Table des matières

1 Introduction générale **3**

2 D-LOTOS **6**

2.1 Introduction 6
2.2 Sémantique de maximalité 7
2.2.1 Syntaxe de Basic LOTOS 7
2.2.2 Principe de la sémantique de maximalité 7
2.2.3 Sémantique opérationnelle structurée de maximalité de Basic LOTOS 9
2.3 Introduction des durées et des contraintes temporelles 12
2.3.1 Sémantique opérationnelle structurée de D-LOTOS 14
2.4 Relations de bissimulation 16
2.4.1 Relation de bissimulation temporelle 16
2.4.2 Relation de bissimulation temporelle de maximalité 17
2.4.3 Relation de performance 18
2.5 Spécification de la latence 20
2.6 Conclusion et perspectives 22
2.7 Preuves des résultats 24

3 Modèles sémantiques pour le temps réel **26**

3.1 Automate temporel 27
3.1.1 Exécution 29
3.2 STEM Temporel 30
3.2.1 Table de transition temporelle 30
3.2.2 Traces d'exécution 32
3.2.3 Conclusion 33

4 Implantation **34**
4.1 Le paradigme de programmation fonctionnelle 34
4.1.1 Principes des langages fonctionnels 34
4.1.2 Présentation du langage de programmation CAML 37
4.2 introduction aux compilateurs : 47
4.2.1 Analyse lexicale : 47
4.2.2 Analyse syntaxique : 48
4.2.3 Analyse sémantique : 48
4.3 Realisation 49
4.3.1 Analyseur lexical : 49
4.3.2 Analyseur syntaxique 51
4.3.3 Générateur du code objet (STEM temporel) 54
4.4 Conclusion 61

5 conclusion **62**

Chapitre 1

Introduction générale

Le progrès technologique observé ces dernières années à touché un bon nombre de secteur, entre autre le domaine des systèmes distribués et critiques. La conception et la réalisation d'applications exploitant ces technologies exige une assistance particulière afin de fournir des produits dont le comportement correcte est certifié. Cette exigence est induite par les domaines d'utilisation de ces produits dont toute défaillance peut avoir des conséquences catastrophiques.

Un système concurrent peut être considéré comme un ensemble de processus coopératifs. Cette coopération est assurée par divers moyens tel que la communication synchrone ou asynchrone, les mécanismes d'exclusion mutuelle et de rendez-vouz, ... etc.

La conception de telles applications fait appel à une démarche formelle dont les principales phases sont comme suit:

- Choix d'un modèle de spécification formel pouvant prendre en compte toutes les fonctionnalités de l'application, tel que le parallélisme, la composition, les excéptions, la communication, les contraites temporelles ... etc. A titre d'exemple, nous pouvons citer la classe des modèles des réseaux de Petri, la classe des modèles algébriques à savoir CCS, CSP, ACP, LOTOS, RT-LOTOS, ET-LOTOS et D-LOTOS.
- La deuxième phase consiste en la capture des besoins utilisateur (issus du cahier de charge). Cette phase aboutie à l'écriture d'une première spécification décrivant l'architecture du système à concevoir. Cette spécification se caractérise par la description du comportement observable du système.
- Différentes étapes de conception seront alors développées afin de prendre en compte, à chaque étape, une partie des contraintes environnementale. La dernière étape donne comme résultat une spécification complète.

Assurer la validité du système revient donc à assurer la validité de chaque spécification résultant de chaque étape de conception. Deux approches de vérification peuvent être utilisées, à savoir l'approche boite noir, appelée par fois approche test, dans laquelle le code de la

spécification n'est pas accéssible ; et l'approche boite blanche pour laquelle le code de la spécification et complètement explorable et pouvant être utilisé pour la génération d'un graphe décrivant le comportement complèt du système à concevoir. Les techniques utilisées dans cette dernière approche exploitent ce graphe de comportement afin de vérifier les propriétés requises du système.

Les propriétés à vérifier peuvent être classées dans deux grandes familles :

- Les propriétés concernant le comportement logique du système telque les comportements perceptibles par un observateur, les possibilités de blocage à certains points d'exécution du système, les traces d'exécution, ...etc. Pour ces propriétés il n'est pas nécessaire d'utiliser des modèles de spécification incluant le temps et les contraintes temporelles. Par voie de conséquence, les modèles sémantiques sous-jacents sont aussi non temporels.

- Les propriétés concernant le comportement quantitatif du système tel que le temps nécéssaire pour son exécution, le respect de certain contraintes temporelles indispensables à son bon fonctionnement, ... etc. Pour prendre en compte cette famille de propriété il indispensable d'utiliser un modèle de spécification temporel pour lequel sa sémantique est exprimée dans un modèle sémantique temporel aussi.

Notre travail s'inscrit dans l'étude des modèles de spécification et sémantiques temporels. Dans ce contexte nous pouvons diviser ces modèles en deux catégories :

- Modèles entrelacés : Ces modèles se caractérise par la considération de l'hypothèse d'atomicité temporelle et structurelle des actions. Il est clair que de tels modèles ne peuvent être utilisés que si cette abstraction ne met pas en cause les résultats de la vérification.

- Modèles non entrelacés : Appelés aussi du vrai parallélisme, ces modèles considère, dès la phase de conception, que les actions ne sont pas atomiques. D'où l'association de durées aux actions. Cette vision est plus proche de la réalité ce qui permet l'obtention de spécifications qui reflètent fidèlement la structure du système à concevoir.

Notre travail consiste à prendre la technique de description formelle temps réel D-LOTOS, pour laquelle une sémantique de maximalité temporelle à été définie, et de développer une représentation des comportement temporels associés sous forme de graphes appelés systèmes de transitions étiquetées maximales temporels (STEM temprels). Les stems temporels peuvent ainsi être utilisé pour le développement d'outils de vérification formels. Il est à noter que le modèle de spécification D-LOTOS se distingue par rapport aux autres modèles existant dans la litérature par la prise en compte aussi bien des durées d'actions que des contraintes temporelles. Ceci permet de faciliter la spécification de systèmes temps réel complexes.

Le mémoire est organisé comme suit :

Le deuxième chapitre rappelle la technique de description formelle temps réel D-LOTOS.

Le troisième chapitre introduit deux modèles sémantiques temporels à savoir les automates temporels et les appelés systèmes de transitions étiquetées maximales temporels (STEM temprels).

Le quatrième chapitre présente la conception et l'implantation de notre outil de génération de STEM temporels à partir de spécifications D-LOTOS. Entre autre le paradigme de programmation fonctionnelle est présenté ainsi que les principales caractéristiques du langage CAML utilisé dans notre projet.

Finalement, le cinquième chapitre donne les conclusions de notre travail ainsi qu'un certain nombre de perspectives pouvant constiuées des travaux futurs.

Chapitre 2

D-LOTOS

2.1 Introduction

L'étude des systèmes critiques en général et des applications temps réel en particulier fait de plus en plus appel à des méthodes formelles permettant de répondre aux exigences aux quelles sont soumises ces applications. L'étude du comportement logique de ces applications implique l'utilisation de modèles de spécification et de vérification dans lesquels les relations temporelles entre occurrences d'actions se réduisent à l'étude de la chronologie d'exécution de ces actions, seules pouvant être vérifiées des propriétés qualitatives et non pas des propriétés quantitatives comme le respect de contraintes temporelles et de performances minimales. Pour répondre à ces besoins quantitatifs, de nouveaux modèles de spécification ont été proposés dans la littérature. Parmi ces modèles nous pouvons citer quelques extensions temporelles des algèbres de processus [18][20][3][2][15] [26][19][10][11][14][16]. Tous ces modèles ont mis en évidence l'importance de l'integration du temps pour la spécification des aspect temporels, par contre une différence majeure est à souligner. A titre d'exemple, les techniques de description formelle RT-LOTOS et ET-LOTOS [10] [11] [16] considèrent des opérateurs permettant d'exprimer des contraintes temporelles, l'objectif principal étant la spécification des systèmes temps réel; par ailleurs, dans [14] , un langage à la CSP est étendu par l'attribution de durées aux actions, le but étant l'étude des performances de systèmes concurrents. Cependant, dans [12][22], il a été montré que la levée de l'hypothèse d'atomicité des actions dans les modèles de spécification nécéssite l'utilisation de sémantiques du vrai parallélisme à la place de la sémantique classique d'entrelacement. Dans cet article nous proposons une approche intégrant à la fois contraintes temporelles et durées des actions dans un langage très proche syntaxiquement de ET-LOTOS, mais pour lequel nous avons défini une sémantique de vrai parallélisme, appelée sémantique temporelle de maximalité. Ceci permettra de spécifier la latence dont l'intérêt a été largement mis en évidence dans RT-LOTOS [9].

Afin d'observer l'incompatibilité de la sémantique d'entrelacement avec l'attribution de

durée aux actions, considérons l'exemple des deux expressions de comportement Basic LOTOS définies par : $E = a; stop \quad ||| \quad b; stop$ et $F = a; b; stop \quad [] \quad b; a; stop$. Tant que les durées des actions a et b sont nulles, les deux comportements paraissent identiques, cependant si nous considèrons que $durée(a) > 0$ 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 $max\{durée(a), durée(b)\}$, alors que le temps minimal pour exécuter les deux actions 'a' et 'b' dans F est de $durée(a) + durée(b)$.

L'article est organisé de la manière suivante: La section 2.2 rappelle la sémantique de maximalité de Basic LOTOS [12][22]. La section 2.3 introduit le concept de durée d'action et donne la syntaxe et la sémantique opérationnelle structurée temporelle de maximalité du langage D-LOTOS (pour LOTOS avec durées attribuées aux actions). La section 2.4 présente trois relations de bissimulation à savoir la relation de bissimulation temporelle, le relation de bissimulation temporelle de maximalité et la relation de performance; les propriétés de chacune d'elles sont présentées. La section 2.5 montre comment la notion de durée d'action permet de spécifier la notion de latence qui a été introduite dans le formalisme RT-LOTOS. La section 2.6 donne quelques conclusions et perspectives de notre travail et finalement la section 2.7 contient les preuves des résultats développés dans cet article.

2.2 Sémantique de maximalité

Dans cette section, nous rappelons la sémantique de maximalité de Basic LOTOS telle qu'elle a été définie en [12][22]. Nous supposons que le lecteur est familier avec Basic LOTOS et sa sémantique d'entrelacement.

2.2.1 Syntaxe de Basic LOTOS

Soit PN l'ensemble des variables de processus parcouru par X et soit $\mathcal{G}$ l'ensemble des noms de portes définies par l'utilisateur (ensemble des actions observables) parcouru par g. Une porte observable particulière $\delta \notin G$ est utilisée pour notifier la terminaison avec succès des processus. L dénote tout sous- ensemble de $\mathcal{G}$, l'action interne est designée par i. $\mathcal{B}$ parcouru par E, F, ... dénote l'ensemble des expressions de comportement dont la syntaxe est:

$$E ::= stop \mid exit \mid X[L] \mid g; E \mid i; E \mid E[]E \mid E|[L]|E \mid hide\, L\, in\, E \mid E >> E \mid E\, [> E$$

Etant donné un processus dont le nom est P et dont le comportement est E, la définition de P est exprimée par $P := E$. Par la suite, l'ensemble de toutes les actions est designé par Act ($Act = \mathcal{G} \cup \{i, \delta\}$).

2.2.2 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. Les dif-

férentes sémantiques se distinguent par le sens associé aux états et aux transitions. A titre d'exemple, dans le cas de la sémantique d'entrelacement, les transitions sont des événements qui correspondent à l'exécution complète des actions associées aux transitions.

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), nous associons, pour distinguer les exécutions de chacune des actions, un identificateur à 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 section 2.2.

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

Dans l'état initial, aucune action n'a encore été exécutée, donc aucun évenement n'est maximal, 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 :

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

x (respectivement 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 tous les deux 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 préfixage, 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 :

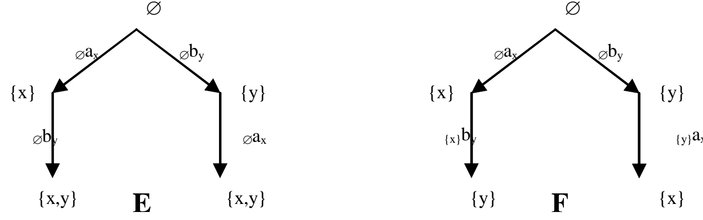


Figure 2.1: Arbres de dérivation de E et F obtenus par la technique de maximalité

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

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

Les arbres de dérivation des expressions de comportement E et F obtenus par l'application de la technique de maximalité sont représentés dans la figure (2.1).

2.2.3 Sémantique opérationnelle structurée de maximalité de Basic LOTOS

Définition 2.1 *L'ensemble des noms des événements est un ensemble dénombrable noté $\mathcal{M}$. Cet ensemble est parcouru par $x, y, \ldots$. $M, N, \ldots$ 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 $M \in 2^{\mathcal{M}}_{fn}$, $x \in \mathcal{M}$ et $a \in Act$, l'atome (M, a, x) sera noté $_{M}a_{x}$. Le choix d'un nom d'évenement peut se faire de manière déterministe par l'utilisation de toute fonction get : $2^{\mathcal{M}} - \{\phi\} \rightarrow \mathcal{M}$ satisfaisant $get(M) \in M$ pour tout $M \in 2^{\mathcal{M}} - \{\phi\}$.

Définition 2.2 *L'ensemble $\mathcal{C}$ des configurations des expressions de comportement de Basic LOTOS est le plus petit ensemble défini par induction comme suit :*

- $\forall E \in \mathcal{B}, \forall M \in 2^{\mathcal{M}}_{fn} : \quad _{M}[E] \in \mathcal{C}$
- $\forall P \in PN, \forall M \in 2^{\mathcal{M}}_{fn} : \quad _{M}[P] \in \mathcal{C}$
- *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 $\mathcal{F} \in \mathcal{C}$* $\quad op \in \{ \; [] \; , \; ||| \; , \; || \; , \; |[L]| \; , \; [> \; \}$
- *si $\mathcal{E} \in \mathcal{C}$ et $\{a_1, \ldots, a_n\}$, $\{b_1, \ldots, b_n\}$ deux sous-ensembles de $\mathcal{G}$ alors $\mathcal{E}[b_1/a_1, \ldots, b_n/a_n] \in \mathcal{C}$*

Etant donné un ensemble $M \in 2^{\mathcal{M}}_{fn}$, ${}_{M}[\cdots]$ est appelée opération d'*encapsulation*. Par la suite nous supposons que cette opération est distributive par rapport aux opérations $[], |[L]|$, *hide*, $[>$ et le renommage des portes. Nous admettons aussi que ${}_{M}[E \gg F] \equiv {}_{M}[E] \gg F$, ce qui est acceptable car la configuration ${}_{M}[E \gg F]$ présente un état dans lequel l'exécution de E suivie de F dépend de la terminaison des actions référencées par l'ensemble M des noms des événements; puisque l'exécution de F dépend de la fin de l'exécution de E, alors le comportement de la configuration est celui de ${}_{M}[E] \gg F$. Une configuration est dite canonique si elle ne peut plus être réduite par la distribution de l'opération d'encapsulation sur les autres opérateurs. Par exemple, la configuration ${}_{\{x,y\}}[a; stop ||| b; stop]$ n'est pas canonique, alors que la configuration ${}_{\{x,y\}}[a; stop] ||| {}_{\{z\}}[b; stop]$ est canonique.

Proposition 2.1 *[22]Toute configuration canonique est sous l'une des formes suivantes:*

$$\begin{array}{lllll} {}_{M}[stop] & {}_{M}[exit] & {}_{M}[a;E] & {}_{M}[P] & \mathcal{E}\ []\ \mathcal{F} \\ \mathcal{E}\ |[L]|\ \mathcal{F} & hide\ L\ in\ \mathcal{E} & \mathcal{E} \gg \mathcal{F} & \mathcal{E}\ [>\ \mathcal{F} & \mathcal{E}\,[b_1/a_1, ..., b_n/a_n] \end{array}$$

où $\mathcal{E}$ et $\mathcal{F}$ sont des configurations canoniques.

Par la suite, nous supposons que toutes les configurations sont canoniques ; la mise sous forme canonique d'une configuration est implicite et interne.

Définition 2.3 *La fonction $\psi : C \rightarrow 2^{\mathcal{M}}_{fn}$, qui détermine l'ensemble des noms des événements dans une configuration, est définie récursivement par :*

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

Définition 2.4 *Soit $\mathcal{E}$ une configuration ; $\mathcal{E}\backslash N$ dénote la configuration obtenue par la suppression de l'ensemble des noms des événements N de la configuration $\mathcal{E}$. $\mathcal{E}\backslash N$ est définie récursivement sur la configuration $\mathcal{E}$ comme suit :*

$$\begin{array}{ll} ({}_{M}[E])\backslash N = {}_{M-N}[E] & (\mathcal{E}\ []\ \mathcal{F})\backslash N = \mathcal{E}\backslash N\ []\ \mathcal{F}\backslash N \\ (\mathcal{E}\ |[L]|\ \mathcal{F})\backslash N = \mathcal{E}\backslash N\ |[L]|\ \mathcal{F}\backslash N & (hide\ L\ in\ \mathcal{E})\backslash N = hide\ L\ in\ \mathcal{E}\backslash N \\ (\mathcal{E} \gg \mathcal{F})\backslash N = \mathcal{E}\backslash N \gg \mathcal{F} & (\mathcal{E}\ [>\ \mathcal{F})\backslash N = \mathcal{E}\backslash N\ [>\ \mathcal{F}\backslash N \\ (\mathcal{E}\,[b_1/a_1, ..., b_n/a_n])\backslash N = \mathcal{E}\backslash N\,[b_1/a_1, ..., b_n/a_n] & \end{array}$$

Définition 2.5 *L'ensemble des fonctions de substitution des noms des événements est Subs (i.e. $Subs = \mathcal{M} \rightarrow 2^{\mathcal{M}}_{fn}$) ; $\sigma, \sigma_1, \sigma_2, ...$ désignent des éléments de Subs. Etant donnés $x, y, z \in \mathcal{M}$ et $M \in 2^{\mathcal{M}}_{fn}$, alors*

- *L'application de σ à x sera écrite σx ;*
- *La substitution identité ι est définie par $\iota x = \{x\}$;*

- $M\sigma = \cup_{x \in M} \; \sigma x$;
- $\sigma\,[y/z]$ *est une fonction de substitution définie par* $\sigma\,[y/z]\,x = \begin{cases} \{y\} & si\ z = x \\ \sigma x & sinon \end{cases}$

Soit σ une fonction de substitution, la substitution simultanée, $\mathcal{E}\sigma$, de toutes les occurrences de x dans $\mathcal{E}$ par σx, est définie récursivement sur la configuration $\mathcal{E}$ comme suit :

$$({}_{M}\,[E])\,\sigma = {}_{M\sigma}\,[E] \qquad (\mathcal{E}\ []\ \mathcal{F})\,\sigma = \mathcal{E}\sigma\ []\ \mathcal{F}\sigma$$
$$(\mathcal{E}\ |[L]|\ \mathcal{F})\,\sigma = \mathcal{E}\sigma\ |[L]|\ \mathcal{F}\sigma \qquad (hide\ L\ in\ \mathcal{E})\,\sigma = hide\ L\ in\ \mathcal{E}\sigma$$
$$(\mathcal{E} \gg F)\,\sigma = \mathcal{E}\sigma \gg F \qquad (\mathcal{E}\ [>\ \mathcal{F})\,\sigma = \mathcal{E}\sigma\ [>\ \mathcal{F}\sigma$$
$$(\mathcal{E}\,[b_1/a_1, ..., b_n/a_n])\,\sigma = \mathcal{E}\sigma\,[b_1/a_1, ..., b_n/a_n]$$

La sémantique opérationnelle de maximalité pour l'ensemble des configurations est donnée par la Définition 2.6.

