Contribution à la conception de systèmes temps-réel s'appuyant sur la technique de description formelle RT-Lotos

Christophe Lohr

Année 2002

THÈSE

Préparée au
Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS

En vue de l'obtention du
Doctorat de l'Institut National Polytechnique de Toulouse

Spécialité :
Programmation et Systèmes

par
Christophe LOHR

Contribution à la conception de systèmes temps-réel s'appuyant sur la technique de description formelle RT-LOTOS

Soutenue le jeudi 19 décembre 2002, devant le jury :

Président	**Patrick**	**SALLÉ**
Directeur de thèse	**Jean-Pierre**	**COURTIAT**
Rapporteurs	**Richard**	**CASTANET**
	Elie	**NAJM**
Examinateurs	**Michel**	**DIAZ**
	Hubert	**GARAVEL**
	François	**VERNADAT**

Remerciements

Mes travaux de recherche ont été menés au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS) au sein du groupe Outils et Logiciels pour la Communication (OLC).

Je remercie donc Jean-Claude Laprie, directeur du LAAS-CNRS pendant ces années, de m'avoir accueilli au sein du laboratoire, ainsi que Michel Diaz, responsable du groupe OLC pour m'avoir accepté dans son groupe.

J'aimerais remercier Richard Castanet, Elie Najm, Michel Diaz, Hubert Garavel, Patrick Sallé, et François Vernadat de m'avoir fait l'honneur de participer à mon jury.

Un grand merci à Aurore Collomb, Hubert Garavel, et Pierre de Saqui-Sannes pour la relecture attentive de mon manuscrit, et à François Vernadat qui m'a aussi aidé à préparer ma soutenance.

Je tiens à remercier tout particulièrement mon directeur de recherche Jean-Pierre Courtiat, pour son dévouement et son aide constante au cours de toutes ces années, grâce auxquels il m'a été possible de mener cette thèse à bien. Je n'oublierai jamais ses maximes pleines de sagesse, et notamment les plus célèbres : « Si c'était facile, ce ne serait pas une thèse ! » et « Le temps passe vite ! ».

Je suis très reconnaissant envers tous ceux qui ont contribué à mes travaux de recherche : tout d'abord Roberto Cruz de Oliviera, génial auteur de l'outil `rtl` *et des nombreux aspects théoriques qui l'entourent ; et je regrette qu'il n'ait pu mener à terme ses projets de recherche. Je remercie Celso Saibel Santos, collègue chaleureux dont les discussions sur son algorithme d'agrégation ont fait germer dans ma tête ce qu'est devenu le TLSA. Je remercie Paulo Sampaio qui a eu le courage de bien vouloir exploiter et mettre à l'épreuve mon TLSA pour ses propres travaux. Je remercie Benaceur Outtaj pour son aide concernant le calcul des invariants* Kronos. *Je tiens à remercier Pierre de Saqui-Sannes et Ludovic Apvrille sans qui* TURTLE *ne serait pas.*

Merci aussi à tous mes compagnons de route depuis mes débuts dans la recherche : Laurent Andriantsiferana, Rachid Benabbou, Emmanuel Castel, Hugues Buisson, Guillermo Hoyos, Roberta Gomes, et tous les autres.

Je veux exprimer également toute ma reconnaissance au personnel du LAAS, ingénieurs, techniciens et administratifs, pour leur concours et leur efficacité essentiels à la conclusion de ce travail.

Je voudrais remercier aussi mes parents, ma famille, pour leur présence et leur soutien malgré les kilomètres.

Je tiens à remercier spécialement tous mes amis (je ne me risquerai pas à faire la liste, c'est trop long et je m'en voudrai d'oublier quelqu'un), pour les nombreuses fêtes, discussions de comptoir, plongées sous-marines, et autres moments de détente indispensables à une thèse épanouissante et réussie.

Enfin, mes pensées vont vers toi, Sidonie, qui partage ma vie depuis quelques années maintenant, pour toute ton affection, ta tendresse, mais aussi ta compréhension et ton soutien de tous les instants. Je t'embrasse.

Table des matières

Introduction **1**

1 Formalisation des systèmes temps-réel et RT-Lotos **7**
1.1 Différents formalismes pour exprimer des contraintes temporelles 7
1.1.1 Extensions temporisées des réseaux de Petri 8
1.1.2 Automates temporisés 9
1.1.3 Algèbres de processus 11
1.1.4 Principales extensions temporelles à Lotos 12
1.2 Le langage RT-Lotos 19
1.2.1 Lotos 19
1.2.2 L'extension RT-Lotos 26
1.2.3 Traitement du non-déterminisme temporel 38
1.3 Validation avec RT-Lotos 40
1.3.1 Simulation 40
1.3.2 Vérification formelle 44
1.3.3 Analyse d'accessibilité 49
1.3.4 Model-checking 55
1.4 Expériences avec RT-Lotos 61
1.5 Conclusion 62

2 Ordonnancement temporel de systèmes cohérents **65**
2.1 Synthèse d'automates d'ordonnancement 65
2.1.1 Motivation pour un automate d'ordonnancement 66
2.1.2 Adéquation des automates temporels existants 67
2.1.3 Problématique de la synthèse d'automates 68
2.2 Une proposition d'automate d'ordonnancement : le `TLSA` 69
2.2.1 Définition formelle d'un `TLSA` 70
2.2.2 Propriétés du formalisme `TLSA` 72
2.3 Un algorithme pour synthétiser un `TLSA` 76
2.3.1 Initialisation de l'algorithme 79
2.3.2 Identification de la configuration accessible au plus tôt de chaque classe 79
2.3.3 Sélection et regroupement des transitions Lotos 89
2.3.4 Découpage des états 95
2.3.5 Pré-traitement du graphe minimal d'accessibilité 97
2.4 Élimination des comportements non cohérents 97
2.4.1 Démarche du filtrage des comportements non cohérents 97
2.4.2 Suppression des configurations indésirables 97

