calculs.

Pannes

Join-calcul fournit également un modèle élémentaire de pannes. L'arrêt brutal d'une machine peut causer la terminaison des emplacements qui y résident, de manière irréversible. De manière plus générale, chaque emplacements peut s'arrêter, entraînant tous ses sous-emplacements. La terminaison d'un emplacement est détectable à partir d'autres emplacements qui continuent à fonctionner, ce qui permet de programmer explicitement la résistance aux pannes si besoin.

Le modèle de pannes et le modèle de migration sont loin d'être indépendants, outre le partage de la notion d'emplacements, la migration peut être utilisée de manière constructive pour se prémunir contre certaines erreurs. Par exemple, un protocole entre deux programmes est considérablement compliqué lorsque chacun des programmes craint une panne de l'autre programme entre chaque message : il faut alors écrire de nombreux codes de rattrapage de pannes. En revanche, si les deux programmes envoient leur agent sur une machine fiable, le problème disparaît presque entièrement. Même dans le cas ou la machine commune n'est pas fiable, elle donne une garantie très utile : les deux agents disparaissent ensemble en cas de pannes, à nouveau, un test unique dans chacun des programmes suffit à se prémunir ce type de pannes.

$P, Q ::=$		**Processus**
	$x < v_1, ..., v_n >$	*Message asynchrone*
	$\mid$ **def** D **in** P	*Définition local*
	$\mid\ P \mid Q$	*Composition parallèle*
	$\mid\ \mathbf{0}$	*Processus inerte*
	$\mid\ go < a, k >$	*Migration de la location courante*
	$\mid\ halt <>$	*Terminaison de la location courante*
	$\mid\ fail < a, k >$	*Détection de panne de location*
$D ::=$		**Définition**
	$J \rhd P$	*Règle réactive*
	$\mid\ D \wedge D\prime$	*Composition*
	$\mid\ \mathsf{T}$	*Définition vide*
	$\mid\ a[D : P]$	*Sous-location*
	$\mid\ \Omega a[D : P]$	*Sous-location bloquée.*
$J ::=$		**Join-patterns**
	$x < y_1, ..., y_n >$	*Patterns message*
	$\mid\ J \mid J\prime$	*Pattern de join*

FIG. 2.24 – *Syntaxe de join-calcul distribué*

Pour les processus :

$$\begin{aligned}
\mathrm{fv}[x < v_1, , v_n >] &\stackrel{def}{=} \{x, v_1, ..., v_n\} \\
\mathrm{fv}[\mathbf{def}\ D\ \mathbf{in}\ P] &\stackrel{def}{=} (\mathrm{fv}[P] \cup \mathrm{fv}[D]) \backslash \mathrm{dv}[D] \\
\mathrm{fv}[P|P'] &\stackrel{def}{=} \mathrm{fv}[P] \cup \mathrm{fv}[P'] \\
\mathrm{fv}[\mathbf{0}] &\stackrel{def}{=} \phi \\
\mathrm{fv}[go < a, k >] &\stackrel{def}{=} \{a, k\} \\
\mathrm{fv}[halt <>] &\stackrel{def}{=} \phi \\
\mathrm{fv}[fail < a, k >] &\stackrel{def}{=} \{a, k\}
\end{aligned}$$

Pour les définitions

$$\begin{aligned}
\mathrm{fv}[J \rhd P] &\stackrel{def}{=} \mathrm{dv}[J] \cup (\mathrm{fv}[P] \backslash \mathrm{rv}[J]) \\
\mathrm{fv}[D \wedge D\prime] &\stackrel{def}{=} \mathrm{fv}[D] \cup \mathrm{fv}[D\prime] \\
\mathrm{fv}[T] &\stackrel{def}{=} \phi \\
\mathrm{fv}[\varepsilon a[D : P]] &\stackrel{def}{=} \{a\} \cup \mathrm{fv}[D] \cup \mathrm{fv}[P]
\end{aligned}$$

$$\begin{aligned}
\mathrm{dv}[J \rhd P] &\stackrel{def}{=} \mathrm{dv}[J] \\
\mathrm{dv}[D \wedge D\prime] &\stackrel{def}{=} \mathrm{dv}[D] \cup \mathrm{dv}[D\prime] \\
\mathrm{dv}[T] &\stackrel{def}{=} \phi \\
\mathrm{dv}[\varepsilon a[D : P]] &\stackrel{def}{=} \{a\} \uplus \mathrm{dv}[D] \\
\mathrm{dv}[a[D : P]] &\stackrel{def}{=} \{a\} \uplus \mathrm{dv}[D]
\end{aligned}$$

Pour les Join-patterns

$$\begin{aligned}
\mathrm{dv}[x < y_1, ..., y_n >] &\stackrel{def}{=} \{x\} & \qquad \mathrm{dv}[J|J'] &\stackrel{def}{=} \mathrm{dv}[J] \uplus \mathrm{dv}[J'] \\
\mathrm{rv}[x < y_1, ..., y_n >] &\stackrel{def}{=} \{y_1, ..., y_n\} & \qquad \mathrm{rv}[J|J'] &\stackrel{def}{=} \mathrm{rv}[J] \uplus \mathrm{rv}[J']
\end{aligned}$$

FIG. 2.25 – *Les portées dans join-calcul distribué*

Représentation des pannes

Soit Ω un marquage $\notin \mathcal{X} \cup \mathcal{N}$ des étiquettes pour des locations en pannes. Pour chaque $a \in \mathcal{X}$, εa dénote aussi a ou Ωa, et φ, ψ des chaînes de εa dans DRCHAM, Ω apparaît dans la chaîne de location en panne $\vdash_\varphi$. On dit que φ est bloquée si elle contient Ω, et activée dans les autres cas.

Conditions pour chaque définition D :

- Chaque nom d'emplacement est défini au maximum une fois.

- Chaque nom de canal est défini par le join-patterns tellque tous apparaissent dans le même emplacement.

str-join	$\vdash P_1 \vert P_2$	$\rightleftharpoons$	$\vdash P_1, P_2$
str-null	$\vdash 0$	$\rightleftharpoons$	$\vdash$
str-and	$D_1 \wedge D_2 \vdash$	$\rightleftharpoons$	$D_1, D_2 \vdash$
str-nodef	$\top \vdash$	$\rightleftharpoons$	$\vdash$
str-def	$\vdash \mathbf{def}\ D\ \mathbf{in}\ P$	$\rightleftharpoons$	$D\sigma_{dv} \vdash P\sigma_{dv}$
Str-loc	$\varepsilon a[D : P] \vdash_\varphi$	$\rightleftharpoons$	$\vdash_\varphi \vert\vert \{D\} \vdash_{\varphi\varepsilon a} \{P\}$
Red	$J \rhd P \vdash_\varphi J\sigma_{rv}$	$\rightarrow$	$J \rhd P \vdash_\varphi P\sigma_{rv}$
Comm	$\vdash_\varphi x < \widetilde{y} > \vert\vert\ J \rhd P \vdash$	$\rightarrow$	$\vdash_\varphi \vert\vert \quad J \rhd P \vdash x < \widetilde{y} >$
Go	$a[D : P \vert go < b, k >] \vdash_\varphi \vert\vert\ \vdash_{\psi\varepsilon b}$	$\rightarrow$	$\vdash_\varphi \vert\vert a[D : P \vert k <>] \vdash_{\psi\varepsilon b}$
Halt	$a[D : P \vert halt <>] \vdash_\varphi$	$\rightarrow$	$\Omega a[D : P] \vdash_\varphi$
Detect	$\vdash_\varphi fail < a, k > \vert\vert \vdash_{\psi\varepsilon a}$	$\rightarrow$	$\vdash_\varphi k <> \vert\vert \vdash_{\psi\varepsilon a}$

Conditions ($\mathcal{S}$ la solution distribuée)

str-def	σ_{dv} instanciation des variables dans dv[D] pour distinguer, les noms frais : $dom(\sigma_{dv}) \cap \text{fv}[\mathcal{S}] = \phi$.
Str-loc	Le nom a n'apparaît pas dans les indices ψ pour toute solution local $\mathcal{S}$
Red	σ_{rv} Substitution des valeurs aux variables reçues rv[J].
Comm	$x \in$dv[J].
Detect	a bloqué : la chaîne $\psi\varepsilon a$ contient le marquage Ω
Red,Comm,Go	φ activé(n'est pas bloqué) : la chaîne φ ne contient pas le marquage
Halt, Detect	Ω

FIG. 2.26 – *La machine chimique réflexive distribuée DRCHAM*

2.4 Calcul des Ambients

Nous décrivons dans cette section un modèle pour la programmation mobile et distribuée : le calcul des ambients[6] [7][8]. C'est un modèle de calcul de processus utilisant la communication de noms, dans l'esprit de π-calcul.

Nous commençons par définir la syntaxe et la sémantique du calcul des ambients sans communication, dans la deuxième section on introduit la notion de communication dans ce calcul.

2.4.1 Présentation

Ambients

Un ambient ce caractérise principalement par le suivant[7] :

- Un ambient est une place bornée où se passe le calcul. La propriété intéressante ici c'est l'existence de limite autour d'un ambient. Les bornes détermine ce qui appartient et ce qui n'appartient pas a l'ambient. Exemple d'ambient dans ce sens, une page web(limitée par un fichier).
- Un ambient est quelque chose qui peut être se niché dans d'autres ambients. Si nous voulons déplacer une application, depuis notre travail vers la maison, l'application doit être enlevée de l'ambient(travail), et insérée dans un ambient différent (maison).
- Un ambient est quelque chose qui peut être déplacé dans son ensemble. Si nous rebranchons un portable à un réseau différent, tous les espaces d'adressage et systèmes de fichier dans le portable, déplacent en conséquence et automatiquement.

2.4.2 Mobilité

Primitifs de mobilité

Le calcul décrit la migration de processus résident dans des domaines administratifs. La syntaxe de calcul est décrite par la figure 2.27, page 51 ; un processus ambient P est soit le processus inerte, soit une composition parallèle $P|P$, soit un processus répliqué $!P$, soit un ambient $n[P]$ nommé n, soit un processus $va.P$ avec un nom local a, soit un processus $M.P$ gardé par la capacité M.

Noms libre

$$\mathrm{fn}((vn)P) \triangleq \mathrm{fn}(P) - \{n\} \qquad \mathrm{fn}(in\ n) \triangleq \{n\}$$
$$\mathrm{fn}(0) \triangleq \phi \qquad \mathrm{fn}(out\ n) \triangleq \{n\}$$
$$\mathrm{fn}(P|Q) \triangleq \mathrm{fn}(P) \cup \mathrm{fn}(Q) \qquad \mathrm{fn}(open\ n) \triangleq \{n\}$$
$$\mathrm{fn}(!P) \triangleq \mathrm{fn}(P)$$
$$\mathrm{fn}(n[P]) \triangleq \{n\} \cup \mathrm{fn}(P)$$
$$\mathrm{fn}(M.P) \triangleq \mathrm{fn}(M) \cup \mathrm{fn}(P)$$

N		**Noms**
$P, Q ::=$		**Processus**
	$(vn)P$	*Restriction*
	$\mid\ !\ P$	*Réplication*
	$\mid\ P\|Q$	*Composition parallèle*
	$\mid\ \mathbf{0}$	*Processus inerte*
	$\mid\ n[P]$	*Ambient*
	$\mid\ M.P$	*Action*
$M ::=$		**Capacité**
	in n	*Migration entrante*
	out n	*Migration sortante*
	open n	*dissolution d'ambient*

FIG. 2.27 – *Syntaxe du calcul des ambient sans communication*

Un ambiant est noté par $n[P]$, où n c'est le nom de l'ambient, et P un processus en cours d'exécution à l'intérieur de l'ambient, il peut être une composition parallèle de processus.

En général, les ambients sont organisés en structure d'arbre. La racine de l'arbre représente le réseau. Les sous-ambients, plus ou moins profonds dans l'arbre, sont des entités mobiles logiques ou physiques. Les migrations sont locales et dépendent de la structure locale de l'arbre. Un ambient peut migrer hors de son père, ou entrer dans un de ses frères. Un ambient peut aussi dissoudre un ambient fils. Pour qu'un ambient atteigne un ambient distant, il doit connaître la structure de l'arbre afin d'effectuer des migrations élémentaires le rapprochant de sa destination.

Le processus $M.P$, exécute la capacité M, puis continu le processus P. Le processus P, ne peut être exécuté jusqu'à ce que l'action soit exécutée.

Capacité d'entrée :

Une capacité d'entrée, *in* m, peut être utilisée dans l'action *in* $m.P$, qui représente une entrée a un ambient nommé m. La règle de réduction est :

$$n[in\ m.P|Q] \mid m[R] \rightarrow m[n[P|Q]|R]$$

Capacité de sortie :

Une capacité de sortie, *out* m, peut être utilisée dans l'action *out* $m.P$, qui représente une sortie de l'ambient nommé m. La règle de réduction est :

$$m[n[out\ m.P|Q]|R] \rightarrow n[P|Q]|m[R]$$

Capacité d'ouverture :

Une capacité d'ouverture, *open* m, peut être utilisée dans l'action *open* $m.P$, L'action fournit une façon de dissoudre la limite d'un ambient nommé m, situé au même niveau que *open*, la règle de réduction est :

$$Open\ m.P|m[Q] \rightarrow P|Q$$

2.4.3 Sémantique opérationnelle

La sémantique du calcul est basée sur une relation de congruence structurelle $\equiv$, entre les processus, et une relation de réduction $\rightarrow$.

La congruence structurelle $\equiv$ est la plus petite relation d'équivalence sur les processus qui satisfait les équations et règles suivantes

Congruence structurelle

$P \equiv P$	$P\|Q \equiv Q\|P$
$Q \equiv P \Longrightarrow P \equiv Q$	$(P\|Q)\|R \equiv P\|(Q\|R)$
$P \equiv Q, Q \equiv R \Longrightarrow P \equiv R$	$!P \equiv P\|!P$
	$(vn)(vm)P \equiv (vm)(vn)P$
$P \equiv Q \Longrightarrow (vn)P \equiv (vn)Q$	$(vn)(P\|Q) \equiv P\|(vn)Q$ si $n \notin \mathrm{fn}(P)$
$P \equiv Q \Longrightarrow P\|R \equiv Q\|R$	$(vn)(m[P]) \equiv m[(vn)P]$ si $n \neq m$
$P \equiv Q \Longrightarrow !P \equiv !Q$	$P\|0 \equiv P$
$P \equiv Q \Longrightarrow n[P] \equiv n[Q]$	$(vn)0 \equiv 0$
$P \equiv Q \Longrightarrow M.P \equiv M.Q$	$!0 \equiv 0$

A noté que les termes suivants sont différents :

$!(vn)P \equiv\ (vn)!P$ La réplication crée de nouveaux noms

$n[P]|n[Q] \equiv n[P|Q]$ Les différents ambients n, ont des identités séparés.

Réduction

La réduction d'ambient $\rightarrow$, est la plus petite relation sur les processus qui satisfait les règles suivantes :

$n[in\ m.P\|Q]\|m[R] \rightarrow m[n[P\|Q]\|R]$	$P \rightarrow Q \Longrightarrow P\|R \rightarrow Q\|R$
$m[n[out\ m.P\|Q]\|R] \rightarrow n[P\|Q]\|m[R]$	$P \rightarrow Q \Longrightarrow (vn)P \rightarrow (vn)Q$
$open\ n.P\|n[Q] \rightarrow P\|Q$	$P \rightarrow Q \Longrightarrow n[P] \rightarrow n[Q]$
$P\prime \equiv P, P \rightarrow Q, Q \equiv Q\prime \Longrightarrow P\prime \rightarrow Q\prime$	

2.4.4 Exemples

Certification d'agent mobile[7]

Un processus dans un niveau supérieur d'un ambient peut être privilégié, parce qu'il peut affecter directement le mouvement des alentours de l'ambient et peut dissoudre le sous-ambient. Supposez qu'un tel processus privilégié veut quitter son ambient *Home*, après retourner et

soit réintégré comme un processus privilégié . L'ambiant $Home$ peut ne pas autoriser juste tout visiteur être privilégié, pour des raisons de sécurité. Donc le processus original doit être certifié d'une façon ou d'une autre.

Une solution est donnée dessous. Le processus niveau supérieur crée un nouveau nom, n, pour être utilisé comme un secret partagé entre lui-même et l'ambient $Home$, *open* n est en place pour certifier le processus quand il revient. Le processus quitte alors $Home$ dans la forme d'un ambient $Agent$. Sur son retour à l'intérieur de $Home$, l'ambient $Agent$ exposés a l'ambient n, cela est ouvert par *open* n pour réintégrer la continuation P du processus original au niveau supérieur de $Home$.

$Home[$
$\quad (vn)(open\ n|$
$\quad\quad Agent[out\ Home.in\ Home.n[out\ Agent.open\ Agent.P]])$
$]$

Voici la trace du calcul :

$Home[(vn)(open\ n|Agent[out\ Home.in\ Home.n[out\ Agent.open\ Agent.P]])]$
$\equiv (vn)Home[open\ n|Agent[out\ Home.in\ Home.n[out\ Agent.open\ Agent.P]]]$
$\rightarrow (vn)(Home[open\ n]|Agent[in\ Home.n[out\ Agent.open\ Agent.P]])$
$\rightarrow (vn)Home[open\ n|Agent[n[out\ Agent.open\ Agent.P]]]$
$\rightarrow (vn)Home[open\ n|n[open\ Agent.P]|Agent[]]$
$\rightarrow (vn)Home[0|open\ Agent.P|Agent[]]$
$\rightarrow (vn)Home[0|P|0]$
$\equiv Home[P]$

Cet exemple illustre la création d'un secret partagé (n) dans un emplacement sûr, ($Home$). La distribution du secret sur le réseau (a porté le long par $Agent$), et la certification de processus nouveaux est basé sur le secret partagé.

Pare-feu[38]

Prenons un processus $Q \stackrel{def}{=} < 4 >$ qui envoie la valeur 4 à une imprimante sécurisée $P \stackrel{def}{=} !(x).Print_x$(nous supposons que $Print_x$ est une primitive qui permet d'imprimer la valeur associée à x). Soit le protocole de l'exemple du pare-feu présenté comme suit

$$(vw.w[k[out\ w.in\ k\prime.in\ w]|open\ k\prime.P])|k\prime[open\ k.Q]$$

Le pare-feu envoie le sous-ambient portant le nom k à k' pour vérifier sa clé. Cette clé correspond au nom k, et la vérification de la clé à l'ouverture de l'ambient k dans k'. L'ambient k contient une capacité *in* w correspondant au nom du pare-feu. Ceci permet à k' de franchir le pare-feu pour y apporter Q qui pourra ainsi interagir avec P.

$\rightarrow vw.w[k[out\ w.in\ k'.in\ w]|open\ k'.P]|k'[open\ k.Q]$ 1
$\rightarrow vw.w[open\ k'.P]|k[in\ k'.in\ w]|k'[openk.Q]$ 2
$\rightarrow vw.w[open\ k'.P]|k'[k[in\ w]|open\ k.Q]$ 3
$\rightarrow vw.w[open\ k'.P]|k'[in\ w|Q]$ 4
$\rightarrow vw.w[open\ k'.P|k'[Q]]$
$\rightarrow vw.w[P|Q]$
$\rightarrow vw.w[P|!(x).Printx| < 4 >]$ 5
$\rightarrow vw.w[P|Printx]$ 6

Le processus de l'exemple est équivalent à 1, après extrusion de la protée de w. Le sous-ambient k commence par sortir du pare-feu, en utilisant la capacité *out*, pour donner 2. L'ambient k peut alors entrer dans l'ambien k' en utilisant sa capacité *in*, donnant 3. Ensuite, l'ambient k est dissout par la capacité open de k', ce qui donne 4. Le calcul se poursuit par des réductions *in*, *open*, *repl* jusqu'à 5. Enfin, les processus P et Q sont voisins dans l'ambient w et peuvent communiquer 6.

2.4.5 Communication

Primitifs de communication

On commence par la syntaxe, les primitives de la mobilité sont les même de la section précédente, mais l'addition des variables de communication change quelques termes (Voir la figure 2.28, page 55).

Noms libres

$\text{fn}(M[P]) \triangleq \text{fn}(M) \cup \text{fn}(P)$ $\quad$ $\text{fn}(x) \triangleq \phi$
$\text{fn}((x).P) \triangleq \text{fn}(P)$ $\quad$ $\text{fn}(\ n) \triangleq \{n\}$
$\text{fn}(< M >) \triangleq \text{fn}(M)$ $\quad$ $\text{fn}(x) \triangleq \phi$
$\quad$ $\quad$ $\text{fn}(M.M') \triangleq \text{fn}(M) \cup \text{fn}(M')$

Variables libres

$\text{fv}((vn)P) \triangleq \text{fv}(P)$ $\quad$ $\text{fv}(in\ M) \triangleq \text{fn}(M)$
$\text{fv}(0) \triangleq \phi$ $\quad$ $\text{fv}(out\ M) \triangleq \text{fn}(M)$
$\text{fv}(P|Q) \triangleq \text{fv}(P) \cup \text{fv}(Q)$ $\quad$ $\text{fv}(open\ M) \triangleq \text{fn}(M)$
$\text{fv}(!P) \triangleq \text{fv}(P)$ $\quad$ $\text{fv}(x) \triangleq \{x\}$
$\text{fv}(M[P]) \triangleq \text{fv}(M) \cup \text{fv}(P)$ $\quad$ $\text{fv}(n) \triangleq \phi$
$\text{fv}((x).P) \triangleq \text{fv}(P) - \{x\}$ $\quad$ $\text{fv}(\varepsilon) \triangleq \phi$
$\text{fn}(< M >) \triangleq \text{fn}(M)$ $\quad$ $\text{fv}(M.M') \triangleq \text{fv}(M) \cup \text{fv}(M')$
$\text{fv}(M.P) \triangleq \text{fv}(M) \cup \text{fv}(P)$

Noté que $P\{x \longleftarrow M\}$ c'est la substitution de la capacité M pour chaque occurrence de la variable x dans le processus P. De la même façon pour $M\{x \longleftarrow M\prime\}$.

N		**Noms**
$P, Q ::=$		**Processus**
	$(vn)P$	*Restriction*
	$\mid\ !\ P$	*Réplication*
	$\mid\ P \mid Q$	*Composition parallèle*
	$\mid\ \mathbf{0}$	*Processus inerte*
	$\mid\ M[P]$	*Ambient*
	$\mid\ M.P$	*Action*
	$\mid (x).P$	*Réception de message*
	$\mid < M >$	*Message asynchrone*

$M ::=$		**Capacité**
	x	
	n	
	in M	*Migration entrante*
	out M	*Migration sortante*
	open M	*Dissolution d'ambient*
	ε	*Nul*
	$M.M\prime$	*Chemin*

FIG. 2.28 – *Syntaxe du calcul des ambient avec communication*

Valeurs de la communication

Les entités qui peuvent être communiquées sont des noms ou capacités. Dans les situations réalistes, la communication de noms devrait être plutôt rare.