Définition 2.6 *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. $\dfrac{}{{}_{M}[exit] \overset{M^{\delta}x}{\longrightarrow} {}_{\{x\}}[stop]}\, x = get(M)$

2. $\dfrac{}{{}_{M}[a;E] \overset{M^{a}x}{\longrightarrow} {}_{\{x\}}[E]}\, x = get(M)$

3. (a) $\dfrac{\mathcal{E} \overset{M^{a}x}{\longrightarrow} \mathcal{E}'}{\mathcal{F}\ []\ \mathcal{E} \overset{M^{a}x}{\longrightarrow} \mathcal{E}' \qquad \mathcal{E}\ []\ \mathcal{F} \overset{M^{a}x}{\longrightarrow} \mathcal{E}'}$

4. (a) i. $\dfrac{\mathcal{E} \overset{M^{a}x}{\longrightarrow} \mathcal{E}' \qquad a \notin L \cup \{\delta\}}{\mathcal{E}\ |[L]|\ \mathcal{F} \overset{M^{a}x}{\longrightarrow} \mathcal{E}'[y/x]\ |[L]|\ \mathcal{F}\backslash M}\, y = get(M - ((\psi(E) \cup \psi(F)) - M))$

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

   (b) $\dfrac{\mathcal{E} \overset{M^{a}x}{\longrightarrow} \mathcal{E}' \qquad \mathcal{F} \overset{M^{a}y}{\longrightarrow} \mathcal{F}' \qquad a \in L \cup \{\delta\}}{\mathcal{E}\ |[L]|\ \mathcal{F} \overset{M \cup N^{a}z}{\longrightarrow} \mathcal{E}'[z/x]\backslash N\ |[L]|\ \mathcal{F}'[z/y]\backslash M}\, z = get(M - ((\psi(E) \cup \psi(F)) - (M \cup N)))$

5. (a) $\dfrac{\mathcal{E} \overset{M^{a}x}{\longrightarrow} \mathcal{E}' \qquad a \notin L}{hide\ L\ in\ \mathcal{E} \overset{M^{a}x}{\longrightarrow} hide\ L\ in\ \mathcal{E}'}$

   (b) $\dfrac{\mathcal{E} \overset{M^{a}x}{\longrightarrow}_{m} \mathcal{E}' \qquad a \in L}{hide\ L\ in\ \mathcal{E} \overset{M^{i}x}{\longrightarrow} hide\ L\ in\ \mathcal{E}'}$

6. (a) $\dfrac{\mathcal{E} \overset{M^{a}x}{\longrightarrow}_{m} \mathcal{E}' \qquad a \neq \delta}{\mathcal{E} >> F \overset{M^{a}x}{\longrightarrow} \mathcal{E}' >> F}$

   (b) $\dfrac{\mathcal{E} \overset{M^{\delta}x}{\longrightarrow} \mathcal{E}'}{\mathcal{E} >> F \overset{M^{i}x}{\longrightarrow} {}_{\{x\}}[F]}$

7. (a) $\dfrac{\mathcal{E} \overset{M^{a}x}{\longrightarrow} \mathcal{E}' \qquad a \neq \delta}{\mathcal{E}\ [>\ \mathcal{F} \overset{M^{a}y}{\longrightarrow} \mathcal{E}'[y/x]\ [>\ \mathcal{F}\backslash M}\, y = get(M - (\psi(E) \cup \psi(F) - M))$

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

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

8. (a) $\dfrac{\mathcal{E} \overset{M^a x}{\longrightarrow} \mathcal{E}' \qquad a \notin \{a_1,\ldots,a_n\}}{\mathcal{E}[b_1/a_1,\ldots,b_n/a_n] \overset{M^a x}{\longrightarrow} \mathcal{E}'[b_1/a_1,\ldots,b_n/a_n]}$

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

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

2.3 Introduction des durées et des contraintes temporelles

Soit $\mathcal{D}$ un ensemble dénombrable, les éléments de $\mathcal{D}$ désigne des valeurs temporelles. Soit $\mathfrak{T}$ l'ensemble de toutes les fonctions $\tau : Act \to \mathcal{D}$ telles que $\tau(i) = \tau(\delta) = 0$. 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 $_\emptyset[a; b; stop]$; à partir de cet état, la transition $_\emptyset[a; b; stop] \overset{\emptyset a x}{\longrightarrow}_{\{x\}} [b; stop]$ 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 execution (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: $_\emptyset[a; b; stop] \overset{\emptyset a x}{\longrightarrow}_{\{x:a:\tau(a)\}} [b; stop]$. 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 de durées d'action dans les algèbres de processus ne permet pas cependant à elle seule de spécifier des systèmes temps réel [14]. Pour combler ce manque, nous considérons des opérateurs classiques de délai similaires à ceux introduits dans des extensions temporelles de LOTOS, telles que ET-LOTOS ou RT-LOTOS, la sémantique de ces opérateurs étant bien entendu exprimée dans notre 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 :

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

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 $g@t[SP]; E$ (respectivement $i@t\{d\}; E$) t est une variable temporelle mémorisant le temps écoulé depuis la sensibilisation de l'action g (respectivement i) et qui sera substituée par zéro lorsque cette action termine son exécution.

Définition 2.7 [1]*L'ensemble $\mathcal{C}_t$ des configurations temporelles est donné par :*

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

Définition 2.8 *L'opération d'encapsulation étant également distributive par rapport à l'opérateur* Δ*, toute configuration canonique est sous l'une des formes suivantes:*

${}_M[stop]$	${}_M[g@t[SP]; E]$	$\mathcal{E}\ []\ \mathcal{F}$	$\mathcal{E} \gg \mathcal{F}$	$hide\ L\ in\ \mathcal{E}$	$\Delta^d\mathcal{E}$	${}_{\{x:g:d\}}[E(t)]$
${}_M[exit\{d\}]$	${}_M[i@t\{d\}; E]$	$\mathcal{E}\ \vert[L]\vert\ \mathcal{F}$	$\mathcal{E}\ [>\ \mathcal{F}$	$\mathcal{E}\,[b_1/a_1, ..., b_n/a_n]$	${}_M[P]$	

Les opérations d'encapsulation, de formes canoniques et de calcul d'ensembles maximaux définies précedement sur les configations s'étendent de manière naturelle aux configurations temporelles. Une précision reste à faire pour les fonctions de substitution, désormais $Subs = \mathcal{M} \times Act \times \mathcal{D} \longrightarrow 2_{fn}^{\mathcal{M}\times Act\times\mathcal{D}}$. Par exemple $[y : a : 3/x : a : 4]x : a : 4 = y : a : 3$. L'extension aux configurations temporelles se déduit de manière directe.

[1]Pour alléger l'exposé nous continuons à utiliser les mêmes symboles désignant les expressions de comportements, les configurations temporelles, ... Le discours étant clarifié par le contexte.

2.3.1 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 $\rightarrow_\tau \subseteq \mathcal{C} \times Atm \cup \mathcal{D} \times \mathcal{C}$.

Processus *stop*

Considérons la configuration $_M[stop]$, à 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 défintion 2.9.

Processus *exit*

Considérons maintenant la configuration $_M[exit\{d\}]$, 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. $\dfrac{\neg Wait(M)}{_M[exit\{d'\}] \xrightarrow{M\delta_x}_\tau\ _{\{x:\delta:0\}}[stop]} \qquad x = get(M)$

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

Préfixage par une action observable Les mêmes contraintes sont imposées à l'occurence d'une action observable préfixant un processus que celles imposées à l'occurence de l'action δ à partir de la configuration $_M[exit\{d\}]$. Avec l'hypothèse que les actions observables 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 de 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.

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

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

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

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

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

Préfixage par une action interne Dans la configuration ${}_M[i@t\{d\};E]$, une fois l'action interne i sensibilisée, elle doit se produire dans l'intervalle temporel $[0,d]$, ce qui est exprimé par les règles sémantiques suivantes:

1. $$\frac{Wait(M^d)\ or\ (\neg Wait(M^d)\ and\ \forall\varepsilon{:}0<\varepsilon<d\ Wait(M^{d-\varepsilon})) \qquad d>0}{{}_M[i@t\{d'\};E] \xrightarrow{d}_{\tau} {}_{M^d}[i@t\{d'\};E]}$$

2. $$\frac{\neg Wait(M) \qquad d>0}{{}_M[i@t\{d'+d\};E] \xrightarrow{d}_{\tau} {}_M[i@t\{d'\};[t+d/t]E]}$$

3. $$\frac{\neg Wait(M)}{{}_M[i@t\{d\};E] \xrightarrow{M^i x}_{\tau} {}_{\{x:i:0\}}[[0/t]E]} \qquad x = get(M)$$

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'instantiation 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 2.6, dans lesquelles les configurations sont temporelles et la relation de transition est remplacée par $\rightarrow_{\tau}$, complétées par les règles suivantes:

$$\frac{}{\Delta^{d'+d}\mathcal{E} \xrightarrow{d}_{\tau} \Delta^{d'}\mathcal{E}}$$

$$\frac{\mathcal{E} \xrightarrow{d} \mathcal{E}'}{\Delta^0\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}'}$$

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

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

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

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

$$\frac{\mathcal{E} \xrightarrow{d} \mathcal{E}' \qquad \forall d'<d\ \mathcal{E}^{d'} \not\xrightarrow{a}_{\tau}\ \forall a\in L}{hide\ L\ in\ \mathcal{E} \xrightarrow{d}_{\tau} hide\ L\ in\ \mathcal{E}'}$$

$$\frac{\mathcal{E} \xrightarrow{M^\delta x}_{\tau} \mathcal{E}'}{\mathcal{E}>>F \xrightarrow{M^i x}_{\tau} {}_{\{x:i:0\}}[F]}$$

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

$$\frac{\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}' \qquad \mathcal{F} \xrightarrow{d}_{\tau} \mathcal{F}'}{\mathcal{E}[>\mathcal{F} \xrightarrow{d}_{\tau} \mathcal{E}'[>\mathcal{F}'}$$

$$\frac{\mathcal{E} \xrightarrow{d}_{\tau} \mathcal{E}'}{\mathcal{E}[b_1/a_1,...,b_n/a_n] \xrightarrow{d}_{\tau} \mathcal{E}'[b_1/a_1,...,b_n/a_n]}$$

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

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

Définition 2.9 *L'opération d'écoulement de temps* $(.)^d$ *dans une configuration est définie récursivement par:*

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

2.4 Relations de bissimulation

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

2.4.1 Relation de bissimulation temporelle

Définition 2.10 *Soit* $\mathcal{L} = (2^{\mathcal{M}}_{fn} \times Act \times \mathcal{M}) \cup \mathcal{D}$ *et* $\mathbf{R} \subseteq \mathcal{C} \times \mathcal{C}$ *une relation binaire entre les configurations temporelles. Soit* $\mathbf{F} : \mathrm{Re}\,l(\mathcal{C}) \to \mathrm{Re}\,l(\mathcal{C})$ *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* $(\mathcal{E}', \mathcal{F}') \in \mathbf{R}$

(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* $\mathcal{F} \xrightarrow{{}_N a_y}_\tau \mathcal{F}'$ *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* $(\mathcal{E}', \mathcal{F}') \in \mathbf{R}$

(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* $(\mathcal{E}', \mathcal{F}') \in \mathbf{R}$

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

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

Notons que cette relation fait abstraction de l'information de maximalité, alors que les durées d'action sont prises en compte. Nous pouvons constater qu'avec une telle abstraction le résultat serait le même par la substitution de toute occurrence d'un préfixage par une

action a dans E et F par $a; \Delta^{\tau(a)}$ et l'utilisation de la sémantique de ET-LOTOS/RT-LOTOS ainsi que la relation de bissimulation temporelle correspondante.

Exemple 2.1 *Soient les expressions de comportement D-LOTOS*

$$E = a; d; stop|[d]|b; d; stop|[d]|c; d; stop \quad et \quad F = a; d; stop|[d]|((b; c; stop|||c; stop)|[c]|(c; d))$$

et soit la fonction τ_1 *définie par* $\tau_1(a) = 2$*,* $\tau_1(b) = 1$*,* $\tau_1(c) = 1$ *et* $\tau_1(d) = 1$*. Ces deux expressions sont liées par la relation* $\sim_t^{\tau_1}$*. Ces deux expressions s'écrivent respectivement*

$$E' = a; \Delta^2 d; \Delta^1 stop|[d]|b; \Delta^1 d; \Delta^1 stop|[d]|c; \Delta^1 d; \Delta^1 stop$$

$$F' = a; \Delta^2 d; \Delta^1 stop|[d]|((b; \Delta^1 c; \Delta^1 stop|||c; \Delta^1 stop)|[c]|(c; \Delta^1 d; \Delta^1 stop))$$

Il est évident que D-LOTOS permet d'écrire des spécifications plus simples dès qu'on s'intéresse aux durées d'actions. Cependant la relations $\sim_t^{\tau}$ n'est pas une congruence vis à vis de la fonction de durée τ. Pour s'en convaincre, considérons l'exemple suivant:

Exemple 2.2 *Soit la fonction* τ_2 *définie par* $\tau_2(a) = 1$*,* $\tau_2(b) = 2$*,* $\tau_2(c) = 1$ *et* $\tau_2(d) = 1$*.* E *et* F *étant les expressions de comportement de l'exemple 2.1. Les dérivations suivantes sont possibles*

$$\emptyset[E] \xrightarrow{\emptyset a_x} \xrightarrow{\emptyset b_y} \xrightarrow{\emptyset c_z} \xrightarrow{1}_{\{x:a:0\}} [d; stop]|[d]|_{\{y:b:1\}}[d; stop]|[d]|_{\{z:c:0\}}[d; stop]$$

$$\emptyset[F] \xrightarrow{\emptyset a_x} \xrightarrow{\emptyset b_y} \xrightarrow{\emptyset c_z} \xrightarrow{1}_{\{x:a:0\}} [d; stop]|[d]|(_{\{y:b:1\}}[c; stop]|||_{\{z:c:0\}}[stop])|[c]|_{\{z:c:0\}}[d; stop]$$

Dans la première configuration résultante la seule évolution possible est le passage du temps, par contre dans la deuxième configuration on peut observer le début de l'action d. Donc $E \not\sim_t^{\tau_2} F$*.*

Ce résultat est dû au fait que la relation $\sim_t^{\tau}$ fait abstraction des informations de maximalité, et nous pouvons par ailleurs voir que E et F ne sont pas liées par la relation de bissimulation de maximalité [12][22].

2.4.2 Relation de bissimulation temporelle de maximalité

Dans [22] il a été montré que la relation de bissimulation de maximalité est une congruence vis à vis du raffinement d'actions, et par dualité vis à vis de l'association de durées aux actions (conjecture). Ceci nous motive à étendre la relation de bissimulation temporelle de maximalité aux configurations temporelles [12][22].

Définition 2.11 *Soient* $\mathcal{L} = (2^{\mathcal{M}}_{fn} \times Act \times \mathcal{M}) \cup \mathcal{D}$*,* $\mathfrak{F} \subseteq 2^{\mathcal{M}\times\mathcal{M}}$ *et* $\mathbf{R} \subseteq \mathcal{C} \times \mathcal{C} \times \mathfrak{F}$ *une relation binaire entre les configurations temporelles. Soit* $\mathbf{F}^{mt} : \mathrm{Re}\, l(\mathcal{C}) \rightarrow \mathrm{Re}\, l(\mathcal{C})$ *une fonction définie par:* $(\mathcal{E}, \mathcal{F}, f) \in F^{mt}(R)$ *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 $f' = (f\lceil(\psi(\mathcal{E}') - \{x\}))\lfloor(\psi(\mathcal{F}') - \{y\}) \cup \{(x, y)\}$*

   (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 $f' = (f\lceil(\psi(\mathcal{E}') - \{x\}))\lfloor(\psi(\mathcal{F}') - \{y\}) \cup \{(x, y)\}$*

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

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

Proposition 2.2 *Les relations $\sim_t^\tau$ et $\sim_{mt}^\tau$sont des relations d'équivalence.*

Remarque 2.1 *Dans [12][22] nous avons défini la relation de bissimulation de maximalité ($\sim_m$) entre les expressions de comportement Basic LOTOS et nous avons montré qu'elle est la plus large relation préservée par le raffinement d'actions. Nous pouvons remarquer que toute expression de comportement D-LOTOS ne contenant pas d'opérateur temporel est une expression Basic-LOTOS; de plus si toutes les actions sont de durée nulle, alors aucune transition temporelle ne sera engendrée par la relation de transition temporelle de maximalité. Dans ce cas la relation de transition de maximalité coincide avec la relation de transition temporelle de maximalité. Nous pouvons noter également que la définition de la relation $\sim_m$ est identique à celle de la définition 2.11 dans laquelle nous supprimons les points (2.b) et (3.b).*

La proposition 2.3 établit le lien entre la notion de durée d'action et la sémantique de maximalité.

Proposition 2.3 *Soient E et F deux expressions de comportement Basic LOTOS telles que $E \sim_{mt}^{\tau_0} F$. Alors pour toute fonction $\tau \in \mathfrak{T}$ telle que $\tau(i) = \tau(\delta) = 0$, $E \sim_{mt}^{\tau} F$.*

2.4.3 Relation de performance

Dans [14], une notion de durée associée aux actions a été proposée pour une algèbre de processus à la CSP. L'idée était de pouvoir qualifier des systèmes qui ont la même performance

sur n'importe quelle machine. Pour cela, il a été supposé que les systèmes s'exécutent sur des machines dotées d'un nombre de processeurs suffisamment important pour que toute actions offertes puissent être exécutées. Sémantiquement parlant, toutes les actions sont considérées urgentes.