2.4.3 Insertion de transitions particulières _t_ . . . 98
2.5 Utilisation du TLSA . . . 99
2.5.1 Présentation d'un cas d'étude . . . 99
2.5.2 Mise en oeuvre de la méthodologie . . . 100
2.5.3 Bilan de la méthodologie . . . 102
2.6 Conclusion . . . 104

3 Méthodologie liant formel et semi-formel 105
3.1 Un profil temps-réel formel pour UML . . . 105
3.1.1 Les projets UML temps-réel . . . 107
3.1.2 L'approche TURTLE . . . 108
3.1.3 Mise en œuvre de TURTLE . . . 114
3.1.4 Étude de cas . . . 118
3.1.5 Les limites de la méthodologie . . . 121
3.2 Algorithmes pour traduire les diagrammes de classes TURTLE . . . 124
3.2.1 Définitions préliminaires . . . 124
3.2.2 Traduction des *Tclasses* à l'origine de relations de *Preemption* . . . 126
3.2.3 Traduction des *Tclasses* à l'origine de relations de *Sequence* . . . 126
3.2.4 Traduction des *Tclasses* qui ne sont à pas l'origine de relations de *Preemption* ou de *Sequence* . . . 129
3.2.5 Réduction des *Tclasses* à l'origine à la fois de relations de *Sequence* et de *Preemption* . . . 129
3.2.6 Traduction des ensembles de *Tclass* en relation de *Parallel* et *Synchro* 135
3.2.7 Traduction des *Tclasses* actives . . . 139
3.3 Extensions par domaine d'application . . . 140
3.3.1 Invocation de classes . . . 140
3.3.2 Suspension/reprise d'activité . . . 142
3.3.3 Traduction du profil temps-réel système en TURTLE . . . 144
3.4 Conclusion . . . 151

Conclusion 155

Publications de l'auteur 161

Références 163

Annexe 175

A Justification de la traduction des domaines d'urgence en invariants 177

B Pré-traitement du graphe minimal d'accessibilité 181
B.1 Reporter les fonctions C et *theta* . . . 181
B.2 Rerouter les régions-point . . . 182
B.3 Projection et minimisation . . . 182

B.3.1 Choix de l'outil CADP 184
B.3.2 Insertion de la minimisation dans la production du `TLSA` 184
B.3.3 Traitement des informations temporelles 184
B.3.4 Reconstruction des fonctions C et *theta* 185

C Spécification RT-LOTOS engendrée depuis TURTLE 187

Liste des figures

1.1 Comparaison des extensions temporelles à LOTOS 14
1.2 Principaux opérateurs de Basic-LOTOS 20
1.3 Illustration des opérateurs de Basic-LOTOS 21
1.4 Syntaxe de Basic-LOTOS 22
1.5 Sémantique opérationnelle de Basic-LOTOS 23
1.6 Représentation d'une action en LOTOS 24
1.7 Règles de synchronisation en LOTOS avec données 25
1.8 Principaux opérateurs LOTOS avec utilisation de données 25
1.9 Sémantique opérationnelle de (Full) LOTOS 26
1.10 Exemple de contraintes temporelles spécifiées avec RT-LOTOS 27
1.11 Illustration des opérateurs RT-LOTOS 29
1.12 Syntaxe de Basic RT-LOTOS 32
1.13 Le modèle Basic RT-LOTOS 33
1.14 Exemple de déclaration d'un type naturel en RT-LOTOS 35
1.15 Exemple d'utilisation de l'opérateur `@` 36
1.16 Syntaxe de (Full) RT-LOTOS 37
1.17 Le modèle (Full) RT-LOTOS 38
1.18 Comparaison entre `latency(`t`)` et `i{`t`}` 39
1.19 Validation avec RT-LOTOS 41
1.20 Exemple : médium de communication 42
1.21 Simulation du médium de communication 43
1.22 Exemple de `DTA` 45
1.23 L'urgence dans le `DTA` avec actions *weak* ou avec domaine temporel U . . . 50
1.24 Optimisation des horloges du `DTA` 51
1.25 Exemple de graphe minimal d'accessibilité 53
1.26 Spécification vérifiable uniquement *à la volée* 54
1.27 Architecture générale de l'outil `Kronos` 56
1.28 Exemples de calcul d'invariant pour quelques comportements RT-LOTOS . 59

1.29 Spécification RT-LOTOS du croisement ferroviaire 60
1.30 Automates temporels issus des processus *Train*, *Gate*, et *Controller* 61

2.1 Illustration des règles de tir 71
2.2 Spécification RT-LOTOS du problème de Dechter 75
2.3 Exemples d'observateur 75
2.4 Mise en place d'un observateur 76
2.5 `TLSA` partiel issu du problème de Dechter 76
2.6 Vue d'ensemble de l'algorithme 77
2.7 Principe de l'algorithme de synthèse d'un `TLSA` 77
2.8 Traitement d'une transition `t` 81
2.9 Transition `t` : valuation strictement supérieure à 83
2.10 Traitement d'une transition LOTOS 84
2.11 Recopie d'un nœud 87
2.12 Graphe des configurations accessibles au plus tôt : `TLSA` avant regroupement des transitions 91
2.13 Fenêtres de tir contiguës 92
2.14 Fenêtres de tir ayant la même expression 93
2.15 Fenêtre de tir ayant des expressions différentes mais dont une seule a des variables 93
2.16 Fenêtre de tir ayant des expressions différentes et dont chacune a des variables 94
2.17 Transitions incluses 94
2.18 Simplifier l'historique des tirs 95
2.19 Découpage des états de contrôle 96
2.20 (*a*) Scénario multimédia (*b*) Spécification RT-LOTOS 100
2.21 (*a*) Graphe minimal d'accessibilité (*b*) `TLSA` associé 101
2.22 (*a*) Graphe minimal d'accessibilité consistant (*b*) `TLSA` associé 103