Il devient maintenant utile de combiner des capacités multiples dans des chemins, surtout quand un ou plus de ces capacités sont représentées par des variables. Par exemple, $(outn.inm).P$ Les variables peuvent être remplacées par des noms ou des capacités.

Le mécanisme de communication le plus simple que nous pouvons imaginer, est une communication locale anonyme dans un ambiant :

$$(x).P \mid < M > \rightarrow P\{x \longleftarrow M\}$$

Sémantique opérationnelle

Congruence structurelle
$P \equiv Q \Longrightarrow M[P] \equiv M[Q]$
$P \equiv Q \Longrightarrow (x).P \equiv (x).Q$
$\varepsilon.P \equiv P$
$(M.M\prime).P \equiv M.M\prime.P$
La règle de réduction :

$$(x).P| < M > \rightarrow P\{x \longleftarrow M\}$$

2.5 Logique de réécriture

Dans cette section, nous définissons les concepts de base liés à la réécriture. Nous commençons par la définition des termes, des substitutions et de l'algèbre de termes. Nous définissons ensuite la logique de réécriture, en particulier nous définissons la forme de ses signatures, formules, théories et ses règles de déduction. Toutes les notions citées dans cette section sont rédiger à partir des documents[27][24] [28] [29][1][32][33].

La logique de réécriture a été proposée par J.Meseguer comme une sémantique concurrente des systèmes de réécriture de termes où les opérateurs peuvent être associatifs, commutatifs, idempotents ou dotés d'un éléments neutre à gauche ou à droite. La logique de réécriture formalise le processus de la réécriture, par opposition à la logique équationnelle qui considère les règles de réécriture comme des égalités et utilise la réécriture pour calculer un représentant unique d'une classe d'équivalence. La logique de réécriture utilise les règles de réécriture pour calculer une relation de " réécrivabilité " entre des termes. Elle permet d'exprimer le fait qu'un terme peut réécrit en un autre. Les objets principaux de la logique de réécriture sont les classes d'équivalence des termes.

Cette logique exprime aussi une équivalence essentielle entre logique et calcul dans une représentation simple. Une axiome logique de réécriture de la forme, $(t \rightarrow t')$, a deux lectures. Dans l'esprit calculatoire, un fragment t d'états de système, peut se transformer en t', en concurrence avec d'autres changements d'états, c'est une transition local concurrente. Dans l'esprit logique, on peut dériver la formule t' à partir de t, on l'appelle une règle d'inférence.

La logique de réécriture, utilisée comme une structure sémantique[24], supporte un large spectre d'applications. Premièrement ; elle est utilisée pour spécifier et unifier beaucoup de modèles de calcul concurant. Deuxièmement ; la logique de réécriture est aussi une structure sémantique pour les langages de programmation. Elle peut être vue comme un langage déclaratif de spécification où les programmes parallèles ou séquentiels peuvent être spécifiés par des règles de réécriture.

La logique de réécriture, utilisée comme une structure logique. Premièrement ; on peut représenter d'autres logiques dans cette logique, et, utiliser son implémentation, pour exécuter les logiques donc à représentées. Deuxièmement ; la même idée peut être appliquée

non seulement à une logique, mais aussi au système d'inférence de preuve de théorème, ou autre outil de la déduction automatisée, dans ce sens nous pouvons spécifier formellement et exécuter les preuves des théorèmes ou outiller dans la logique de réécriture, et nous pouvons utiliser la même spécification comme un nouveau outil.

2.5.1 Définitions de base

La logique de réécriture repose sur les définitions des algèbres universelle, en premier lieu la définition d'une signature : multi-sortes, ordre-sortes, ou mono sortes, pour simplifier la théorie nous traitent le cas mono sorte[1].

Définition 2.15 *(Opérateurs)* Σ *est un ensemble d'opérateurs (symboles de fonctions) muni de la fonction arité :* $\Sigma \rightarrow \mathcal{N}$ *qui associe à un symbole de fonction l'arité du symbole .* Σ_n *désigne le sous-ensemble de fonctions d'arité n.*

Définition 2.16 *(Signature) Une signature dans la logique de réécriture est une paire (*Σ*, E) avec* Σ *alphabet de symboles de fonctions et E un ensemble de* Σ*-équations. La réécriture opère sur les classes d'équivalence modulo l'ensemble d'équations E.*

Définition 2.17 *(Termes) les termes sur* Σ *sont définis comme un ensemble* $\mathcal{T}(\Sigma,\mathcal{X})$*, par les règles :*

· *Chaque variable* $x \in \mathcal{X}$ *est un terme de* $\mathcal{T}(\Sigma,\mathcal{X})$*,*

· *Si* $t_1, ..., t_n$ *sont des termes de* $\mathcal{T}(\Sigma,\mathcal{X})$*, respectivement, et si* $f \in \Sigma$ *alors* $f(t_1, ..., t_n)$ *est un terme de* $\mathcal{T}(\Sigma,\mathcal{X})$*.*

Définition 2.18 *(* Σ*-algèbre) une* Σ*-algèbre* $\mathcal{A}$ *est alors un ensemble non vide de valeurs* $\mathcal{A}$ *et avec une fonction* $f_A : A^n \rightarrow A$ *pour chaque* $f \in \Sigma_n$ *avec* $n \in \mathcal{N}$ *.*

Définition 2.19 *(Algèbre quotient des termes) soient* $\pm$ *un ensemble d'opérateurs,* $\mathcal{X}$ *un ensemble de variables et* $\equiv_E$ *la plus petite relation de congruence contenant l'ensemble des axiomes E, alors l'ensemble quotient des termes* $\mathcal{T}_{\Sigma,E}(\mathcal{X})$ *est l'ensemble* $\mathcal{T}_{\Sigma,E}(\mathcal{X}) = \{[u]_E \mid u \in \mathcal{T}(\Sigma, \mathcal{X})\}$ *où* $[u]_E$ *est la classe d'équivalence définie par* $[u]_E = \{v \mid v \in \mathcal{T}(\Sigma, \mathcal{X}), v \equiv_E u\}$*.*

On note par $\mathcal{T}_\Sigma$, Σ-algèbre des Σ-termes(Σ-termes clos), et par $\mathcal{T}_\Sigma(\mathcal{X})$ Σ-algèbre des Σ-termes avec des variables dans l'ensemble $\mathcal{X}$. De la même façon, soit E un ensemble de Σ-équations, $\mathcal{T}_{\Sigma,E}$ représente Σ-algèbre des classes d'équivalence des Σ-termes modulo les équations E, $\mathcal{T}_{\Sigma,E}$ $(\mathcal{X})$: Σ-algèbre des classes d'équivalence des Σ-termes avec des variables dans l'ensemble $\mathcal{X}$ modulo les équations dans E.

Définition 2.20 *(formules de la logique de réécriture) les formules de la logique de réécriture pour une signature (*$\mathcal{S}$*, E) sont de la forme* $[t]_E \Longrightarrow [t']_E$*. Où* $[t]_E, [t']_E$ *sont des classes d'équivalences de termes* t, t' *dans l'ensemble quotient des termes modulo E.*

Définition 2.21 *(substitution) soit* $t \in T_\Sigma(\{x_1, ..., x_n\})$*, et les termes* $u_1, ..., u_n$*, c'est le terme obtenu à partir de* t*, par une les substitutions simultanées* u_i *pour* $x_i, i = 1, ..., n$*. Notée* $t(\widetilde{u}/\widetilde{x})$

Définition 2.22 *(Séquent de réécriture) Soit la signature (*Σ*, E), les formules dans la logique de réécriture sont des " séquents " de la forme (*$\pi : [t]_E \rightarrow [t']_E$*), où* $t, t' \in \mathcal{T}(\Sigma,\mathcal{X})$*,* $\pi \in \mathcal{T}(L \cup \{;\} \cup F \cup T_{\Sigma,E}(X))$*, où (;) est un opérateur binaire infixe.* π *est appelé terme de preuve.*

Le sens informel du séquent $(\pi : [t]_E \rightarrow [t']_E)$, est que le calcul π permet de dériver t' de t . Les termes de preuves formalisent les preuves de la logique de réécriture en affectant à chaque preuve un terme d'une structure algébrique des preuves. Un terme de preuve est une formalisation d'une preuve en logique de réécriture. Il contient les informations sur la construction d'une formule valide, (les informations sur les règles de déduction appliquées et sur leurs paramètres).

Définition 2.23 *(Théorie de réécriture) Une théorie de réécriture* $\mathcal{R}$ *est un quintuple* $\mathcal{R}$ *= (*Σ*, E, L, R) où* Σ *alphabet de symboles de fonction, E un ensemble de* Σ*-équations, L c'est un ensemble d'étiquettes, et R un ensemble de paires* $R \subseteq L \times \mathcal{T}_{\Sigma,E}(\mathcal{X})^2$ *dont le premier composant est une étiquette, et le deuxième composant est une paire de classes E-équivalence de termes, avec* $\mathcal{X}$*=*$\{x_1, ..., x_n, ...\}$ *un ensemble infinie de variables. Un élément de* $\mathcal{R}$ *sont appelé règle de réécriture. On note une règle* $(r, ([t(x_1, ..., x_n)], [t'(x_1, ..., x_n)]))$ *de la façon suivante*

$$(r, ([t(x_1, ..., x_n)] \rightarrow [t'(x_1, ..., x_n)]))$$

ou tout simplement

$$(r, ([t(\overline{x})] \rightarrow [t'(\overline{x})]))$$

Définition 2.24 *(Règles de déduction de la logique de réécriture) Soit une théorie de réécriture* $\mathcal{R}$*= (*Σ*, E, L, R), nous disons que* $\mathcal{R}$ *implique un séquents* $[t] \rightarrow [t']$*et on écrit* $\mathcal{R} \vdash [t] \rightarrow [t']$ *si et seulement si le séquent peut être obtenu par une suite finie d'application des règles de détection suivantes :*

1. Réflexivité *: Pour chaque* $[t] \in \mathcal{T}_{\Sigma,E}(X)$, $\dfrac{}{[t] \rightarrow [t]}$.

2. Congruence *: Pour chaque* $f \in \Sigma_n$ *,* $n \in \mathcal{N}$*,*

$$\frac{[t_1] \rightarrow [t'_1] ... [t_n] \rightarrow [t'_n]}{[f(t_1, ..., t_n)] \rightarrow [f(t'_1, ..., t'_n]}$$

3. Remplacement *: Pour chaque règle de réécriture* $(r, [t(x_1, ..., x_n)] \rightarrow [t\prime(x_1, ..., x_n)])$ *dans* $\mathcal{R}$*,*

$$\frac{[w_1] \rightarrow [w'_1] ... [w_n] \rightarrow [w'_n]}{[t(\widetilde{w}/\widetilde{x})] \rightarrow [t'(\widetilde{w}'/\widetilde{x}]}$$

4. Transitivité :

$$\frac{[t_1] \rightarrow [t_2] \qquad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

La règle de transitivité dit comment créer un séquent par composition de séquents. La congruence dit que la relation de réécriture est compatible avec la structure algébrique et exprime la réduction parallèle. La règle de remplacement décrit la réduction simultanée de deux radicaux compatibles[1].

Définition 2.25 *Soit une théorie de réécriture $\mathcal{R}$ = (Σ, E, L, R), un (Σ, E)-séquent $[t] \rightarrow [t\prime]$ est appelé $\mathcal{R}$-réécriture concurrente si et seulement si, il peut être dériver a partir de $\mathcal{R}$ par l'application finie des règles 1-4.*

La logique de réécriture c'est une logique pour raisonner sur les systèmes concourants, qui ont des états, et évolu en terme de transitions. La signature de la théorie de réécriture décrit une structure particulière pour les états d'un système - mutiset, arbre binaire, etc -. Le changement basique local dans le système est axiomatisé par des règles de réécriture correspondant à des patterns locaux pouvant être changés en d'autres partterns. Donc chaque étape de réécriture c'est une transition local parallèle dans un système concurrent. Donc on obtient[29].

Etat	$\longleftrightarrow$	*Terme*	$\longleftrightarrow$	*Proposition*
Transition	$\longleftrightarrow$	*Réécriture*	$\longleftrightarrow$	*Déduction*
Structure distribuée	$\longleftrightarrow$	*Structure algébrique*	$\longleftrightarrow$	*Structure propositionnelle*

Exemple 2.2 *Considérons la théorie de réécriture $\mathcal{R} = (\Sigma, R)$ avec $\Sigma = (f, E, X)$*

$f_0 = \{zero\}, f_1 = \{s\}, f_2 = \{plus\}, f = f_0 \cup f_1 \cup f_2$
$E = \{plus(x, y) = plus(y, x)\}, X = \{x, y\}, L = \{l_0, l_1\}$
Soit $R = \left\{ \begin{array}{ll} l_0 & : plus(zero, zero) \rightarrow x \\ l_1(x, y) & : plus(s(x), y) \rightarrow s(plus(x, y)) \end{array} \right\}$

En prenant la règle de déduction **Réflexivité** pour obtenir la formule $[zero]_E \Longrightarrow [zero]_E$, cette formule est ensuite utilisée comme prémisse pour la règle de déduction **Remplacement** avec la règle l_1 (la règle est appliquable grâce à la commutativité de plus) afin d'obtenir la formule $[plus(zero, s(zero)]_E \Longrightarrow [s(plus(zero, zero)]_E$. Puis nous utilisons encore une fois la déduction **Remplacement**, mais avec la règle l_0 afin d'obtenir $[plus(zero, zero]_E \Longrightarrow [zero]_E$ et nous finissons la preuve en utilisant sur cette formule la règle de déduction **Congruence** pour le symbole s et la règle de **Transitivité** pour concaténer les deux applications des règles l_0 et l_1.

Donc cette théorie implique le séquent :

$$[l_0(s(zero))]_E : [plus(zero, s(zero))]_E \Longrightarrow [(s(zero))]_E$$

Qui exprime une application de la règle l_0 en appliquant les règles de déduction de la logique.

Soit la formule $[plus(zero, plus(s(zero), zero))]_E \Longrightarrow [(s(zero))]_E$.

Deux preuves π_1 et π_2 de cette formule sont :

$$\pi_1 = l_0([plus(s(zero), zero)]_E); l_1([zero]_E, [zero]_E)$$

$$\pi_2 = plus(zero, l_1([zero]_E, [zero]_E)); l_0([s(plus(zero, zero))]_E)$$

le terme de preuve π_1 exprime la réduction en tête de l_0 puis la réduction de l_1. Ce terme de preuve exprime donc une séquence de réduction.

Le séquent associé à la règle de remplacement offre aussi la possibilité de décrire une réduction simultanée de plusieurs réécriture comme le montre le terme de preuve suivant :

$$l_0(l_1([zero]_E, [zero]_E))$$

Deux réécritures ne peuvent être exécutées simultanément que lorsqu'elles s'appliquent sur des occurrences disjointes du terme. Ainsi, chaque séquent peut exprimer la réduction parallèle. Néanmoins, ce type de concurrence est de nature disjointe. Il est difficile de déterminer des positions disjointes si nous considérons le terme comme une structure distribuée. D'un point de vue logique, une architecture une-occurrence/un-processus ne peut être déduite facilement des règles de la logique de réécriture[1].

2.5.2 Structure sémantique pour modèles de concurrence

Programmation fonctionnelle parallèle

Les calculs fonctionnels, bien que susceptible de parallélisme, sont néanmoins déterminant, dans le sens que le résultat final d'une expression fonctionnelle c'est une valeur unique - s'il existe - calculé selon la composition de fonctions décrites dans l'expression. Nous pouvons distinguer, entre des langages fonctionneles premier ordre, aussi appelés langages équational et les langages d'ordre supérieur qui sont typiquement basés sur le lambda-calcul typé ou non typé, pour que les fonctions puissent être définies par des lambda-expressions, où le lambda-calcul en question satisfait aussi la propriété Church-Rosser[29].

Bien sûr, la programmation fonctionnelle premier ordre peut être vue comme un cas spécial dans lequel les règles se récrivant sont Church-Rosser. De cette façon, une intégration continue de la programmation fonctionnelle parallèle dans plus de structure générale de la logique de réécriture est naturellement réalisée. Pour les fonctions d'ordre supérieur, la clef est que cette logique de réécriture permet de réécrire les axiomes équationales.

Systèmes de transition étiquetées

Un système de transition étiquetée représente une théorie de réécriture $\mathcal{R} = (\Sigma, E, L, R)$ dans lequel Σ c'est un ensemble de constants ; E ensemble vide, et les règles sont toutes de la forme $r : a \rightarrow b$ avec a et b des constantes dans Σ[29].

Machine Chimique Abstraite

Les transitions concurrentes dans la machine chimique abstraite(cham) de Berry et Boudol[5], sont vues comme des réactions chimiques qui peuvent arriver simultanément dans plusieurs points de la solution[29]. Donc cham spécifier que les classes d'une théorie de réécriture dans lesquelles les axiomes $E = ACI$ sont l'associativité et commutativité de multiset avec un opérateur d'union de -, - et ayant le multiset vide λ ; comme son identité.

Chaque cham peut être alors exprimée comme une théorie se réécriture $\mathcal{C}$ =(Σ, ACI, L, R).

Objets concurant

Dans un système orienté objet concurant, l'état concurant, appelé une configuration, a typiquement la structure d'un multiset, composé d'objets et des messages. Donc, nous pouvons voir la création des configurations par un opérateur binaire d'union de multiset que nous pouvons le représenter par la syntaxe suivante [29]

subsorts Object Msg < Configuration .

op – : Configuration Configuration → Configuration

[assoc comm id : null]

Où l'opérateur d'union des multiset ,– , est déclaré pour satisfaire les lois structurelles de l'associativité et la commutativité et avoir le null comme identité. La déclaration de subsort à la première ligne ci-dessus déclare que les états de l'objet et des messages sont des configurations de multiset de singleton.

Un objet dans un état donné est représenté comme un terme

$$< O : c | a_1 : v_1, ..., a_n : v_n >$$

Où O est le nom de l'objet ou l'identificateur, C est sa classe, a_i est les attributs de l'objet et les v_i est les valeurs correspondantes. L'ensemble de toutes les paires valeur-attribut d'un état d'objet est formé par application répétée de l'opérateur d'union binaire -, - qu'obéit aussi aux lois structurelles d'associativité, commutativité et l'identité.

Chaque classe est déclarée par la syntaxe :

$$Class\ C | a_1 : S_1, ..., a_n : S_n$$

C : nom de la classe, a_i des attributs, S_i : type de chaque attribut.

L'envoi dun message :

$$msg\ m : p_1, ..., p_n \rightarrow Msg$$

Exemple 2.3 *Une version simplifiée de problème de Milner de téléphone portable[12]*

$$Class\ Car|base : Oid, phone - book : Set[Oid]$$

Chaque classe Car à les attributs : base qui contrôle ce véhicule. phone-book l'ensemble des numéros téléphones de ce véhicule.

$$Class\ Base|cars : Set[Oid], center : Oid$$

Chaque Base à les attributs : cars l'ensemble des véhicules, contrôlées par cette base, centre qui contôle cette base.

$$Class\ Centre|control : PFun[Oid, Oid], bases : Set[Oid], cars : Set[Oid]$$

Chaque Centre à les attributs : bases l'ensemble des bases. Cars l'ensemble des véhicules, Control : table de correspondance entre les bases et les véhicules.

Le premier message est talk :

$$\begin{aligned} rl[talk] :< O : Car|base : O', phone - book : O'' + OS > \&none \\ \Rightarrow < O : Car| > \&(toO' : talk(O'')). \end{aligned}$$

Le deuxième message est switch :

$$\begin{aligned} crl[switch] :< O : Centre|control : PF, bases : O' + O''' + OS, \\ cars : O'' + OS' > \&none \\ \Rightarrow < O : Centre|control : PF[O'' \rightarrow O'] > \&(toO' : alert(O'')). \\ (to\ O''' : release\ O''\ to\ O')\ if\ PF[O''] == O''' \end{aligned}$$

Le dernier message est :

$$\begin{aligned} rl[release] :< O : Base|cars : O' + OS > (toO : release\ O'\ to\ O'')\&none \\ \Rightarrow < O : Base|cars : OS > \&(to\ O' : switch(O'')) \end{aligned}$$

2.5.3 MAUDE

Maude[26][25] est un langage haut niveau. Très performant pour la construction des applications basées sur la logique équationnelle et la logique de réécriture. L'interprétations calculatoire et logique de la logique de réécriture sont comme deux côtés de la même pièce de monnaie, les mêmes raisons font une bonne structure sémantique au niveau calculatoire et aussi une bonne structure logique au niveau logique, c'est-à-dire une métalogique dans laquelle beaucoup d'autres logiques peuvent être naturellement représentées et mises en oeuvre [25]. Par conséquent, certaines applications intéressantes écrite en Maude sont des applications meta-langage, dans lesquelles Maude est utilisé pour créer des environnements exécutables pour différentes logiques, prouver des théorèmes, langage et modèles de calculs.

Caractéristiques de Maude [26] :

- *Basé sur la logique de réécriture.* Où les programmes sont des théories, et les règles de déduction logique de réécriture correspond exactement au calcul concourant.
- *Spectre large(wide-spectrume).* La logique de réécriture est une structure sémantique et logique dans laquelle la spécification, rapide et efficace pour l'exécution parallèle et distribuée, aussi bien que la transformation formelle des spécifications des programmes.
- *Multi-paradigme* Toutes les applications basée sur la logique équationnelle sont supportées facilement dans la logique de réécriture, donc la programmation fonctionnelle style équationnelle est naturellement représenté par un sous-langage. Le style déclaratif de programmation concurante, orientée-objets est aussi supportés avec une sémantique logique simple. La logique de HORN aussi est supporté par la logique de réécriture avec des facilités pour l'unification.
- *Réflective* Une caractéristique fondamentale de la logique de réécriture est d'être *réflective*. Il exite une théorie finie de réécriture $\mathcal{U}$ dite universelle. Cela signifie qu'il existe
  - une représentation finie $\overline{\mathcal{R}}$ sous forme de terme de $\mathcal{U}$ de toute théorie finie $\mathcal{R}$
  - une représentation finie $\overline{t}$ sous forme de terme de $\mathcal{U}$ de tout terme t fini de $\mathcal{R}$
  - une représentation $\langle\overline{\mathcal{R}}; t\rangle$ de tout couple ($\mathcal{R}$,t) vérifiant ($\mathcal{R} : t \rightarrow t'$) $\Longleftrightarrow$ ($\mathcal{U}$: $\langle\overline{R}; \overline{t}\rangle \rightarrow \langle\overline{R}; \overline{t}'\rangle$)
- *Stratégies internes.* Cette propriété permet de guider le processus de réécriture dont la sémantique peut être définie à l'intérieur de la logique par des règles de réécriture.