Pour rapprocher les notations, la relation de transition définie dans [14] peut être exprimée par $E \xrightarrow{a,n}_{p\tau} E'$ indiquant que le temps écoulé depuis le démarage du système jusqu'à la fin de a est égal à n. ($E, E' \in \mathcal{C}_p$ étant des configurations exprimant les états du système [14]). La relation de performance est définie par:

Définition 2.12 *Soit τ une fonction de durée, une relation binaire $R^{p\tau} \subseteq \mathcal{C}_p \times \mathcal{C}_p$ est une τ-performance si $(E, F) \in R^{p\tau}$ implique $\forall(a, n) \in Act \times \mathcal{D}$:*

- *Si $E \xrightarrow{a,n}_{p\tau} E'$ alors il existe $F \xrightarrow{a,n}_{p\tau} F'$ tel que $(E', F') \in R^{p\tau}$.*
- *Si $F \xrightarrow{a,n}_{p\tau} F'$ alors il existe $E \xrightarrow{a,n}_{p\tau} E'$ tel que $(E', F') \in R^{p\tau}$.*

Deux configurations E et F sont liées par cette relation, notée $E \sim_p^\tau F$, s'il existe une relation τ-performance contenant (E, F).

Une première remarque concerne l'urgence imposée aux actions. Dans notre modèle, ceci peut être mis en oeuvre en intériorisant toutes les actions (utilisation de l'opérateur *hide*) ($i(a)$ dans une transition signifie que l'action interne i correspond à l'intériorisation de l'action a). Par conséquent, deux expressions de comportement Basic LOTOS dont toutes les actions sont intériorisées ont les mêmes performances si elles sont liées par la relation de bissimulation temporelle.

La deuxième remarque est liée à la prise en compte du début d'exécution des actions sachant que d'autres actions sont en cours d'exécution. Cet aspect ne peut pas être pris en compte par la relation de transition $\longrightarrow_{p\tau}$, alors que la sémantique de maximalité peut parfaitement prendre en compte cet aspect. Cette différence a un impact direct sur les systèmes pouvant être identifiés par chacune des approches. Dans ce but, nous considèrons l'exemple des deux systèmes E et F ci-dessous issus de [14]. Les deux systèmes sont exprimés dans le modèle des structures d'événements primaires.

Soient τ_1 et τ_2 deux fonctions de durée définies par $\tau_1(a) = \tau_1(b) = \tau_1(c) = 1$ et $\tau_2(a) = 5$, $\tau_2(b) = 1$, $\tau_2(c) = 1$. Comme mentioné dans [14], nous avons les relations suivantes $E \sim_p^{\tau 1} F$ et $E \nsim_p^{\tau 2} F$. En fait, en considérant la fonction τ_2, l'action c causée par a dans F peut être accomplie à la date 6 ce qui n'est pas le cas dans E. Cependant, étant donné que les actions sont urgentes, l'exécution de l'action b dans F se termine à la date 1 et l'action c causée par b commence son exécution dès que possible, donc l'action c causée par a ne devrait pas être exécutée. Cette discrimination n'a pas pu être prise en compte par la relation de bissimulation $\sim_p^\tau$ parce que la relation de transition $\longrightarrow_{p\tau}$n'est pas assez fine

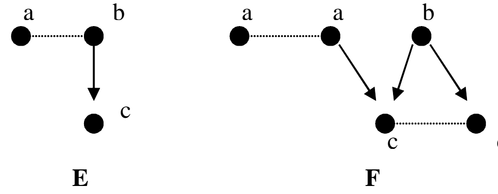


Figure 2.2: Structures d'événements primaires

pour prendre en compte l'exécution concurrente des actions avec des durées non nulles. Bien que, que dans le cas général, la relation $\sim_t^\tau$ ne soit pas une congruence vis à vis de la fonction de durée τ (voir exemples 2.1 et 2.2), le lecteur peut vérifier que $E \sim_t^{\tau 1} F$ et $E \sim_t^{\tau 2} F$. De la proposition 2.3 nous pouvons déduire le corollaire suivant:

Corollaire 2.1 *Soient E et F deux expressions de comportement Basic LOTOS dont les actions sont toutes intériorisées. Les actions intériorisées étant distinguées par les actions observables correspondantes dans la sémantique, tel que $E \sim_{mt}^{\tau_0} F$. Alors pour toute fonction $\tau \in F$ tel que $\tau(i) = \tau(\delta) = 0$, $E \sim_{mt}^{\tau} F$.*

On considérant que l'éxécution d'une action prend des durées différentes sur des processeurs différents, le corollaire 2.1 nous donne le moyen de décider si deux systèmes ont les mêmes performances sur n'importe quelle machine et cela en faisant abstraction des durées, c'est à dire en vérifiant que les deux systèmes sont liés par la relation $\sim_{mt}^{\tau_0}$.

2.5 Spécification de la latence

L'opérateur de latence de RT-LOTOS est d'un intérêt considérable pour la spécification de systèmes temps réel [11][6][8][17][24]. Il peut être introduit sans problème particulier dans le langage D-LOTOS. Nous préférons cependant montrer ici comment l'utilisation conjointe de la notion de durée d'action associée à l'opérateur @ permet de spécifier une latence similaire à celle exprimée par RT-LOTOS.

Expliquons tout d'abord la finalité de l'opérateur de latence à travers l'exemple de deux expressions RT-LOTOS $E_1 = a\{u\}; F$ et $E_2 = \Omega^l a\{u\}; F$ en faisant l'hypothèse que $l < u$. Dans E_1 l'action a peut s'exécuter dans l'interval temporel $[0, u]$ sous condition que l'environnement accepte de se synchroniser sur cette action dans cet intervalle. Au delà de cet intervalle, l'action a ne pourra plus être offerte, dans ce cas E_1 se transforme dans le processus *stop*. Dans l'expression E_2, nous distinguons deux intervales temporels, $I = [0, l]$ et $J = [l, u]$; durant l'intervalle I, l'action a peut s'exécuter sous condition que l'environnement et le processus E_2 acceptent de se synchroniser ensemble sur cette action; ainsi, nous pouvons spécifier que le processus peut refuser de se synchoniser pour des raisons non explicitées ! Par contre, durant l'intervalle J , l'action a peut ne pas s'exécuter en raison d'un refus de

l'environnement. Cependant, dans le cas où l'action a ne pourra pas s'exécuter, nous ne pouvons pas avoir connaissance de l'origine de ce refus d'exécution. Donc, d'un point de vue observable, ces deux systèmes sont identiques. La différence de comportement peut être perçue dès que l'on intériorise l'action a. Dans l'expression $E_1' = hide\ a\ in\ E_1$, l'action a est urgente et sera executée dès qu'elle est offerte. Par contre dans l'expression $E_2' = hide\ a\ in\ E_2$, l'action a peut ne pas s'exécuter durant l'intervalle I, et elle ne devient urgente qu'à la fin de cet intervalle. L'utilité de l'opérateur de latence réside donc dans la préservation de l'indéterminisme d'exécution des actions intériorisées.

Remarquons que la levée de l'hypothèse d'atomicité des actions implique que toute action peut être considérée comme un processus en exécution, cependant la durée d'éxécution du processus n'a pas nécessairement une valeur déterminée due au comportement non déterministe éventuel de ce processus. Ceci nous fait penser qu'il serait plus judicieux de considérer des durées d'action variables de la forme $[m, M]$ indiquant que la durée d'exécution d'une action est comprise entre une durée minimale égale à m et une durée maximale égale à M. Donc, si une action a munie d'une durée $[m, M]$ débute son exécution à un instant ta, cette action peut terminer son exécution dans l'intervalle temporel $[ta + m, ta + M]$. Cette idée est facilement réalisable en reconsidérant les points suivants:

- Nous définissons deux fonctions temporelles $min, max : \mathcal{G} \rightarrow \mathcal{D}$ qui associent respectivement une durée minimale et une durée maximale à toute action observable. L'action interne i et l'action δ sont par hypothèse de durée nulle. Evidement pour toute action $g \in \mathcal{G}$, $min(g) \leq \max(g)$.
- A chaque fois qu'une action observable g commence son exécution, nous lui associons la durée $max(g)$ dans la configuration résultante (voir les règles sémantiques).
- Etant donné un ensemble $M \in 2_{fn}^{\mathcal{M}\times Act\times\mathcal{D}}$, désormais le prédicat $wait$ est défini par $wait(M) = \exists x : g : d \in M\ telque\ \max(g) - d < \min(g)$

  Il est clair que la sémantique présentée dans la section 2.3.1 est un cas particulier de celle-ci dans laquelle $min(g) = max(g)$ pour toute action observable $g \in \mathcal{G}$.

Soit l'exemple de la spécification d'un médium de communication, introduit dans [11], dont le délai de transmission est compris dans un intervalle $[m, M]$. Soit a l'action correspondant à l'émission d'un message sur le médium, et b l'action de réception de ce message après un délai non déterministe. L'action $error$ caractérise la situation d'erreur dans laquelle l'environnement n'est pas prêt à recevoir le message offert par le médium de transmission. En utilisant l'opérateur de latence de RT-LOTOS, la spécification peut être exprimée ainsi [11]:

$$Medium = a; (\Delta^m \Omega^{M-m} b\{M - m\};\ Medium\ []\ \Delta^{M+e} error;\ stop)$$

En considérent la notion de durée des actions, nous pouvons distinguer deux types d'actions: d'une part, les actions a et b, qui représentent respectivement l'émission et la

réception d'un message, et que nous pouvons considérer de durée nulle, et l'opération de transmission proprement dite qui peut durer entre m et M, qui est représentée par l'action c de durée comprise entre m et M. Ceci nous conduit á la spécification suivante avec D-LOTOS:

$$Medium = hide\ c\ in\ a; (c@t; (b; medium\ []\ \Delta^{M-t+e} error;\ stop))$$

2.6 Conclusion et perspectives

Dans cet article, nous avons défini le modèle de spécification D-LOTOS qui intègre deux concepts à savoir la spécification des contraintes temporelles et la prise en compte de durées d'actions.

D'un point de vue syntaxique, nous sommes restés très proches des formalismes ET-LOTOS, mais d'un point de vue sémantique nous nous sommes abstrait de l'hypothèse d'atomicité des actions qui est imposée par la sémantique d'entrelacement du parallélisme.

Dans ce but, nous avons donné une sémantique de maximalité temporelle à D-LOTOS. Cette approche nous a permis de définir des relations de bissimulation temporelle pour analyser le comportement de systèmes temps réel d'une part ainsi qu'une relation de performance permettant de caractériser des systèmes concurrents ayant des performances identiques quelque soit l'architecture sous-jacente sur laquelle ces systèmes s'exécutent.

Un autre résultat concerne la relation de bissimulation temporelle de maximalité. Nous avons montré que pour les expressions de comportement Basic LOTOS, deux expressions de comportement qui sont maximalement bissimilaire sont également maximalement temporellement bissimilaires pour toute fonction de durée τ. Ce résultat n'est pas tout à fait surprenant parce qu'il confirme notre idée de départ qui considère que les deux notions de rafinement d'actions et de durée associée aux actions sont similaires pour l'étude des sémantiques dites de vrai parallélisme. Notons qu'il a été montré que la relation de bissimulation de maximalité est la plus large relation préservée par le raffinement d'actions [13][12][22].

Différents résultats restent à établir pour le langage D-LOTOS, en particulier la définition du sous-langage le plus large pour lequel les propriétés précédentes restent valables. Pour voir le problème, considons les expressions de comportement $E = hide\ b\ in\ a\{2\}; stop|[a]|b; a\{1\}; stop$ et $F = hide\ b\ in\ b; a\{1\}; stop$ et soient τ_1 et τ_2 deux fonctions de durée définies respectivent par $\tau_1(a) = \tau_1(b) = 1$ et $\tau_2(a) = 1$ et $\tau_2(b) = 3$. Nous pouvons vérifier que $E \sim_{mt}^{\tau_1} F$ et $E \not\sim_{mt}^{\tau_2} F$.

L'introduction de durées dynamiques a permi en outre de formaliser la notion de latence telle qu'elle a été définie dans le formalisme RT-LOTOS; il reste encore à étudier l'impact de cette forme de durée dynamique sur les propriétés des relations de bissimulation temporelles que nous avons définies.

Par ailleur nous proposons l'extension de notre approche au langage RT-LOTOS avec des durées statiques et sa comparaison formelle avec le langage D-LOTOS avec des durées dynamiques. Nous pensons que les deux langages devraient avoir le même pouvoir d'expression.

La prise en compte des données pouvant se faire de manière similaire à ce qui a été développé pour d'autres extensions temporelles de LOTOS, le développement d'outils de vérification reste à faire et nous faisons le pari que la sémantique utilisée permettra d'hériter des résultats disponibles dans la littérature sur les sémantiquesde vrai parallélisme [21][23].

2.7 Preuves des résultats

Preuve de la Proposition 2.2: Les preuves pour ces deux relations sont respectivement similaires à celles de la relation de bissimulation temporelle entre les expressions RT-LOTOS [10][11] et la relation de bissimulation de maximalité sur les expressions Basic LOTOS et les arbres maximaux [12][22].<>

Preuve de la Proposition 2.3: $E \sim_{mt}^{\tau 0} F$ par hypothèse, donc il existe une relation minimale de bissimulation temporelle de maximalité R telle que $({}_{\emptyset}[E],{}_{\emptyset}[F],\emptyset) \in R$. Soit $\tau \in F$ une fonction qui associe une durée à toute action telle que $\tau(i) = \tau(\delta) = 0$. L'idée est de construire une relation de bissimulation temporelle de maximalité R^{τ} dont les éléments sont liés par la relation de transition $\rightarrow_{\tau}$ et $({}_{\emptyset}[E],{}_{\emptyset}[F],\emptyset) \in R^{\tau}$.

Remarquons que $\forall(\mathcal{E},\mathcal{F},f) \in R$, si Si $\mathcal{E} \xrightarrow{Ma_x}_{\tau} \mathcal{E}'$, alors il existe $\mathcal{F} \xrightarrow{Na_y}_{\tau} \mathcal{F}'$ tel que

1. (a) pour tout $(u,v) \in f$ si $u \notin M$ alors $v \notin N$; et
   (b) $(\mathcal{E}',\mathcal{F}',f') \in \mathbf{R}$, avec $f' = (f\lceil(\psi(\mathcal{E}') - \{x\}))\lfloor(\psi(\mathcal{F}') - \{y\}) \cup \{(x,y)\}$

Cependant, dans la sémantique de maximalité, en considérant une configuration $\mathcal{E}$ toute seule (E n'est pas considérée dans un contexte de relation de bissimulation de maximalité), $\psi(\mathcal{E})$ représente l'ensemble des actions qui sont potentiellement en cours d'exécution. Par contre la mise en relation de $\mathcal{E}$ avec une configuration $\mathcal{F}$ dans la relation $\mathbf{R}$ peut nous faire déduire les évenements maximaux correspondant aux actions qui ont effectivement terminées leur exécution. En effet: d'après la règle (1), si une action indexée par $u \in \psi(\mathcal{E})$ tel que $u \in Dom(f)$ reste maximale ($u \notin M \Rightarrow u \in \psi(\mathcal{E}')$). Alors l'action correspondante indexée par $v = f(u)$ reste maximale aussi ($v \notin N \Rightarrow v \in \psi(\mathcal{F}')$). L'inverse n'est pas vrai, si $u \in M$ il se peut que l'action correspondante reste maximale, mais dans ce cas nous pouvons déduire qu'elle a terminé son exécution. Ceci peut être mieux perçu par l'association des durées aux actions. En fait la transition $\mathcal{E} \xrightarrow{Ma_x}_{\tau} \mathcal{E}'$ n'est possible que si toutes les actions indexées par M teminent leur exécution, ce qui revient à dire que les actions associées liées par la fonction f ont aussi terminé leur exécution même si elles sont maximales.

D'autre part, les ensembles $\psi(\mathcal{E})$ et $\psi(\mathcal{F})$ sont liés par la relation suivante (E et F sont respectivement issues de ${}_{\emptyset}[E]$ et ${}_{\emptyset}[F]$) [22]:

Soit $A \subseteq \psi(E)$ tel que $\forall x \in A$, $x \notin Dom(f)$ alors il existe $\{(\mathcal{E}_1,\mathcal{F}_1,f_1),...,(\mathcal{E}_n,\mathcal{F}_n,f_n)\} \subseteq R$ tel que $(\mathcal{E}_n,\mathcal{F}_n,f_n) = (\mathcal{E},\mathcal{F},f)$ et $\forall 1 \leq i < n$, $\mathcal{E}_i \xrightarrow{M_i a i_{xi}} \mathcal{E}'_{i+1}$ et $\mathcal{F}_i \xrightarrow{N_i a i_{yi}} \mathcal{F}'_{i+1}$ tel que $x \in Dom(f_1)$ et $\forall 1 < j \leq n$, $x \notin Dom(f_j)$ et $x \in \psi(\mathcal{E}_j)$ et $f_1(x) \in N_1$. Donc, malgré que x reste maximal dans $\mathcal{E}_2$, l'action correspondante a terminé de s'exécuter. De ce fait on déduit que pour tout $(\mathcal{E},\mathcal{F},f)$, seules les actions qui sont liées par f sont potentiellement en cours d'exécution, toutes les autres (celles indéxées par $\psi(\mathcal{E}) - Dom(f)$ et $\psi(\mathcal{F}) - Codom(f)$) ont terminé de s'exécuter. Cette remarque nous conduit à la proposition de la relation R^{τ} définie comme suit:

Pour tout $(\mathcal{E}, \mathcal{F}, f) \in R$ avec $Dom(f) = \{x_1, ..., x_n\}$ et $\sigma = [x_1 : |x_1| : \tau(|x_1|)/x_1 : |x_1| : 0]...[x_n : |x_n| : \tau(|x_n|)/x_n : |x_n| : 0]$, on pose :

- $(\mathcal{E}\sigma, \mathcal{F}\sigma, f) \in R^\tau$, et
- Pour tout t tel que $0 < t < \max(\tau(|x_1|), ..., \tau(|x_n|))$, $((\mathcal{E}\sigma)^t, (\mathcal{F}\sigma)^t, f) \in R^\tau$.

  Où $|x_i|$ désigne le nom de l'action indexée par x_i. Evidement cette définition de R^τ est implicite (R^τ étant infinie si le domaine de temps est dense). On peut constater que la relation R^τ est une relation de bissimulation temporelle de maximalité contenant $({}_{\emptyset}[E], {}_{\emptyset}[F], \emptyset)$ et satisfaisant la relation de transition $\rightarrow^\tau_{mt}$, donc $E \sim^\tau_{mt} F$.<>

Preuve du Corollaire 2.1 : Les termes considérés par ce corollaire sont aussi des termes Basic LOTOS, de ce fait la proposition 2.3 implique le résultat.<>

Chapitre 3

Modèles sémantiques pour le temps réel

Ce chapitre présente certains modèles de spécification et certains modèles sémantiques pour la description des systèmes temps réel.