3.1 Une méthodologie `UML` intégrant la validation RT-LOTOS 106
3.2 Le type abstrait *Gate* 109
3.3 Structure d'une classe stéréotypée *Tclass* 110
3.4 Opérateurs de composition héritant de *Composer* 111
3.5 Règles de composition des diagrammes de classes `TURTLE` 112
3.6 Opérateurs non temporels des diagrammes d'activité `TURTLE` 113
3.7 Opérateurs temporels des diagrammes d'activité `TURTLE` 114
3.8 Règles de composition des symboles d'activité `TURTLE` 115
3.9 De la modélisation `TURTLE` à la vérification 115
3.10 Traduction RT-LOTOS des activités `TURTLE` 117
3.11 Diagramme de classes n° 1 de la machine à café 119
3.12 Extrait du graphe minimal d'accessibilité de la machine à café n° 1 120
3.13 Diagramme de classes n° 2 de la machine à café 121

3.14 Graphe minimal d'accessibilité de la machine à café n° 2 avec une offre sur les boutons limitée à 40 (graphe (a)) puis à 60 (graphe (b)) 122
3.15 Instance de classes 125
3.16 *Tclass* à l'origine de plusieurs relations de *Preemption* 126
3.17 *Tclass* à l'origine de plusieurs relations de *Sequence* 128
3.18 *Tclasses* à l'origine à la fois de relations de *Sequence* et de *Preemption* . . 130
3.19 Fonction *String Rewrite*(*process* : *String*, *delta* : *String*, *proc_pre* : *String*) 134
3.20 Règles de composition de l'*Invocation* 141
3.21 Exemple d'utilisation de *Invocation* 141
3.22 Traduction de *Invocation* en `TURTLE` natif 142
3.23 Exemple d'utilisation de *Suspend* 142
3.24 Règles de composition de *Suspend* 143
3.25 Opérateurs temporels suspensibles 144
3.26 Outillage des *Tclasses* suspensibles 145
3.27 Outillage des opérateurs suspensibles : tester la parité des interruptions . . 145
3.28 Appel suspensible sur une *Gate* 146
3.29 Transformation simple du choix dans les *Tclasses* suspensibles 146
3.30 Transformation générale du choix dans les *Tclasses* suspensibles 147
3.31 Transformation simple d'un délai suspensible 147
3.32 Transformation générale d'un délai suspensibles 148
3.33 Traduction d'une offre limitée suspensible 149
3.34 Transformation d'une offre limitée non suspensible 150
3.35 Outillage des compositions parallèles suspensibles 151
3.36 Exemple de transformation d'activité suspensible 152
3.37 Conformité entre implémentation et `TLSA` 158

B.1 Reporter les fonctions `C` et `theta` 182
B.2 Rerouter le graphe minimal d'accessibilité 183

Liste des algorithmes

2.1 Recherche des configurations accessibles au plus tôt 80
2.2 Traitement des transitions temporelles 82
2.3 Traitement des transitions LOTOS 86
2.4 Réplication d'un nœud 87
3.1 Traduction des *Tclasses* à l'origine de relations de *Preemption* 127

3.2 Traduction des *Tclasses* à l'origine de relations de *Sequence* 128
3.3 Traduction des *Tclasses* qui ne sont à pas l'origine de relations de *Preemption* ou de *Sequence* 129
3.4 Fonction $String\ Make\_Pre(T, \mathcal{T}, \mathcal{R})$ 131
3.5 Fonction $String\ Make\_Seq(T, \mathcal{T}, \mathcal{R})$ 132
3.6 Réduction des *Tclasses* à l'origine de relations de *Sequence* et de *Preemption* 133
3.7 Amélioration de la fonction $String\ Make\_Seq(T, \mathcal{T}, \mathcal{R})$ 135
3.8 Amélioration de la réduction des *Tclasses* à l'origine de relations de *Sequence* et de *Preemption* 136
3.9 Construire l'ensemble de renommage des portes 137
3.10 Traduction des relations *Parallel* et *Synchro* 138

Introduction

Au cours des décennies passées, les informaticiens ont eu pour défi d'élaborer des théories et des techniques qui puissent garantir que les systèmes informatiques fonctionnent correctement, c'est-à-dire, selon des caractéristiques prescrites exprimant leur comportement désiré. Les manières traditionnelles d'obtenir de telles «garanties» ont été la simulation et le test. Ces techniques permettent d'obtenir un certain degré de confiance dans la conformité du fonctionnement du système. Cependant, dans beaucoup de cas cette approche prend excessivement de temps et fournit souvent seulement une évaluation probabiliste de la conformité.

Dans la littérature, il a été proposé un grand nombre de techniques dotées d'un support mathématique pour raisonner sur la conformité des systèmes informatiques [Hoa69, Dij75, Pnu77, Lam77, Hoa78, Mil89, Hol91]. L'idée générale est de décrire dans un cadre formel les systèmes informatiques à étudier, puis d'appliquer des méthodes rigoureuses pour démontrer que la description des systèmes est correcte, qu'elle répond bien à certaines exigences formellement définies. L'avantage de cette approche est qu'elle peut être employée tôt dans le cycle de conception pour détecter des erreurs de conception logique avant même que les systèmes n'aient encore été mis en œuvre.

L'étude exposée dans ce document s'intéresse à la conception de systèmes temps-réel en s'appuyant sur la méthode formelle RT-LOTOS, extension temporelle à l'algèbre de processus LOTOS. Nous abordons plusieurs points relatifs à la spécification, la validation et l'ordonnancement de systèmes concurrents sujets à des contraintes logiques et temporelles.