Exemple 2.4 *(Valeurs booléennes)*

Cet exemple, définit les valeurs Booléennes et quelques opérations logiques sur eux (négation, conjonction et disjonction). Pour construire les données ; on a deux constantes différentes, true et false.

fmod $BOOLEAN$ **is**
sort $Bool$.
op $true :\rightarrow Bool[ctor]$.
op $false :\rightarrow Bool[ctor]$.
op not_ : $Bool \rightarrow Bool$.
op _and_ : $Bool\ Bool \rightarrow Bool$.
op _or_ : $Bool\ Bool \rightarrow Bool$.
var $A : Bool$
eq not $true = false$.
eq not $false = true$.
eq $true$ and $A = A$.
eq $false$ and $A = false$.
eq $true$ or $A = true$.
eq $false$ or $A = A$.
endfm
L'attribut *ctor* pour indiquer les opérations employées comme des constructeurs.

Exemple 2.5 (*nombres naturels*)

Dans ce cas, il n'est pas pratique d'avoir un ensemble infini de constantes différentes pour dénoter les valeurs. Au lieu de cela, nous pouvons employer un constant 0 et une fonction de successeur s.
fmod $BASIC - NAT$ **is**
sort NAT.
op $0 :\rightarrow NAT[ctor]$.
op $s : NAT \rightarrow NAT[ctor]$.
op $\_ + \_ : NAT\ \ NAT \rightarrow NAT$.
var $M\ N : NAT$
eq $0 + N = N$.
eq $s(M) + N = s(M + N)$.
endfm

CCS en MAUDE

Les auteurs des[24][39], ont montrés deux voies différentes dans lesquelles CCS de Milner peut être naturellement représenter dans la logique de réécriture. Une représentation traite essentiellement les transitions comme des règles de réécriture, avec quelque précision syntactique pour enregistrer dans le terme les actions qui ont été exécutées. L'autre représentation considère les règles sémantique opérationnelles de CCS comme des règles de réécriture d'une théorie de réécriture et fournit une représentation plus déclarative.
fth $LABEL$ **is**
sort $Label$.

op $\tilde{}\_ : Label \rightarrow Label.$
op N : $Label.$
eq $\tilde{}\tilde{}N = N.$
endft
fmod $ACTION[X : LABEL]$ **is**
sort Act.
subsort $Label < Act.$
op $tau :\rightarrow Act.$
endfm
fth $PROCESSUSID$ **is**
sort ProcessusId.
endft
*** Syntaxe de CCS
fmod $PROCESSUS[X :: LABEL, Y :: PROCESSUSID]$ **is**
protecting $ACTION[X]$
sort $Process.$
subsort $ProcessId < Process.$
op $0 :\rightarrow Process.$ *** processus inerte.
op $\_.\_ : Act\ \ Process \rightarrow Process.$ *** préfixe.
op $\_ + \_ : Process\ Process \rightarrow Process.$ $[assoc\ comm\ idem\ id : 0]$
op $\_|\_ : Process\ \ Process \rightarrow Process.$ $[assoc\ comm\ idem\ id : 0]$
op $\_[\_/\_] : Process\ Label\ Label \rightarrow Process.$
op $\_\&\_ : Process\ Label \rightarrow Process.$
endfm
*** transition de CCS
mod $CCS[X :: LABEL, Y :: PROCESSUSID, C :: CCS - CONTEXTE * [X, Y]]$ **is**
sort $Configuration.$ $***multiensemble\ de\ transitions.$
op $(\_ : \_?\_) : Act\ Process\ Process \rightarrow Configuration.$
op $empty :\rightarrow Configuration.$
op $-- : Configuration\ Configuration \rightarrow Configuration\ [assoc\ comm\ id : empty].$
var $L\ M : Label.$
var $A : Act.$
vars $P\ P\ Q\ Q : Process.$
var $X : ProcessId.$

rl empty
$\Rightarrow$——————— ****Préfixe
$(A : (A.P) \rightarrow P)$
rl $(A : P) \rightarrow P')$

$\Rightarrow$——————****sommation.
$(A : P + Q \rightarrow P')$
rl $(A : P \rightarrow P')$
$\Rightarrow$—————— ****composition.
$(A : P|Q \rightarrow P'|Q)$
rl $(L : P \rightarrow P')(\sim L : Q \rightarrow Q')$
$\Rightarrow$————————————
$(tau : P|Q \rightarrow P'|Q')$
crl $(A : P \rightarrow P')$
$\Rightarrow$—————— ****réstricution.
$(A : P\&L \rightarrow P'\&L)$
if not ((A==L) **and** (A== ∼L)).
endm

2.5.4 Système temps-réel

La première contribution de la recherche dans le domaine de la logique de réécriture est la spécification des applications temps réel[32]. Ce travail a été initier par Kosiuczenko et Wirsing sur la logique de réécriture avec le temps (TRL) [29], une extension de la logique de réécriture où la relation de réécriture est étiquetée par le temps. Les axiomes dans TRL sont des séquences de la forme $t \xrightarrow{r} t'$. Leur signification intuitive est que t se développe à t' dans un temps r. les règles de déduction standard de la logique de réécriture sont étendues, avec des exigences de temps, permettre seulement la déduction dans laquelle toutes les parties différentes d'un système se développent dans le même temps. En fait, les spécifications de réécriture orientées objet dans le langage Maude ont une extension naturelle à la spécification TRL orientée objet dans Timed Maude [32].

Bien qu'il est possible de considérer la logique de réécriture comme un cas particulier de TRL dans lequel toutes les règles prennent le temps zéro, Peter Olveczky et José Meseguer [32] ont examiné une alternative différente, à savoir, en employant la logique de réécriture standard pour directement spécifier des systèmes temps-réel. L'idée de base est de mettre le temps explicitement, comme un additif monoïde qui agit sur les états du système selon des règles de réécriture.

Essentiellement, nous avons besoin d'inclure dans la spécification un type de données Time satisfaisant les propriétés équationnelle. Cependant, il est parfois utile de mettre en évidence l'aspect temps réel en faisant apparaître l'information de durée d'une façon explicite pour certains règles de réécriture. Cette idée a été formalisée au moyen d'une théorie de réécriture temps réel, P.Olveczky et J.Meseguer ont montré que, en ajoutant une horloge explicite, ils sont réductibles à une théorie de réécriture ordinaire, en préservant toutes les propriétés attendues. Les systèmes temps-réel peuvent être spécifiés dans la logique de réécriture ordinaire et que leur comportement n'exige pas de système d'inférence spécial, comme celui proposé dans TRL [32].

2.5.5 Modèles de temps et théorie de réecriture temps réel

P.Olveczky propose dans[32] une notion générale de théorie de réécriture temps réel, consistant d'une ordinaire théorie de réécriture, où les règles de réécriture affectant le système entier ont associé des expressions de durée-temps. Il a montré aussi que cette logique de réécriture temps-réel peut être réduite à une théorie de réécriture ordinaire, en ajoutant un horloge explicite à l'état global de la façon qui préserve tout les propriétés attendus.

Model de temps

Le temps($Time$) est modélisé abstraitement par un monoïde commutatif($Time$, +, 0) avec des opérateurs complémentaires $\leq$, $<$ et - (minus) satisfaisant la théorie de Maude suivante

fth $TIME$ **is**
protecting $Bool$
sort $Time$
op $0 : \rightarrow Time$
op _+_ : $Time\ Time \rightarrow Time$ [$assoc\ comm\ id : 0$]
ops _<_, _≤_ : $Time\ Time \rightarrow Bool$
op _- _ : $Time\ Time \rightarrow Time$
vars $x_r, y_r, z_r, w_r : Time$
cep $y_r = z_r$ if $x_r + y_r == x_r + z_r$
eq $(x_r < x_r) = false$
ceq $x_r < z_r = true$ if $x_r < y_r$ and $y_r < z_r$
eq $(xr \leq yr) = (xr < yr)$ or $(x_r == y_r)$
eq $0 \leq x_r = true$
ceq $x_r + y_r \leq z_r + w_r = true$ if $x_r \leq z_r$ and $y_r \leq w_r$
eq $x_r \leq (x_r - y_r) + y_r = true$
ceq $(x_r - y_r) + y_r = x_r$ if $y_r \leq x_r$
ceq $x_r - z_r \leq y_r - z_r = true$ if $x_r \leq y_r$
endft

Le temps linéaire peut être spécifié selon la théorie suivante :

fth $LTIME$ **is**
including $TIME$
op min : $Time\ Time \rightarrow Time$ [$comm$]
vars $x_r, y_r : Time$
ceq $x_r = y_r$ if not $(x_r < y_r)$ and not $(y_r < x_r)$
ceq $min(x_r, y_r) = y_r$ if $y_r \leq x_r$
endft

Théories de réécriture temps-réel

Les règles dans la théorie de réécriture temps réel sont divisées en des règles tick, et des règles instantanées. Une règle [tick] : $t \rightarrow t'$ pourra mener à la réécriture $f(t,u) \rightarrow f(t',u)$, où le temps s'écoule seulement dans une partie du système. Pour assurer que le temps avances uniformément dans toutes les parties d'un état, ils ont introduit un nouveau sorte *System*, sans sous-sortes, et un constructeur $\{-\} : State \rightarrow System$ avec la signification que $\{t\}$ dénote le système entier, qui est dans l'état t. Le temps uniforme s'écoule est alors assuré si l'état global a toujours la forme $\{t\}$ et chaque règle tick est de la forme $[tick] : \{t\} \rightarrow \{t'\}$.

Définition 2.26 *Une théorie de réécriture temps réel $\mathcal{R}_{\phi,\tau}$ est un tuple ($\mathcal{R}$, ϕ, τ) où $\mathcal{R}$ = (Σ, E, L, R) c'est une théorie de réécriture tel que :*

- – *ϕ est une théorie équationnelle morphisme $\phi : TIME \rightarrow (\Sigma, E)$ où $TIME$ est la théorie définie dans la section précédente.*
  – *Le domaine de temps est fonctionnel, c'est-à-dire chaque fois que $\alpha : r \rightarrow r'$ est un terme de preuve dans $\mathcal{R}$ et r est un terme de sorte $\phi(Time)$, alors $r = r'$ et α est équivalent à la preuve d'identité r,*
  – *(Σ, E) contient une sorte désigné, que nous appelons State et une sorte spécifique System sans sous-sortes ou des super-sortes et avec seulement un opérateur*

$$\{-\} : State \rightarrow System$$

et chaque $f : s_1...s_n \rightarrow s$ dans Σ, le sorte System n'apparaît pas parmi les sortes $s_1, ..., s_n$,
  – *τ est une nomination d'un terme $\tau_1(x_1, ..., x_n)$ de sorte $\phi(Time)$ à chacune des règles de réécriture dans R de la forme*

$$(\Psi) \qquad [l] : u(x_1, ..., x_n) \rightarrow u\prime(x_1, ..., x_n) \quad if \quad C(x_1, ..., x_n)$$

Où u et u' sont les termes de sorte *System*.

Nous appelons les règles de la forme (Ψ) des règles globales. Une règle globale l est une règle tick si sa durée $\tau_l(x_1, ..., x_n)$ diffère 0_ϕ pour quelque instances de ces variables, Et est une règle instantanée autrement. Les règles qui ne sont pas de la forme (Ψ) sont appelés des règles locales, parce qu'ils n'agissent pas sur le système dans l'ensemble, mais seulement sur quelques composants de système. Les règles locales sont toujours vues comme des règles instantanée qui prennent le temps zéro.

Le temps total qui s'écoule $\tau(\alpha)$ de la réécriture $\alpha : \{t\} \rightarrow \{t'\}$ de sorte *System* est défini comme la somme du temps écoulé dans chacune règle tick application α.

2.5.6 Théorie temps réel intériorisée dans la logique de réecriture

En ajoutant une horloge à l'état, une théorie de réécriture temps réel ($\mathcal{R}$, ϕ, τ) peut être transformé dans une logique de réécriture ordinaire, sans perdre l'information de temps[32].Un état dans un système est chronométré, s'il a la forme $< t, r >$ avec l'état t global de sorte $System$ et r une valeur de sorte $Time_\phi$, qui dénote le temps total écoulé dans un calcul si dans l'état initial l'horloge avait la valeur 0_ϕ

Définition 2.27 *l'opérateur d'intériorisation (_)C de la catégorie RTRWTh de la logique de réécriture temps réel à la catégorie RWTh de la logique de réécriture prend une logique de réécriture temps réel($\mathcal{R}$, ϕ, τ) à une logique de réécriture* $\mathcal{R}^C_{\phi,\tau} = (\Sigma^C_{\phi,\tau}, E^C_{\phi,\tau}, L^C_{\phi,\tau}, R^C_{\phi,\tau})$ *comme suit :*

- *Les sortes dans* $\Sigma^C_{\phi,\tau}$ *sont ceux dans* $\mathcal{R}$ *, avec un nouveau sorte ClockedSystem,*
- *Les opérations dans* $\Sigma^C_{\phi,\tau}$ *sont ceux dans* $\mathcal{R}$*, avec un nouveau constructeur libre*

$$< -, - >: System\ Time_\phi \rightarrow ClockedSystem,$$

- *Les axiomes dans* $E^C_{\phi,\tau}$ *sont inchangé de ceux dans* $\mathcal{R}$*,*
- $R^C_{\phi,\tau}$ *Contient les règles locales dans R de sortes d'autre que System, avec une règle*

$$[L^C_{\phi,\tau}(x_1, ..., x_n, x_r)] :< u(x_1, ..., x_n, x_r) > \rightarrow \\ < u'(x_1, ..., x_n), x_r +_\phi \tau_1(x_1, , x_n) > \text{ if } C(x_1, ..., x_n)$$

Pour chaque règle

$$[l(x_1, ..., x_n)] : u(x_1, ..., x_n) \rightarrow u'(x_1, ..., x_n) \text{ if } C(x_1, ..., x_n)$$

Dans $\mathcal{R}$ de sorte System, où x_r est une variable de sorte $Time_\phi$ qui n'est pas dans la liste $x_1, ..., x_n$.

2.6 Conclusion et discussion

Dans ce chapitre nous avons exploré plusieurs formalismes qui adapte les besoins de la programmation répartie et mobile, nous avons vu que le formalisme CCS c'était le point de départ, il n'inclu pas la notion de mobilité. π-calcul : un calcul de processus algébrique avec la mobilité. Il y a eu un nombre des formalismes qui traitent la mobilité, mais n'ont pas développé leurs théories algébriques spécifiques. Beaucoup de particularités du π-calcul sont dues a eux, $\pi-$calcul présente une théorie algébrique complète.

Le join-calcul hérite la plupart des propriétés de π-calcul asynchrone. La différence essentielle entre les deux, est que dans le join-calcul on connaît statiquement tous les récepteurs

a un nom de canal donné. Le comportement de synchronisation de noms est entièrement déclaré comme des join-patterns, quand les noms sont présentés dans un processus de définition, compilés dans l'ensemble.

La logique de réécriture représente une méthode générale pour la spécification des systèmes temps réel, ces systèmes sont supposés distribués et peuvent exposer des calculs concurrants, dans lesquells plusieurs composants de système peuvent changer simultanément et indépendamment.

2.6.1 π-calcul et join-calcul

Nous nous intéressons aux calculs d'agents mobiles, c'est à dire la formalisation de programme s'exécutant en parallèle sur plusieurs machines et pouvant migrer d'une machine à l'autre, et plus précisément à des calculs remplissant les conditions : être suffisamment riche pour être proche d'un langage de programmation, être conçu de telle sorte qu'une implémentation distribuée soit aisée, se prêter facilement à l'écriture de preuves ou de systèmes de types.

Une notion cruciale pour les calculs distribués est ; la notion de communication et de synchronisation permettant de modéliser les interactions entre agents. Cette notion se traduit habituellement soit par des rendez-vous, comme dans CCS ou le π-calcul, soit par des messages asynchrones comme dans le join-calcul. L'envoi de messages entre agents peut, soit se faire directement, soit utiliser un troisième agent qui transporte le message et le libère à sa destination.

Dans son calcul de systèmes communiquant (CCS), Milner simplifie grandement le modèle en utilisant un médium idéalisé, qu'il nomme l'éther pour la communication entre agents. Les propriétés de l'éther permettent de modéliser la communication d'une manière uniforme, abstraite, qui passe le médium sous silence. Ainsi, deux processus communiquent par rendez-vous, cette étape de communication affectant simultanément l'émetteur et le récepteur. Intuitivement, le rendez-vous présuppose, que les deux agents sont adjacents, mais quand ce n'est pas réaliste il est toujours possible de réintroduire le médium comme un agent intermédiaire, et de décrire la communication entre l'émetteur et le récepteur comme une succession de rendez-vous intermédiaires. Le π-calcul constitue une amélioration significative en terme d'expressivité.

Dans la pratique, la programmation répartie est plus compliquée. Pour programmer un ensemble d'ordinateurs connectés en réseau, l'interface des systèmes d'exploitations fournit des primitives de plus bas niveau que la communication par canaux dans l'éther. En général, il s'agit d'envois de messages asynchrones d'un point fixe du réseau à un autre, de manière peu fiable, comme dans le protocole IP, des primitives de plus haut niveau rendent l'adressage plus commode - par exemple l'appel d'une procédure ou d'une méthode à distance masque le routage et l'encodage des données, la résolution dynamique de l'adressage permet de rediriger les messages d'un point à un autre - et fournissent davantage de garanties - par exemple, l'absence de duplication des messages, leur intégrité, ou la confirmation ultérieure de leur réception. Par ailleurs, ces mécanismes de communication ne sont pas intégrés aux langages

de programmation traditionnels, ils sont disponibles dans des bibliothèques de protocole ou dans des extensions, et se juxtaposant aux mécanismes de base du langage.

Il est bien sûr possible de ne pas considérer ces problèmes dans un langage de haut niveau fondé sur un calcul de processus, à condition d'implémenter son modèle de communication à partir de primitives de bas niveau. Outre d'évidents problèmes de performance, cette approche n'est pas satisfaisante parce que certains détails de l'implémentation doivent nécessairement être révélés au programmeur, qui ne peut alors plus raisonner uniquement sur son programme indépendamment d'une implémentation particulière. Le modèle abstrait perd alors beaucoup de son intérêt.

L'écart entre l'implémentation et le modèle cache par exemple le nombre de messages et de machines requis pour implémenter une opération élémentaire. Dès lors, l'efficacité d'un programme mesurée par exemple par le volume de messages émis sur le réseau ne peut pas se déduire facilement du code source.

Dans certaines situations relativement courantes, cet écart devient évident. Lorsque certaines machines sont plus lentes que d'autres, ou sont en panne, l'utilisation de ces machines ralentit ou fait échouer le calcul. Si l'utilisation de ces machines n'apparaît pas dans le langage, ce comportement devient incompréhensible.

Dans le cas particulier de CCS asynchrone, le problème provient de certains aspects dynamiques de la communication : Lorsqu'un message est émis sur un canal par une machine, et qu'il existe un récepteur en attente sur une autre machine, le message devrait en théorie arriver sur cette machine sans autre détour. Cependant, la localisation des récepteurs sur un canal donné varie en fonction des autres étapes de calcul, elle est une propriété globale dont chaque machine ne peut maintenir qu'une image approximative, par exemple par envoi de messages de mise à jour.

Par exemple si une machine attend un seul message sur un canal donné, plusieurs autres machines peuvent en même temps produire un message sur ce canal et tenter de le communiquer à cette machine. Dans ce cas, seul le premier message est reçu, tandis que les suivants doivent être réexpédiés à d'autres récepteurs potentiels. Outre la présence de nombreux messages inutiles, cela pose un sérieux problème en cas de panne d'une machine qui a reçu trop de messages : Si un processus reçoit un seul message, en théorie sa disparition peut causer la perte de plus un message, tandis que d'autres récepteurs reçoivent les autres messages émis sur ce canal. En pratique, avec l'implémentation esquissée ci-dessus, de nombreux messages peuvent être perdus.

Le join-calcul distribué, c'est un calcul proche au langage de programmation, et suffisamment riche pour exprimer les concepts liés aux interactions entre agents mobiles : communication, migration, pannes, contrôle. Du point de vue interaction entre localité et communication ou migration, le join-calcul c'est un calcul transparent, c'est à dire interaction inexistante.

Un autre aspect c'est que toute synchronisation présente dans le calcul soit locale. Par exemple un rendez-vous entre deux agents sur deux sites distincts est interdit. Le join-calcul ne fournisse aucune primitive de synchronisation distribuée. C'est la raison pour laquelle une

implémentation distribuée basée sur le π-calcul n'est pas immédiate, et loin d'être évidente.

Le join-calcul correspond au noyau d'un langage de programmation directement utilisable, en particulier il est implémentable de manière répartie, il préserve la plupart des propriétés formelles du π-calcul, mais il présente de meilleurs propriétés de localité au cours du calcul

Cette analyse souligne que le π-calcul ne nous paraît pas adéquat pour la programmation répartie, même s'il permet d'en étudier certains aspects. Le join-calcul vérifie l'implémentation pour tout programme, indépendamment de la localisation des processus.

Pour la logique de réécriture, le problème c'est que la règle de transivité crée des problèmes dans les relations de branchement telque la bisimulation.

2.6.2 Les ambients et le join-calcul

De point de vue interaction entre localité et communication ou migration : le join-calcul est complètement transparent, alors que les ambients sont très locaux. La localité des interactions dans le calcul des ambients rend complexe la migration à distance. En effet, il est nécessaire dans ce cas de spécifier intégralement le chemin que devra suivre l'ambient. De plus, aucune garantie n'est donnée que l'ambient arrivera à bon port. En effet, non seulement d'autres migrations peuvent modifier le chemin, mais celui-ci peut être équivoque : le calcul ne garantit pas l'unicité des noms d'ambients. Ainsi, la programmation distribuée distante repose intégralement sur le programmeur qui doit s'assurer que certains invariants sont satisfaits. C'est la raison pour laquelle le calcul des ambients ne remplit pas le rôle d'un langage de programmation.

Aussi la localité des interactions entre ambients n'est pas suffisante pour garantir une implémentation distribuée aisée : deux ambients sur deux machines distinctes peuvent avoir besoin de se synchroniser pour décider si une réduction peut avoir lieu, de l'autre coté join-calcul ne fournisse aucune primitive de synchronisation distribuée. Cependant, le calcul des ambients motive l'importance de donner une réelle signification à la localité. En effet, si l'on considère pas les pannes, tout terme du join-calcul distribué s'exécute indépendamment de l'endroit où il est.