Ces modèles peuvent être considérés comme des extensions des modèles de spécification du parallélisme par l'introduction de mécanismes et d'opérateurs permettant l'expression de contraintes temporelles auxquelles sont soumises les comportements des systèmes critiques. Parmi ces modèles nous citons les réseaux de Petri temporels, la technique de description formelle ET-LOTOS, la technique de description formelle RT-LOTOS. Ces modèles ont comme particularité d'avoir une sémantique temporelle d'entrelacement dont l'atomicité temporelle et structurelle est une hypothèse inhérente à cette sémantique. Un modèle plus récent, appelé D-LOTOS, a été défini pour la spécification des systèmes temps réel tout en offrant la possibilité de considérer l'approche de raffinement de spécifications consistant en le remplacement des actions par des processus tout au long de la trajectoire de conception de ces systèmes. Ceci est rendu possible grâce à l'adoption de la sémantique de maximalité temporelle pour ce langage ; cette dernière est une sémantique du vrai parallélisme.

Les modèles sémantiques temps réel sont utilisés pour exprimer la sémantique des modèles de spécification sus-indiqués. Deux familles de modèles sémantiques sont à souligner :

- Les modèles dits d'entrelacement : Ils se caractérisent par l'interprétation de l'exécution parallèle de deux actions par leurs exécutions alternées. Cette abstraction peut être acceptable dans le cas où les actions sont atomiques. Cependant, une fois le système est exprimé dans ces modèles, il devient impossible de retrouver toute information relative à l'exécution des actions parallèles.

- Les modèles dits du vrai parallélisme : Ils se caractérisent par la prise en compte de l'hypothèse de non atomicité des actions, et ceci dès les premières étapes de conception.

Les retombés de cette approche est double ; elle permet la définition de méthodes de conception formelle basées sur le raffinement d'actions ainsi que l'attribution de durées aux actions, ce qui facilite la spécification de systèmes réels.

3.1 Automate temporel

Définition 3.1 *Une séquence de temps est définie comme étant un ensemble de points aléatoires appartenant à l'ensemble des réels positifs :* $t = t_1t_2t_3 \ldots t$ *; avec la satisfiabilité des deux conditions suivantes.*

- – ***La monotonicité*** *:* $ti < t_i+1$ *pour* $i \geq 1$ *cette propriété nous assure que l'instant qui va venir dans le future il est tout à fait supérieur à celui de présent c'est pour assurer que le temps évolue toujours.*
  – ***La progression :*** *Pour chaque instant* t *qui appartient à* $\mathbb{R}^+$*, il existe un instant* $t\prime$ *appartient au même ensemble et qui vérifie la relation* $t\prime > t$ *d'après cette définition on constate que le temps n'est pas continu il est discret donc il ne modélise pas le temps réel[1].*

Définition 3.2 *Soit* ■**Σ** *l'ensemble d'alphabet. Un mot temporel de* E *est le couple* (δ, t) *où* δ *est un mot* $\delta = \delta_1\delta_2\delta_3...$ *de* Σ^* *et* $t = t_1t_2t_3 \ldots$ *est une séquence de temps. A chaque instant* t_i *on a une lecture du symbole* δ_i*.*

Remarque 3.1 *Notons que l'occurrence des symboles* δ_i *arrivent et terminent à l'instant* t_i*, entre autre ils obéissent à l'hypothès d'atomicité temporelle.*

Définition : Un langage temporel sur Σ est un ensemble de mots temporels engendrés par Σ c'est à dire $LT = \{ \ (\delta, t) \mid (\delta, t) \ est \ un \ mot \ temporel \ \}$

Définition 3.3 *[1]Un automate temporel est une table de transition de la forme* (Σ, S, S_0, H, E) *tel que*

- – Σ *: ensemble d'alphabets*
  – S *: ensemble fini d'états*
  – S_0 *: ensemble fini d'états initiaux*
  – H *: ensemble fini d'horloges.*

vérifiant les conditions suivantes

- Initialisation des horloges

A toute transition, une ou plusieurs initialisations d'horloges sont possibles.

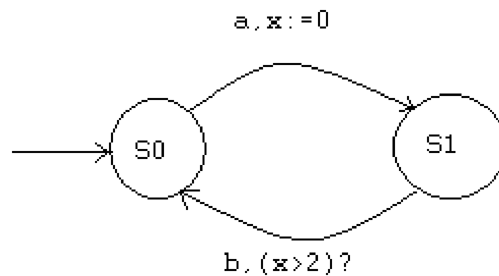


Figure 3.1: Table de transition temporelle.

- Contraintes temporelles

Le franchissement d'une transition temporelle n'est possible que si les contraintes temporelles exprimées en fonctions des valeurs des horloges au moment du franchissement de cette transition sont vérifiées.

- Continuité

Le temps est considéré comme global, en d'autre terme quand le temps progresse toutes les horloges progressent de la même valeur.

E : ensemble de transitions, une trnasition de la forme $(s_i,\ s_{i+1}, a_i,\delta_i,\ \lambda_i)$ signifie que le passage de $s_i$vers s_{i+1} $(s_i \rightarrow\ s_{i+1})$,à l'instant t_i en lisant le symbole a_i , est possible ssi:

- Les contraintes δ_i des horloges sont valides.
- Les horloges aui appartiennent à l'ensemble λ_i se réinitialisent à zero. .

Voici deux figures qui illustrent deux tables de transition temporelle .

L_1 est le langage temporel suivant :

$L_1 = \{((ab)^\omega, \tau) \mid \blacksquare i.(\tau_{2i} > \tau_{2i-1} + 2)\}$.

En considérant la table de transition de la figure3.1, l'état initial est s_0. Il existe une seule horloge x. une notation de la forme $x := 0$ sur l'arc correspond à la transition temporelle de s_0 vers s_1 qui signifie l'initialisation de l'horloge x dès que cette transition est franchise. De la même manière, la contrainte $(x > 2)$? sur l'arc correspond définie une condition de franchissement de la transition s_1 vers s_0. Durant cette transition, l'horloge x se réinitialise à 0. A l'etat s_1, après la lecture du symbole a, l'horloge x peut s'incrémentée. La transition s_1 vers s_0 ne peut être franchis que durant l'intervalle $[2, +\infty[$, c'est à dire si la valeur de

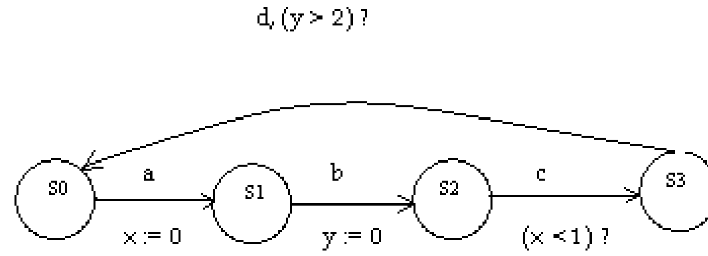


Figure 3.2: Table de transition Temporelle à deux horloges

l'horloge x est supérieur à 2 et ainsi de suite. Nous pouvons constater que le délai entre la lecture du symbole a et le symbole b suivant est toujours supérieur à 2.

L_2 est le langage temporel suivant :

$L_2 = \{((abcd)^{\omega}, \tau) \mid\blacksquare\ ((\tau_{4j+3} < \tau_{4j+1} + 1)\ et\ (\tau_{4j+4} > \tau_{4j+2} + 2))\}$.

Cette figure représente un automate temporel constitué de quatre états $s_0,\ s_1,\ s_2\ et\ s_3$. L'initialisation de l'horloge x est faite à chaque transition temporelle s_0 vers s_1 permettant la lecture du symbole a. L'occurrence de l'action c est conditionnée par la contrainte $(x < 1)$? associé à la transition temporelle s_2 vers s_3. Ceci signifie que le temps séparant l'occurrence du symbole a et l'occurrence du symbole c suivant est inférieure à 1. Les contraintes sur l'horloge y assure que le temps séparant l'occurrence du symbole b et l'occurrence du symbole d suivant est supérieur à 2.

3.1.1 Exécution

Définition 3.4 *[1]Une exécution Ex, dénotée par* $(\overline{s}, \overline{v})$*, de la table de transition* $< \Sigma, S, s_0, H, E >$ *sur le mot temporel* (δ, τ) *est une séquence infinie de la forme :* $Ex : < s_0, v_0 > \longrightarrow^{\sigma_1}_{\tau_1} < s_1, v_1 > \longrightarrow^{\sigma_2}_{\tau_2} < s_2, v_2 > \ldots$ *Avec* $s_i \in S,$ *et* $v_i \in [C \to \mathbb{R}]$*, pour tout* $i \geq 0$*, satisfaisant les deux contraintes suivantes :*

- – ***Initialisation :*** $s_0 \in S,$ *et* $v_0(x) = 0$ *pour tout* $x \in C$.
  – ***Séquencement :*** *pour tout* $i \geq 1$*, il y a un arc dans E de la forme* $< s_{i-1}, s_i, \sigma_i, \delta_i, \lambda_i >$ *tel que* $(v_{i-1} + \tau_i - \tau_{i-1})$ *vérifie* δ_i *et* v_i *égale à* $[\lambda \ \to 0](v_{i-1} + \tau_i - \tau_{i-1})$.

Exemple 3.1 *Considérant la table de transition temporelle de l'exemple illustré par la figure3.2 suivante* $(a, 2) \to (b, 2.7) \to (c, 2.8) \to (d, 5) \to \ldots$

Soit la liste $[x, y]$ exprimant les valeurs des deux horloges x et y, le segment initial de l'exécution est :

$< s,[0,0] > \rightarrow_2^a < s1,[0,2] > \rightarrow_{2.7}^b < s2,[0.7,0] > \rightarrow_{2.8}^c < s3,[0.8,0.1] > \rightarrow_5^d < s0,[3,2.3] >$ ...

Au cours de l'exécution $Ex = (\overline{s}, \overline{v})$ sur le mot temporel (σ, τ), les valeurs des horloges à l'instant t entre τ_i et τ_{i+1} sont calculées à partir de la relation $(v_i + t - \tau_i)$. Lorsqu'il y a une transition de l'état s_i vers s_{i+1}, on utilise la relation $(v_i + \tau_{i+1} - \tau_i)$ pour évaluer la contrainte sur l'horloge de cette transition. Par contre, dans le l'instant τ_{i+1} les horloges qui se sont initialisées sont représentées par 0.

Remarque 3.2 *Nous pouvons noter que les automates d'états finis sont des automates temporels dont l'ensemble des horloges est vide.*

3.2 STEM Temporel

Dans cette section nous étendons le modèle des systèmes de transitions étiquetées maximales par l'introduction des horloges locales tel est le cas des automates temporels[1]. Une différence est à souligner et qui concerne la prise en compte des durées d'actions intrinsèquement présentes dans la sémantique de maximalité. D'où la différence entre les deux approches, entrelacée et vrai parallélisme[22].

3.2.1 Table de transition temporelle

Définition 3.5 *Soit* $A : (Atmt, S, s_0, H, Tr)$

- – S *: l'ensemble des états tel que* $[s \in S \Longrightarrow s \in 2_{fn}^{\mathcal{M} \times\ act\ \times \mathcal{D}}]$.
  - – s_0 *: l'état initial.*
  - – H *: l'ensemble des horloges parcouru par* C_x, C_y *où* x *et* y *sont des noms d'événements du stem associé.*
  - – Tr *: l'ensemble de transitions tel que :*
  - – $Tr \in S \times S \times Atmt \times Cont \times 2_{fn}^{H}$*; à titre d'exemple* $(s_i, s_{i+1}, {}_M a_x, \delta_i, \lambda_i)$ *représente une transition de l'état* s_i *vers l'état* s_{i+1} *en lisant l'atome temporel* ${}_M a_x$. δ_i *est la contrainte associée* ,$\lambda_i \subseteq H$ *est l'ensemble des horloges qui seront initialisées par cette transition.*

Remarque 3.3 δ *: une expression booléenne dont la syntaxe.*

$$\delta ::= true \mid false \mid \delta\ or\ \delta \mid \delta\ and\ \delta \mid not\ \delta \mid min \sqsubseteq\ id \sqsubseteq max \quad \sqsubseteq \in \{<, \leq\}$$

Exemple 3.2 *Soit la spécification D_LOTOS suivante :*

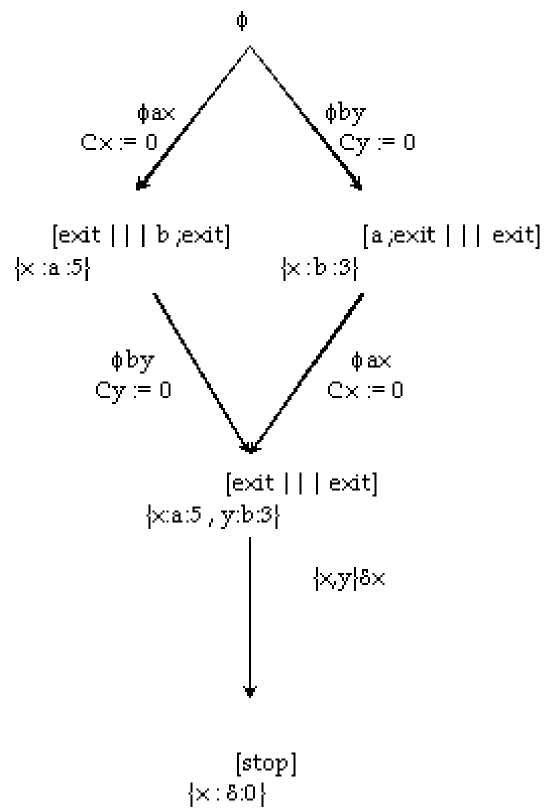


Figure 3.3: STEM Temporel

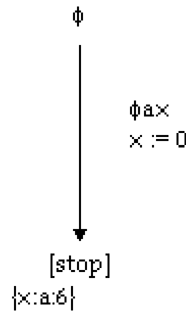


Figure 3.4: Evénement x comme horloge locale

$SYSTEM \quad proc[a[5], b[3]] := a; exit \,|||\, b; exit \quad ENDSYS$

Le STEM temporel associé à cette spécification est donné par la figure3.3

Dans un stem temporel, une transition représente toujours le début d'exécution d'une action. Etant donnée que les horloges sont utilisées pour calculer le temps dissipé après le début d'exécution des actions, nous identifions toute horloge par le nom d'événement associé à l'action correspondante. Dans l'exemple précédent, le nom d'événement associé à l'action a étant x, de ce fait nous définissons l'horloge notée $_{C_x}$.

Lorsqu'il n y a pas un risque d'ambiguïtés, nous pouvons identifier les noms des horloges aux noms des événements. D'où la figure3.4. qui représente la spécification D-LOTOS $SYSTEM\ proc[a[6]] := a; stop\ ENDSYS$.

Dans ce modèle on propose d'utiliser un ensemble d'horloges locales H tel que chaque horloge locale $C_x \in H$ est associée à un événement x qui identifie le début d'exécution de l'action a. La terminaison de l'action a peut être connue dès que l'horloge C_x prend la valeur $\tau(a)$.

A l'état initial s_0, aucune action n'a commencé son exécution, ce qui correspond à $\psi(s_0) = \phi$. Par conséquent l'ensemble des horloges à l'état initial est vide.

La méthode de génération des stems temporels associés à des spécifications D-LOTOS sera présentée en détail dans le chapitre Implantation.

3.2.2 Traces d'exécution

Définition 3.6 *Une exécution Ex, dénotée par* $(\overline{s}, \overline{v})$*, de la table de transition* $< ATMt, S, s_0, H, Tr >$ *est une séquence infinie de la forme :* $Ex :< s_0, v_0 > \rightarrow_{t_1}^{atm_1} < s_1, v_1 > \rightarrow_{t_2}^{atm_2} < s_2, v_2 >$ $\ldots$*Avec* $s_i \in S$*, et* $v_i \in [2^H_{fn} \rightarrow \mathbb{R}]$*, pour tout* $i \geq 0$*, satisfaisant les deux contraintes suivantes :*

- – ***Initialisation :*** $s_0 \in S$*, et* $v_0(x) = 0$ *pour tout* $x \in H$*.*
  – ***Séquencement :*** *pour tout* $i \geq 1$*, il existe un arc dans* Tr *de la forme* $<$

$s_{i-1}, s_i, atm_i, \delta_i, \lambda_i >$ *tel que* $(t_{i-1} + \tau_i - \tau_{i-1})$ *vérifie* δ_i *et* t_i *est égale à* $[\lambda \to 0](t_{i-1} + \tau_i - \tau_{i-1})$

Exemple 3.3 *Soit l'expression de comportement D-LOTOS* $P := delay(5)\ a[9]\{2\}; stop\ |||\ b[13]\{6\}; stop.$ *Soit la liste* $[C_x, C_y]$ *exprimant les valeurs des deux horloges, le segment d'exécution initiale est:* $< s, \phi > \ \ \to_2^{\phi b_x} < s_1, [C_x = 0] > \ \ \to_6^{\phi a_y} < s_2, [C_x = 4, C_y = 0] >$

Au cours de l'exécution de $Ex = (\overline{s}, \overline{v})$ sur le mot temporel (σ, τ), les valeurs des horloges à l'instant t situé entre τ_i et τ_{i+1} sont calculées à partir de la relation $(v_i + t - \tau_i)$. Lorsqu'il y a une transition de l'état s_i vers l'état s_{i+1}, on utilise la relation $(v_i + \tau_{i+1} - \tau_i)$ pour évaluer la contrainte de l'horloge de cette transition. Cependant, à l'instant τ_{i+1} les horloges qui se sont initialisées sont représentées par 0.

3.2.3 Conclusion

Ce chapitre a été consacré à la présentation du modèle sémantique temporel d'entrelacement, à savoir le modèle des automates temporelles[7][25][16], et l'introduction du modèles des systèmes de transitions étiquetées maximales temporels. Ce dernier, étant basée sur la sémantique de maximalité dont les actions sont intrinsèquement non atomiques, intègre dans sa définition la prise en compte explicite des durées d'actions et des contraintes temporelles. L'identification des horloges étant faite par l'utilisation des noms d'événement associés aux actions, ceci à permis de construire les horloges à la volée. Ce point nous distingue des autres approches dans lesquelles l'ensemble des horloges est défini statiquement dès le départ par l'utilisation d'algorithmes spécifiques[7].

Chapitre 4

Implantation

.

Ce chapitre est consacré à la conception et l'implantation de notre compilateur de génération de stem temporels à partir de spécifications D-lotos, il est organisé comme suit :

- La première partie : introduit le paradigme de programmation fonctionnelle et présente les principales caractéristiques du langage CAML.
- La deuxième partie : donne une introduction générale à l'architecture des compilateurs.
- La troisième partie : décrit les phases de conception de notre outils entre autre l'analyse lexicale , syntaxique , sémantique et la génération du code objet.