Systèmes temps-réel

Un *système temps-réel* est un système informatique doté d'un comportement qui est contraint par le temps. À la différence de systèmes non temporisés, un système temps-réel doit interagir correctement avec son environnement non seulement au regard des informations échangées, mais également au regard des instants auxquels ces interactions se réalisent.

Les systèmes temps-réel sont fréquemment soit des systèmes embarqués, soit des systèmes critiques, qui apparaissent comme des composants de systèmes complexes plus importants et dont la sûreté de fonctionnement est critique [Sto96]. On distingue les systèmes temps-réel *durs*, qui doivent toujours réagir en temps opportun, et les systèmes temps-réel *mous*, qui peuvent occasionnellement ne pas satisfaire leurs contraintes temporelles sans compromettre leur fonctionnement normal [TH97]. Certains systèmes peuvent combiner des tâches temps-réel dures et des tâches temps-réel molles. Les systèmes temps-réel ont pour caractéristique commune qu'ils peuvent être constitués de nombreux composants fonctionnant en parallèle; ce sont donc des *systèmes concurrents*. Lorsqu'un système temps-réel doit réagir à chaque stimulus de l'environnement, il est appelé *système réactif* [Pnu86].

Le qualificatif «temps-réel» est parfois employé pour désigner des systèmes qui doivent réagir aux stimuli de leur environnement «aussi vite que possible». Implicitement, de tels systèmes ont pour contrainte de devoir terminer la réaction à un premier stimulus avant que le suivant ne se produise. Pour répondre à ce besoin, de tels systèmes doivent être «rapides». En fait, un système temps-réel, n'est pas nécessairement rapide, seulement il doit effectuer ses travaux au bon moment. Les systèmes temps-réel sont souvent sujets à des contraintes complexes de synchronisation et doivent accomplir plusieurs tâches en parallèle. De telles caractéristiques rendent complexes leur conception et leur exécution. Par ailleurs, s'il s'agit souvent de systèmes critiques, et l'on doit s'assurer de la correction de leur comportement. Des méthodes formelles peuvent alors être employées pour aider le développement de tels systèmes en fournissant un cadre formel dans lequel la vérification de leur fonctionnement correct peut être menée.

Méthodes formelles

Les *méthodes formelles* ou *techniques de description formelle* jouent un rôle fondamental dans les différentes étapes du processus de l'ingénierie des systèmes informatiques. Les techniques de description formelle peuvent être définies comme un ensemble de notations pourvues d'une sémantique formelle et d'outils utilisés pour spécifier sans ambiguïté un système complexe dans les différentes phases de son cycle de vie (expression des besoins, architecture fonctionnelle, architecture détaillée), et pour valider (simuler et vérifier formellement) un certain nombre de ses propriétés [Win90, Hal90]. Les méthodes formelles sont généralement caractérisées par quatre concepts de base : l'abstraction, l'indépendance vis-à-vis de l'implémentation, la sémantique formelle, et les méthodes de vérification utilisables.

À la fin des années 80, plusieurs techniques ont été développées pour décrire des systèmes de plus en plus complexes, d'une manière complète, non ambiguë, et à un niveau d'abstraction élevé. Ces techniques reposent en général sur des modèles mathématiques puissants qui leur permettent d'assurer une vérification a priori de ces systèmes.

Langage de description formelle

Les *langages de description formelle* sont issus du besoin de décrire complètement et sans ambiguïté les systèmes informatiques. La définition de la sémantique du langage est essentielle pour peu que l'on désire établir dans un cadre formel que tel système résout bien tel problème [Liv78, Gue81, AG92]. C'est en effet la sémantique qui donne le sens à un système, permettant ainsi de vérifier que le système fait effectivement ce que l'on attend de lui. L'utilisation de techniques de description formelles, complétées par des outils informatiques adaptés, offrant des fonctionnalités de simulation, de prototypage rapide, de validation et de vérification formelle, permet la conception de systèmes sûrs.

Les travaux présentés dans ce mémoire s'articulent autour de la méthode formelle RT-LOTOS, un langage qui permet de spécifier des systèmes temps-réel et de les valider par analyse d'accessibilité. RT-LOTOS appartient à la famille des algèbres de processus, un processus se définissant comme une machine à états finis fonctionnant de manière concurrente et autonome avec les autres processus et se synchronisant éventuellement avec eux lors de rendez-vous multiple.

Analyse d'accessibilité

La *vérification* est une approche s'appuyant sur un raisonnement mathématique qui permet de prouver que la description formelle d'un système satisfait certaines propriétés souhaitées. En d'autres termes, il s'agit de vérifier le bon fonctionnement d'un système de processus qui interagissent (réseau de communication, programmes qui s'exécutent en parallèle, etc.), c'est-à-dire l'absence de comportements indésirables (états bloquants, boucles infinies d'actions sans progression du temps, etc.) et la conformité des comportements possibles à des spécifications préétablies [Arn92]. L'analyse automatique suppose une modélisation du comportement des processus considérés, et une formalisation des propriétés à vérifier.

Nous distinguons deux approches pour vérifier le bon fonctionnement d'un système. L'approche *énumérative* consiste à construire de manière exhaustive tous les états du système d'après sa description formelle. Cette approche, relativement simple à mettre en place, n'est possible que si la modélisation du système est finie; son inconvénient majeur est la taille du modèle engendré qui croit exponentiellement avec le nombre de composantes parallèles du système. L'approche *symbolique* consiste à représenter les ensembles caractéristiques d'états par des prédicats. Cette approche, qui peut s'appliquer à des systèmes dont la modélisation est infinie, a l'avantage principal d'éviter l'explosion combinatoire; par contre, il est crucial de disposer d'une procédure de décision efficace pour comparer les prédicats avec les propriétés à vérifier.