Dans le join-calcul distribué, chaque canal est défini dans une unique location. Un message sur un canal donné est envoyé dans la location où le canal est défini, quelle que soit la position courante du message, donc la liaison entre message et définition est statique. Alan Schmitt définit dans sa thèse[38] , une extension du join-calcul distribué possédant des canaux dynamiques, afin de modéliser des comportements dépendant de la position des messages. Contrairement aux canaux statiques, un canal dynmique peut être defini dans plusieurs locations, et un message sur un canal dynamique est envoyé à la location englobante la plus proche qui défini ce canal. Ainsi, lors de la migration de location, la liaison entre messages dynamiques et leurs définitions peut changer. De nouvelles définitions pour des canaux dynamiques existant, ainsi que de nouveaux canaux dynamiques peuvent être crées lors de l'exécution d'un programme. Schmitt définit des synchronisations supplémentaires pour modéliser la liaison dynamique, mais sont tous sont purement locales.

Chapitre 3

Modèles algébriques de spécification des applications temps réel

Le besoin de spécifier formellement des systèmes dont le comportement dépend étroitement du temps n'est pas nouveau. La plupart des protocoles contiennent des mécanismes de temporisation essentiels pour la sûreté de fonctionnement. La retransmission de messages dans un protocole de liaison de données ou de transport en est un exemple évident.

Dès 1976, Merlin et Farber[34] proposent une extension temporelle des réseaux de Petri visant précisément à spécifier ces mécanismes de récupération d'erreur basés sur des temporisations.

Depuis lors, plusieurs nouveaux mécanismes de protocoles, ainsi que les facilités de services correspondantes, ont fait leurs apparitions et rendent ce besoin encore plus fondamental. Citons par exemple les transferts de données isochrones, le contrôle de débit d'émission, la resynchronisation de flux multimédia,

Vu l'importance du problème, la recherche d'une technique de description formelle apte à spécifier ces systèmes s'est intensifiée durant ces dernières années. La plupart des travaux portent sur l'extension de modèles existants et en particulier sur les algèbres de processus. Parmi ces travaux, portent spécifiquement sur l'extension de LOTOS[2][18][20][19][22].

Dans ce chapitre nous explorons deux extensions de LOTOS, le langage RT-LOTOS qui intègre les contraintes temporelles, et le langage D-LOTOS qui prend en charge les durées des actions.

3.1 Extension temporelle de LOTOS

Principales extensions temporelles à LOTOS La problématique de l'expression explicite du temps dans LOTOS a engendré une série d'approches différentes. Nous listons les principaux modèles ci-dessous :

- TIC-LOTOS (Quemada 1987, UPM - Espagne)
- LOTOS-T (Miguel 1992, UPM - Espagne)
- T-LOTOS et U-LOTOS (Bolognesi 1991, CNUCE - Italie)
- TLOTOS (Leduc 1992, ULG - Belgique)
- Time LOTOS et ET-LOTOS (Leduc, Leonard 1993-94, ULG - Belgique)
- TE-LOTOS (Leduc, Leonard, Quemada, Miguel et al., 1995)
- LOTOS/T (Nakata 1993, ES-Osaka - Japon)
- RT-LOTOS (Courtiat, de Camargo, Saidouni 1993, LAAS - France)
- E-LOTOS (ISO/IEC 15437 :2001)
- D-LOTOS (Saidouni, Courtiat 2003, LIRE - Algérie et LAAS - France)

Certaines de ces approches ont servi de prémisses et de base de réflexion aux approches qui ont suivi. Pour différencier ces approches, les aspects suivants sont considérés :

Choix du domaine temporel

1. Soit *discret* : les grandeurs temporelles sont définies sur les entiers naturels ($\mathbb{Z}$), une progression d'une unité de temps peut alors être transcrite par l'occurrence d'une action spécifique (fréquemment notée `tic`). Ceci facilite la définition formelle, car le modèle rapproche du modèle non temporisé, mais occasionne un fort risque d'explosion combinatoire.

2. Soit *dense* et *dénombrable* : les grandeurs temporelles sont définies sur les rationnels positifs $\mathbb{Q}^+$. Cela semble parfaitement convenir à la grande majorité des cas pratiques d'utilisation d'un langage de spécification formelle de systèmes temps-réel. Le modèle mathématique sous-jacent est par contre, plus complexe à définir et à mettre en œuvre dans le cadre de la vérification.

3. Soit *réel* : quelques rares modèles évoquent la possibilité de définir les grandeurs temporelles dans $\mathbb{R}^+$. Notons qu'alors, presque aucune technique de validation n'a été proposée.

Temporisation des actions Cette temporisation se fait soit par l'ajout d'un opérateur temporel dédié («*opérateur* $< \ldots >$ P»), soit par une extension de l'opérateur de préfixe (préfixe d'un processus par une action : «$g < \ldots >; P$»), soit les deux.

Hypothèses d'urgence des actions Une action est dite *urgente* lorsqu'elle doit se réaliser immédiatement, sans progression possible du temps, dès qu'elle est sensibilisée. Certains modèles présupposent implicitement que toutes les actions sont urgentes. La plupart associent l'urgence aux actions internes (l'action i ou une action intériorisée par hide). Certains modèles introduisent un mot-clé spécifique pour déclarer urgentes des actions internes

ou observables. D'autres, enfin, proposent des mécanismes pour relâcher l'urgence des actions internes sous certaines conditions.

Opérateurs additionnels Certains langages proposent des facilités d'écriture pour spécifier des comportements réputés classiques par le biais d'opérateurs de haut niveau, qui toutefois n'introduisent pas véritablement de fonctionnalités nouvelles dans le modèle (en d'autres termes, les comportements spécifiés par ces opérateurs de haut niveau peuvent également être spécifiés au moyen d'opérateurs du modèle de base, mais de manière plus lourde).

Le tableau 3.1, page 77 reprend quelques points de comparaison entre les différentes extensions temporelles à LOTOS.

L'extension *Real-Time* LOTOS (RT-LOTOS)

RT-LOTOS [10][23], extension temporelle de LOTOS qui a été définie en considérant une sémantique d'entrelacement. Nous en évoquerons quelques aspects dans cette section, dans la perspective d'expliquer certains opérateurs temporels qui ont été repris dans D-LOTOS avant d'introduire celui-ci.

L'introduction de RT-LOTOS a été faite dans le but de permettre la description de l'aspect quantitatif (et non pas seulement de l'aspect qualitatif de l'ordonnancement des événements) des instants auxquels les événements se produisent réellement. L'extension temporelle RT-LOTOS reprend les concepts et l'essentiel des opérateurs de LOTOS, et apporte de nouveaux opérateurs permettant d'exprimer des contraintes temporelles quantifiées.

L'occurrence des actions peut être contrainte de la manière suivante :

- en retardant l'occurrence d'un processus de manière déterministe ;
- en retardant l'occurrence d'un processus de manière non-déterministe ;
- en limitant le temps pendant lequel une action est offerte à son environnement.

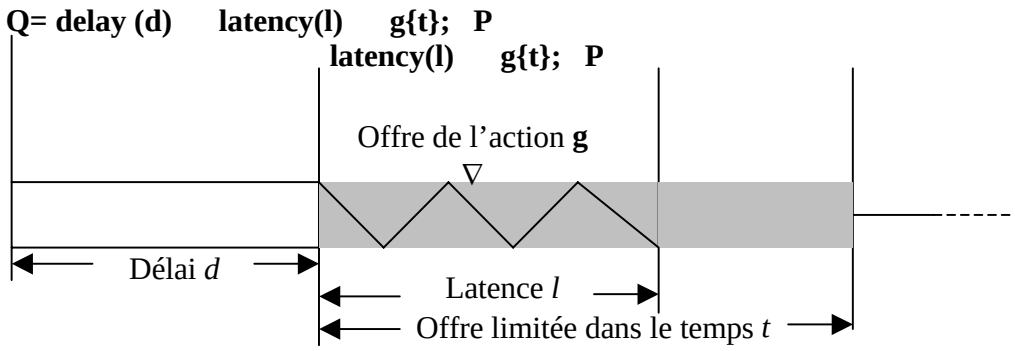
RT-LOTOS propose essentiellement trois opérateurs pour décrire, de manière intuitive, l'expression du temps dans le comportement de processus LOTOS.

- L'opérateur de délai (noté `delay`(d) ou Δ^d) permet de retarder un processus d'une certaine quantité de temps d.
- L'opérateur de latence (noté `latency`(l) ou Ω^l) permet de retarder un processus d'une certaine quantité de temps choisie de manière non-déterministe au sein de l'intervalle de latence $[0, l]$. Notons qu'en fait, cet opérateur porte sur la ou les premières actions du processus auquel il est appliqué. Par ailleurs, il n'a d'effet que s'il porte sur une action interne, l'occurrence d'une action observable étant, en tout état de cause, soumise à la date de l'offre faite par l'environnement. En d'autres termes, l'opérateur de latence

permet de relâcher la contrainte d'urgence d'une action interne. Il permet d'introduire de manière générale le non-déterminisme temporel.

- L'opérateur de restriction temporelle (noté $g\{t\}$) limite le temps pendant lequel une action observable g peut être offerte à son environnement. Le délai d'expiration commence à courir à partir du moment où l'action est offerte.

Présentation informelle des opérateurs de RT-LOTOS Considérons par exemple le processus $Q = delay(d)latency(l)g\{t\};P$ qui combine trois opérateurs. La figure suivante représente sur une ligne temporelle les contraintes spécifiées : dans l'intervalle $[0, d]$, aucune action n'est offerte ; à une date choisie dans l'intervalle $[d, d+l]$, l'action g est offerte ; l'action g est offerte jusqu'à la date $d + t$, après laquelle elle n'est plus offerte à son environnement.



Exemple de contraintes temporelles spécifiées avec RT-LOTOS

Syntaxe formelle et sémantique opérationnelle de Basic RT-LOTOS La syntaxe de Basic RT-LOTOS est donnée par la figure 3.1 .

$$
\begin{aligned}
P ::= \ & \texttt{stop} \mid \texttt{exit} \mid X[L] \mid i;P \mid g;P \\
& \mid P \ [] \ P \mid P \ |[L]| \ P \mid \texttt{hide}\ L\ \texttt{in}\ P \\
& \mid P \gg P \mid P \ [> P \\
& \mid \Delta^u P \mid \Omega^u P \\
& \mid i\{u\};P \mid g\{u\};P
\end{aligned}
$$

FIG. 3.1 – *Syntaxe de Basic RT-LOTOS*

Les notations suivantes seront utilisées lors de l'expression des différentes règles d'inférence de la sémantique opérationnelle :

- $P \xrightarrow{a}$ signifie que $\exists P'$ tel que $P \xrightarrow{a} P'$.

	Domaine temporel	**Temporisation**	**Urgence**
TIC-LOTOS	Discret	Préfixe et opérateur	Toute action
LOTOS-T	Discret ou dense	Préfixe	Actions cachées
T-LOTOS	Discret	Opérateur	Mot-clé spécifique
U-LOTOS	Discret	Opérateur	Mot-clé spécifique
TLOTOS	Discret	Opérateur	Relâchée
Time LOTOS	Dense	Opérateur	Actions cachées
ET-LOTOS	Dense	Préfixe et opérateur	Actions cachées
TE-LOTOS	Dense	Préfixe et opérateur	Toute action
LOTOS/T	Discret	Préfixe	Mot-clé spécifique
RT-LOTOS	Dense	Préfixe et opérateur	Actions cachées
E-LOTOS	Dense	Préfixe et opérateur	Actions cachées
D-LOTOS	Dense	Préfixe et opérateur	Actions cachées

TAB. 3.1 – *Comparaison des extensions temporelles à Lotos*

- $P \stackrel{a}{\not\rightarrow}$ signifie que le processus P ne peut pas réaliser l'action a.
- $P \stackrel{t}{\rightarrow} P'$ signifie que le processus P ne peut exécuter aucune action pendant une période de t unités de temps et se comporte ensuite comme le processus P'.

Afin de traiter les actions urgentes et non urgentes dans le modèle sémantique, deux actions sémantiques, notées g_w et g_s, sont associées à chaque action $g \in \mathcal{G}$ du modèle syntaxique :

- g_s, appelée action g forte (ou g *strong*),
- g_w, appelée action g faible (ou g *weak*).

Ces actions sémantiques présentent les caractéristiques d'urgence suivantes :

1. g_s et g_w ($g \in \mathcal{G}$) ne sont pas urgentes par définition dès lors que g est observable
2. i_s est urgente, de même que g_s ($g \in \mathcal{G}$) lorsqu'elle est cachée, ainsi que δ_s lorsqu'elle apparaît à droite de l'opérateur $\gg$
3. i_w n'est pas urgente, de même que g_w ($g \in \mathcal{G}$) lorsqu'elle est cachée, ainsi que δ_w lorsqu'elle apparaît à droite de l'opérateur $\gg$

L'*urgence* en RT-LOTOS est définie formellement par l'assertion suivante :

$$P \stackrel{i_s}{\rightarrow} \Rightarrow \forall t \neq 0,\ P \stackrel{t}{\not\rightarrow}$$

La sémantique opérationnelle d'entrelacement de Basic RT-LOTOS est définie dans le tableau suivant :

(1.*a*)	$exit \xrightarrow{\delta_s} stop$	(1.*b*)	$stop \xrightarrow{t} stop$
		(1.*c*)	$exit \xrightarrow{t} exit$
(2.*a*)	$g\{u\};P \xrightarrow{g_s} P \ (g \in \mathcal{G},\ u > 0)$	(2.*b*)	$g\{u+t\};P \xrightarrow{t} g\{u\};P \ (g \in \mathcal{G})$
		(2.*c*)	$g\{0\};P \xrightarrow{t} stop \ (g \in \mathcal{G})$
(3.*a*)	$i\{v\};P \xrightarrow{i_w} P \ (v > 0)$	(3.*c*)	$i\{u+t\};P \xrightarrow{t} i\{u\};P$
(3.*b*)	$i\{0\};P \xrightarrow{i_s} P$		
(4.*a*)	$\dfrac{P \xrightarrow{g} P'}{P[]Q \xrightarrow{g} P'} \ (g \in \mathcal{G}^{i,\delta})$	(4.*b*)	$\dfrac{P \xrightarrow{t} P' \quad Q \xrightarrow{t} Q'}{P[]Q \xrightarrow{t} P'[]Q'}$
(5.*a*)	$\dfrac{P \xrightarrow{g_x} P' \quad Q \xrightarrow{g_y} Q'}{P\|[L]\|Q \xrightarrow{g_{x \wedge y}} P'\|[L]\|Q'} \ (g \in L \cup \{\delta\})$	(5.*c*)	$\dfrac{P \xrightarrow{t} P' \quad Q \xrightarrow{t} Q'}{P\|[L]\|Q \xrightarrow{t} P'\|[L]\|Q'}$
(5.*b*)	$\dfrac{P \xrightarrow{g} P'}{P\|[L]\|Q \xrightarrow{g} P'\|[L]\|Q} \ (g \in \mathcal{G}^{i,\delta} \setminus L)$		
(6.*a*)	$\dfrac{P \xrightarrow{g} P' \quad (g \in \mathcal{G}^{i,\delta} \setminus L)}{hide\ L\ in\ P \xrightarrow{g} hide\ L\ in\ P'}$	(6.*c*)	$\dfrac{P \xrightarrow{t} P' \quad P \not\xrightarrow{g_s} \ (\forall g \in L)}{hide\ L\ in\ P \xrightarrow{t} hide\ L\ in\ P'}$
(6.*b*)	$\dfrac{P \xrightarrow{g_x} P' \quad (g \in L)}{hide\ L\ in\ P \xrightarrow{i_x} hide\ L\ in\ P'}$		
(7.*a*)	$\dfrac{P \xrightarrow{g} P'}{P \gg Q \xrightarrow{g} P' \gg Q} \ (g \in \mathcal{G}^{i})$	(7.*c*)	$\dfrac{P \xrightarrow{t} P' \quad P \not\xrightarrow{\delta_s}}{P \gg Q \xrightarrow{t} P' \gg Q}$
(7.*b*)	$\dfrac{P \xrightarrow{\delta_x} P'}{P \gg Q \xrightarrow{i_x} Q}$		
(8.*a*)	$\dfrac{P \xrightarrow{g} P'}{P[>Q \xrightarrow{g} P'[>Q} \ (g \in \mathcal{G}^{i})$	(8.*d*)	$\dfrac{P \xrightarrow{t} P' \quad Q \xrightarrow{t} Q'}{P[>Q \xrightarrow{t} P'[>Q'}$
(8.*b*)	$\dfrac{Q \xrightarrow{g} Q'}{P[>Q \xrightarrow{g} Q'} \ (g \in \mathcal{G}^{i,\delta})$		
(8.*c*)	$\dfrac{P \xrightarrow{\delta} P'}{P[>Q \xrightarrow{\delta} P'}$		
(9.*a*)	$\dfrac{P_X[g_1/g'_1 \cdots g_n/g'_n] \xrightarrow{h} Q' \quad X[g'_1 \cdots g'_n] := P_X}{X[g_1 \cdots g_n] \xrightarrow{h} Q'} \ (h \in \{g_s, g_w \mid g \in \mathcal{G}^{i,\delta}\} \cup D)$		
(9.*b*)	$\dfrac{P \xrightarrow{g} P'}{P\phi \xrightarrow{\phi(g)} P'\phi} \ (\phi = [g_1/g'_1 \cdots g_n/g'_n] \text{ et } g \in \mathcal{G}^{i,\delta})$	(9.*c*)	$\dfrac{P \xrightarrow{t} P'}{P\phi \xrightarrow{t} P'\phi} \ (\phi = [g_1/g'_1 \cdots g_n/g'_n])$
(10.*a*)	$\dfrac{P \xrightarrow{g} P'}{\Delta^0 P \xrightarrow{g} P'} \ (g \in \mathcal{G}^{i,\delta})$	(10.*b*)	$\Delta^{u+t} P \xrightarrow{t} \Delta^u P$
		(10.*c*)	$\dfrac{P \xrightarrow{t} P'}{\Delta^0 P \xrightarrow{t} P'}$
(11.*a*)	$\dfrac{P \xrightarrow{g} P'}{\Omega^u P \xrightarrow{g_w} P'} \ (g \in \mathcal{G}^{\delta})$	(11.*d*)	$\dfrac{P \xrightarrow{t} P'}{\Omega^{u+t} P \xrightarrow{t} \Omega^u P'}$
(11.*b*)	$\dfrac{P \xrightarrow{i} P'}{\Omega^u P \xrightarrow{i} P'}$	(11.*e*)	$\dfrac{P \xrightarrow{t} P'}{\Omega^0 P \xrightarrow{t} P'}$
(11.*c*)	$\dfrac{P \xrightarrow{g} P'}{\Omega^0 P \xrightarrow{g} P'} \ (g \in \mathcal{G}^{i,\delta})$		

3.2 Le langage D-LOTOS

Dans cette section nous introduisons une approche intégrant à la fois contraintes temporelles et durées des actions dans un langage très proche syntaxiquement de ET-LOTOS[20][21], mais pour lequel les auteurs de[37] ont défini une sémantique de vrai parallélisme, appelée sémantique temporelle de maximalité. Pour mettre en évidence l'incompatibilité de la sémantique d'entrelacement avec l'attribution de durées aux actions, considérons l'exemple des deux expressions de comportement Basic LOTOS défnies par : $E = a; stop|||b; stop$ et $F = a; b; stop[]b; a; stop$. Tant que les durées des actions a et b sont nulles, les deux comportements paraissent identiques, cependant si nous considèrons que durée $(b) > 0$ et durée $(b) > 0$, il en découle que dans E l'exécution des actions 'a' et 'b' peut être faite dans un temps égal à max{durée(b), durée(a) }, alors que le temps minimal pour exécuter les deux actions 'a' et 'b' dans F est de durée(b).

3.2.1 Sémantique de maximalité

Dans cette section, nous introduisons la sémantique de maximalité de Basic LOTOS telle qu'elle a été définie en [37]. La syntaxe de Basic LOTOS est définie dans la page 15.

Principe de la sémantique de maximalité

La sémantique d'un système concurrent peut être caractérisée par l'ensemble des états du système et des transitions par lesquelles le système passe d'un état à un autre. Dans l'approche basée sur la maximalité, les transitions sont des événements qui ne représentent que le début de l'exécution des actions. En conséquence, l'exécution concurrente de plusieurs actions devient possible, c'est-à-dire que l'on peut distinguer exécutions séquentielles et exécutions parallèles d'actions.

Etant donné que plusieurs actions qui ont le même nom peuvent s'exécuter en parallèle (auto-concurrence), pour distinguer les exécutions de chacune des actions, un identificateur a été associer a chaque début d'exécution d'action, c'est-à-dire à la transition ou à l'événement associé. Dans un état, un événement est dit maximal s'il correspond au début de l'exécution d'une action qui peut éventuellement être toujours en train de s'exécuter dans cet état là. Associer des noms d'événements maximaux aux états nous conduit à la notion de configuration qui sera formalisée dans la définition 3.2.

Pour illustrer la maximalité et cette notion de configuration, considérons les expressions de comportement E et F introduites précédemment.

Dans l'état initial, aucune action n'a encore été exécutée, donc l'ensemble des événements maximaux est vide, d'où les configurations initiales suivantes associées à E et F : ${}_{\phi}[E]$ et ${}_{\phi}[F]$. En appliquant la sémantique de maximalité, les transitions suivantes sont possibles :

$$ {}_{\phi}[E] \xrightarrow{\phi_x^a}_{m\{x\}} [stop] \quad ||| \quad {}_{\phi}[b; stop] \xrightarrow{\phi_x^b}_{m\{x\}} [stop] \quad ||| \quad {}_{\{y\}}[stop] $$

x (resp y) étant le nom de l'événement identifiant le début de l'action 'a' (respectivement 'b'). Etant donné que rien ne peut être conclu à propos de la terminaison des deux actions 'a' et 'b' dans la configuration ${}_{\{x\}}[stop]|||{}_{\{y\}}[stop]$, x et y sont alors maximaux dans cette configuration. Notons que x est également maximal dans l'état intermédiaire représenté par la configuration ${}_{\{x\}}[stop]|||{}_{\phi}[b; stop]$. Pour la configuration initiale, associée à l'expression de comportement F, la transition suivante est possible :

$$ {}_{\phi}[F] \xrightarrow{\phi^{a}_{x}} {}_{m\{x\}}\ [b; stop] $$

Comme précédemment, x identifie le début de l'action 'a' et il est le nom du seul événement maximal dans la configuration $\{x\}[b; stop]$. Il est clair que, au vu de la sémantique de l'opérateur de préfxage, le début de l'exécution de l'action 'b' n'est possible que si l'action 'a' a terminé son exécution. Par conséquent, x ne reste plus maximal lorsque l'action 'b' commence son exécution ; l'unique événement maximal dans la configuration résultante est donc celui identifié par y qui correspond au début de l'exécution de l'action 'b'. L'ensemble des noms des événements maximaux a donc été modifié par la suppression de x et l'ajout de y, ce qui justifie la dérivation suivante :

$$ {}_{\{x\}}[b; stop] \xrightarrow{\{x\}^{a}_{y}} {}_{m\{y\}}\ [stop] $$

La configuration ${}_{\{y\}}[stop]$ est différente de la configuration ${}_{\{x\}}[stop]\ |||\ {}_{\{y\}}[stop]$, car la première ne possède qu'un seul événement maximal (identifié par y), alors que la deuxième en possède deux (identifiés par x et y). La configuration $\{y\}[stop]$ est différente de la configuration ${}_{\{x\}}[stop]\ |||\ {}_{\{y\}}[stop]$, car la première ne possède qu'un seul événement maximal (identifié par y), alors que la deuxième en possède deux (identifiés par x et y).