4.1 Le paradigme de programmation fonctionnelle

4.1.1 Principes des langages fonctionnels

Exécution et évaluation

Les langages de programmation classiques présentent deux paradigmes de calcule assez différents. Le premier, est le paradigme impératif, dans lequel un programme est une suite d'instruction donnée à une machine pour être exécutée. L'opération d'affectation est l'exemple typique d'une instruction purement impérative. Lorsqu' on écrit :

$\alpha := \beta$

On donne l'ordre à la machine de remplacer le contenu de l'emplacement mémoire correspondant à la variable α par le contenu de l'emplacement mémoire correspondant à la variable β par contre si on veut calculer la valeur de l'expression $\sqrt{\frac{(\alpha/2)}{(\beta * 3)}}$, on écrie $sqrt((a/2)/(b*3))$ et on donne à la machine cette expression à évaluer et on fait cofinance au compilateur pour

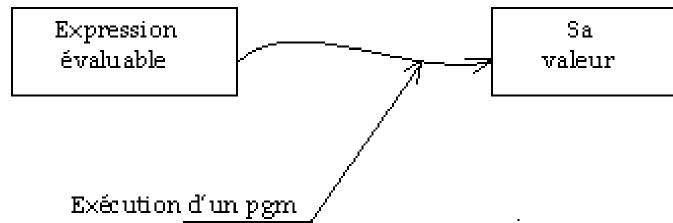


Figure 4.1: évaluation d'une expression

Exécution	Evaluation
Commande	Expression
Procédure	Fonction
Affectation	Liaison

Figure 4.2: Approche impérative et approche fonctionnelle

qu'il fabrique la suite d'instruction nécessaire à son calcule. On ne dit pas, par exemple, s'il faut calculer $(a/2)$ avant $(b * 3)$ ou le contraire.

L'évaluation d'une expression n'est pas ici de nature impérative. L'évaluation de cette dernière expression n'est pas considérée comme une exécution d'une suite d'instruction donnée à une machine mais comme un processus plus abstrait, fonction, d'évaluation qui conduit d'une expression à sa valeur. Comme nous le verrons, l'approche fonctionnelle permet de généraliser ce point de vue en systématisant la vision d'un calcul comme une évaluation d'expression.la figure4.2donne la correspondance entre les notions principale utilisées dans les deux approches de la programmation.

Seul les langages machine et les langages d'assemblage s'inscrivent exclusivement dans le point de vue impératif. Depuis FORTRAN et l'invention de la notion de compilateur, les langages de programmation offrent des combinaisons impératives et évaluatives. Toutefois, ces langages restent principalement imprégnés de la vision. Par exemple, la notion de fonction

n'y a pas du tout le statut qu'elle a en mathématique : une fonction y est vue comme une procédure qui retourne une valeur avec toute la charge impérative que comporte la notion de procédure.

Vision fonctionnelle

Le terme de « programmation fonctionnelle » ou celui presque équivalent à « programmation impérative » que l'on utilise pour parler de la programmation dans des langages tels que CAML ou LISP fait référence au fait que dans ces langages, la définition et l'application des fonctions jouent un rôle primordial dans la construction de programmes. Toutefois, il n'est pas sûr que ces qualificatifs seront les plus appropriés.

Ce qui caractérise véritablement ce style de programmation, c'est la place centrale accordée à la notion d'expression. Un programme est vu essentiellement comme une expression évaluable et l'exécution d'un programme comme un mécanisme d'évaluation qui conduit d'une expression à sa valeur.

Si l'on adopte ce point de vue, on est naturellement conduit ensuite à privilégier la définition et l'application des fonctions. En effet, une expression n'est rien d'autre qu'une combinaison d'opérateurs (c'est –à- dire de fonction) et d'arguments. On ne peut programmer avec des expressions que si l'on peut construire un ensemble suffisamment riche de fonctions. Par ailleurs, une fonction se définit comme une expression dans laquelle on abstrait certaines variables pour en faire des paramètres.

De plus, le fait de privilégier les notions d'expression et de valeur, reliées par un mécanisme d'évaluation, a aussi d'autres conséquences. Il est naturel de demander, par exemple, que toute valeur puisse être dénotée par une expression, ce qui n'est pas le cas dans les langages de programmation classiques. Par exemple, si on définit en Pascal ou en C le type point comme un enregistrement comportant les champs X_c et Y_c correspondant aux coordonnées du point, il n'existe aucune façon directe de dénoter le point de coordonnées (x, y) où x et y sont deux variables réelles. Pour construire ce point, il faut d'abord disposer d'une variable p de type point puis effectuer les affectations $p.X_c := x$ et $p.Y_c := y$. Pourquoi les langages traditionnels ne permettent-il pas de noter directement le point de coordonnées (x, y) ? Parce que cela supposerait une allocation implicite de mémoire, ce que ces langages n'autorisent pas. On voit donc que le fait de privilégier les notions d'expression et de valeurs conduit aussi directement à introduire l'allocation implicite de mémoire et son corollaire nécessaire, la récupération automatique.

En résumé, le rôle primordial joué par les fonctions et l'allocation dynamique de mémoire sont des traits de langages fonctionnels qui découlent de leur option de base : la vision de la programmation comme évaluation d'expressions.

Fonctions comme valeurs

A partir du moment où l'on dispose de fonctions définies simplement à partir d'expressions par abstraction de certains paramètres, on débouche sur une utilisation des fonctions qui est beaucoup plus proche de la notion mathématique de fonction que celle qu'on trouve dans les langages de programmation traditionnels. Il devient alors naturel d'autoriser des opérations sur les fonctions comme la composition. Qu'est ce que la composition ? c'est une fonction qui prend en argument deux fonctions f et g et rend en résultat leur composition fog. la composition est un exemple de fonction d'ordre supérieur.

Le fait de passer des fonctions en argument ou de les récupérer comme résultat d'autres fonctions oblige à considérer les fonctions comme une catégorie de valeurs : les fonctions doivent faire partie des valeurs du langage. Et comme les autres valeurs, il doit être possible de leur associer une expression pour les désigner. C'est ainsi que l'on est amené à introduire une construction pour noter les valeurs fonctionnelles.

Typage

Quand on manipule des objets complexes comme des fonctions d'ordre supérieur, il est encore plus nécessaire que l'habitude de disposer d'une assistance pour vérifier la cohérence des programmes. Le typage s'impose donc naturellement. Par rapport au système de type d'un langage traditionnel, dans certains langages ajoutent des types fonctionnels pour typer les fonctions à définir, par exemple, la fonction $fact\ n$ qui est une fonction qui nous retourne le factoriel du nombre n c'est une fonction typer, par exemple, en CAML par : $int \rightarrow int$.

4.1.2 Présentation du langage de programmation CAML

Introduction

Cette introduction à CAML est indispensable au début de ce chapitre. En effet, pour exposer de façon précise les méthodes de programmation et les algorithmes dans ce chapitre, il est préférable de commencer par s'accorder sur une « langue commune ». Nous aurions pu choisir de rester dans la généralité en écrivant les programmes en pseudo-langage comme :

début
 tant que l'étudiant ne comprend pas
 faire répéter l'explication
fin

Afin que le lecteur puisse connaître la satisfaction de voir s'exécuter son programme sous ces yeux, il valait mieux choisir un langage de programmation existant sur ordinateur. Le langage Caml .

Caml-Light, possède, outre sa gratuité, de nombreux attraits pédagogiques (interactivité, clarté, justesse) et ne déroutera pas le lecteur familier d'autres langages de programmation

répandus. La version utilisée dans ce mémoire est la v 0.74 française. Les commandes de Caml sont bien documentées dans l'aide fournie avec le logiciel.

Caml est un langage de programmation, à la fois facile à apprendre et cependant étonnamment expressif. Il est développé et distribué par l'INRIA depuis 1984 et il est disponible gratuitement pour les machines Unix, PC ou Macintosh.

Il existe deux dialectes de Caml: Caml Light et Objective Caml. Caml Light est un sous-ensemble d'Objective Caml, plus spécialement adapté à l'enseignement et à l'apprentissage de la programmation. En plus du cœur du langage de Caml Light, Objective Caml comporte un puissant système de modules, des objets et un compilateur optimisant.

On donne ici un bref aperçu du langage, qui permet de se faire une idée des traits saillants de Caml

Sûreté

Le langage Caml est très sûr. Le compilateur fait de nombreuses vérifications avant la compilation des programmes. De nombreuses erreurs de programmation deviennent ainsi impossibles en Caml: confusions de types de données, accès erronés à l'intérieur des données par exemple. En effet, tous ces points sont vérifiés et gérés automatiquement par le compilateur, ce qui garantit l'intégrité parfaite des données manipulées par les programmes.

Caml est un langage fortement typé, mais il est inutile d'ajouter des informations de type dans les programmes (comme en Ada, en Pascal ou en C): les annotations de typage sont automatiquement calculées par le compilateur.

Types de données

Rappelons que Caml ne se contente pas d'afficher le résultat d'une commande de l'utilisateur, il détermine également son type (on dit que Caml a « typé l'expression »). Nous avons vu les types int et float qui avertissent l'utilisateur que le résultat de son calcul est un entier ou un flottant. Cette indication de type est aussi donnée pour toutes les fonctions connues de Caml, par exemple :

Il existe en Caml de nombreux types de données prédéfinis:

- – Types de base: entiers, flottants, booléens, caractères, chaînes de caractères.
  – Types de données plus complexes: n-uplets, tableaux, listes, ensembles, tables de hachage, files, piles, flux de données.

Au-delà de ces types prédéfinis, Caml propose de puissants moyens de définir de nouveaux types: types enregistrements, types énumérés, et des types sommes généraux. Les types sommes sont une généralisation des types unions, à la fois simple, sûre et facile à maîtriser. Ils permettent la définition de types de données qui présentent des valeurs hétérogènes repérées par des constructeurs de valeurs.

Au gré du programmeur, tous ces types sont définissables concrètement (les constructeurs sont disponibles à l'extérieur du module) ou abstraitement (l'implémentation est restreinte au module de définition et les constructeurs sont invisibles à l'extérieur).

Ce mécanisme autorise un contrôle fin du degré d'encapsulation des données manipulées par les programmes, ce qui est indispensable pour la programmation à grande échelle.

Dans ce paragraphe, on présente la construction prédéfinie liste puis on montre comment le programmeur peut définir ses propres types.

Type listes Les listes sont des suites finies d'éléments de même type. Les listes se notent entre crochets et leurs éléments sont séparés par des points-virgules.

```
#let liste = [1+2 ;3+4 ;5+6] ;;
val liste : int list = [3 ;7 ;11]
```

les opérateurs **::** et **@** permettent d'ajouter un élément en tête d'une liste et de concaténer deux listes.

```
#2 :: liste ;;
-: int list = [2 ;3 ;7 ;11]
#liste @ liste ;;
- : int list = [3 ;7 ;11 ;3 ;7 ;11]
```

Pour définir une fonction sur les listes, on doit, en général, distinguer le cas où la liste est vide (notée []) et le cas où elle ne l'est pas. Lorsqu'une liste n'est pas vide, elle s'écrie sous la forme **x :: rest** où x est le premier élément de la liste et *rest* est le reste de la liste, ce qui conduit à des définition comportant deux cas. Voici une fonction pour calculer la longueur d'une liste.

```
#let rec longueur l =
match l
   with [ ] -> 0
  | (x :: rest) -> 1 + longueur(rest);;
val longueur : 'a list -> int = <fun>
```

On note que cette fonction est indépendante du type des éléments de la liste, d'où le polymorphisme. Voici une fonction qui prend en argument une fonction et une liste, applique la fonction à tous les éléments de la liste et rend la liste des résultats.

```
#let rec map (f,l)
match l
  with [ ] -> [ ]
  |(x :: rest) -> f(x) :: map(f,rest);;
val map : ('a -> 'b) * 'a list -> 'b list = <fun>
# map ((function x -> x*x), [1,2,3,4,5]);;
- : int list = [1;4;9;16;25]
```

Type tableaux Pour les tableaux, on utilise un type '*a array* semblable au type '*a list* et une syntaxe assez proche :

#[|1+2 ;3+4 ;5+6|] ;;

- : int array = [|3 ;7 ;11|]

Type enregistrement Les types enregistrement font l'objet d'une déclaration dans laquelle le programmeur définit les champs de l'enregistrement et leur type. La syntaxe des enregistrement utilise des accolades.

#type point = {xc :int ;yc :int} ;;

La syntaxe des valeurs est proche de celle du type.

#{xc=2 ;xy=3} ;;

- : point = {xc=2 ;xy=3}

Type définie par constructeurs L'exemple le plus simple de types avec constructeurs est un type énuméré défini par une liste finie de constructeurs constants. Les constructeurs sont des identificateurs commençant par une majuscule.

#type couleur = Trèfle | Carreau | Cœur | Pique ;;

#Trèfle ;;

- : couleur = trèfle

#fonction Trèfle -> 1 | Carreau -> 2 | Cœur -> 3 | Pique -> 4 ;;

- : couleur -> int = <fun>

Les constructeurs peuvent également comporter des arguments. Pour définir des arbres binaires tels que celui de la figure suivante, il suffit de déclarer qu'un arbre est soit une feuille (comportant une information qui est un entier), soit un nœud binaire (comportant deux sous arbres). On utilisera un constructeur **F** pour les feuilles et un constructeur **N** pour les nœuds binaires.

#type arbre = F of int | N of arbre * arbre ;;

L'arbre de la figure suivante s'écrit :

#N(F(7),N(F(2),N(F(8), F(13)))) ;;

- : arbre = N(F 7, N(F 2, N(F 8, F 13)))

Fonctions

Caml est un langage de programmation fonctionnel: il n'y a pas de restriction à la définition et à l'usage des fonctions, qu'on peut librement passer en argument ou retourner en résultat dans les programmes.

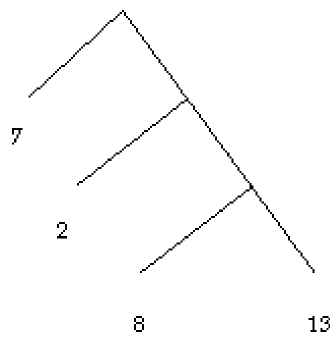


Figure 4.3: arbre binaire

Construction de fonctions Considérons l'expression mathématique $\frac{\sin\ x}{x}$. On peut la considérer comme représentant une fonction de la variable x . En CAML, pour noter cette fonction, il suffira d'expliciter qu'elle dépend de x :

#function x -> sin(x) /.x ;;

- : float -> float = <fun>

il est facile d'appliquer directement une telle fonction à un argument :

#(function x -> sin(x)/.x)(0.1) ;;

- : float = 0.998334166468

fonction d'ordre supérieur Dans l'expression $\frac{\sin\ x}{x}$ on peut aussi abstraire la variable sin et construire pour toute fonction f la nouvelle fonction $\frac{f(x)}{x}$:

#let h(f) = function x -> f(x)/.x ;;

val h: (float -> float) -> (float -> float) = <fun>

#k = h(sin);;

val k : float -> float = <fun>

#k(0.1);;

- : float = 0.998334166468

Gestion mémoire automatisée et incrémentale

Caml offre une gestion automatique de la mémoire: l'allocation et la libération des structures de données est implicite (il n'y a pas de primitives de manipulation explicite de la mémoire comme "**new**" ou "**free**" ou "**dispose**"), et laissée à la charge du compilateur. On obtient ainsi une programmation bien plus sûre, puisqu'il n'y a jamais de corruption inattendue des structures de données manipulées.

De plus le gestionnaire mémoire opère en parallèle avec l'application, sans jamais l'arrêter de façon notable (récupération mémoire incrémentale).

Traits impératifs

Caml offre la panoplie complète des traits de la programmation impérative, en particulier les tableaux modifiables en place, les boucles et les variables affectables, les enregistrements avec champs physiquement modifiables.

Changeons à présent de style et passons à la programmation impérative. Nous allons atteindre notre propos aux preuves de boucles, les boucles jouent un rôle des appels récursifs d'une fonction en programmation récursive. La méthode consiste à déterminer une propriété, dite invariant de boucle, liant les éléments variables d'une boucle, et qui reste vraie quel que soit le nombre de passages celle-ci. Détaillons cela sur cet exemple.

Reprenons le cas de la fonction somme, cette fois dans sa version impérative :

```
#let somme n =
let resultat = ref 0 in
  for i = 1 to n do
resultat := !resultat + i
done;
!resultat;;
```

Compilateur rapide, code exécutable rapide

Caml propose un compilateur de fichiers, et la compilation séparée est assurée par un système de modules. De surcroît, le compilateur Caml comporte une option qui maximise la vitesse de compilation, la portabilité des programmes obtenus, et minimise la taille des exécutables (compilation en code-octets).

Le compilateur d'Objective Caml comporte en plus une option "optimisante" qui privilégie la vitesse d'exécution (compilation en code natif): le compilateur optimisant d'Objective Caml produit des programmes dont la vitesse d'exécution est digne des meilleurs compilateurs disponibles actuellement.

Interactivité

Caml offre également un système interactif (une boucle de lecture-évaluation-impression des résultats), qui est très pratique pour apprendre le langage ou essayer et corriger ses programmes: il n'y a pas besoin d'utiliser forcément des fichiers, ni d'ajouter des ordres d'impression dans les programmes puisque les résultats sont imprimés automatiquement par le système interactif.