Une modélisation *discrète* du temps permet de vérifier les systèmes temps-réel en utilisant directement l'approche énumérative. Si l'on souhaite une modélisation *dense* du temps l'approche énumérative n'est pas utilisable directement, le système ayant alors une modélisation infinie. Pour ce faire, le modèle utilisé doit pouvoir représenter des ensembles infinis

d'états le long d'une progression temporelle.

Le langage RT-LOTOS est pourvu d'une méthode de vérification par analyse d'accessibilité, qui est basée sur l'approche énumérative avec un traitement du temps dense. L'*analyse d'accessibilité* consiste à rechercher dans l'ensemble des états possibles du système modélisé, depuis l'état initial, une suite d'actions atteignant un état particulier [AD91]. Cette technique passe par la construction exhaustive de tous les états (appelés également *configurations*) du modèle. Ce résultat est obtenu, lorsque c'est possible, par la construction d'un système à états finis appelé *graphe de régions*. Dans un graphe de régions, chaque région est constituée essentiellement d'un ensemble de configurations qui sont équivalentes dans le sens où elles peuvent évoluer vers la même région dans le futur.

L'approche symbolique pour la vérification systèmes temps-réel a également été définie dans le cadre de vérification par model-checking d'automates temporisés [Yov93].

Les travaux exposés dans ce mémoire reposent sur l'analyse d'accessibilité de RT-LOTOS [CRdO95].

Contributions de la thèse

Les principales contributions de notre étude portent sur trois aspects.

- Dans un premier temps, nous proposons un ensemble de travaux concernant les techniques de validation liées au langage RT-LOTOS. Plus précisément, nous nous intéressons à l'automate temporisé (appelé un `DTA`) dérivé d'une spécification RT-LOTOS, d'une part en terme de simulation (nous avons développé une technique de simulation rapide basée sur une exécution du `DTA` et non pas sur une exécution des actions sémantiques de la spécification RT-LOTOS), et d'autre part en termes de validation formelle (nous proposons un interfaçage entre des outils de vérification du type *model-checker* et RT-LOTOS au moyen d'une traduction entre les différents types d'automates temporisés employés).
- Dans un deuxième temps, nous proposons une technique ainsi qu'un modèle formel pour extraire les comportements cohérents d'un système et réaliser l'ordonnancement dans le temps de ses actions, c'est-à-dire que nous cherchons à exploiter sous un angle nouveau les informations présentes dans les graphes de régions obtenus par analyse d'accessibilité. Ainsi, nous proposons un moyen de raffiner les graphes d'accessibilité, d'en élaguer certaines branches jugées non souhaitables, d'extraire les dates de tir possible des actions, et de les présenter sous la forme d'un nouveau type automate temporisé (appelé un `TLSA`) ayant pour vocation l'ordonnancement dans le temps des actions d'un système.
- Dans un troisième temps, nous proposons un lien possible entre méthodes formelles et méthodes semi-formelles. Dans ce cadre, nous proposons une sémantique formelle pour les diagrammes `UML` s'appuyant sur RT-LOTOS, après avoir défini une extension temps-réel à `UML` (appelée `TURTLE`). Nous définissons ainsi une méthodologie qui s'inscrit dans les techniques de développement industriel classiques et qui permet une vérification formelle des systèmes temps-réel.

Plan du mémoire

Ce document est organisé en trois chapitres :

Chapitre 1 : Ce chapitre introduit la problématique de la description formelle des systèmes temps-réel. Il expose plus particulièrement le formalisme RT-LOTOS, et présente les diverses techniques de validation développées pour RT-LOTOS.

Chapitre 2 : Ce chapitre traite de la problématique de l'ordonnancement de systèmes temps-réel. Il présente la solution que nous avons élaborée, en donne la formalisation, les techniques et algorithmes à mettre en œuvre, et propose une étude de cas.

Chapitre 3 : Ce chapitre propose une méthodologie permettant de lier une approche formelle et approche semi-formelle. Il présente notre solution s'appuyant sur `UML` et RT-LOTOS et expose les techniques sous-jacentes que nous avons élaborées.

Certains algorithmes et détails d'implémentation sont décrits en annexe pour faciliter la lecture du mémoire.

1

Formalisation des systèmes temps-réel et RT-Lotos

Ce chapitre introduit différentes techniques de description formelle pour la spécification des systèmes temps-réel. La première section expose différentes techniques pour la formalisation de systèmes temporels, tels que les réseaux de Petri *temporisés, les automates temporisés, et les algèbres de processus. La seconde section expose le formalisme* RT-Lotos, *extension temporelle du formalisme* Lotos. *Finalement, la troisième section présente différentes techniques de validation développées et mises en œuvre pour* RT-Lotos.

1.1 Différents formalismes pour exprimer des contraintes temporelles

Un système *temps-réel* est un système dont le comportement doit satisfaire des contraintes temporelles strictes. Les systèmes temps-réel sont en général caractérisés par des interactions complexes avec leur environnement et par des contraintes temporelles dont la violation peut être critique. Dans un tel contexte, une approche formelle permet de définir les différents types de contraintes en les étayant d'une sémantique précise, de poser également les hypothèses de travail nécessaires comme par exemple la manière dont le temps progresse par rapport à l'exécution du système et les actions qu'il réalise, puis finalement, de vérifier a priori la conformité de l'ensemble des comportements possibles du système vis-à-vis de propriétés fixées a priori.

De nombreuses méthodes formelles existent pour modéliser les systèmes temps-réel [HD96]. Nous nous attacherons ici aux systèmes avec un domaine temporel dense. Dans ce contexte, nous retiendrons les extensions temporisées des réseaux de Petri, les automates

temporisés, et les algèbres de processus.