Sémantique de maximalité de Basic LOTOS

Définition 3.1 *L'ensemble des noms des évenements est un ensemble dénombrable noté $\mathcal{M}$. Cet ensemble est parcouru par $x, y, ...$ $M, N, ...$ dénotent des sous-ensembles finis de $\mathcal{M}$. L'ensemble des atomes de support Act est Atm=$2^{\mathcal{M}}_{fn} \times Act \times \mathcal{M}$,$2^{\mathcal{M}}_{fn}$ étant l'ensemble des parties finies de $\mathcal{M}$. Pour $M \in 2^{\mathcal{M}}_{fn}$, $x \in M$ et $a \in Act$, l'atome (M, a, x) sera noté M^{a}_{x} . Le choix d'un nom d'événement peut se faire de manière déterministe par l'utilisation de toute fonction get : $2^{\mathcal{M}} - \{\phi\} \longrightarrow \mathcal{M}$ satisfaisant $get(\mathcal{M}) \in \mathcal{M}$ pour tout $M \in 2^{\mathcal{M}} - \{\phi\}$.*

Définition 3.2 *L'ensemble $\mathcal{C}$ des configurations des expressions de comportement de Basic LOTOS est le plus petit ensemble défini par induction comme suit :*

- $\forall E \in \mathcal{B}, \forall M \in 2^{\mathcal{M}}_{fn}\ :\ {}_{M}[E] \in \mathcal{C}$
- $\forall P \in PN, \forall M \in 2^{\mathcal{M}}_{fn} :\ {}_{M}[P] \in \mathcal{C}$
- *si $\mathcal{E} \in \mathcal{C}$,alors hide L in $\mathcal{E} \in \mathcal{C}$*

- *si* $\mathcal{E} \in \mathcal{C}$,*et* $F \in \mathcal{B}$ *alors* $\mathcal{E} \gg F \in \mathcal{C}$
- *si* $\mathcal{E}, \mathcal{F} \in \mathcal{C}$, *alors* $\mathcal{E}$ *op* $\mathcal{F} \in \mathcal{C}$ $\qquad op \in \{[]\,, |||, ||, |\,[L]\,|, [>\}$
- *si* $\mathcal{E} \in \mathcal{C}$,*et* $\{a_1, ..., a_n\}, \{b_1, ..., b_n\}$ *deux sous ensembles de* $\mathcal{G}$ *alors* $E[b_1/a_1, ..., b_n/a_n] \in \mathcal{C}$

Etant donné un ensemble $M \in 2^{\mathcal{M}}_{fn}$, $_M[...]$ est appelée opération d'encapsulation.

Définition 3.3 *La fonction* $\psi : \mathcal{C} \longrightarrow 2^{\mathcal{M}}_{fn}$ *, qui détermine l'ensemble des noms des événements dans une configuration, est définie récursivement par :*

$$\begin{array}{lll} \psi\,(_M\,[E]) = M & \psi\,(\mathcal{E}\,[]\,\mathcal{F}) = \psi\,(\mathcal{E}) \cup \psi\,(\mathcal{F}) & \psi\,(\mathcal{E}|\,[L]\,|\mathcal{F}) = \psi\,(\mathcal{E}) \cup \psi\,(\mathcal{F}) \\ \psi\,(\text{hide } L \text{ in } \mathcal{E}) = \psi\,(\mathcal{E}) & \psi\,(\mathcal{E} \gg \mathcal{F}) = \psi\,(\mathcal{E}) & \psi\,(\mathcal{E}[> \mathcal{F}) = \psi\,(\mathcal{E}) \cup \psi\,(\mathcal{F}) \\ & \psi\,(\mathcal{E}\,[b_1/a_1, ..., b_n/a_n]) = \psi\,(\mathcal{E}) & \end{array}$$

Définition 3.4 *La relation de transition de maximalité* $\longrightarrow \subseteq \mathcal{C} \times Atm \times \mathcal{C}$ *est définie comme étant la plus petite relation satisfaisant les règles suivantes :*

1. $\dfrac{}{{}_M[exit] \overset{M^{\delta x}}{\rightarrow}\, _{\{x\}}[stop]}\, x = get(\mathcal{M})$
2. $\dfrac{}{{}_M[a;E] \overset{M^{ax}}{\rightarrow}\, _{\{x\}}[E]}\; x = get(\mathcal{M})$
3. (a) $\dfrac{\mathcal{E} \overset{M^{ax}}{\rightarrow} \mathcal{E}'}{\mathcal{F}[]\mathcal{E} \overset{M^{ax}}{\rightarrow} \mathcal{E}' \quad \mathcal{E}[]\mathcal{F} \overset{M^{ax}}{\rightarrow} \mathcal{E}'}$
4. (a) i. $\dfrac{\mathcal{E} \overset{M^{ax}}{\rightarrow} \mathcal{E}' \;\; a \notin L \cup \{\delta\}}{\mathcal{E}|[L]|\mathcal{F} \overset{M^{ax}}{\rightarrow} \mathcal{E}'[y/x]\; |[L]|\; \mathcal{F} \setminus M}\, y = get(\mathcal{M} - ((\psi(\mathcal{E}) \cup \psi(\mathcal{F})) - M)$

   ii. $\dfrac{\mathcal{E} \overset{M^{ax}}{\rightarrow} \mathcal{E}' \;\; a \notin L \cup \{\delta\}}{\mathcal{F}|[L]|\mathcal{E} \overset{M^{ax}}{\rightarrow} \mathcal{F} \setminus M\; |[L]|\; \mathcal{E}'[y/x]}\, y = get(\mathcal{M} - ((\psi(\mathcal{E}) \cup \psi(\mathcal{F})) - M)$

   (b) $\dfrac{\mathcal{E} \overset{M^{ax}}{\rightarrow} \mathcal{E}' \;\; \mathcal{F} \overset{M^{ay}}{\rightarrow} \mathcal{F} \quad a \in L \cup \{\delta\}}{\mathcal{E}|[L]|\mathcal{F} \overset{M \cup N^{ax}}{\rightarrow} \mathcal{E}'[z/x] \setminus N \;\; |[L]|\; \mathcal{F}'[z/y] \setminus M}\, z = get(\mathcal{M} - ((\psi(\mathcal{E}) \cup \psi(\mathcal{F})) - (M \cup N)))$
5. (a) $\dfrac{\mathcal{E} \overset{M^{ax}}{\rightarrow} \mathcal{E}' \;\; a \notin L}{hide\ L\ in\ \mathcal{E} \overset{M^{ax}}{\rightarrow} hide\ L\ in\ \mathcal{E}'}$

   (b) $\dfrac{\mathcal{E} \overset{M^{ax}}{\rightarrow_m} \mathcal{E}' \;\; a \in L}{hide\ L\ in\ \mathcal{E} \overset{M^{ix}}{\rightarrow} hide\ L\ in\ \mathcal{E}'}$
6. (a) $\dfrac{\mathcal{E} \overset{M^{ax}}{\rightarrow_m} \mathcal{E}' \;\; a \neq \delta}{\mathcal{E} \gg \mathcal{F} \overset{M^{ax}}{\rightarrow} \mathcal{E}' \gg \mathcal{F}}$

   (b) $\dfrac{\mathcal{E} \overset{M^{\delta x}}{\rightarrow} \mathcal{E}'}{\mathcal{E} \gg F \overset{M^{ix}}{\rightarrow}\, _{\{x\}}[F]}$
7. (a) $\dfrac{\mathcal{E} \overset{M^{ax}}{\rightarrow} \mathcal{E}' \;\; a \neq \delta}{\mathcal{E}[> \mathcal{F} \overset{M^{ay}}{\rightarrow}\; \mathcal{E}'[y/x][> \mathcal{F} \setminus M}\, y = get(\mathcal{M} - (\psi(\mathcal{E}) \cup \psi(\mathcal{F})) - M)$

(b) $\dfrac{\mathcal{E} \overset{M^{\delta x}}{\rightarrow} \mathcal{E}'}{\mathcal{E}[>\mathcal{F} \overset{M^{\delta y}}{\rightarrow} \mathcal{E}'[y/x][>_{\psi(\mathcal{F})-M}[stop]} y = get(\mathcal{M} - (\psi(\mathcal{E}) \cup \psi(\mathcal{F})) - M)$

(c) $\dfrac{\mathcal{F} \overset{M^{\delta x}}{\rightarrow} \mathcal{F}'}{\mathcal{E}[>\mathcal{F} \overset{M^{\delta y}}{\rightarrow}_{\psi(\mathcal{E})-M}[stop]\ [>\ \mathcal{F}'[y/x]} y = get(\mathcal{M} - (\psi(\mathcal{E}) \cup \psi(\mathcal{F})) - M)$

8. (a) $\dfrac{\mathcal{E} \overset{M^{a_x}}{\rightarrow} \mathcal{E}' \quad a \notin \{a_1,...,a_n\}}{\mathcal{E}[b_1/a_1,...,b_n/b_n] \overset{M^{a_x}}{\rightarrow} \mathcal{E}'[b_1/a_1,...,b_n/b_n]}$

(b) $\dfrac{\mathcal{E} \overset{M^{a}x}{\longrightarrow} \mathcal{E}' \qquad a=a_i\ (1 \leq i \leq n)}{\mathcal{E}[b_1/a_1,...,b_n/a_n] \overset{M^{bi}x}{\longrightarrow} \mathcal{E}'[b_1/a_1,...,b_n/a_n]}$

9. $\dfrac{P:=E \qquad {}_M[E] \overset{M^{a}x}{\longrightarrow} \mathcal{F}}{{}_M[P] \overset{M^{a}x}{\longrightarrow} \mathcal{F}}$

3.2.2 Introduction des durées et des contraintes temporelles

Soit $\mathcal{D}$ un ensemble dénombrable, les éléments de $\mathcal{D}$ désigne des valeurs temporelles. Soit $\mathcal{T}$ l'ensemble de toutes les fonctions $\tau : Act \longrightarrow \mathcal{D}$ telles que $\tau(i) = t(\delta) = 0$. τ_0 est la fonction constante définie par $\tau_0(a) = 0$ pour tout $a \in Act$.

La fonction de durée τ étant fixée, considérons l'expression de comportement $G = a; b; stop$. Dans l'état initial, aucune action n'est en cours d'exécution et la configuration associée est donc ${}_\phi[a; b; stop]$; à partir de cet état, la transition ${}_\phi[a; b; stop] \overset{\phi^{a_x}}{\longrightarrow}_{\{x\}} [b; stop]$ est possible. L'état résultant interprète le fait que l'action a est potentiellement en cours d'exécution. Selon la sémantique de maximalité, nous n'avons pas le moyen de déterminer si l'action a a terminé ou pas son exécution, sauf dans le cas où l'action b a débuté son exécution (le début de b dépend de la fin de a) ; ainsi, si b a débuté son exécution, nous pouvons déduire que a a terminé de s'exécuter. Nous pouvons ainsi constater que les durées d'action sont présentes de manière intrinsèque mais implicite dans l'approche de maximalité ; leur prise en compte de manière explicite va nous permettre de raisonner sur des propriétés quantitatives du comportement d'un système.

En prenant en compte la durée de l'action a, nous pouvons accepter la transition suivante : ${}_\phi[a; b; stop] \overset{\phi^{a_x}}{\longrightarrow}_{\{x:a:\tau(a)\}} [b; stop]$. La configuration résultante montre que l'action b ne peut débuter son exécution que si une durée égale à $\tau(a)$ s'est écoulée, cette durée ne représentant rien d'autre que le temps nécessaire à l'exécution de l'action a. Nous pouvons bien sûr également considérer les états intermédiaires représentant l'écoulement d'un laps de temps $t \leq \tau(a)$ par ${}_{\{x:a:\tau(a)\}}[b; stop] \overset{t}{\longrightarrow}_{\{x:a:\tau(a)-t\}} [b; stop]$; de telles configurations seront appelées par la suite configurations temporelles, ce qui nous amène à constater qu'une configuration générée par la sémantique de maximalité représente en fait une classe de configurations temporelles. La prise en compte explicite des durées d'action dans les algèbres de processus ne permet pas cependant a elle seule de spécifier des systèmes temps réel. Pour combler ce manque, les opérateurs classiques de délai similaires a ceux introduits dans des extensions temporelles de LOTOS sont utilisés, telles que ET-LOTOS ou RT-LOTOS, la sémantique de ces opérateurs étant bien entendu exprimée dans le contexte de maximalité.

Du fait que les actions ne sont pas atomiques, les contraintes temporelles concernent dans ce contexte le début d'exécution des actions et non pas l'exécution complète des actions. Le langage ainsi défini est appelé D-LOTOS, pour LOTOS avec durées d'action.

La syntaxe de D-LOTOS est définie comme suit :

$$\begin{aligned} E ::= \quad & stop \mid exit\{d\} \mid \Delta^d E \mid X[L] \mid g@t[SP];E \mid i@t\{d\};E \mid \\ & E[]E \mid E|[L]|E \mid \ hide\, L\, in\, E \mid E \gg E \mid E\,[>\, E \end{aligned}$$

Soient a une action (observable ou interne), E une expression de comportement et $d \in \mathcal{D}$ une valeur dans le domaine temporel. Intuitivement $a\{d\}$ signifie que l'action a doit commencer son exécution dans l'intervalle temporel $[0, d]$. $\Delta^d E$ signifie qu'aucune évolution de E n'est permise avant l'écoulement d'un délai égal à d. Dans $g@t[SP];E$ (resp. $i@t\{d\};E$) t est une variable temporelle mémorisant le temps écoulé depuis la sensibilisation de l'action g (resp. i)et qui sera substituée par zéro lorsque cette action termine son exécution.

Définition 3.5 *L'ensemble $\mathcal{C}_t$ des configurations temporelles est donné par :*

- $\forall E \in \mathcal{B}, \forall M \in 2^{\mathcal{M}\times Act\times\mathcal{D}}_{fn} \ : \ {}_M[E] \in \mathcal{C}_t$
- $\forall P \in PN, \forall M \in 2^{\mathcal{M}\times Act\times\mathcal{D}}_{fn} : \ {}_M[P] \in \mathcal{C}_t$
- *si* $\mathcal{E} \in \mathcal{C}_t$ *alors hide* L *in* $\mathcal{E} \in \mathcal{C}_t$
- *si* $\mathcal{E} \in \mathcal{C}_t$ *et* $F \in \mathcal{B}$ *alors* $\mathcal{E} \gg F \in \mathcal{C}_t$
- *si* $\mathcal{E}, \mathcal{F} \in \mathcal{C}_t$, *alors* $\mathcal{E}$ *op* $\mathcal{F} \in \mathcal{C}_t$ $\qquad op \in \{[]\,, |||, ||, |\,[L]\,|, [>\}$
- *si* $\mathcal{E} \in \mathcal{C}_t$ *et* $\{a_1, ..., a_n\}\,, \{b_1, ..., b_n\} \in 2^{\mathcal{G}}_{fn}$ *alors* $E[b_1/a_1, ..., b_n/a_n] \in \mathcal{C}_t$
- $\forall \mathcal{E} \in \mathcal{C}_t, \forall d \in \mathcal{D} : \Delta^d \mathcal{E} \in \mathcal{C}_t$
- ${}_{(x:g:d)}[E(t)] \in \mathcal{C}_t$

Définition 3.6 *L'opérateur d'écoulement de temps (.)d dans une configuration est définie récursivement par :*

$\phi^d = \phi$	$(\mathcal{E}[]\mathcal{F})^d = \mathcal{E}^d[]\mathcal{F}^d$
$(x : a : d')^d = x : a : d' - d$ *tel que* d'-$d = 0$ *si* d>d'	$({}_M[E])^d = {}_{M^d}[E]$
$(M \cup \{x : a : d'\})^d = Md \cup \{(x : a : d')^d\}$	$(\mathcal{E}\|[L]\|\mathcal{F})^d = \mathcal{E}^d\|[L]\|\mathcal{F}^d$
$(hide\ L\ in\ \mathcal{E})^d = hide\ L\ in\ \mathcal{E}^d$	$(\mathcal{E} \gg F)^d = \mathcal{E}^d \gg F$
$(\mathcal{E}[b_1/a_1, ..., b_n/a_n])^d = (\mathcal{E}^d[b_1/a_1, ..., b_n/a_n])$	$(\mathcal{E}[> \mathcal{F})^d = \mathcal{E}^d[> \mathcal{F}^d$
${}_{\{x:g:d'\}}[E(t)]^d = {}_{\{x:g:d'\}^d}[[t + d/t]E(t)]$	

3.2.3 Sémantique opérationnelle structurée de D-LOTOS

La fonction de durée τ étant fixée, la relation de transition temporelle de maximalité entre les configurations temporelles est notée $\rightarrow_\tau \subseteq \mathcal{C}_t \times Atm \cup \mathcal{D} \times \mathcal{C}_t$.

Processus *stop* Considérons la configuration ${}_M[stop]$. A la différence du processus *stop* de LOTOS, cette configuration représente des évolutions potentielles en fonction des actions indexées par l'ensemble M. L'évolution cesse dès que toutes ces actions terminent, ce qui est caractérisé au moyen du prédicat $Wait : 2_{fn}^{\mathcal{M}\times Act\times\mathcal{D}} \longrightarrow \{true, false\}$ défini sur tout $M \in 2_{fn}^{\mathcal{M}\times Act\times\mathcal{D}}$ par : $Wait(M) = \exists\, x : a : d \in M \ \ tel\ que\ d > 0$. Intuitivement $Wait(M) = true$ s'il existe au moins une action référencée dans M qui est en cours d'exécution. Ainsi, tant que $Wait(M) = true$, le passage du temps a un effet sur la configuration ${}_M[stop]$, d'où la règle sémantique ${}_M[stop] \xrightarrow{d}_\tau {}_{M^d}[stop]$ avec M^d donné par la définition 3.6.

Processus *exit* Considérons maintenant la configuration ${}_M[exit]$. La terminaison avec succès ne peut se produire qu'une fois les actions indexées par l'ensemble M ont terminé leur exécution, ce qui est conditionné par la valeur de $Wait(M)$ qui doit être égale à $false$ dans la règle 1. Les règles 2 et 3 expriment le fait que le temps attaché au processus *exit* ne peut commencer à s'écouler que si toutes les actions référencées par M sont terminées. La règle 4 impose que l'occurence de l'action δ ait lieu dans la période d, dans le cas contraire la terminaison avec succès ne se produira jamais.

1. $\dfrac{\neg Wait(M)}{{}_M[exit\{d'\}] \xrightarrow{M^\delta x}_\tau {}_{\{x:\delta:0\}}[stop]} \qquad x = get(\mathcal{M})$

2. $\dfrac{Wait(M^d)\ or\ (\neg Wait(M^d)\ and\ \forall\varepsilon{>}0.\ Wait(M^{d-\varepsilon})) \qquad d{>}0}{{}_M[exit\{d'\}] \xrightarrow{d}_\tau {}_{M^d}[exit\{d'\}]}$

3. $\dfrac{\neg Wait(M)}{{}_M[exit\{d'{+}d\}] \xrightarrow{d}_\tau {}_M[exit\{d'\}]}$

4. $\dfrac{\neg Wait(M)\ and\ d'{>}d}{{}_M[exit\{d\}] \xrightarrow{d'}_\tau {}_M[stop]}$

Opérateur de préfixage Les mêmes contraintes sont imposées à l'occurrence d'une action observable préfixant un processus que celles imposées à l'occurrence de l'action δ à partir de la configuration ${}_M[exit\{d\}]$. Avec l'hypothèse que les actions ne sont pas urgentes, nous considérons l'opérateur @ introduit dans ET-LOTOS. L'expression SP dans les règles suivantes représente un prédicat sur l'exécution de l'action g. Les règles 1, 2 et 5 montrent que la prise en compte de l'écoulement du temps dans E commence uniquement lorsque l'action g est sensibilisée ou a commencé son exécution. La règle 3 exprime le fait qu'une fois que l'action g est sensibilisée et que le prédicat SP est vrai à cet instant, l'action g peut commencer son exécution. L'expression de comportement E ne peut évidemment évoluer que si l'action g se termine, ce qui est exprimé par la règle 4. Il est à noter que le préfixage peut être soit par une action observable ou interne avec la condition que l'action interne i et de durée nulle $(\tau(i) = 0)$.