Nous appellerons session Caml tout ce qui se passe entre le lancement de Caml et son arrêt par la commande **quit**();;. Une fois Caml lancé, l'utilisateur travaillant avec un système d'exploitation à fenêtres comme Windows, Mac-OS, X11, ... dispose d'une fenêtre d'Entrée, Caml light input en anglais, dans la quelle il saisit ses commandes et d'une fenêtre de Résultats, Caml light output en anglais, où le compilateur Caml affiche ses résultats ; les sorties

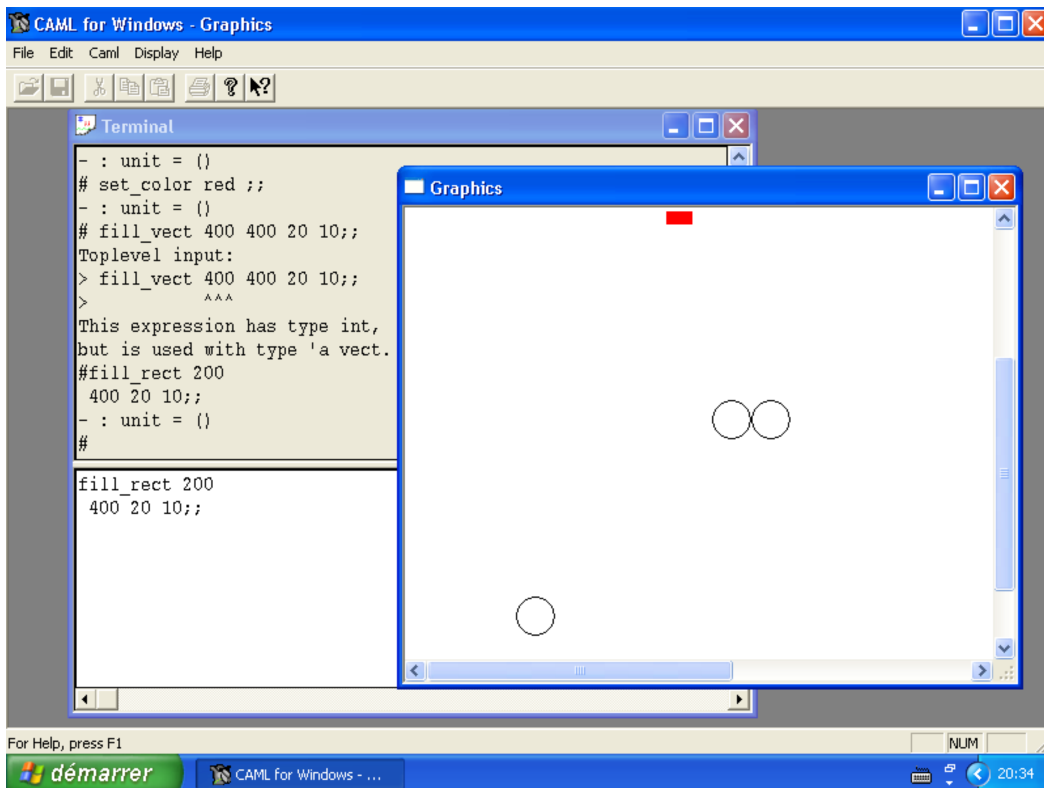


Figure 4.4: interface interactive de CAML Light

graphiques sont représentées dans la fenêtre Graphique.

Forte capacité de traitement symbolique

Le filtrage apporte un confort inégalé dans le traitement symbolique des données.

Le vérificateur de filtrage procure un niveau de sécurité dans la programmation et un degré de qualité inégalé des programmes qui manipulent des données symboliques.

Traitement des erreurs

Caml possède un mécanisme général d'exceptions, pour traiter ou corriger les erreurs ou les situations exceptionnelles.

Observons ce qui se produit lorsque l'on demande :

#hd [] ;;

Exception non rattrapée : Failure "hd"

CAML renvoie un message d'erreur : il n'y a pas de premier élément dans la liste vide !

En fait c'est un peut plus précis que cela :

CAML déclenche une exception.

Une exception est le cas (ou les cas) ou la fonction ne marche pas.

Gestion des exceptions

En CAML, nous pouvons :

1. Utiliser les exceptions prédéfinies
2. Ou bien en créer de nouvelles.

Pour l'instant, passons en revue quelques exceptions prédéfinies en CAML. On distingue deux types d'exceptions en CAML : les exceptions constantes et les exceptions paramétrées.

Parmi les exceptions constantes, citons l'exception **Not_found** qui apparaît lorsqu'une fonction de rechrche n'aboutit pas, **Division_by_zero**, que se passe de commentaire, et enfin, celle que l'on ne souhaite pas :

Out_of_memory.

#1/(2-2);;

Exception non rattrapée : Division_by_zero

#index"Walid" ["Walid" ;"Adel"] ;;

- : int = 2

#index "Mounir" ["Walid" ;"Adel"] ;;

Exception non rattrapée : Not_found

Jusqu'à présent nous n'avons pas fait grande chose des exceptions. En effet, on peut rattraper une exception, principalement lorsque une fonction appelante reçoit un déclenchement d'exception de la art d'une fonction appelée et qu'elle sait comment « récupérer la situation ». La syntaxe CAML est alors la suivante :

try résultat_récupérée
with attitude_à_adopter ;;
« l'attitude à adopter » est un filtrage. Elle est de la forme :
exception1 → action1
|**exeption2 → action2**
|**. . .**
Par exemple :
let résultat_récupérée n m =
try n/m
with Division_by_zero → 0
résultat_récupérée : int → int → int = <fun>
#résultat_récupérée 8 4 ;;
- : int = 2
#résultat_récupérée 8 0 ;;
- :int = 0

Mise au point des programmes

Plusieurs méthodes de mise au point des programmes s'offrent à vous en Caml:

- le système interactif offre une méthode élémentaire mais très simple et rapide pour tester des (petites) fonctions: on vérifie simplement les résultats obtenus sur quelques exemples tapés directement dans la boucle d'interaction.
- Dans les cas plus complexes, le système interactif permet également à très peu de frais de suivre la progression des calculs avec le mécanisme de trace des appels de fonctions.
- Enfin le débogueur symbolique avec retour arrière permet de suivre très finement le déroulement de l'exécution, de l'arrêter à tout moment pour examiner l'état courant des variables et des fonctions en attente, et même de revenir en arrière dans les calculs pour reprendre l'exécution au moment où un évènement intéressant se produit.

Polymorphisme

Caml est doté d'un puissant typage "**polymorphe**": certains types peuvent rester indéterminés, représentant alors "n'importe quel type".

De nombreuses fonctions sont naturellement polymorphes. Par exemple, les fonctions **FST** et **SND** qui permettent d'accéder à la première et à la seconde composante d'un couple doivent pouvoir opérer sur n'importe quel couple, c'est-à-dire quels que soient les types de ses deux composantes. Ces types seront donc des variables de types pour ces fonctions. En CAML, les variables de types sont notées 'a, 'b, 'c, etc.

#let fst (x,y) = x ;;

*val fst : ‘a * ’b → ‘a = <fun>*

#let snd(x, y) = y ;;

*val snd : ‘a * ’b → ‘b = <fun>*

Ainsi, les fonctions et procédures qui sont d'usage général s'appliquent à n'importe quel type de données, sans exception (par exemple les routines de tri s'appliquent à tout type de tableaux).

Méthode et stratégies d'évaluation

Caml est un langage "strict", par opposition aux langages paresseux. Cependant la pleine fonctionnalité permet de créer des suspensions et donc de coder l'évaluation paresseuse de données potentiellement infinies.

Cependant, certains calculs ne se terminent pas et une même expression peut donner lieu à la fois des calculs qui se terminent et à d'autre qui ne se termine pas. Par exemple, si on définit le fonction boucle par :

#let rec boucle(x) = boucle(x+1)

l'expression :

(function x → 1)(boucle(0))

s'évalue en la valeur 1 si on applique tout de suite la règle

(fonction x ⟹ e)(é) → e[x ← é] à l'expression tout entière mais donne lieu à un calcul infini si on essaie d'évaluer l'argument *(boucle(0))*.

Il est possible de démontrer que la stratégie consistant à l'appliquer systématiquement la règle précédente avant d'évaluer l'argument d'une application conduit toujours au résultat lorsque celui-ci existe. Cette stratégie est appelée « par nécessité » car l'argument n'est finalement évalué que lorsque sa valeur est nécessaire au calcul. La stratégie « par valeur » consiste à évaluer l systématiquement l'argument avant d'effectuer une application peut, par contre, boucler inutilement comme c'est le cas dans l'exemple précédent.

Le langage CAML adopte cependant l'évaluation par valeur, de même que SML ou les langages de la famille LISP. On peut se demander pourquoi. La raison principale n'est pas liée à l'efficacité, comme on pourrait le croire, mais plutôt au problèmes d'interfaçages. Dans l'évaluation par nécessité, le programmeur perd le contrôle de l'ordre d'évaluation : les différentes parties de sont programme sont évaluées en fonction des besoins du calcul et l'ordre qui en résulte dépend des données et n'est donc pas complètement prédictible. Cela n'a pas d'inconvénient pour des programmes purement fonctionnels mais devient gênant quand ces programmes doivent interagir avec d'autres programmes ou plus généralement avec le monde extérieur. Dans ce cas, l'ordre d'évaluation doit être complètement maîtrisée par le programmeur et c'est ce que permet l'évaluation par valeur.

Programmation en vraie grandeur

Les programmes Caml sont formés d'unités de compilation que le compilateur compile séparément. Ces organisation est parfaitement compatible avec l'utilisation d'outils traditionnels de gestion de projets (comme l'utilitaire make d'Unix). Le système de module du langage est puissant et sûr (toutes les interactions entre modules sont statiquement vérifiées par le contrôleur de types). Les modules d'Objective Caml peuvent comporter des sous-modules (à un degré d'emboîtement quelconque) et les fonctions des modules dans les modules sont autorisées (ce qui permet de définir des modules paramétrés par d'autres modules).

Programmation orientée objets

Objective Caml propose des objets qui permettent d'utiliser le style orienté objets dans les programmes Caml. Fidèle à la philosophie du langage, cette extension orientée objets obéit au paradigme du "typage fort": en conséquence, aucune méthode ne peut être appliquée à un objet qui ne pourrait y «répondre» («les méthodes sont toujours bien comprises»). Encore une fois, cette vérification systématique du compilateur évite de nombreuses erreurs. Ceci offre au programmeur Caml, outre un confort insoupçonné dans l'écriture de ses programmes orientés objets, un niveau inégalé de qualité des programmes qu'il produit.

Puissantes bibliothèques

De nombreuses bibliothèques et contributions sont disponibles en Caml, en particulier des primitives de dessin indépendantes de la machine (libgraph), une arithmétique rationnelle exacte en multi-précision (camlnum), et de nombreuses interfaces avec des technologies bien connues: générateurs d'analyseurs lexicaux et syntaxiques avec camllex et camlyacc. Sous Unix, on dispose aussi d'un débogueur avec retour arrière, d'un navigateur dans les fichiers sources (camlbrowser), d'une interface graphique à l'aide de Tk/Tcl (camltk) et d'une interface poussée avec le système (libunix).

4.2 introduction aux compilateurs :

Tout simplement, un compilateur est un programme, constitué d'un ensemble fini de phases, qui admet en entrée un programme source, écrit dans un langage bien déterminé ,le traite et le transforme en un programme écrit dans un langage cible.

Pour effectuer cette transformation, il faut passer par un ensemble d'étapes :

4.2.1 Analyse lexicale :

Cette première phase de compilation est consacrée à la reconnaissance des unités lexicales du programme,.

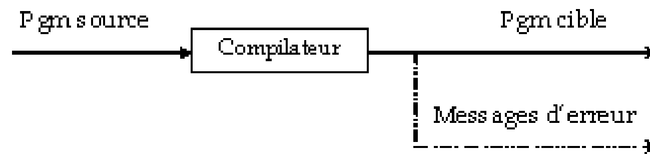


Figure 4.5: Compilation

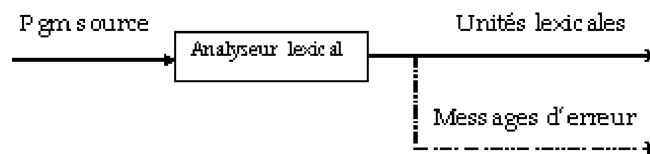


Figure 4.6: Analyse lexicale

4.2.2 Analyse syntaxique :

Cette deuxième phase de compilation détermine si la suite des unités lexicales respecte la grammaire du langage, et retourne un arbre syntaxique ou un message d'erreur ..

4.2.3 Analyse sémantique :

Au cours de cette étape le compilateur effectue certaines opérations de vérification de typage , déclarations et utilisation des variables. . .

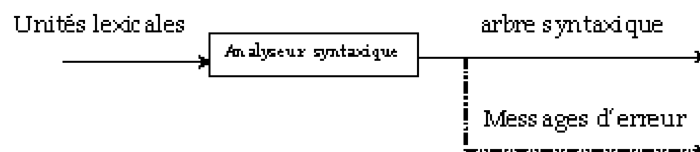


Figure 4.7: Analyse syntaxique

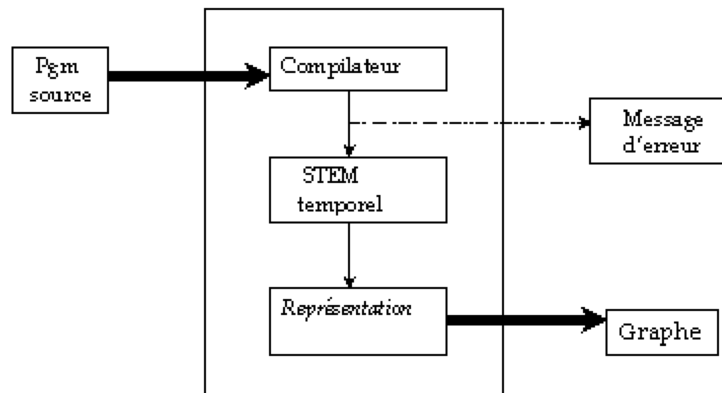


Figure 4.8: Fonctionalitées du compilateur D-LOTOS

4.3 Realisation

Dans cette partie nous expliquons en détail les trois phases constituant le compilateur: l'analyse lexical, syntaxique, sémantique et la génération du code objet.

4.3.1 Analyseur lexical :

Unités lexicales du langage D-LOTOS :

- mots clés

system, endsys, process, endproc, hide, in, exit, nil, where, delay.

- les opérateurs

  - ∗ · Opérateur de composition , '**;**'.
    · Opérateur de choix :'[]'
    · Opérateurs de composition parallèle '|[**gates**]|', '||', '|||',
    · Opérateur d'interruption '[>',
    · Opérateur de séquencement '>>'

- les identificateurs

définies comme suit
id → alpha suite_alpha
alpha → 'a'..'b' | 'A'..'B'
suite_alpha → _ alpha | digit suite_alpha|eps

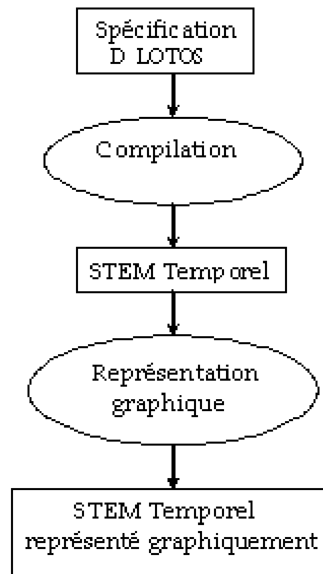


Figure 4.9: Phases de conception

- le type digit

définie comme :
digit → num suite_num | .
num → 0..9
suite_num → num suite_num | . gauche
gauche → num suite_num

- caractères spéciaux : ’(‘, ‘)’, ‘[‘, ‘]’, ‘,’, ‘{‘, ‘}’.

Ces unités lexicales sont représentées par le type « token »,définie en Caml comme suit :
type token = id of string
digit of real
spécial of string
clé of string
Clé(system), digit(11) est la représentation de : system, 11.
La fonction principale de l’analyse lexical est :
main_lex de type char list → token list .

Exemple 4.1 *soit data = [‘p′; ′r′; ′o′; ′c′; ′e′; ′s′; ′s′; ‘p′; ‘ : ′; ′ = ′; ‘n′; ′i′; ′l′] l’application de la fonction main_lex sur la liste data donne :main_lex (data) = [clé⌋process⌋ ; id⌋p⌋ ; spécial⌋ := ‘′; clé⌋nil⌋]; ;*

4.3.2 Analyseur syntaxique

Représentation des expressions de comportement D-lotos en caml

Dans les langages imératifs les processus sont implémentés à l'aide d'une structure d'arbre .Par exemple le processus ($P1[]\ P2$) est implémenté par un arbre binaire comme suit:

- – La racine est le symbol "[]"
  – le fils gauche est l'arbre correspondant au processus :$p1$.
  – le fils droit est l'arbre correspondant au processus $P2$.

En plus de ça ,la gestion de l'arbre binaire(insertion ,parcour,suppression d'un fils) est faite d'une manière **directe(explicite)** par le programmeur ,ce dernier doit gérer explicitement l'éspace mémoire (allocation, désallocation).

Mais les langages fonctionnels fournissent d'autre techniques ,car la gestion de la mémoire est faite automatiquement par le compilateur(ou l'interpreteur) .Le langage **Caml** fournie les types récursifs.

Exemple 4.2 *pour représenter le processus* ($p1[]\ p2$) *,on définie un type process comme suit:type process* $=$ *choix of process* $*$ *process Ce qui donne un arbre binaire étiqueté par choix,les deux fils sont écrits :process* $*$ *process.*

Le processus D_LOTOS sont implémentés par le type *process* ,définie récursivement en **Caml** comm suit

```
type process =
    stop
  | exit of real
  | comp of (string * real) *process
  | choix of process * process
  | parallel of process * string list * process
  | hiden of string list * process
  | seq of process * process
  | rupt of process * process
  | delays of int * process
  | idt of string * (string * real )list ;;
```

- – * Le type *process* est la conjonction de
- l'unité *stop* .
- sous type $exit(u) \mid u \in \mathbb{R}$

qui correspond à l'expression D_lotos $exit\{u\}$

- sous type $comp((a,u),p) \mid a \in string, u \in \mathbb{R}, p \in process$

qui correspond à l'expression D_lotos $a\{u\};p$.

- sous type $choix(p1,p2) \mid p1,p2 \in process$

qui correspond à l'expression D_lotos $p1[\ ]\ p2$.

- sous type $parallel(p1,L,p2) \mid p1,p2 \in process, L \in string\ list$

qui correspond à l'expression D_lotos $p1|[L]|\ p2$.

- sous type $hiden(L,p) \mid p \in process, L \in string\ list$

qui correspond à l'expression D_lotos $hide\ L\ in\ p$.

- sous type $seq(p1,p2) \mid p1,p2 \in process$,

qui correspond à l'expression D_lotos $p1 >> p2$.

- sous type $rupt(p1,p2) \mid p1,p2 \in process$,

qui correspond à l'expression D_lotos $p1\ [> p2$.

- sous type $delays(u,p) \mid p \in process, u \in \mathbb{R}$,

qui correspond à l'expression D_lotos $delay(u)\ p$.

- sous type $idt(nom, portes) \mid nom \in string, portes \in (string * \text{real})\ list$

qui correspond à l'expression D_lotos $nom[portes]$.tel que les portes ont la forme $(a,d_a);(b,d_b),...$ie :les actions et leurs durées

- – * Expressions D_lotos en Caml:
- L'expression $a\{8\};stop$

est écrite en **Caml** $comp(("a",8),stop)$.