1.1.1 Extensions temporisées des réseaux de PETRI

Les réseaux de PETRI (notés *RdP*), du fait de leur sémantique du parallélisme et de leur représentation graphique, apparaissent comme particulièrement attractifs pour la modélisation d'architectures. Le modèle initial des réseaux de PETRI est atemporel. Plusieurs extensions ont été proposées pour prendre en compte le temps. Une comparaison détaillée est présentée dans [Boy01]. On y distingue principalement quatre grandes «familles» de réseaux :

Les réseaux de PETRI temporisés qui associent une durée aux transitions ou aux places.

- *RdP t-temporisés* (C. RAMCHANDANI) : le tir d'une *transition* n'est plus instantané (de durée nulle) mais nécessite un certain temps [Ram74].
- *RdP p-temporisés* (J. SIFAKIS) : un jeton déposé dans une *place* reste indisponible pendant un certain temps [Sif77].
- *RdP temporisés* (R.R. RAZOUK et C.V. PHELPS) : une transition doit rester sensibilisée pendant un certain temps avant de pouvoir être tirée, puis le tir commence et se poursuit durant un certain temps [RP84].

Les réseaux de PETRI temporels qui préfèrent la notion de délai (entre événements) à celle de durée (d'un état ou d'une action).

- *RdP temporels* (P. MERLIN) : un intervalle de temps $[a,b]$ est associé à chaque transition; si la transition est sensibilisée de façon continue, elle va être tirée entre a et b unités de temps [Mer74, MF76].
- *RdP à arcs temporels* (B. WALTER) : le temps porte sur les arcs incidents aux transitions; une transition ne peut être tirée que s'il existe, pour chaque place en entrée de la transition, un jeton dont l'âge est bien dans l'intervalle de l'arc reliant la place à la transition, mais rien ne force le tir de la transition lorsque la borne maximale de son intervalle temporel est atteinte [Wal83].
- *RdP p-temporels* (W. KHANSAS) : un intervalle temporel est associé aux places; un jeton dans une place annotée d'un intervalle $[a,b]$ ne peut pas quitter cette place avant d'y avoir passé au moins a unités de temps, et doit la quitter avant b unités de temps; si un jeton n'arrive pas à quitter une place avant b unités de temps, il devient un jeton mort [Kha97].
- *RdP statiquement temporisés* (A. CERONE et A. MAGGIOLO-SCHETTINI) : un modèle très général qui associe un intervalle temporel sur les places, les transitions et les arcs incidents aux transitions, et pourvu de deux sémantiques de tir *forte* (une transition est forcée lorsqu'elle atteint la borne maximum de son intervalle de tir) ou *faible* (un jeton peut rester indéfiniment dans une place) [CMS99].

Les systèmes à gardes algébriques qui utilisent des contraintes temporelles de la forme $(3 \leq x) \vee (x \leq 7)$ (où x représente une valeur d'horloge) plutôt que de simples

durées ou délais. Ce type de contraintes est proche des *gardes* que l'on trouve sur les automates temporisés (voir section 1.1.2). Les *réseaux RT* (H. Boucheneb), définis dans [Bou99], se présentent comme un modèle unificateur qui ajoute des conditions temporelles aux réseaux de Petri prédicat/transition de H.J. Genrich ([Gen89]).

Les systèmes à synchronisation, c'est-à-dire des modèles conçus pour modéliser un système comme la composition de sous-éléments en les «synchronisant». Les réseaux de Petri à *flux temporels* (RdPFT) (M. Diaz et P. Sénac), introduits pour la modélisation de scénarios multimédia [DS93], associent des durées aux places, des règles de synchronisation entre éléments, une condition temporelle sur les arcs, et différents modes possible de synchronisation entre flux. Une extension hiérarchique des RdPFT a été introduite [SDLdSS96, Sén96] pour permettre de modéliser des scénarios complexes de façon modulaire.

1.1.2 Automates temporisés

Les automates temporisés ont étés introduits par Rajeev Alur et David L. Dill [ACD90, AD91, AD94]. Un automate temporisé est un automate à états finis classique étendu par un ensemble de variables à valeur réelles appelées *horloges*. Ces horloges représentent les horloges des entités concurrentes d'un système. Elles sont supposées croître toutes à la même vitesse, et mesurer le temps passé depuis leur dernière remise à zéro. Les horloges peuvent êtres testées (comparées à des entiers naturels) et remises à zéro. Un *état* (ou *configuration*) du système est composé d'un état de contrôle de l'automate et d'un jeu de valeurs des horloges. Les transitions entre les états de contrôle de l'automate sont composées (i) d'un label appelé *action*, (ii) d'une condition temporelle de sensibilisation appelée *garde* définie sur les horloges du système qui doit être satisfaite pour autoriser le système à réaliser la transition, et (iii) d'un ensemble d'horloges qui doivent être remises à zéro. Les transitions sont *atomiques*, c'est-à-dire qu'elles se réalisent en un temps nul. Par contre, le temps peut s'écouler au sein des états de contrôle de l'automate.

De nombreux travaux ont proposé des extensions à ce modèle.

La notion d'*urgence* a été introduite :

- soit en distinguant des actions urgentes des actions non urgentes, comme dans l'algèbre de processus CCS [Mil89];
- soit en ajoutant à chaque état de contrôle de l'automate une condition temporelle appelée *invariant* définie sur les horloges du système qui doit rester satisfaite lors d'une progression du temps à l'intérieur de l'état de contrôle [HNSY94];
- soit en ajoutant à chaque transition une condition temporelle qui indique, lorsqu'elle est satisfaite, que le temps ne peut plus progresser (*Timed Automata with Deadlines* [BST97], *Dynamic Timed Automata* (`DTA`) section 1.3.2).