1. $\dfrac{Wait(M^d)\ or\ (\neg Wait(M^d)\ and\ \forall\varepsilon{:}0{<}\varepsilon{<}d\ Wait(M^{d-\varepsilon})) \qquad d{>}0}{{}_M[g@t[SP];E] \xrightarrow{d}_\tau {}_{M^d}[g@t[SP];E]}$

2. $\dfrac{\neg Wait(M) \qquad d{>}0}{{}_M[g@t[SP];E] \xrightarrow{d}_\tau {}_M[g@t[[t{+}d/t]SP];[t{+}d/t]E]}$

3. $\dfrac{\neg Wait(M)\ and \vdash [0/t]SP}{{}_M[g@t[SP];E] \xrightarrow{M^{g}x}_{\tau} {}_{\{x:g:\tau(g)\}}[E(t)]} \qquad x = get(\mathcal{M})$

4. $\dfrac{\neg Wait(x{:}g{:}d)\ and\ {}_{\{x:g:0\}}[[0/t]E] \xrightarrow{M^{a}x} \mathcal{E}}{{}_{\{x:g:d\}}[E(t)] \xrightarrow{M^{a}x}_{\tau} \mathcal{E}}$

5. $\dfrac{}{{}_{\{x:g:d'+d\}}[E(t)] \xrightarrow{d}_{\tau} {}_{\{x:g:d'\}}[[t+d/t]E(t)]}$

Les autres opérateurs La sémantique des opérateurs de délai, de choix, de composition parallèle, d'intériorisation, de séquencement, d'interruption, de renommage de portes et d'instanciation de processus est donnée par les règles 3a, 4(a)i, 4(a)ii, 4b, 5a, 5b, 6a, 7a, 7b, 7c, 8a et 9 de la définition 3.4, 81 dans lesquelles les configurations sont temporelles et la relation de transition est remplacée par $\rightarrow_\tau$, complétées par les règles suivantes :

$$\frac{}{\Delta^{d'+d}\mathcal{E} \xrightarrow{d}_{\tau} \Delta^{d'}\mathcal{E}} \qquad \frac{\mathcal{E} \xrightarrow{d} \mathcal{E}'}{\Delta^0\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}'} \qquad \frac{\mathcal{E} \xrightarrow{M^{a}x}_{\tau} \mathcal{E}'}{\Delta^0\mathcal{E} \xrightarrow{M^{a}x}_{\tau} \mathcal{E}'}$$

$$\frac{\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}' \qquad \mathcal{F} \xrightarrow{d}_{\tau} \mathcal{F}'}{\mathcal{E}\ []\ \mathcal{F} \xrightarrow{d}_{\tau} \mathcal{E}'[]\mathcal{F}'} \qquad \frac{\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}' \qquad \mathcal{F} \xrightarrow{d}_{\tau} \mathcal{F}'}{\mathcal{E}\ |[L]|\ \mathcal{F} \xrightarrow{d}_{\tau} \mathcal{E}'|[L]|\mathcal{F}'}$$

$$\frac{P{:=}E \qquad {}_M[E] \xrightarrow{d}_{\tau} \mathcal{F}}{{}_M[P] \xrightarrow{d}_{\tau} \mathcal{F}}$$

$$\frac{\mathcal{E} \xrightarrow{d} \mathcal{E}' \qquad \forall d'{<}d \quad \mathcal{E}^{d'} \not\xrightarrow{a}_{\tau} \qquad \forall a{\in}L}{hide\ L\ in\ \mathcal{E} \xrightarrow{d}_{\tau} hide\ L\ in\ \mathcal{E}'} \qquad \frac{\mathcal{E} \xrightarrow{M^{\delta}x}_{\tau} \mathcal{E}'}{\mathcal{E}{>>}F \xrightarrow{M^{i}x}_{\tau} {}_{\{x:i:0\}}[F]}$$

$$\frac{\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}' \qquad \mathcal{E} \not\xrightarrow{\delta}_{\tau}}{\mathcal{E}{>>}F \xrightarrow{d}_{\tau} \mathcal{E}'{>>}F} \qquad \frac{\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}' \qquad \mathcal{F} \xrightarrow{d}_{\tau} \mathcal{F}'}{\mathcal{E}[{>}\mathcal{F} \xrightarrow{d}_{\tau} \mathcal{E}'[{>}\mathcal{F}'}$$

$$\frac{\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}'}{\mathcal{E}[b_1/a_1,...,b_n/a_n] \xrightarrow{d}_{\tau} \mathcal{E}'[b_1/a_1,...,b_n/a_n]} \qquad \frac{\mathcal{E} \xrightarrow{M^{a}x}_{\tau} \mathcal{E}' \qquad a{=}a_i\ (1{\leq}i{\leq}n)}{\mathcal{E}[b_1/a_1,...,b_n/a_n] \xrightarrow{M^{bi}x}_{\tau} \mathcal{E}'[x{:}bi{:}dbi/x{:}a{:}da][b_1/a_1,...,b_n/a_n]}$$

Ces règles sont similaires à celles introduites dans les algèbres de processus temps-réel.

La fonction de durée τ étant fixée, la relation de transition temporelle de maximalité entre les configurations temporelles est notée $\longrightarrow \subseteq \mathcal{C} \times Atm \cup \mathcal{D} \times \mathcal{C}$.

3.2.4 Relations de bissimulation

Dans cette section nous définissons quelques relations de bissimulation caractérisant le comportements d'applications concurrentes.

Relation de bissimulation temporelle

Définition 3.7 *Soit $\mathcal{L} = (2^{\mathcal{M}}_{fn} \times Act \times \mathcal{M}) \cup \mathcal{D}$ et $\boldsymbol{R} \subseteq \mathcal{C} \times \mathcal{C}$ une relation binaire entre les configurations temporelles. Soit $\boldsymbol{F}$: $Rel(\mathcal{C}) \longrightarrow Rel(\mathcal{C})$ une fonction définie par : $(E, F) \in \mathbf{F}(\mathbf{R})$ si :*

1. *(a) Si $\mathcal{E} \xrightarrow{M^{a}x}_{\tau} \mathcal{E}'$ avec ${}_{M^{a}x} \in 2^{\mathcal{M}}_{fn} \times Act \times \mathcal{M}$, alors il existe $\mathcal{F} \xrightarrow{N^{a}y}_{\tau} \mathcal{F}$ tel que $(\mathcal{E}', \mathcal{F}') \in \boldsymbol{R}$*

   (b) $\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}'$ avec $d \in \mathcal{D}$, alors il existe $\mathcal{F} \xrightarrow{d}_{\tau} \mathcal{F}'$ tel que $(\mathcal{E}', \mathcal{F}') \in \mathbf{R}$

2. *(a) Si $\mathcal{F} \xrightarrow{N^{a}y}_{\tau} \mathcal{F}\prime$ avec ${}_{N^{a}y} \in 2^{\mathcal{M}}_{fn} \times Act \times \mathcal{M}$, alors il existe $\mathcal{E} \xrightarrow{M^{a}_{x}}_{\tau} \mathcal{E}'$ tel que $(\mathcal{E}', \mathcal{F}) \in \boldsymbol{R}$*

(b) Si $\mathcal{F} \xrightarrow{d}_{\tau} \mathcal{F}'$ *avec* $d \in \mathcal{D}$, *alors il existe* $\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}'$ *tel que* $(\mathcal{E}', \mathcal{F}') \in \boldsymbol{R}$

$\boldsymbol{R}$ *est dite une relation de bissimulation temporelle forte ssi* $\boldsymbol{R} \subseteq \boldsymbol{F}(\boldsymbol{R})$. *Si* $(\mathcal{E}, \mathcal{F}) \in \mathbf{R}$ *pour une relation de bissimulation temporelle* $\boldsymbol{R}$, *alors* $\mathcal{E}$ *et* $\mathcal{F}$ *sont dites fortement temporellement bissimilaires, et on note* $\mathcal{E} \backsim_t^{\tau} \mathcal{F}$. *Ce qui peut être exprimé par* $\backsim_t^{\tau} = \cup\{\boldsymbol{R} : \boldsymbol{R}$ *est une relation de bissimulation temporelle forte}.*

Deux expressions de comportement D-LOTOS E et F sont dites fortement temporellement bissimilaires, noté $E \backsim_t^{\tau} F$ *, s'il existe une relation de bissimulation temporelle forte* $\mathbf{R}$ *tel que* $({}_{\phi}[E]_{,\phi}\,[F]) \in \mathbf{R}$.

Relation de bissimulation temporelle de maximalité

Dans[36] il a été montré que la relation de bissimulation de maximalité est une congruence vis à vis du rafinement d'actions, et par dualité vis à vis de l'association de durées aux actions (conjecture). Ceci a motivé à étendre la relation de bissimulation de maximalité aux configurations temporelles.

Définition 3.8 *Soient* $\mathcal{L} = (2^{\mathcal{M}}_{fn} \times Act \times \mathcal{M}) \cup D$, $\mathfrak{F} \subseteq 2^{\mathcal{M} \times \mathcal{M}}$) *et* $\boldsymbol{R} \subseteq \mathcal{C} \times C \times$ *Funerelationbinaireentr* $Rel(\mathcal{C}) \longrightarrow Rel(\mathcal{C})$ *une fonction définie par :* $(\mathcal{E}, \mathcal{F}, f) \in \boldsymbol{F}^{mt}(R)$ *si :*

1. $Dom(f) \subseteq \psi(\mathcal{E})$ *et* $Codom(f) \subseteq \psi(\mathcal{F})$,
2. *(a) Si* $\mathcal{E} \xrightarrow{M^{a_x}}_{\tau} \mathcal{E}'$ *, alors il existe* $\mathcal{F} \xrightarrow{N^{a_y}}_{\tau} \mathcal{F}'$ *tel que*
   - *i. pour tout* $(u, v) \in f$ *si* $u \notin M$ *alors* $v \notin N$; *et*
   - *ii.* $(\mathcal{E}', \mathcal{F}', f') \in \mathbf{R}$, *avec* $f' = (f(\psi(\mathcal{E}') - \{x\}))(\psi(\mathcal{F}') - \{y\}) \cup \{(x, y)\}$

   (b) Si $\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}'$ *, alors il existe* $\mathcal{F} \xrightarrow{d}_{\tau} \mathcal{F}'$*tel que* $(\mathcal{E}', \mathcal{F}', f) \in \mathbf{R}$
3. *(a) Si* $\mathcal{F} \xrightarrow{N^{a_y}}_{\tau} \mathcal{F}'$ *, alors il existe* $\mathcal{E} \xrightarrow{M^{a_x}}_{\tau} \mathcal{E}'$ *tel que*
   - *i. pour tout* $(u, v) \in f$ *si* $v \notin N$ *alors* $u \notin M$; *et*
   - *ii.* $(\mathcal{E}', \mathcal{F}', f') \in \mathbf{R}$, *avec* $f' = (f(\psi(\mathcal{E}') - \{x\}))(\psi(\mathcal{F}') - \{y\}) \cup \{(x, y)\}$

   (b) Si $\mathcal{F} \xrightarrow{d}_{\tau} \mathcal{F}'$ *, alors il existe* $\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}'$*tel que* $(\mathcal{E}', \mathcal{F}', f) \in \mathbf{R}$

$\mathbf{R}$ *est une relation de bissimulation temporelle de maximalité forte ssi* $\mathbf{R} \subseteq \mathbf{F}^{mt}(\mathbf{R})$. *Si* $(\mathcal{E}, \mathcal{F}, f) \in \boldsymbol{R}$ *pour une certaine relation de bissimulation temporelle de maximalité forte* $\boldsymbol{R}$, *alors les configurations* $\mathcal{E}$ *et* $\mathcal{F}$ *sont dites liées par cette relation, ce qui est noté symboliquement* $\mathcal{E} \backsim_{mt}^{\tau} \mathcal{F}$. *(*$(f[A)[B]$ *désigne la fonction résultante de la restriction du domaine de* f *à* A *de son codomaine à* B*).*

3.3 Limites du langage D-LOTOS

Les applications mobiles et distribuées, sont naturellement dynamique c'est à dire, le nombre d'agents qui intervient dans le système, n'est pas stable. A chaque instant il peut y avoir des nouvelles créations ou suppressions d'agents.

Pour pouvoir étudier ce type d'applications, il faut utiliser des modèles dynamiques c'est à dire, des modèles qui fournissent formellement des primitives et des outils pour gérer cette aspect. Dans le cas D-LOTOS, il n'offre pas des primitives de création dynamique de localités, ou des nouveaux emplacements de calculs.

Deuxième aspect dans ces applications, c'est le mécanisme de communication élémentaire entre agents. Traditionnellement la communication entre processus est modélisée par la transmission de message d'un processus émetteur vers un processus récepteur. Dans la pratique, la programmation répartie utilise la communication asynchrone, il s'agit d'envois de messages asynchrones d'un point fixe du réseau à un autre. D-LOTOS ne fournit aucune primitive de communication asynchrone. Par ailleurs, il fournit des primitives de communication synchrone. Donc une implémentation du D-LOTOS dans un cadre distribué est loin d'être évidente à cause de la synchronisation distribuée présente dans le calcul. Pour que le modèle soit directement implémentable dans un cadre parallèle, il est nécessaire que toute synchronisation présente dans le modèle soit locale. C'est à dire nous interdisons par exemple qu'un rendez-vous puisse avoir lieu entre deux agents sur deux sites distincts.

L'autre insuffissance dans le langage D-LOTOS, est la mobilté, on peut pas spécifier des applications mobile en D-LOTOS, a cause de l'absence de la notion de localité.

Cette analyse souligne que D-LOTOS n'est pas adéquat pour la programmation mobile, même s'il permet d'en étudier certains aspects.

3.4 Conclusion

Dans ce chapitre, nous avons exposé dans la première section l'extension temporelle RT-LOTOS. Ensuite dans la deuxième section, nous avons étudier le modèle de spécification D-LOTOS qui intègre deux concepts à savoir la spécification des contraintes temporelles et la prise en compte des durées d'action. D'un point de vue syntaxique, D-LOTOS restés très proche du formalisme RT-LOTOS, mais d'un point de vue sémantique les auteurs [37], abstrait de l'hypothèse d'atomicité des actions qui est imposée par la sémantique d'entrelacement du parallélisme. Dans ce but, la sémantique de maximalité temporelle a été adoptée pour D-LOTOS.

Chapitre 4

MD-LOTOS (mobil D-LOTOS) : Un modèle de spécification des applications temps réel et mobiles.

Ce chapitre présente un nouveau calcul de processus réparti, temps réel. Ce calcul constitue un modèle formel de spécification des applications temps réel, avec mobilité de processus. Dont la motivation est d'étendre le langage D-LOTOS afin de prendre en compte la mobilité dans les applications temps réel.

Dans un système réparti de grande taille, les localités peuvent représenter les agents, ou des machines sur lesquelles les agents migrent et s'exécutent. D'autre part dans les algèbres de processus temps réel (RT-LOTOS, ET-LOTOS, D-LOTOS, etc) étudier dans le chapitre 3, nous constatons qu'il n'y a pas de notion de localité. D'ou l'incapacité pour la spécification de la mobilité.

Notre calcul de processus vise a pallier la limitation de langage D-LOTOS, en fournissant un modèle de programmation réparti, temps réel qui autorise la programmation explicite des localités.

4.1 Présentation de MD-LOTOS

Notre objectif dans cette section est de développer un modèle inspiré a la fois du Join-calcul distribué et de langage D-LOTOS, tout en résolvant les difficultés associées à ces deux modèles qui sont décrites dans les sections 2.6, page 69 et 3.3, page 87.

Nous devons prendre en compte le mécanisme de création automatique des processus, et de localités qui est nécessaire dans les applications temps réel, et mobiles, cette propriété exprime le dynamisme du modèle. Les applications réparties sont de nature dynamique, c'est à dire on connaît pas au départ le nombre d'agents (processus, localités) qui intervient dans le système. Ici, on peut spécifier ce mécanisme par l'utilisation des définitions locales de Join-calcul, et en introduisant la notion de localité dans D-LOTOS.

Nous nous intéressons aux langages de processus qui représentent explicitement l'existence d'objets, nommés localités, qui abstraient les lieux physiques, comme les ordinateurs, ou logiques, comme les agents. Pour leur permettre de remplir tous ces rôles, les localités sont nommées et structurées selon une hiérarchie en arbre, c'est une manière de programmer les localités explicitements. Une localité $L0$ sera représentée par $\vdash_{L0}$. Situés à l'intérieur des localités, les processus exécutent des actions et sont les responsables de l'évolution globale du système. La hiérarchie des localités peut être modifiée par l'activité des processus : on parle de mobilité

Les localités déterminent la résistance aux pannes. Ainsi, chaque groupe de processus et de définitions peut migrer d'une machine à une autre, ou s'arrêter. En terme de langage de programmation, cela correspond à un modèle très expressif d'agents mobiles. Ces localités ont une structure hiérarchique, ce qui permet de modéliser les sites comme des localités particulières.

Dans le join-calcul, nous désirons conserver la facilité à communiquer à distance, en nous affranchissant de la nécessité de spécifier le chemin que le message doit parcourir pour parvenir à destination. Nous voulons également conserver la facilité d'implémentation, en évitant toute forme de synchronisation distribuée. Enfin, nous considérons que la notion de définitions sous forme de filtre de messages associé à un processus est une construction riche très utile à la programmation.

Du D-LOTOS, nous retenons les aspects liés au temps tel que durées des actions, les opérateurs expriment les contraintes temporelles.

Nous présentons notre calcul, à partir d'un exemple, qui représente une version simplifiée d'un protocole de communication sur un réseau local.

On suppose que notre système contient une source d'information, et deux récepteurs. La source envoie des messages sur le réseau pour les deux récepteurs, d'autre part, chaque récepteur retourne un acquittement négatif pour signaler la perte du message. La source envoie le message suivant si elle ne reçoit pas d'acquittement négatif aprés t unités de temps. L'état du calcul est représenté par deux sortes de termes, des messages et des règles.

Par exemple, nous modélisons l'interface de la source par deux noms, *send* pour les paquet d'informations envoyer par la source, $NACK$ pour les acquittements négatifs.

Dans la source on veut exprimer trois règles :

1. Si un message est présent sur le nom *send* un processus P est déclenché pour envoyer le message aux deux récepteurs en parallèle.

2. Si la source ne reçoit pas d'acquittement négatif des deux récepteurs aprés t unités de temps, un processus est déclenché pour envoyer le prochain message.

3. Si la source reçoit un acquittement négatif de l'un des deux récepteurs, ou les deux, elle doit déclencher la retransmission du message perdu.

L'interface des récepteurs est modélisé par un nom *paquet* pour les paquets réçus. Dans les récepteurs on veut exprimer la règle :

1. Concaténation des messages réçus.

Pour exprimer ces règles on utilise la règle de Join-calcul, $J \rhd P$, cette règle consomme un ensemble de messages de la forme décrite dans le filtre J, et déclenche l'exécution d'une copie du processus P dans lequel les paramètres formels de J sont remplacés par les arguments transmis dans ces messages. La même règle peut être utilisée à plusieurs reprises, tant qu'il y a des messages à consommer.

Nous définissons le comportement de la source par une règle qu'est l'union des trois règle, notée D, cette règle décrit comment les messages envoyés sur les noms *send*, $NACK$ sont traités :

$$D1 = send < m > \rhd P$$

Lorsque on reçcoit un message sur *send* le processus P est déclenché. Le processus P représente la duplication du message $paquet < m >$, autant que fois qu'on veut envoyer :

$$P \stackrel{def}{=} \textbf{def } k <> \rhd \ \ paquet < m > | k <> \quad \textbf{in } k <>$$

$D2 = k <> \rhd \Delta^t \ send < m\_suivant >$

$D3 = NACK < N > \rhd paquet < m >$

Le comportement des récepteurs est défini par la règle D', cette règle décrit comment est faites la concaténation des messages :

$$D' = paquet < m1 >, paquet < m2 > \rhd paquet < m1 \& m2 >$$

Chaque objet (source, récepteur) dans le système représente une localité du calcul. Ainsi le système est sous la forme :

$$D \vdash_{L0} \quad || \vdash_{L0L1} NACK < 1 > \quad || \vdash_{L0L2} paquet < m >$$

Ce système se décompose de trois localités, la source($\vdash_{L0}$), deux récepteurs($\vdash_{L0L1}, \vdash_{L0L2}$) connectés a la source. Dans ce système on à une perte de message dans le récepteur $L1$, et une bonne réception du message m dans le récepteur $L2$.

Nous pouvons regrouper la définition de la source et l'envoi du message dans un seul processus :

$$P \stackrel{def}{=} \textbf{def } D \textbf{ in } send < m >$$

Le nom $NACK$ est défini localement à la source, il doit être consommé localement. Pour cela on a besoin d'une règle de communication globale, qui transporte ce message de la localité de récepteur $L1$ à la localité de la source.

La première étape transmet le message $NACK < 1 >$ de la machine du récepteur $L1$ à la source, cette réduction a lieu parce que le nom $NACK$ est uniquement défini sur la source. Ensuite, le message d'acquittement négatif est maintenant sur la source, qui déclenche la

$$\begin{array}{lllll}
 & D \vdash_{L0} & ||D' \vdash_{L0L1} NACK < 1 > & ||D' \vdash_{L0L2} paquet < m > \\
\overset{Comm}{\rightarrow} & D \vdash_{L0} NACK < 1 > & ||D' \vdash_{L0L1} & ||D' \vdash_{L0L2} paquet < m > \\
\overset{RED}{\rightarrow} & D \vdash_{L0} paquet < m > & ||D' \vdash_{L0L1} & ||D' \vdash_{L0L2} paquet < m > \\
\overset{Comm}{\rightarrow} & D \vdash_{L0} & ||D' \vdash_{L0L1} paquet < m > & ||D' \vdash_{L0L2} paquet < m >
\end{array}$$

TAB. 4.1 – Protocole de communication simplifée

retransmission du message m adressé au récepteur L1. A nouveau, ce message est d'abord transmis à cette machine, puis traité localement.

Jusqu'à maintenant nous avons modéliser un système statique, c'est à dire on connaît au départ le nombre d'objets qui intervient dans le système. Le dynamisme c'est une caractéristique principale dans un système réparti. Donc on doit rajouter une primitive pour la création dynamique des récepteurs, de l'autre coté la suppression dynamique. Dans notre exemple nous voulons exprimer le cas où un nouveau récepteur se connecte à la source.

La définition $L3[\intercal : P]$ permet de créer une sous-localité, appelée $L3$ avec la définition vide et un processus P en cours d'exécution. Dans le comportement de la source on rajoute la définition suivante :

$$C = creation <>\triangleright\ X[D'' : P]$$

Cette définition permet de créer dynamiquement des récepteurs només $X \in \{L0, L1, ...\}$. Cette nouvelle règle doit être rajouter aux règles de la source donc :

$$R = D \cup C$$

$$\begin{array}{llll}
 & R \vdash_{L0} creation <> & ||D' \vdash_{L0L1} & ||D' \vdash_{L0L2} \\
\overset{création}{\rightarrow} & R \vdash_{L0} & ||D' \vdash_{L0L1} & ||D' \vdash_{L0L2} \quad ||D'' \vdash_{L0L3}
\end{array}$$

A partir de cet exemple nous décrivons les constructeurs de base de notre calcul :

- Nous rajoutons le constructeur pour les définitions locales, constructeur pour la migration des localités, et le message asynchrone.

- Pour éviter la synchronisation de deux agents sur deux localités distinctes, nous introduisons la notion de filtre qu'est local à une localité. Donc tout synchronisation est faites au niveau local.

- Les récepteurs associés aux messages sont tous simplement des définitions du Join-calcul, c'est à dire des récepteurs associant un processus à un filtre de message.