- L'expression $(a\{8\};stop\ []\ b\{45\};stop)$

est écrite en **Caml** $choix(comp(("a",8),stop),comp(("b",45),stop))$

- L'expression $((a\{12\};stop)\ |[a]|\ a\{15\};stop)$

est écrite en **Caml :** $parallel(comp(("a", 12), stop), ["a"], comp(("a", 15), stop)\ \ )$

- L'expression $hide\ a, b\ in\ a\{45\}; b\{12\}; stop$

est écrite en **Caml** $hiden(["a"; "b"]\ , comp(("a", 45), comp(("b", 12), stop)))$

- L'expression $a\{48\}; next_proc[b[15], c[23], d[10]]$

est écrite en **Caml** $comp(("a", 48), idt("next_proc", [("b", 15); ("c", 23); ("d", 10)]))$

- L'expression $delay(10)(a\{4\}; stop)$

est écrite en **Caml** $delays(10, comp(("a", 4), stop))$

génération du code intermédiaire

Soit **code** La fonction de génération de l' arbre syntaxique de type *process* définie dans la partie précédente,qui admet en entrée une suite d'unités lexicales définies précédemment ,et retourne un arbre syntaxique de type *process*.Nous allons donner les règles sémantiques correspondantes se basant sur la syntaxe du langage:

$(exp_1!\ \ stop) \Longrightarrow (\ exp_1.code := stop)$
$(exp_1!\ \ exit\{u\}) \Longrightarrow (\ exp_1.code := exit(u))$
$(exp_1!\ \ id\{u\}; exp_2) \Longrightarrow (exp_1.code := comp(("id.nom", u),\ exp_2.code),$
$(exp_1!\ \ exp_2\ [\ ]\ exp_3) \Longrightarrow (exp_1.code := choix(exp_2.code\ \ , exp_3.code))$
$(exp_1!\ \ exp_2\ |[\ L\ ]|\ exp_3) \Longrightarrow (exp_1.code := parallel(exp_2.code, L.elem\ \ , exp_3.code))$
$(exp_1!\ \ hide\ L\ in\ exp_2\ ) \Longrightarrow (exp_1.code := hiden(L.elem\ , exp_2.code))$
$(exp_1!\ \ exp_2 >> exp_3) \Longrightarrow (exp_1.code := seq(exp_2.code\ ,\ exp_3.code))$
$(exp_1!\ \ exp_2\ [> exp_3) \Longrightarrow (exp_1.code := rupt(exp_2.code\ ,\ exp_3.code))$
$(exp_1!\ \ id[portes]) \Longrightarrow (exp_1.code := idt(id.nom, portes.elem))$

$(L\ !\ id\ suite_L) \Longrightarrow (L.elem := id.nom\ :: suite_L.elem\ )$
(**::** fonction d'insertion d'un élement dans une liste)
$(suite_L\ !\ \epsilon\ ) \Longrightarrow (suite_L.elem := []\ )$ (la liste vide)
$(suite_L_1\ !\ , id\ suite_L_2) \Longrightarrow (suite_L_1.elem :=\ id.nom\ ::\ suite_L_2.elem)$

$(Portes\ !\ id[u]\ \ suite_Portes) \Longrightarrow (\ Portes.elem := (id.nom\ ,\ u)\ :: suite_Portes.elem.)$
$(suite_Portes\ !\ \epsilon\ ) \Longrightarrow (suite_Porte.elem := []\ )\ .$
$(suite_Portes_1\ !\ , id[u]\ \ suite_Portes_2) \Longrightarrow (\ suite_Portes_1.elem := (id.nom, u)\ :: suite_Portes_2.elem)$

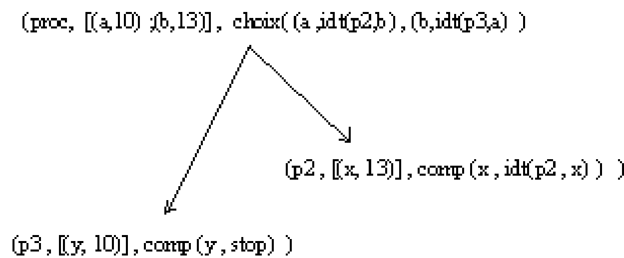


Figure 4.10: Arbre de processus

Principe

L'approche utilisée est l'analyse par descente récursive où chaque non terminal correspond à une fonction (voir cours de compilation) .

La fonction de génération de code intérimaire accepte en entrée une liste de token et retourne toutes les informations des processus y compris le **nom** de chaque processus, ses **portes** avec leurs **durées**, son **expression** de comportement, ses **processus fils**, donc d'une manière précise le résultat est un arbre où :

- chaque nœud à la forme (nom, portes, exp) tel que :
  - nom : est le nom d'un processus de type string.
  - portes : sont les portes du processus de type (string * int) list c'est une liste d'action sa durée .
  - exp : arbre syntaxique de l'expression de comportement, de type process défini présedement .
- Les fils d'un nœud sont les sous processus d'un tel processus.

Exemple 4.3 *System proc[a[10], b[13]] := (a ;p2[b]) [ ] (b;p3[a])*
Where process p2[x] := x;p2[x] endproc
process p3[y] := y;stop endproc endsys

le code intermédiaire généré par l'analyseur syntaxique est illustré par figure4.10

4.3.3 Générateur du code objet (STEM temporel)

Règles sémantiques

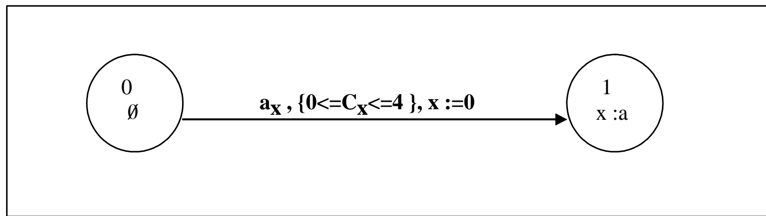
Dans cette partie nous donnons les régles sémantiques qui permettent de générer un stem temporel à partir d'une expression **D-Lotos** donnée. Avant de donner les régles sémantiques

nous présentons quelques exemples illustratifs:

Exemples

Example 1 *Soit l'expression D-LOTOS* ***exp1:=***$_\phi\,[a[12]\{4\};stop]$ *,le début d'exécution de l'action a de durée* 12 *doit avoir lieu dans un intervalle de longueur 4, d'où la dérivation suivante:*

$$exp1 \xrightarrow{\langle_\phi a_x,\{0\leq C_x\leq 4\},\{x\}\rangle} >\ _{\{x:a:12\}}[stop]$$



STEM Temporel de exp1

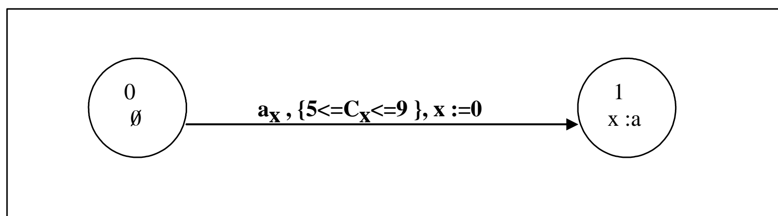
A l'état '0' l'horloge c_x sera déclenchée, le début d'exécution de l'action a est conditionné par la contrainte $\{0 \leq c_x \leq 4\}$.Quand a commence on lui affecte le nom d'événement x.L'horloge c_x subie alors les opérations suivantes :

- On réinitialise c_x à la valeur 0, et
- on passe à l'état '1'.

remarques:

Remark 1 *L'initialisation de l'horloge c_x est importante car, dans les états futurs, elle indique si l'action a a terminé son exécution ou non , en d'autres termes, si $c_x > 12$ alors a est terminée sinon a est en cours d'éxécution. Ceci est dû au fait que l'horloge c_x calcule le laps du temps écoulé à partir du début de a.*

Example 2 *Soit l'expression D-LOTOS* ***exp2 :=*** $_\phi\,[delay(5)a[12]\{4\};stop]$ *. L'expression exp2 differt de exp1 par l'ajout du délai d'attente de 5 ut. ($exp2 := delay(5)exp1$). D'où la dérivation* $exp2 \xrightarrow{\langle_\phi a_x,\{5\leq C_x\leq 5+4\},\{x\}\rangle} >\ _{\{x:a:12\}}[stop]$

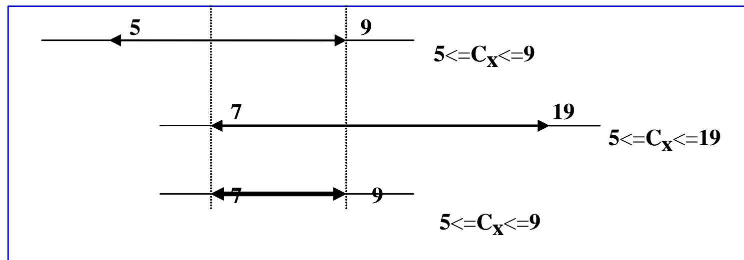


STEM Temporel de exp2

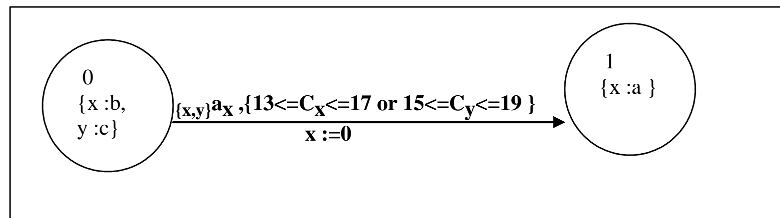
La seule différence avec la génération 1 est que l'arrivée de a est conditionnée par $\{5 \leq C_x \leq 9\}$résultant de $delay(5)a[12]\{4\}$.

Example 3 *Considérons maintenat l'expression D-LOTOS* ***exp3:=***$_{\{x:b:13,y:c:15\}}$ $[a[12]\{4\}; stop]$ *.Dans cette configuration (à la différence avec exp1) on a deux actions en exécution* $\{x : b : 13, y : c : 15\}$*.La question qui se pose est : Comment conditionner l'arrivée de l'action a par les deux horloges* c_x *et* c_y*.Rappelons que* c_x *(respectivement* c_y*) mesure le laps de temps écoulé à partir du début de l'action a (respectivent celui de b). Donc si* $c_x > 13$ *nous concluons que l'action b est achevée, par conséquent le laps de temps écoulé aprés la terminaison de b est égale à* $(c_x - 13)$ *(de même pour l'action c le laps de temps est de* $c_y - 15$*). De plus l'arrivée de a est conditionnée par l'intervalle* $\{0..4\}$*,ce dernier n'est calculé qu'après la terminaison de la dernière action achevée c-à-d soit* $x : b$ *ou* $y : c$*. En coclusion l'arrivée de a est conditionnée par:*

- La *terminaison de {x,y}*
- La contrainte $\{13 < c_x \leq 17 \;\; or \;\; 15 \leq c_y \leq 19\}$ (voir figure suivante **??**)



Intersection des contraintes

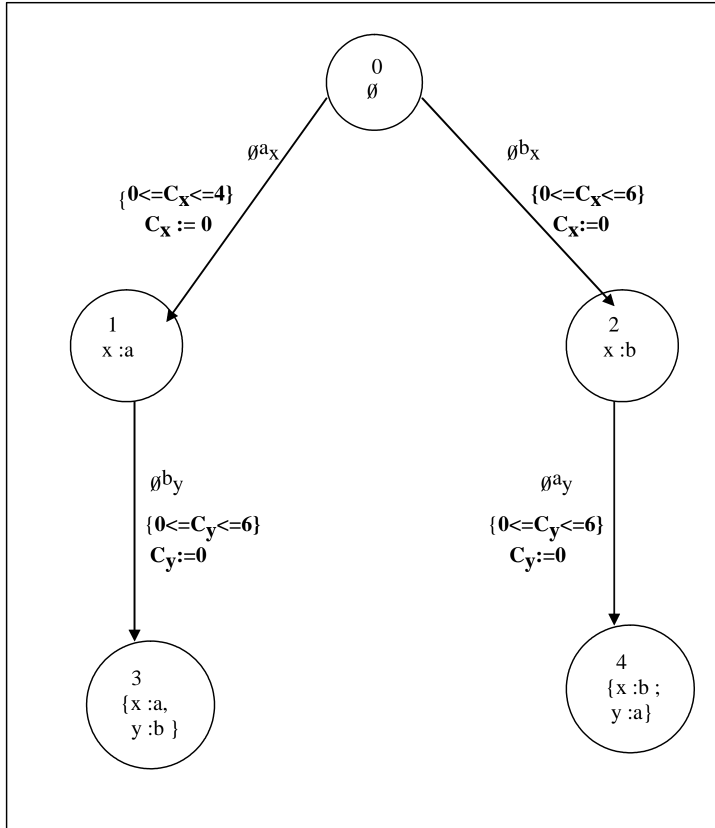


STEM Temporel de exp3

Example 4 *Pour l'expression D-LOTOS* ***exp4:=***$_{\phi}$ $[a[12]\{4\}; stop]$ $|||$ $_{\phi}$ $[b[13]\{6\}; stop]$ *on a deux chemins de dérivations possibles:*

$$exp4 \xrightarrow{\langle_{\phi} a_x, \{0 \leq C_x \leq 4\}, \{c_x\}\rangle} \; _{\{x:a:12\}}[stop] \; ||| \; _{\phi}[b[13]\{6\}; stop]$$

$$exp4 \xrightarrow{\langle_\phi b_x,\{0\leq C_x\leq 6\},\{c_x\}\rangle} > [a[12]\{4\};stop] \;|||\; _{\{x:b:13\}}[stop]$$

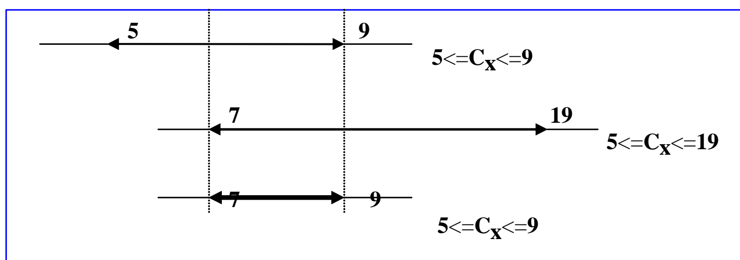


STEM Temporel de exp 4

5)exp5:= $_\phi\,[delay(5)a[12]\{4\};stop]\;\;|||\;_\phi\,[delay(7)a[12]\{10\};stop]$ .On a la dérivation :

$$exp5 \xrightarrow{\langle_\phi a_x,\{5\leq C_x\leq 9\ and\ 7\leq C_x\leq 17\},\{c_x\}\rangle} > {}_{\{x:a:12\}}[stop] \;|||\; _{\{x:a:12\}}[stop]$$

On a donc l'intersection des deux contraintes $\{5 \leq C_x \leq 9\ \}, \{7 \leq C_x \leq 17\}$ qui donne $\{7 \leq C_x \leq 9\}$.



Intersection des attentes

Le stem temporel étant trivial.

Généralisation: régles sémantiques La relation de transition $\rightarrow \subseteq \mathcal{C} \times atm \cup \mathcal{D} \times Contrainte \times Init \times \mathcal{C}$, qui génère le STEM Temporel , après une série de dérivation sur les configurations est définie de manière opérationnelle comme suit :

1. *processus exit* :

   (a) $$\frac{}{_{\phi}[exit\{u\}] \xrightarrow{\langle_{\phi}\delta_x,(0\leq c_x\leq u),\{c_x\}\rangle}_{\{x:\delta:d_\delta\}}[stop]} \; x = get(M)$$

   (b) $$\frac{M\neq\phi}{_{M}[exit\{u\}] \xrightarrow{\langle_{M}\delta_x,\{add_u(u,creat_const(\tau(\delta),M))\},\{c_x\}\rangle}_{\{x:\delta:d_\delta\}}[stop]} \; x = get(M)$$

2. *Opérateur de préfixage* :

   (a) $$\frac{}{_{\phi}[a\{u\};E] \xrightarrow{\langle_{\phi}a_x,(0\leq c_x\leq u),\{c_x\}\rangle}_{\{x:a:d_a\}}[E]} \; x = get(\mathcal{M})$$

   (b) $$\frac{M\neq\phi}{_{M}[a\{u\};E] \xrightarrow{\langle_{M}a_x,\{add_u(u,creat_const(\tau(a),M))\},\{c_x\}\rangle}_{\{x:a:d_a\}}[E]} \; x = get(\mathcal{M})$$

3. *Opérateur de choix* :

   (a) $$\frac{\mathcal{E} \xrightarrow{\langle trans\rangle} \mathcal{E}'}{\mathcal{E}[]\mathcal{F} \xrightarrow{\langle trans\rangle} \mathcal{E}'}$$

   (b) $$\frac{\mathcal{E} \xrightarrow{\langle trans\rangle} \mathcal{E}'}{\mathcal{F}[]\mathcal{E} \xrightarrow{\langle trans\rangle} \mathcal{E}'}$$

4. *Opérateur de composition parallèle* :

   (a) **(i)** $$\frac{\mathcal{E} \xrightarrow{\langle_{M}a_x,const,init\rangle} \mathcal{E}' \quad a\notin L\cup\{\delta\}}{\mathcal{E}|[L]|\mathcal{F} \xrightarrow{\langle_{M}a_y,const[c_y/c_x],init[c_y/c_x]\rangle} \mathcal{E}'[y/x]|[L]|\mathcal{F}\backslash M} \; y = get(\mathcal{M} - ((\psi(\mathcal{E})\cup\psi(\mathcal{F}))-M))$$

   (ii) $$\frac{\mathcal{E} \xrightarrow{\langle_{M}a_x,const,init\rangle} \mathcal{E}' \quad a\notin L\cup\{\delta\}}{\mathcal{F}|[L]|\mathcal{E} \xrightarrow{\langle_{M}a_y,\; const[c_y/c_x],\; init[c_y/c_x]\rangle} \mathcal{F}\backslash M|[L]|\mathcal{E}'[y/x]} \; y = get(\mathcal{M} - ((\psi(\mathcal{E})\cup\psi(\mathcal{F}))-M))$$

   b. $$\frac{\mathcal{E} \xrightarrow{\langle_{M}a_x,const1,init1\rangle} \mathcal{E}' \;,\mathcal{F} \xrightarrow{\langle_{N}a_y,const2,init2\rangle} \mathcal{E}', \;\; a\in L\cup\{\delta\}}{\mathcal{E}|[L]|\mathcal{F} \xrightarrow{\langle_{M\cup N}a_z,\; (const1[c_z/c_x]\;\cup\; const2[c_z/c_y]),\; (init1[c_z/c_x]\;\cup\; init2[c_z/c_y])\rangle} \mathcal{E}'[y/x]|[L]|\mathcal{F}\backslash M} \quad z = get(\mathcal{M} - ((\psi(\mathcal{E}) \cup \psi(\mathcal{F})) - (M \cup N)))$$