Dans un but d'optimisation, des travaux ont proposé de réduire le nombre d'horloges. Une première approche consiste à essayer de repérer les horloges non utilisées et les horloges qui sont toujours égales [DY96]. Une autre approche consiste à définir un ensemble

d'horloges non plus globales à tout le système mais locales à chacun des états de contrôle de l'automate. Des fonctions de recopie d'horloges sont alors ajoutées aux transitions. Le nombre d'horloges par état de contrôle est a priori inférieur ou égal au nombre d'horloges globales du système (voir également la définition du `DTA` dans la section 1.3.2).

Des travaux ont proposé une extension radicale du modèle en introduisant des horloges ne progressant plus nécessairement toutes au même rythme. Ces automates sont appelés automates *hybrides* [NSY93].

On trouvera dans [Pet99] un historique très clair des différents modèles, depuis les premiers travaux de ALUR et DILL jusqu'au modèle de l'outil UPPAAL [UPP].

i) Composition d'automates

Un système est divisé habituellement en parties, ou sous-systèmes. Dès lors, il peut être pratique de décrire la *composition* de sous-systèmes qui s'exécutent en parallèle et qui communiquent et se synchronisent entre eux.

Le modèle traditionnel des automates temporisés est basé sur un passage synchrone du temps pour tous les sous-systèmes et un *entrelacement* des actions atomiques de chaque sous-système. On parle alors de *temps universel* par opposition à une perception locale de l'écoulement du temps que pourrait avoir chacun des sous-systèmes (cette perception pouvant être soumise à des dérives, de la gigue, etc.)

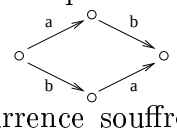
Dans le cas de la synchronisation d'automates, il est plus courant de définir un opérateur qui définit la synchronisation. Du fait de l'absence de parallélisme intrinsèque au modèle, la composition de deux automates crée un automate avec un nombre d'états de contrôle important (de l'ordre du produit du nombre des états de contrôle des automates composés), qui dissuade de les construire à la main. Ainsi, l'opérateur de synchronisation doit générer automatiquement le nouvel automate, fruit de la synchronisation des deux autres : ses états de contrôle, ses transitions, mais aussi ses contraintes temporelles. En pratique, la synchronisation est généralement faite à la volée (on ne construit pas le résultat explicitement). Par contre, la sémantique de la composition doit être effectivement définie.

Dans [ACD90], la composition parallèle d'automates temporisés est interprétée comme une conjonction logique similaire à l'opérateur de (multi-)synchronisation forte des algèbres de processus. Intuitivement cela signifie que pour être composés, deux systèmes doivent réaliser les mêmes actions aux mêmes dates. Cette restriction est trop forte pour être utilisée en pratique. Par conséquent, des travaux ont proposé d'autres sémantiques de composition d'automates temporisés. Certains auteurs, toujours en suivant le modèle d'algèbre de processus, proposent d'une part une fonction de relation de *complémentarité* entre action qui dit qu'une action a peut se synchroniser avec son action complémentaire $\bar{a}$ en donnant une action invisible τ, et d'autre part une restriction '\' telle que $A_1 \parallel A_2 \backslash_{\{a,b\}}$ oblige les automates A_1 et A_2 à se synchroniser sur les actions a,$\bar{a}$ et b,$\bar{b}$. Ce modèle de synchronisation deux à deux a obligé l'introduction de la notion de nœuds *commited*, c'est-à-dire (en simplifiant) des états de contrôle où on ne peut pas rester, pour garantir une atomicité lors de synchronisations entre plusieurs automates. Toutes ces variations sont présentées

de façon claire dans [Bur98]. Pour traiter la synchronisation entre automates, [LL98a] propose un cadre plus général que la synchronisation deux à deux : une fonction qui permet de synchroniser n automates et d'associer n'importe quel nom à cette synchronisation (au lieu de nécessairement la cacher en la transformant en une action invisible τ).

ii) Vrai parallélisme et entrelacement

La composition d'automates temporisés est basée sur une notion de parallélisme par entrelacement des actions. La sémantique de parallélisme par *entrelacement* s'oppose traditionnellement à la sémantique dite de *vrai parallélisme*.

Dans le cas d'un parallélisme par entrelacement, les actions sont nécessairement atomiques; typiquement, une activité qui consomme du temps est spécifiée par une action de début et une action de fin. L'occurrence de «deux actions parallèles a et b au même instant» sera représentée par un losange que l'on ne pourrait différencier de «a suivi de b ou b suivi de a», qui donnerait le même losange : 

Cependant, cette interprétation de la concurrence souffre de plusieurs inconvénients majeurs dès que l'on veut en particulier s'abstraire des hypothèses d'atomicité (temporelle et/ou spatiale) des actions. Pour résoudre ces problèmes, différentes sémantiques alternatives (connues globalement sous le nom de sémantiques de vrai parallélisme) ont été étudiées, comme, par exemple, les sémantiques de causalité [CdC93] et de maximalité [Saï96]. Ces sémantiques imposent un traitement mathématique plus complexe et n'ont à ce jour pas réellement percé pratiquement.

1.1.3 Algèbres de processus

Les algèbres de processus sont des langages abstraits conçus pour la spécification, la conception, et l'analyse fonctionnelle de systèmes concurrents. De nombreux modèles ont été proposés. Les modèles émergents sont LOTOS (*Language of Temporal Ordering Specification* [BB87]), CCS (*Calculus of Communicating Systems* de R. MILNER [Mil89]), CSP (*Communicating Sequential Process* de C.A.R. HOARE [Hoa89]), ACP (*Algebra of Communicating Processes* de J.A. BERGSTRA [BK85]), auxquels peuvent être ajoutés des modèles qui introduisent le temps, comme TCCS (*Temporal CCS* [MT90]), ainsi que les langages du type SPA (*Stochastic Process Algebras* de U. HERZOG [HR98]).