- Les différents emplacements qui apparaissent au cours du calcul sont structurés en arbre de localités. Chaque localité de l'arbre représente un emplacement (site) de calcul qui contient des définitions, et des processus en cours d'exécution. Donc chaque localité est de la forme :

$$\{D\} \vdash \text{chemin de la localité} \{P\}$$

Le modèle opère sur deux types de multi-ensembles, $\{D\}$ multi-ensemble de définitions, $\{P\}$ multi-ensemble de processus en cours d'exécution.

Le système complet c'est un ensemble de localités, tel que chaque localité est identifiée par un chemin représentant une chaîne qui permette de définir le chemin d'accès qui mène à cette localité dans l'arbre.

4.1.1 Traitement local et global

Dans notre calcul on a deux types de traitements, le traitement local qui se passe au niveau de chaque localité c'est à dire calcul local. Et le traitement global entre les localités qui peut être, soit une transmission d'un message d'une localité vers une autre, soit une migration de localité avec ces définitions et processus en cours d'exécution, soit la création d'une nouvelle localité, ou suppression de localité.

Traitement local

Dans un système réparti, l'application tourne sur plusieurs sites différents. Chaque site représente une localité. D'après notre exemple introductif chaque localité peut contenir des règles et des processus en cours d'exécution.

Chacune des localités évolue localement comme dans la machine abstraite de Join-calcul : certains messages locaux sont consommés par des définitions locales et remplacés par de nouveaux processus, tandis que les processus peuvent introduire de nouveaux noms avec leurs définitions.

Traitement global

Afin de modéliser la présence de plusieurs sites de calcul, chaque localité représente un calcul local, et une règle de communication globale entre emplacements : Ainsi, l'état du calcul est représenté par une famille de paires de multi-ensembles $\{Di, Pi\}$ qui contiennent respectivement les définitions locales et les processus en cours d'exécution dans chaque emplacement.

Par ailleurs, une règle supplémentaire décrit la communication globale : lorsqu'un message est émis dans un emplacement et que ce message est défini dans un autre emplacement, une étape de calcul transporte ce message de l'emplacement émetteur vers l'emplacement récepteur. Par la suite, ce message pourra être consommé localement, par la règle réduction, peut être avec d'autres messages.

Informellement, ce nouveau mécanisme de calcul reflète le routage des messages d'un point à l'autre du réseau.

4.1.2 Syntaxe et sémantique opérationnelle de maximalité

Nous définissons dans cette section la syntaxe et la sémantique du notre calcul.

Syntaxe

La syntaxe de notre modèle est une extension du langage D-LOTOS par l'introduction des mécanismes de mobilité de Join-calcul, c'est à dire nous retenons la notion de définition de canaux par des filtres de messages et la composition parallèle. Ainsi que le mécanisme de migration et la notion de calcul local.

La syntaxe du calcul est décrite en figure 4.1, page 94.

- Soit PN l'ensemble des variables de processus parcouru par X ,

- Soit $\mathcal{G}$ l'ensemble des noms de portes définies par l'utilisateur (ensemble des actions observables) parcouru par g, toutes ces actions sont définies localement.

- Une porte observable particulière $\delta \in \mathcal{G}$ est utilisée pour notifier la terminaison avec succès des processus,

- L dénote tout sous ensemble de $\mathcal{G}$,

- L'action interne est désignée par i,

- Soit $\mathcal{C}$ l'ensemble des canaux définis par l'utilisateur parcouru par $\overline{a}$,$\overline{b}$,...

- Soit $\gamma \notin \mathcal{G}$ une action particulière utilisée pour notifier les opérations de migration, communications des messages, et la création et suppression d'une nouvelle localité.

- $\mathcal{B}$ parcouru par E, F,... dénote l'ensemble des expressions de comportement dont la syntaxe est dans la figure 4.1.

- Soit LC l'ensemble des noms de localités parcouru par $L1, L2$, et $\varphi, \psi, ... \in LC^*$ dénote des chaînes de noms de locations.

- l'ensemble des actions est :

$$Act = \mathcal{G} \cup \{i, \delta, \gamma\}$$

La signification informelle des différents constructeurs du notre modèle est la suivante :

- Informellement $a\{d\}$ signifie que l'action a doit commencer son exécution dans l'intervalle temporel $[0, d]$. $\Delta^d E$ signifie qu'aucune évolution de E n'est permise avant l'écoulement d'un délai égal à d. Dans $g@t[SP]\,;\, E$ (resp. $i@t\{d\}; E$) t est une variable temporelle mémorisant le temps écoulé depuis la sensibilisation de l'action g (resp. i)et qui sera substituée par zéro lorsque cette action termine son exécution.

- Un processus Go $a\{d\}; E$ provoque la migration de la location repliée contenant la requête de migration Go vers la location portent le nom correspondant à la requête. Il permet le déplacement de localité courante dans l'arbre vers une nouvelle position dans l'arbre. En d'autre terme en parle de notion de mobilité, donc la mobilité c'est la migration d'un sous arbre.

$E ::=$		**Comportement**
	$stop$	*Inaction*
	$\mid exit\{d\}$	*Terminaison avec succés*
	$\mid \Delta^d E$	*Opérateur de délai*
	$\mid g@t[SP]; E$	*Action avec prédicat*
	$\mid i@t\{d\}; E$	*Restriction temporelle*
	$\mid E[]E$	*Choix non − déterministe*
	$\mid E\mid[L]\mid E$	*Composition parallèle*
	$\mid hide\, L\, in\, E$	*Intériorisation*
	$\mid E \gg E$	*Composition séquentielle*
	$\mid E [> E$	*Préemption*
	$\mid$**def** D **in** E	*Définition locale*
	$\mid Go\ \psi\{d\}; E$	*Migration*
	$\mid \overline{a} < \overline{b} > \{d\}$	*message*
	$\mid X[L]$	
$D ::=$		**Définition**
	T	*définition vide*
	$\mid D, D'$	*Composition de définitions*
	$\mid J \triangleright E$	*Règle*
	$\mid \psi[D : E]$	*Création de localité*
$J ::=$		**Filtres**
	$\overline{a} < \overline{b} >$	*Messgae*
	$\mid J, J\prime$	*Composotion de filtres*
$L ::=$		**Localités**
	ϕ	*Localité vide*
	$\mid L \mid\mid L\prime$	*Composition*
	$\mid L0[D : E]$	*Localité dépliée*

FIG. 4.1 – *Syntaxe de MD-LOTOS*

Exemple

Soit l'état du système suivant :

$$L1[\intercal : Go\ L2\ \{d\}; E]\ \vdash_{L0} ||\ \vdash_{L2}$$

La migration doit être réalisé dans l'espace du temps de 0 à d, si le temps écouler dépasse d, l'action est plus offerte à l'environnement.

- Un processus **def** D **in** E correspond à une définition locale, les définitions de canaux expriment le comportement associé à la réception de messages sur ces canaux. Une définition peut être, soit la définition vide $\intercal$, soit une composition de définitions D, D', soit une règle $J \rhd E$ consommant les messages correspondant au filtre J pour exécuter E, soit une location repliée $L1[D : E]$ ayant pour nom $L1$, pour définition D et pour processus P.

- Informellement, la définition $L1[D' : P]$ correspond à la création d'une localité $D' \vdash_{L0L1} P$, tel que $L0L1$ c'est le chemin de la localité $L1$. On dit que $\vdash_{L1}$ c'est une sous-localité de $\vdash_{L0}$

Exemple :

$$\begin{array}{lllll}
 & L1[D' : E],\ x < u > \rhd Q \vdash_{L0} F & ||D \vdash_{L2} x < y > & \\
\overset{creation}{\rightarrow} & x < u > \rhd Q \vdash_{L0} F & ||D \vdash_{L2} x < y > & ||D' \vdash_{L0L1} E \\
\overset{Comm}{\rightarrow} & x < u > \rhd Q \vdash_{L0} F,\ x < y > & ||D \vdash_{L2} & ||D' \vdash_{L0L1} E \\
\overset{Comm}{\rightarrow} & x < u > \rhd Q \vdash_{L0} F,\ Q\{y/x\} & ||D \vdash_{L2} & ||D' \vdash_{L0L1} E
\end{array}$$

- Les filtres modélisent la réception et la synchronisation de messages : un filtre de message $x < y >$ attend qu'un message y sur le canal x soit présent, le filtre $J|J'$ attend que le filtre J et $J\prime$ soient satisfaits pour être satisfait.

- Les localités représentent le système global. Une localité peut être soit la localité vide ϕ, soit une conjonction de localités $LC||LC'$, soit une localité dépliée $LC[D : E]$ ayant pour chemin LC, pour définition D et pour comportement E.

Sémantique opérationnelle structurée du calcul :

La relation de transition de maximalité $\rightarrow \subseteq \mathcal{C}\text{x}Atm\text{x}\mathcal{C}$ est définie comme état la plus petite relation satisfaisant les règles suivantes :

- **Processus def** D **in** E **(Définition locale)** :

Dans la configuration [**def** D **in** E], l'exécution du comportement E ne peut se produire que si le comportement a la forme d'un filtre d'une règle permet les règles décrites dans D.

Exemple :

Soit l'exemple de serveur d'impression [13] :

$$D \stackrel{def}{=} accepter < imprimante > |imprimer < fichier > imprimante < fichier >$$

Cette règle consomme deux message, l'un sur *accepter*, l'autre sur *imprimer*, et déclenche l'impression en envoyant le fichier à l'imprimante. Nous pouvons regrouper la définition du serveur d'impression est sont état courant dans un seul processus.

$$P = \text{def } D \text{ in } imprimer < 1 > |accepter < laser >$$

Suivant l'explication informelle de la règle D, le message $accepter < laser >$ et le message $imprimer < 1 >$ vont d'être consommés,

$$D \vdash accepter < laser > |imprimer < 1 >$$
$$\rightarrow D \vdash laser < 1 >$$

La règle sémantique est donnée par :

$$\frac{\exists\ J \rhd B\ /\ \ J\overline{\sigma}_{FA} = E}{\text{def } D \text{ in } E \rightarrow\ \text{def } D \text{ in } B\overline{\sigma}_{FA}}$$

$\overline{\sigma}_{FA}$: dénote la substitution des arguments formels décrite dans J par les paramètres transmis dans E.

Suivant notre exemple :

$$E \stackrel{def}{=} imprimer < 1 > |accepter < laser >$$
$$B \stackrel{def}{=} imprimante < fichier >$$
$$J \stackrel{def}{=} accepter < imprimante > |imprimer < fichier >$$

Les paramètres formels : *imprimante* est substituée par *laser*, et $fichier$ par 1.

$$B\overline{\sigma}_{FA} \stackrel{def}{=} laser < 1 >$$
$$J\overline{\sigma}_{FA} \stackrel{def}{=} accepter < laser > |imprimer < 1 >$$

• **Processus** $Go\ L\{d\}; E$ $(Migration)$:

Dans un système sans contraintes temporelles, se processus ce comporte comme suit :

La localité qui contient le constructeur de migration, migre vers la localité décrite dans la primitive. Si on considère les contraintes temporelles :

Exemple :

$$L0[D : P|Go\ L2\{d\}; E] \vdash_{L1} \quad || \quad \vdash_{L1L2}$$

Dans cette configuration on a plusieurs cas :

Cas 1 :

La migration peut avoir lieu avant que le temps écouler ne dépasse pas d unités, ce qui donne :

$$\vdash_{L1} || \; L0[D : P|E] \vdash_{L1L2}$$

Cas 2 :

Il existe une durée d', tel que $d = d' + d''$, et le temps écouler depuis que l'action de migration est sensibilisée est d'', ce qui donne :

$$L0[D : P|Go\ L2\{d'\}; E] \vdash_{L1} \quad || \quad \vdash_{L1L2}$$

Cas 3 :

Le temps écouler dépasse la durée d, donc la migration ne se produit jamais, ce qui donne :

$$L0[D : P|E] \vdash_{L1} \quad || \quad \vdash_{L1L2}$$

Dans ce cas on doit imposé que l'occurrence de l'action de migration ait lieu dans la période d, dans le cas contraire la migration ne se produira jamais.

Donc les règles sémantiques pour les trois cas sont :

1. $$\frac{}{L0[D : P|Go\ L2\{d'\}; E] \vdash_{L1} \quad || \quad \vdash_{L1L2} \xrightarrow{\gamma} \vdash_{L1} || \; L0[D : P|E] \vdash_{L1L2}}$$
2. $$\frac{}{L0[D : P|Go\ L2\{d' + d\}; E] \vdash_{L1} \quad || \quad \vdash_{L1L2} \xrightarrow{d} L0[D : P|Go\ L2\{d'\}; E] \vdash_{L1} || \quad \vdash_{L1L2}}$$
3. $$\frac{d \succ d'}{L0[D : P|Go\ L2\{d'\}; E] \vdash_{L1} \quad || \quad \vdash_{L1L2} \xrightarrow{d} L0[D : P|E] \vdash_{L1} \quad || \quad \vdash_{L1L2}}$$

- **Règle de réduction :**

Exemple :

Soit la localité du serveur d'impression : $D \vdash_s accepter < laser > |imprimer < 1 >$.

Cette configuration et de la forme : $J \rhd P \vdash_s J\overline{\sigma}_{FA}$. Le filtre J de la règle est satisfait, donc on lance une copie de processus P.

Dans notre exemple est $laser < 1 >$, c'est à dire on obtient la forme : $J \rhd P \vdash_s P\overline{\sigma}_{FA}$

La règle sémantique est donnée par :

$$\frac{\exists\ J \rhd P \ \in D \ / \ \ E = J\overline{\sigma}_{FA}}{D \vdash_{L0} E \xrightarrow{\gamma} \ D \vdash_{L0} P\overline{\sigma}_{FA}}$$

- **Communication globale :**

Les localités s'échangent des messages entre elles, le comportement $x < y > \{d\}$, exprime que le message $x < y >$ est offert pour une durée d, c'est à dire qu'il doit être émis avant d unités de temps, dans le cas contraire il est plus offert.

Exemple :

$$\vdash_{L0} x < y > \{d\} \;\; || \;\; J \rhd E \vdash_{L1}$$

Cas 1 :

La transmission du message peut avoir lieu avant que le temps écouler ne dépasse d unités, ce qui donne :

$$\vdash_{L0} \;\; || \;\; J \rhd E \vdash_{L1} x < y >$$

Le message peut être consommé localement, si le message satisfait le filtre J.

Cas 2 :

Le temps écouler dépasse la durée d , donc on obtient :

$$\vdash_{L0} \;\; || \;\; J \rhd E \vdash_{L1}$$

Le message, est plus offert.

Cas 3 :

Il existe une durée $d\prime$, tel que $d = d\prime + d"$, et le temps écouler depuis que l'action de migration est sensibilisée est $d"$, ce qui donne :

$$\vdash_{L0} x < y > \{d'\} \;\; || \;\; J \rhd E \vdash_{L1}$$

Les règles sémantiques sont :

1. $$\frac{}{\vdash_{L0} x < \widetilde{y} > \{d\} \;\; || \;\; J \rhd E \vdash_{L1} \;\; \xrightarrow{\gamma} \vdash_{L0} || \; J \rhd E \vdash_{L1} x < \widetilde{y} >}$$
2. $$\frac{d' \succ d}{\vdash_{L0} x < \widetilde{y} > \{d\} \;\; || \;\; J \rhd E \vdash_{L1} \;\; \xrightarrow{d'} \vdash_{L0} || \; J \rhd E \vdash_{L1}}$$
3. $$\frac{}{\vdash_{L0} x < \widetilde{y} > \{d' + d''\} \;\; || \;\; J \rhd E \vdash_{L1} \;\; \xrightarrow{d''} \vdash_{L0} x < \widetilde{y} > \{d'\} || \; J \rhd E \vdash_{L1}}$$

- **Règle de création des localités :**

$$\frac{}{L1[D : E] \vdash_{L0} \quad \xrightarrow{\gamma} \quad \vdash_{L0} \quad || \quad D \vdash_{L0L1} E}$$

- **Règle de suppression des localités :**Si l'ensemble des processus en cours d'éxecution dans une localité est *Stop*, on supprime cette localité.

$$\frac{D \vdash_{L0} Stop}{D \vdash_{L0} Stop \quad || \quad D' \vdash_{L1} E \;\; ||... \quad \xrightarrow{\gamma} \quad D' \vdash_{L1} E||...}$$

4.2 Etude de cas (Gigue d'information)

Dans cette section, on va montrer comment spécifier le comportement d'un système mobile, temps-réel dans le langage MD-LOTOS.

Soit une gigue d'information, située dans une source, celle-ci envoie la gigue à toutes les récepteurs, le système est shématisé dans la figure 4.2.

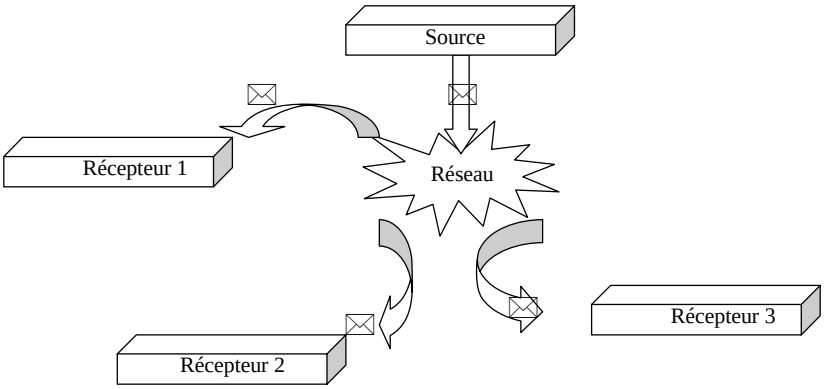


FIG. 4.2 – Schéma d'exécution de la gigue d'information

Principe de fonctionnement

On suppose que chaque gigue contient un message(*Start*) indiquant le début de la gigue, et un autre(*End*) indiquant la fin de la gigue.

- **Comportement de la source :**

La source doit assurer, deux tâches principales
- L'envoi de la gigue
- La création des nouvelles localités.

On définit le processus :

$$send[m] \stackrel{def}{=} \textbf{def}\ k <>\triangleright paquet < m > |k <> \textbf{in}\ k <>$$

Ce processus permet de dupliquer le message m, autant de fois qu'on veut (suivant le nombre de récepteurs).

Le processus *gigue* d'envoi de la gigue, est défini dans la figure 4.3, page 101, ce processus attend une gigue g de type $flux$ de données sur la porte b, suivi du processus $envoyer[g]$, après on fait appel au processus *gigue* pour une nouvelle gigue(définition récursive).

Le processus *envoyer* : commence par l'envoi du message *start* (début de la gigue), le message suivant est retardé par l'opérateur Δ, d'une durée d unités de temps, le processus est récursive jusqu'à le dernier message *end*. Donc la gigue est envoyée, on termine par l'action *exit* pour passer à la prochaine gigue.

La deuxième tâche de la source est assurée par la définition D défini dans la figure 4.3. Cette règle permet de créer les localités d'une façon dynamique. Le choix d'un nom d'une localité peut se faire de manière déterministe par l'utilisation de toute fonction *get* : $(\mathcal{L} - \mathcal{LC}) \rightarrow \mathcal{L}$ tel que :

$\mathcal{L}$ =$\{L0, L1, ...\}$ c'est l'ensemble des noms de localités.

$\mathcal{LC}$ c'est l'ensemble des noms de localités déjà créer. X nom de la nouvelle localité.

L'état du calcul dans la localité source est représenté par :

$$D \vdash_{L0} gigue$$

- **Comportement des récepteurs :**

Chaque récepteur doit assurer :

- la réception des messages.
- la concaténation des message réçus.

On définit le processus *reception* donné dans la figure 4.3. Ce processus consomme le premier paquet *start*, est continu à recevoir les messages suivants, jusqu'à le dernier message *end*, on arrêt le processus recevoir. Ensuite soit on reçoit une nouvelle gigue ou la sortie.

La concaténation est assuré par la règle R, défini dans la figure 4.3.

L'état du calcul dans chaque récepteur est :

$$R \vdash_{L0L1} reception$$

Spécification

La spécification complète est donnée par la figure 4.3, page 101.

4.3 Outil de compilation

La définition et la mise au point d'outils d'analyse lexicale et syntaxique est une étape importante dans le processus de mise en oeuvre d'un langage de programmation. Ocamllex et Ocamlyacc, sont deux générateurs d'analyseurs lexicaux et syntaxiques pour n'importe quel langage développé en langage Objective caml[11].

Dans les cas simples comme dans les cas compliqués, le problème qui doit résoudre l'analyse lexical et syntaxique est la transformation d'un flot linéaire de caractères en une donnée à

Spécifiction $GigueInformation[b] :=$
$[] \quad : flux;$
$message \quad : flux;$
$message.flux : flux;$
$\_\&\_ \quad : flux\ flux \rightarrow flux;$

$D ::= new <>\triangleright X[R : recevoir];$
$R ::= paquet < m1 >, paquet < m2 > \triangleright paquet < m1\&m2 >$

behavior *source*
$D \vdash_{L0} gigue$
behavior *récepteur*
$R \vdash_{L0L1} reception$

processus $send[m : flux] ::=$
(**def** $k <>\triangleright paquet < m > |k <>$ **in** $k <>$)
endproc ;
processus $gigue[b] ::=$
$(b?g : flux; envoyer[g]$
$\gg$
$gigue[b])$
endproc ;
processus $envoyer[g] ::=$
$([g = start.g\prime] \rightarrow$
$send[start] \gg \Delta^d envoyer[g']$
$[g = m.g\prime] \rightarrow$
$send[m] \gg \Delta^d envoyer[g']$
$[g = end.g\prime] \rightarrow$
$send[end]; exit)$
endproc ;
processus $reception[] ::=$
($paquet < start >; recevoir$
$\gg$
$(reception \ [] \ exit))$
endproc ;
processus $recevoir[] ::=$
($paquet < m > \rightarrow recevoir$
$paquet < end > \rightarrow exit)$
endproc ;

endspec.

FIG. 4.3 – *Spécification de la gigue d'information en MD-LOTOS*

la structure plus riche : une suite de mots, une structure d'enregistrements, l'arbre de syntaxe abstraite d'un programme...etc.

Tout langage possède un vocabulaire (le lexique) et une grammaire permettant d'assembler correctement les éléments du vocabulaire (la syntaxe). Pour qu'une machine, ou un programme, soit capable de traiter correctement un langage, celui-ci doit obéir à des règles lexicales et syntaxiques très précises. Une machine n'a pas $<<l'esprit\ de\ finesse>>$ requis pour prendre une décision face à l'ambiguïté des langues naturelles. Il convient donc de s'adresser à elle selon des règles clairement fixées ne souffrant pas d'exception.