5. *Opérateur d'intériorisation* :

a. $$\frac{\mathcal{E} \xrightarrow{\langle_{M}a_x,const,init\rangle} \mathcal{E}' \quad a\notin L}{hide\ L\ in\ \mathcal{E} \xrightarrow{\langle_{M}a_x,const,init\rangle} hide\ L\ in\ \mathcal{E}'}$$

b. $$\frac{\mathcal{E} \xrightarrow{\langle_{M}a_x,const,init\rangle} \mathcal{E}' \quad a\in L}{hide\ L\ in\ \mathcal{E} \xrightarrow{\langle_{M}i_x,const,init\rangle} hide\ L\ in\ \mathcal{E}'}$$

6. *Opérateur de séquencement* :

a $\dfrac{\mathcal{E} \xrightarrow{\langle M^{a_x, const, init}\rangle} \mathcal{E}' \quad a \neq \delta}{\mathcal{E} >> F \xrightarrow{\langle M^{a_x, const, init}\rangle} \mathcal{E}' >> F}$

b. $\dfrac{\mathcal{E} \xrightarrow{\langle M^{\delta_x, const, init}\rangle} \mathcal{E}'}{\mathcal{E} >> F \xrightarrow{\langle M^{i_x, const, init}\rangle}_{\{x:\delta:d_\delta\}} [F]}$

7. *Opérateur d'interuptiont :*

a. $\dfrac{\mathcal{E} \xrightarrow{\langle M^{a_x, const, init}\rangle} \mathcal{E}' \quad a \neq \delta}{\mathcal{E} \quad [> \mathcal{F} \xrightarrow{\langle M^{a_y, const[c_y/c_x],\ init[c_y/c_x]}\rangle} \mathcal{E}'[y/x] \ [> \ \mathcal{F} \backslash M}$ $y = get(\mathcal{M} - (\psi(\mathcal{E}) \cup \psi(\mathcal{F}) - M))$

b. $\dfrac{\mathcal{E} \xrightarrow{\langle M^{\delta_x, const, init}\rangle} \mathcal{E}' \quad a \neq \delta}{\mathcal{E} \quad [> \mathcal{F} \xrightarrow{\langle M^{\delta_y, const[c_y/c_x], init[c_y/c_x]}\rangle}_{\psi(\mathcal{F})-M} [stop]}$ $y = get(\mathcal{M} - (\psi(\mathcal{E}) \cup \psi(\mathcal{F}) - M))$

c. $\dfrac{\mathcal{F} \xrightarrow{\langle M^{a_x, const, init}\rangle} \mathcal{F}'}{\mathcal{E} \quad [> \mathcal{F} \xrightarrow{\langle M^{a_y, const[c_y/c_x], init[c_y/c_x]}\rangle}_{\psi(\mathcal{E})-M} [stop] \ [> \ \mathcal{F}'[y/x]}$ $y = get(\mathcal{M} - (\psi(\mathcal{E}) \cup \psi(\mathcal{F}) - M))$

8. *Opérateur d'attente(delay) :*

$$\dfrac{\mathcal{E} \xrightarrow{\langle M^{a_x, const, init}\rangle} \mathcal{E}'}{delay(u)\mathcal{E} \xrightarrow{\langle M^{a_x, add_u(u, const), init}\rangle} \mathcal{E}'}$$

9. *Renomage des portes :*

a. $\dfrac{\mathcal{E} \xrightarrow{\langle M^{a_x, const, init}\rangle} \mathcal{E}' \qquad a \notin \{a_1,...,a_n\}}{\mathcal{E}[b_1/a_1,...,b_n/a_n] \xrightarrow{\langle M^{a_x, const, init}\rangle} \mathcal{E}'[b_1/a_1,...,b_n/a_n]}$

b. $\dfrac{\mathcal{E} \xrightarrow{\langle M^{a_x, const, init}\rangle} \mathcal{E}' \qquad a \ = a_i \ (1 \leq i \leq n)}{\mathcal{E}[b_1/a_1,...,b_n/a_n] \xrightarrow{\langle M^{b_{i_x}, const, init}\rangle} \mathcal{E}'[x{:}a'/x{:}a][b_1/a_1,...,b_n/a_n]}$

10. *Instantiation de processus :*

$$\dfrac{P{:=}E \quad ,_M[E] \xrightarrow{M^{a_x}} \mathcal{F}}{_M[P] \xrightarrow{M^{a_x}} \mathcal{F}}$$

Définition des fonctions **(a)** $creat_const \in D \times 2^{\mathcal{M} \times act}_{f_n}$! *Contraintes :est définie récursivement comme suit:*

$creat_const(d, \phi) = false.$

$creat_const(d, M \cup \{(x : a : d_a)\}) = (d_a \leq c_x \leq \ d_a + d) \ \ or \ \ creat_const(d, M)$

(b) $add_const \in D \times Contraintes$! *Contraintes ::est définie récursivement comme suit:*

$add_const(u, true) = false$

$add_const(u, true) = true$

$add_const(u, (min \sqsubseteq c_x \sqsubseteq max)) = (min + u \sqsubseteq c_x \sqsubseteq max + u)$. $(\sqsubseteq \in \{<, \leq\})$

$add_const(u, const1 \ or \ const2) = (add_const(u, const1)) \ or \ (add_const(u, const2))$

(c) L'ensembles des fonctions de substitution des noms d'horloges est sub_clock (ie $sub_clock = clocks!2^{clocks}$: $\sigma, \sigma_1, \sigma_2, ...$*désignent des éléments de sub_clocks.Etant donnés* $c_x, c_y \in clocks$ *et* $clk \in 2^{clocks}$, $Const \in 2^{clocks}$,*alors:*

- *L'application de σ à c_x* sera écrite σc_x.
- $clk\ \sigma = \cup_{c_x \in clk} \sigma$
- $\sigma[c_y/c_z]$ *est une fonction de substitution définie par* $\sigma[c_y/c_z]c_x = \{c_y\}$ *si* $c_z = c_x$

σx

sinon

soit σ une fonction de substitution:

- *La substitution simultanée Const σ ($const \in Contraintes$) de toutes les occurrences de c_x dans const par σc_x,est définie récursivement par*

$$true\ \sigma = true$$
$$false\ \sigma = false$$
$$(min \sqsubseteq c_x \sqsubseteq max)\sigma = \{(min \sqsubseteq \sigma c_x \sqsubseteq max)\}$$
$$(const1\ or\ const2)\sigma = (const1\ \sigma)\ or\ (const2\ \sigma).$$

- *La substitution simultanée init σ ($init \in Initialisation$) de toutes les occurrences de c_x dans init par σc_x,est définie récursivement par*

$$\phi\sigma = \phi$$
$$(init \cup \{c_x\}) = (\ init\ \sigma) \cup \{\sigma c_x\}$$

(d) La fonction $\cup$ entre les contrainte est définie récursivement comme suit :

- $true \cup const = const$
- $false\ \cup\ const = false$
- $((min1 \sqsubseteq c_x \sqsubseteq max1) \cup (min2 \sqsubseteq c_x \sqsubseteq max2)) = (\max(min1, min2) \sqsubseteq c_x \sqsubseteq \min(max1, max2))$

tel que min, ($resp$ max) est la fonction qui calcule le minimum, (resp maximum) des deux réels.

- $((min1 \sqsubseteq c_x \sqsubseteq max1) \cup (min2 \sqsubseteq c_y \sqsubseteq max2)) = ((min1 \sqsubseteq c_x \sqsubseteq max1)\ or\ (min2 \sqsubseteq c_y \sqsubseteq max2))$
- $A \cup (B\ or\ C) = (A \cup B)\ or\ (A \cup C)$.

Détails d'implantation Elle a comme entrée le résultat de l'analyseur syntaxique (arbre de processus) et génère en sortie un STEM Temporel. Les deux fonctions principales sont :

- – **gen_succ : configuration → (transition * configuration) list**

Cette fonction accepte en entrée une configuration, et retourne une liste de toutes les dérivations possibles de cette configuration sous forme d'un couple (trans, config).

Exemple 4.4 *gen_succ*$(C) = (tr_1, C_1) :: (tr_2, C_2) :: []$. *Qui signifie* $C \overset{tr_1}{\rightarrow} C_1$,$C \overset{tr_2}{\rightarrow} C_2$.

- – **gen_all : configuration → (configuration* transition * configuration) list** c'est la fonction principale de génération de code (STEM Temporel), elle a comme paramètre une configuration initiale et génère en premier temps les dérivations immédiates et puis elle s'appelle elle-même pour ces configurations résultantes, le résultat final est une liste de triplet (C_1, tr, C_2) qui reflète la dérivation $C_1 \overset{tr}{\rightarrow} C_2$

4.4 Conclusion

Ce chapitre est constitué de trois parties, dans la première partie nous avons abordé la notion de programmation fonctionnelle avec une présentation brève du langage utilisé CAML. Des exemples simples ont été utilisés pour l'explication des principaux concepts du langage. Dans la deuxième partie nous avons présenté la technique de compilation et nous avons montré les différentes phases classiques de compilateur avec illustration par des schémas. Dans la troisième partie nous avons présenté les différentes étapes de l'implantation de notre outil , ces étapes peuvent être divisées en deux phases : La phase de compilation et la phase de la représentation graphique.

Entre autre, nous avons présenté les techniques utilisées pour le développement du compilateur D-LOTOS (phases : lexical, syntaxique, génération de code). Chacune des phase étant illustrée par un schéma. Les fonctions développées sont représentées en détail et des exemple ont été utilisés pour l'illustration du fonctionnement de ces fonctions.

Il est à noter que le choix du paradigme fonctionnel a écourté considérablement le temps de développement de la phase de génération de code.

Chapitre 5

conclusion

Le travail présenté dans ce mémoire concerne la conception et l'implantation d'un outil pour traduction de spécifications écrites dans le langage temps réel D-LOTOS en graphes de comportements appelés systèmes de transitions étiquetées temporels (STEMs temporels)

Après avoir donné la définition complète du langage D-LOTOS et de sa sémantique de maximalité temporelle, nous avons consacré un chapitre pour la présentation du modèle sémantique des automates temporels. La liaison de ce modèle avec la sémantique d'entrelacement à été largement soulignée. Entre autre nous avons marqué l'importance de la levée des hypothèses d'atomicité temporelle et structurelle pour la spécification des systèmes critiques avec plus de facilités. Ces constats ont motivé l'utilisation des sémantiques dites du vrai parallélisme, en particulier la sémantique de maximalité temporelle. Pour permettre le développement d'outils de vérification basés sur le langage D-LOTOS, le modèle sémantique des STEMtemporels a été défini.

En marge des résultats obtenus dans notre travail, nous avons voulu confirmé l'idée que les langages fonctionnelles conviennent mieux pour le prototypage. En fait, dans des travaux antérieurs dirigés par notre encadreur le Dr Djamel-Eddine SAIDOUNI, l'environnement de vérification formelle du parallélisme FOCOVE a été développé dans un paradigme de programmation impérative. La taille des programmes développés et de l'ordre de 8000 lignes de code. L'objectif qui nous a été tracé par notre encadreur a été atteint par le choix du langage fonctionnel ML. Les programmes obtenus sont en fait de tailles très réduites relativement, environ 600 lignes de code, à ceux obtenues dans un style de programmation impérative.

Un chapitre a été consacré pour la présentation de la conception et l'implantation de notre outil. Pour des raison de clarté de l'exposé nous avons introduit le langage de programmation CAML. Une comparaison avec le paradigme impératif a été faite. Nous avons donné également les différents composants de notre outil à savoir l'analyseur lexical, l'analyseur syntaxique et le générateur de stems temporels. Les fonctions que nous avons écrites ont été présentées en détail.

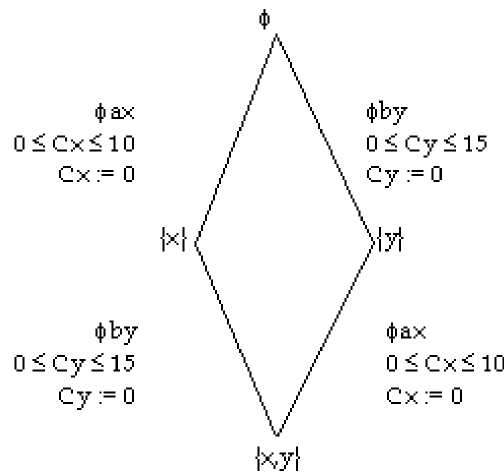


Figure 5.1: STEM Temporel classique

Notre travail peut être considéré comme un premier pas pour le développement d'un environnement de vérification des systèmes temps réel dont D-LOTOS est le langage de spécification.

Cependant plusieurs travaux reste à développer, parmi lesquels nous pouvons citer :

L'exploitation de l'information sur le parallélisme disponible directement dans les structures de stems temporels à fin de procéder à des réductions plus intéressantes. Notons que des travaux similaires sur des modèles non temporels ont été faites dans la littérature. En fait, nous indiqué qu'une réduction possible d'un STEM Temporel consiste à regrouper les transitions parallèles en une seule transition étiquetée par plusieurs actions qui s'exécutent en parallèle. A titre d'exemple, la figure5.2 présente une réduction possible du système de la figure5.1 dans lequel les actions a et b sont en parallèle.

Soit la spécification :

$SYSTEM \;\; proc[a[3], b[5]] \;\; := \;\; a\{10\}; stop \;\; ||| \;\; b\{15\}; stop \;\; ENDSYS$

Nous pouvons remarquer, sur cet exemple, que nous somme passé d'un système de transitions qui comporte 4 places et 4 transitions à un système qui comporte 2 places et 1 transition. Evidement, de telles réductions devraient se faire en respectant soit les sémantiques linéaires ou arborescentes, et ceci selon les abstractions jugées acceptables pour la vérification des propriétés requises de l'application.

Un autre travail consiste en l'utilisation de formalismes pour l'expression des propriétés à vérifier. Nous pensons que les logiques temporelles TCTL, DC (Duration Caculus)[5] et DIL(Duration Interval Logic)[4]peuvent constituées de tels modèles.

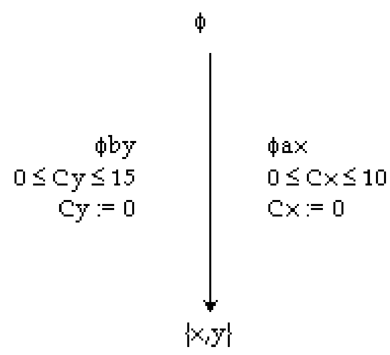


Figure 5.2: STEM Temporel condensé

Bibliographie

[1] R. Alur and D. Dill. A theory of timed automata. *TCS*, 126:183–235, 1994.

[2] J. Baeten and J. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3:142–188, 1991.

[3] T. Bolognesi. LOTOS-like process algebras with urgent or timed interaction. In *FORTE*, pages 255–270. North Holland, 1991.

[4] A. Bouajjani, Y. Lakhnech, and R. Robbana. From duration calculus to linear hyblid automata. In *CAV*, volume 939 of *LNCS*, pages 198–210. Springer-Verlag, 1995.

[5] Z. Chaochen. Duration calculus, a logical approach to real-time systems. In *AMAST*, volume 1548 of *LNCS*, pages 1–7. Springer-Verlag, 1998.

[6] J. Courtiat and R. de Oliveira. On RT-LOTOS and its application to the formal design of multimedia protocols. *Annals of Telecommunications*, 50:11–12, 1995.

[7] J. Courtiat and R. de Oliveira. A reachability analysis of RT-LOTOS specifications. In *FORTE*, London, 1995. Chapman and Hall.

[8] J. Courtiat, R. de Oliveira, and L. Andriantsiferana. Specification and validation of multimedia protocols using RT-LOTOS. In *Fifth IEEE International Conference on Future Trends of Distributed Computing Systems*, Cheju Island, Korea, 1995.

[9] J. Courtiat, C. . A. S. Santos, C. Lohr, and B. Outtaj. Experience with RT-LOTOS, a temporal extension of the LOTOS formal description technique. *Computer Communications*, 23:1104–1123, 2000.

[10] J. P. Courtiat, M. S. de Camargo, and D. E. Saïdouni. RT-LOTOS: LOTOS temporisé pour la spécification de systèmes temps réel. In R. Dssouli, G. Bochmann, and L. Le'vesque, editors, *Ingénierie des Protocoles (CFIP'93)*, pages 427–441. Hermes, 1993.

[11] J. P. Courtiat and R. C. de Oliveira. About time nondeterminism and exception handling in a temporal extension of LOTOS. In S. T. Vuong and S. T. Chanson, editors, *PSTV'94*, pages 37–52. Chapman & Hall, 1994.

[12] J. P. Courtiat and D. E. Saïdouni. Relating maximality-based semantics to action refinement in process algebras. In D. Hogrefe and S. Leue, editors, *IFIP TC6/WG6.1, 7th Int. Conf. on Formal Description Techniques (FORTE'94)*, pages 293–308. Chapman & Hall, 1995.

[13] R. Devillers. Maximality preserving bisimulation. *TCS*, 102:165–183, 1992.

[14] R. Gorrieri, M. Roccetti, and E. Stancampiano. A theory of processes with durational actions. *Theoretical Computer Science*, 140:73–94, 1995.

[15] M. Hennessy and T. Regan. A temporal process algebra. In J. M. J. Quemada and E. Vazquez, editors, *FORTE*, pages 33–48. North-Holland, 1991.

[16] 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.

[17] 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.

[18] F. Moller and C. Tofts. A temporal calculus of communicating systems. In J. C. Baeten and J. W. Klop, editors, *CONCUR*, volume 458 of *LNCS*, pages 401–415. Springer-Verlag, 1990.

[19] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In K. G. Larsen and A. Skou, editors, *CAV'91*, volume 575 of *LNCS*, pages 376–398. Springer-Verlag, 1991.

[20] J. Quemada, A. Azcorra, and D. Frutos. TIC: a timed calculus for LOTOS. In S. T. Vuong, editor, *Formal Description Techniques (FORTE'89)*, pages 195–209. North-Holland, 1990.

[21] P. Ribet, F. Vernadat, and B. Berthomieu. On combining the persistent sets method with the covering steps graph method. Technical Report 02388, LAAS-CNRS, 7 avenue du colonl Roche, 31077, Toulouse Cedex France, 2002.

[22] 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.

[23] D. E. Saïdouni and O. Labbani. Maximality-based symbolic model checking. In *ACS/IEEE International Conference on Computer Systems and Applications*, Tunisia, July 2003.

[24] P. N. M. Sampaio. *Conception Formelle de Documents Multimédia Interactifs: Une Approache S'appuyant sur RT-LOTOS*. PhD thesis, LAAS-CNRS, 7 avenue du colonel Roche, 31077 Toulouse cedex, France, 2003.

[25] M. Yannakakis and D. Lee. An efficient algorithm for minimizing real-time transition systems. *Formal Methods in System Design*, 11:113–136, 1997.

[26] W. Yi. CCS + Time = an interleaving model for real time systems. In J. L. Albert, B. Monien, and M. R. Artalejo, editors, *ICALP'91*, volume 510 of *LNCS*, pages 217–228. Springer-Verlag, 1991.