Cette section présente un outil de compilation basé sur les deux outils Ocamllex et Ocamlyacc, pour l'analyse lexicale et l'analyse syntaxique. Cette dernière suppose en règle générale que la première à été effectuée. Dans la première section, nous présentons l'outil Ocamllex. Ensuite nous définissons la grammaire de notre langage (MD-LOTOS), et nous illustrons l'utilisation de l'outil Ocamlyacc. Et enfin nous présentons notre outil pour le langage MD-LOTOS.

4.3.1 L'outil Ocamllex

L'outil Ocamllex est générateur d'analyse lexical construit pour Objective Caml. Il produit un fichier source Objective CAML à partir d'un fichier contenant la description des éléments du lexique à reconnaître sous forme d'ensembles d'expressions rationnelles. L'usage est de donner aux fichiers de description lexicale l'extension *mll*. Pour obtenir un source Objective CAML à partir d'un fichier *mll* on tape la commande

$$Ocamllex \qquad nom\_fichier.mll$$

Exemple 4.1 *Soit l'exemple d'évaluation d'expressions arithmétiques, le fichier de description des élements du lexique à reconnaître est :*

```
(*File lexer.mll*)
{
open Parser
}
rule Token = parser
        [' ' 't'] {Token lexbuf}
        | ['n'] {EOL}
        | [' 0' - '9']+ {INT (Int_of_string(get_lexeme_lexbuf)}
        | '+' {Plus}
        | '-' {Minus}
        | '*' {Times}
        | '/' {Div}
        | '(' {LPAR}
```

```
    | ')' {RPAR}
```

La traduction de ce fichier par Ocamllex fournit la fonction lexer de type lexing.lexbuf->lexeme

4.3.2 L'outil Ocamlyacc

Cet outil prend en entrée un fichier de description d'une grammaire dont chaque règle est attachée à une section sémantique et il engendre deux fichier Objective CAML d'extension *ml* et *mli* contenant la fonction d'analyse syntaxique et son interface.

Reprenons le même exemple de la section précédente, le fichier du description

```
/*parseur pour les opérations arithmétiques*/
% token <int> INT
% token Plus Minus Times Div LPAR RPAR
% left Plus Minus
% left Times Div
% start maintoken
% type <ast.exp> main
% %
main : expr Eol
expr : INT {$1}
    | LPAR expr RPAR      {$2}
    | expr Plus expr      {$1+$3}
    | expr Minus expr     {$1-$3}
    | expr Times expr     {$1*$3}
    | expr Div expr       {$1/$3}
```

On peut composer les deux outils Ocamllex et Ocamlyacc de sorte que la transformation du flot de caractères en flot de lexèmes soit l'entrée de l'analyseur syntaxique. Pour se faire, le type lexème doit être commun aux deux. Ce type est défini dans les fichiers d'extension *mli* et *ml* engendrés par Ocamlyacc à partir des déclarations des token du fichier de description *mly* correspondant. Le fichier *mll* importe ce type, Ocamllex traduit ce fichier en une fonction Objective Caml de type lexing.lexbuf ->lexème.

Syntaxe abstraite

Une fois reconnue une phrase du langage, il est nécessaire de la transformer en une forme adapté aux phases successives du compilateur, qui vont explorer à plusieurs reprises cette représentation pour vérifier le typage, construire les tables des symboles, calculer la durée de vie des variables et d'autres attributs. L'arbre de dérivation syntaxique associé à la grammaire utilisée pour l'analyse n'est pas bien adapté, parce qu'il contient un grand nombre de noeuds internes (les non terminaux de la grammaire) qu'il n'ont aucun intérêt lors de visite de la structure.

Une arbre de syntaxe abstraite est un arbre dont la structure ne garde plus trace des détails de l'analyse mais seulement de la structure de programme.

Dans le langage Objective CAML on définit la syntaxe abstraite par :

Type exp-ast = Int of int
| Add of exp
| Mult exp-ast* exp-ast
| Sub exp-ast* exp-ast expr
| Div Times exp-ast* exp-ast

4.3.3 Outil de compilation pour MD-LOTOS

A partir du la syntaxe du langage MD-LOTOS (page 94), on à définit une grammaire ci-dossous.

Grammaire

Les terminaux sont en gras, les non-terminaux en italiques.

Les spécifications sont introduites par le mot-clé specification constituées de déclarations des nouveaus objets utilisées par la spécification, déclarations des nouveaux opérateurs, déclarations ou définitions des règles(ces déclarations sont optionnelles), de définitions des comportements, et d'une liste de processus. Autrement dit :

$specification$: := **specification** $ident\ decl\_types\ decl\_operateurs\ decl\_regles\ comport\ proc$ **endspec**.

$decl\_types$: := $meg : type$;

$type$: := **int** | **float** | **flux**.

meg ::= $var \mid var, meg$

var ::= $ident \mid ident.type$

$binop$ ::= $\_op\_ : type\ type \rightarrow type$;

$unop$ ::= $\_op : type \rightarrow type$;

$decl\_operateurs$ ::= $binop \mid unop \mid binopdecl\_operateurs \mid unopdecl\_operateurs$

$regle$ ::= $ident$ ::= $definition$

$decl\_regles$ ::= $regle \mid regle; decl\_regles$

$chiffre$: := 0-9

$alpha$: := a-z | A-Z

$ident$: := $alpha\ (alpha \mid chiffre)$*

$entier$: := $chiffre$+

$duree$: := $entier$

L ::= $ident$

op : := & | ! | ? | $ | ~| # | ≥ | ≤ | = | &&

sp : := **true** | **false**

La déclaration de comportements sont introduites par le mot-clé **behavior**.

comportement : := **behavior** *ident* *ident* ⊢ *localite* *ident*

comport : := *comportement* | *comportement comport*

Les déclarations de processus sont introduites par le mot-clé **processus**.

processus : := **processus** [*arguments*] : := *expression* **endproc**

arguments : := *ident* | *ident* : *type* | *ident*, *arguments* | *ident* : *type*, *arguments*

proc : := *processus* | *processus* ; *proc*

expression : := **stop**
| **exit** {*duree*}
| Δ*duree expression*
| *ident*@ *entier* [*sp*] ; *expression*
| *i* @ *entier* {*duree*} ; *expression*
| *expression operateur expression*
| *expression*|[*L*]|*expression*
| **hide** *L* **in** *expression*
| **def** *definition* **in** *expression*
| **go** *localite* {*duree*} ; *expression*
| *canal* < *canal* > {*duree*}

operateur : := [] | ≫ | [>

localite : :=*l entier*

canal : := *ident*

definition : := ⊤
| *definition*, *definition*
| *filtre* ▷ *expression*
| *localite*[*definition* : *expression*]

filtre : := *canal* < *canal* >
| *filtre*, *filtre*

Fichier de description pour ocamllex

L'étape suivante est la définition d'un fichier de description (MD-LOTOS.mll) qui représente une entrée pour Ocamllex. A partir de ce fichier de description l'outil Ocamllex, produit un programme CAML capable de reconnaître qu'une suite de lexèmes appartient au langage MD-LOTOS définit par cette grammaire.

```
(* Analyse lexicale *)
{
    open Lexing
    open Parser
    open Ast
```

```
(* Erreurs lexicales *)
exception Lexical_error of string

let id_or_keyword =
    let h = Hashtbl.create 15 in
    List.iter (fun (s,k) -> Hashtbl.add h s k)

        [ "specification", SPECIFICATION; "int", INT; "flux", FLUX; "behavior", BEHA-
VIOR; "processus", PROCESSUS;"endproc", ENDPROC; "hide",HIDE; "in", IN; "def", DEF;
"go", GO; "stop", STOP; "exit", EXIT; "endspec", ENDSPEC; "true",TRUE; "false",FALSE ];

    fun s -> try Hashtbl.find h s with Not_found -> IDENT s

}

let alpha = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let ident = (alpha | '_') (alpha | '_' | digit)*

let user_op = ['&' |'!' |'?'  '$'  ' ~'  '#'  '≥'   ' ≤ ' '='    '&&']
let lang_ops=['>>' '[>' '[]' ]
Let t_mg=["flux"   "int"]

Let sp=["true"   "false"]
rule token = parse
        | '\t'                { newline lexbuf; token lexbuf }
        | [' ' '\t' '\r']+    { token lexbuf }
        | ident               { id_or_keyword (lexeme lexbuf) }
        | digit+              { INTEGER (int_of_string (lexeme lexbuf)) }
        | user_op             { OPS (lexing.lexeme lexbuf)) }
        | lang_op             { OPL (lexing.lexeme lexbuf)) }
        | t_mg                {Type}
        | sp                  {SP}
        | '{'                 { LBRACE }
        | '}'                 { RBRACE }
        | '('                 { LPAR }
        | ')'                 { RPAR }
        | '['                 { LSQUARE }
        | ']'                 { RSQUARE }
        | ','                 { COMMA }
        | ';'                 { SEMICOLON }
        | '.'                 { DOT }
```

```
| ' :'                { DDOT }
| "->"               { ARROW }
| '⊢'                { LOC }
| '>'                { SUP }
| '<'                { INF }
| '/'                { SLASH }
| 'Δ'                { DELTA }
| "||"               { VERTICALBARVERTICALBAR }
| '| '               { VERTICALBAR }
| "="                { EQ }
| 'ᴛ'                { VIDE }
| '@'                { AD }
| '▷'                { DELTAP }
| "::="              { AFF }
| 'i'               {I}
| '_'                {DOP }
| eof                { EOF }
```

Fichier de description pour ocamlyacc

Le fichier de description pour ocamlyacc est définit par :

```
%{
open Ast
%}
%token <string> IDENT
%token <int> INTEGER

 /* Mots clés */
%token SPECIFICATION BEHAVIOR HIDE ENDSEPC ENDPROC
%token STOP EXIT PROCESSUS DEF IN GO FLUX INT TRUE FALSE

/* Symboles */
%token LPAR RPAR LBRACE RBRACE LSQUARE RSQUARE
%token SEMICOLON COMMA DOT ARROW
%token SUP INF SLASH VERTICALBAR VIDE DOP
%token EOF I
%token Type SP
/* Opérateurs */
%token OPS OPL
%token EQ VERTICALBARVERTICALBAR DDOT
%token ARROW LOC DELTA AD DELTAP AFF DELTAP
```

```
/*s Précédences */
%right EQ
%left VERTICALBARVERTICALBAR                    /* || */
%nonassoc DOT ARROW  LOC DELTA AD DELTAP AFF DELTAP DDOT

%token <int> INTEGER
%token <string> IDENT

/*s Point d'entrée */
%start specif
%type <Ast.spc> specif
 %%

/*Règle de production*/
specif :
      | SECIFICATION IDENT D_type D_ops D_reg compt procs ENDSEPC {Specification($2)}
D_type :
      | /*Epsilon*/ {[]}
      |meg DDOT Type SEMICOLON {Types($1,Type)}
meg :
      | var {Vars($1)}
      | var COMMA Type {Vars($1)}
var :
      |IDENT {Vars($1)}
      |IDENT DOT Type {Vars($1)}
binop :
      | CO OPS CO DDOT Type Type ARROW Type SEMICOLON {Op_bin(OPS,Type)}
unop :
      | CO OPS DDOT Type ARROW Type SEMICOLON {Op_uni(OPS,Type)}
D_ops :
      |binop {$1}
      |unop {$1}
      |binop D_ops {$1}
      |unop D_ops {$1}
regle :
      | IDENT AFF definition {Regle($1,$3)}
D_reg :
      | /*Epsilon*/ {[]}
      | regle SEMICOLON D_reg {$1}
comp :
      | BEAHAVIOR IDENT IDENT LOC localite IDENT {Behavior($2,$5) }
```

compt :

| /*Epsilon*/ {[]}

| comp compt {$1}

processus :

| PROCESSUS LSQUARE arguments RSQUARE AFF expression ENDPROC {Processus($3,$5)}

arguments :

| /*Epsilon*/ {[]}

| IDENT {$1}

| IDENT DDOT Type {Arg($1,Type)}

| IDENT SEMICOLON arguments {$1}

| IDENT DDOT Type COMMA arguments {Arg($1,Type)}

procs :

| /*Epsilon*/ {[]}

|processus COMMA procs {$1}

expression :

| STOP {Stop}

| EXIT LBRACE INTEGER RBRACE {Exit($3)}

| DELTA INTEGER expression {Delay($2,$3)}

| IDENT AD INTEGER LSQUARE SP RSQUARE SEMICOLON expression {Exp($1,$3,SP,$8)}

| I AD INTEGER LBRACE INTEGER RBRACE SEMICOLON expression {Exp(I,$3,$8)}

|expression OPL expression {Exp($1,OPL,$3)}

|expression VERTICALBAR LSQUARE IDENT RSQUARE VERTICALBAR expression {Exp($1,$4,$7)}

|HIDE IDENT IN expression {Hide($2,$4)}

|DEF definition IN expression {Def($2,$4)}

|GO localite LBRACE INTEGER RBRACE SEMICOLON expression {Go($2,$4,$7)}

|IDENT INF IDENT SUP LBRACE INTEGER RBRACE {Mgs($1,$3,$6)}

localite :

| L INTEGER {Loca(L,$2)}

definition :

|VIDE {Def(vide)}

|definition COMMA definition {Deft($1,$3)}

|filtre DELTAP expression {Deft($1,DELTAP,$3)}

|localite LSQUARE definition DDOT expression RSQUARE {Deft($1,$3,$5)}

filtre :

|IDENT INF IDENT SUP {Filtre($1,$3)}

|filtre COMMA filtre {Filtre($1,$3)}

4.3.4 Etape d'analyse

La compilation du couple analyse lexical, analyse syntaxique doit être faite en suivant un certain ordre. Ceci est dû à la dépendance mutulle dans la déclaration des lexèmes. Ainsi, dans notre exemple, il faudra suivre la séquence de commandes :

ocamlc -c md-lotos_types.mli
ocamlyacc md-lotos_parser.mly
ocamllex md-lotos_lexer.mll
ocamlc -c md-lotos_parser.mli
ocamlc -c md-lotos_lexer.ml
ocamlc -c md-lotos_parser.ml

Ce qui engendera les fichiers md-lotos_lexer.cmo et md-lotos_parser.cmo.

4.4 Conclusion

Dans ce chapitre nous avons proposé un langage temps réel et mobile, MD-LOTOS, qui permet de prendre en compte les aspects de base des systèmes temps réel et mobile. A savoir la distribution, la mobilité, le dynamisme, et les contraintes temporelles et durées d'actions, en supportant la non-atomicité temporelles et structurelle des actions.

MD-LOTOS, s'inspire de deux modèle Join-calcul et D-LOTOS, il hérite de D-LOTOS la spécification des contraintes temporelles et la prise en compte des durées d'action. De Join-calcul nous gardons la notion de définition local, et nous introduissons la notion de mibilité, et le calcul dynamique. MD-LOTOS ne fournit aucune primitive de synchronisation distante, toute synchronisation distribuée est interdite, ce qui rend l'implémentation du modèle facile.

De point de vue syntaxique, MD-LOTOS reste trés proche du D-LOTOS, et de point de vue sémantique MD-LOTOS partage la même sémantique de maximalité avec D-LOTOS.

Chapitre 5

Conclusion et perspectives

Le but de ce travail était d'étudier les modèles algébriques de spécification des sytèmes mobiles, et les modèles de spécification des systèmes temps réel, ensuite proposer un nouveau modèle capable de spécifier des systèmes temps réel et mobiles.

Bilan

Dans l'objectif de spécifier des systèmes temps réel et mobiles, nous avons étudier en détaille deux familles de modèles algébriques a savoir les modèles mobiles et les modèles temps réel, et nous avons étudier aussi une autre approche logique de réécriture.

Ensuite nous avons présenté un modèle adapté à la programmation répartie. Ce modèle s'inspire des deux modèles algébriques Join-calcul et D-LOTOS, et se caratérise par :

- Pour exprimer les contraintes temporelles et durées des actions nous avons gardé la syntaxe de D-LOTOS avec des nouveaux constructeurs.
- D-LOTOS n'offre pas de primitive de création dynamique des agents(processus, localités,...). MD-LOTOS c'est un modèle dynamique c'est à dire il permet la création de nouveaux agents, il offre des primitives pour cela.
- Toute synchronisation présentée dans MD-LOTOS est locale, ce qui rend l'implémentation du modèle plus facile.
- MD-LOTOS traite les localités d'une manière explicite c'est à dire il offre des primitives de création et de suppression des nouvelles localités. Chaque localité contient un ensemble de processus en cours d'éxecution et un ensemble de règles, ces localités sont structrées en hiérarchie d'arbre, et peuvent migrer dans l'arbre ce qui rend le calcul mobile.

Perspectives

Le travail présenté dans ce document peut être continué dans plusieurs directions :

- Valider le modèle par des techniques de vérifications formelles, a savoir l'approche comportementale, l'approche logique basé sur les model-checking et l'approche teste.

- Développer une théorie algébrique pour le calcul avec des équivalences, des théorèmes,...
- Etudier le cas des réseaux sans fil en MD-LOTOS.

Bibliographie

[1] E. Alouini. *Etude et mise en oeuvre de la réécriture conditionnelle concurrente sur des machines parallèle à mémoire distribuée*. PhD thesis, Université Henri Poincaré- Nancy I, Mai 1997.

[2] T. Bolognesi. LOTOS-like process algebras with urgent or timed interaction. In *FORTE*, pages 255–270. North Holland, 1991.

[3] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14 :25–59, 1987.

[4] G. Boudol. Some chemical abstract machines. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *LNCS*, pages 92–123. Springer-Verlag, 1994.

[5] G. Boudol and G.Berry. The chemical abstract machine. *Theoretical Computer Science*, 96 :217–248, 1992.

[6] L. Cardelli. Mobile ambient synchronization. Technical report, July 1998.

[7] L. Cardelli and A. D. Gordon. Mobile ambients. volume 1378, pages 140–155. Springer-Verlag, 1998.

[8] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *POPL'99*, pages 79–92, January 1999.

[9] J.-J. L. L. M. Cédric Fournet, Georges Gonthier and D. Rémy. A calculus of mobile agents. *Montanari and Sassone*, 103 :406–421, 1996.

[10] J. P. Courtiat, M. S. de Camargo, and D. E. Saïdouni. RT-LOTOS : LOTOS temporisé pour la spécification de systmès temps rél. Research Report 93040, LAAS-CNRS, 7 av. du Colonel Roche, 31077 Toulouse Cedex France, February 1993.

[11] P. M. Emmanuel CHAILOUX and B. PAGANO. *Objective CAML*.

[12] P. F.Dupan, S.Eker and J.Meseguer. Mobile maude. Technical report, Computer Science Laboratory, SRI International, 2000.

[13] C. Fournet. *The Join-Calculus : a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, Palaiseau, INRIA, Novembre 1998.

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[15] ISO8807. *LOTOS, A Formal Description Technique Based on the Ordering of Observational Behaviour*. ISO, November 1988.

[16] ISO/IEC. *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, International Standard 8807*. International Organisation of Standardization – Information Processing Systems – Open Systems Interconnection, Genève, September 1988.

[17] J. L. Krivine. *Lambda-calcul, types et modèles*. Masson, Paris, 1990.

[18] G. Leduc. A timed lotos supporting a dense time domain and including new timed operators. In R. G. M. Diaz, editor, *FORTE*, pages 87–102. North Holland, 1993.

[19] L. Léonard and G. Leduc. Comment rendre lotos apte à spécifier des systèmes temps réelă. *Ingénierie des Protocoles*, 1993.

[20] L. Léonard and G. Leduc. An enhanced version of timed lotos and its application to a case study. May 1993.

[21] L. Léonard and G. Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 29 :271–292, 1997.

[22] L. Léonard and G. Leduc. A formal definition of time in LOTOS - extended abstract. *Formal Aspects of Computing*, 10(3) :248–266, 1998.

[23] C. Lohr. *Contribution à la Conception de Systèmes Temps-Réel S'appuyant sur la Technique de Description Formelle RT-LOTOS*. PhD thesis, LAAS-CNRS, 7 avenue du colonel Roche, 31077 Toulouse Cedex France, 2002.

[24] N. Marti-Oliet and J.Meseguer. Rewriting logic as a logical and semantic framwork. In J.Meseguer, editor, *First Intl. Workshop on Rewriting Logic and its Applications*, volume 4, 1996.

[25] N. M.-O.-F. P. J. M.Clavel, S.Eker and J. Quesada. A maude tutorial. Technical report, Computer Science Laboratory, SRI International, March 2000.

[26] P. M.Clavel, S.Eker and J.Meseguer. Principales of maude. In J.Meseguer, editor, *First Intl. Workshop on Rewriting Logic and its Applications*, volume 4, pages 73–155, 1996.

[27] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96 :73–155, 1992.

[28] J. Meseguer. Rewriting logic as a semantic framwork for concurrency. In *CONCUR'96*, volume 1119 of *LNCS*, pages 331–372. Springer-Verlag, August 1996.

[29] J. Meseguer. Research dirctions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *CONCUR'96*, volume 165 of *F*, pages 347–398. Springer, 1997.

[30] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[31] R. Milner. The polyadic -calculus : a tutorial. *Logic and Algebra of Specification*, 1991.

[32] P. C. Ölveczky. *Specification and Verification of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, Department of Computer Science, University of Bergen, Norway, Research done at the Computer Science Laboratory, SRI International, Menlo Park, USA, December 2000.

[33] P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In T. Margaria and M. Wermelinger, editors, *Fundamental Approaches to Software Engineering (FASE'2004)*, volume 2984 of *LNCS*, pages 354–358. Springer-Verlag, 2004.

[34] D. F. P.M. Merlin. Recoverability of communication protocols implications of a theoretical study. In *ACS/IEEE International Conference on Computer Systems and Applications*, pages 1036–1043, Septembre 1976.

[35] J. P. Robin Milner and D. Walker. A calculus of mobile processes, parts i and ii. *Information and Compution*, 100 :1–77, 1992.

[36] D. E. Saïdouni. *Sémantique de Maximalité : Application au Raffinement d'Actions en LOTOS*. PhD thesis, LAAS-CNRS, 7 av. du Colonel Roche, 31077 Toulouse Cedex France, 1996.

[37] D. E. Saïdouni and J. P. Courtiat. Prise en compte des durées d'action dans les algèbres de processus par l'utilisation de la sémantique de maximalité (version étendue). Technical Report 03243, LAAS-CNRS, 7 avenue du colonel Roche, 31077, Toulouse Cedex France, May 2003.

[38] A. Schmitt. *Conception et Implémentation de Calculs d'Agents Mobiles*. PhD thesis, Ecole Polytechnique, Palaiseau, INRIA, Septembre 2002.

[39] A. Verdejo and N. Marti-Oliet. Implementing ccs in maude.