Dans les algèbres de processus, les systèmes sont modélisés par un ensemble de processus, des entités appelées *agents*, qui exécutent des *actions* atomiques. Ces actions sont les briques de base du langage, et des opérateurs sont utilisés pour décrire des comportements séquentiels qui peuvent s'exécuter de manière concurrente et se synchroniser (et communiquer) entre eux.

Avec CCS, deux agents communiquent lorsque l'un réalise une action, par exemple a, et que l'autre réalise l'action complémentaire $\bar{a}$. L'action résultant de la communication est distinguée par le label τ, qui représente une action *interne* invisible à l'environnement. Les agents peuvent réaliser leurs actions internes indépendamment les uns des autres, en

parallèle, avec une sémantique d'entrelacement. Les opérateurs du langage permettent de construire un agent pour lequel on spécifie une première action (préfixe); un choix entre plusieurs alternatives (choix); ou des activités concurrentes (composition).

Le mécanisme de communication de CSP diffère car il n'y a pas de notion de complémentarité des actions : deux agents communiquent en réalisant simultanément la même action (avec le même label). Lors de la communication, l'action conjointe reste visible à l'environnement, et peut être utilisée par d'autres processus; ainsi, plus de deux processus peuvent participer à une communication.

L'approche ACP se caractérise par une approche purement algébrique. Elle partage avec CSP la même notion d'équivalence (équivalence par bisimulation).

Les approches CCS, CSP et ACP ont pour objectif une analyse qualitative et non quantitative, et font ainsi abstraction du temps. Plusieurs travaux ont proposé d'incorporer le temps dans les algèbres de processus. Une vue d'ensemble de la problématique est proposée dans [NS91]; le lecteur trouvera également dans [Ver97] un exposé détaillé, précis et très complet des algèbres de processus, notamment celles intégrant le temps. Par exemple, TCCS étend CCS avec un délai fixe et une attente pour réaliser les synchronisations, en considérant que les actions consomment un temps nul (actions atomiques). Par opposition, les SPAs associent généralement une variable aléatoire représentant la durée de chaque action. Par contre, les SPAs adoptent le mécanisme de communication (multiple) de CSP.

LOTOS appartient également à la famille des algèbres de processus. Il s'agit d'un langage basé sur les concepts de CCS et CSP pour la description des aspects contrôle, et les types abstraits algébriques pour la description des données. LOTOS sera présenté plus en détail dans le paragraphe 1.2.1.

Les algèbres de processus apparaissent comme des modèles de description formelle et de spécification «orientés contraintes» [VSvSB91, Led94] qui semblent bien adaptés à l'intuition et à la compréhension humaine. Ajoutée à cela la faculté d'automatiser le processus de vérification, ces modèles connaissent un certain succès.

1.1.4 Principales extensions temporelles à LOTOS

LOTOS [Lot88] est une algèbre de processus permettant d'exprimer la structure logique et temporelle de comportements. Ce modèle formel sera détaillé dans la sous-section 1.2.1.

LOTOS appartient à la famille des *algèbres de processus*; il s'appuie sur le langage CCS de MILNER (étendu par un mécanisme de synchronisation multipoint hérité de CSP de HOARE) pour la spécification de la partie comportementale ; la partie description des structures de données est inspirée de Act-One, un formalisme de description de types de données abstraits algébrique. LOTOS est un langage asynchrone, avec synchronisation par rendez-vous multiples. Les processus offrent des synchronisations à leur environnement au travers de *portes* de communication (ou *actions*).

La problématique de l'expression explicite du temps dans LOTOS a engendré une série

d'approches différentes. Nous listons les principaux modèles ci-dessous :

- TIC-LOTOS (QUEMADA 1987, UPM - Espagne)
- LOTOS-T (MIGUEL 1992, UPM - Espagne)
- T-LOTOS et U-LOTOS (BOLOGNESI 1991, CNUCE - Italie)
- TLOTOS (LEDUC 1992, ULG - Belgique)
- Time LOTOS et ET-LOTOS (LEDUC, LÉONARD 1993-94, ULG - Belgique)
- TE-LOTOS (LEDUC, LÉONARD, QUEMADA, MIGUEL et all., 1995)
- LOTOS/T (NAKATA 1993, ES-Osaka - Japon)
- RT-LOTOS (COURTIAT 1993, LAAS - France)
- E-LOTOS (ISO/IEC 15437:2001)

Certaines de ces approches ont servi de prémices et de base de réflexion aux approches qui ont suivi. Certains concepts émergents semblent poindre à travers l'ensemble de ces travaux. Pour différencier ces approches, nous considérerons les aspects suivants :

- Choix du domaine temporel :
  1. Soit *discret* : les grandeurs temporelles sont définies sur les entiers naturels ($\mathbb{Z}$), une progression de une unité de temps peut alors être transcrite par l'occurrence d'une action spécifique (fréquemment notée `tic`). Ceci facilite la définition formelle, car le modèle se rapproche du modèle non temporisé, mais occasionne un fort risque d'explosion combinatoire.
  2. Soit *dense* et *dénombrable* : les grandeurs temporelles sont définies sur les rationnels positifs ($\mathbb{Q}_+^{0,\infty}$). Cela semble parfaitement convenir à la grande majorité des cas pratiques d'utilisation d'un langage de spécification formelle de systèmes temps-réel. Le modèle mathématique sous-jacent est, par contre, plus complexe à définir et à mettre en œuvre dans le cadre de la vérification.
  3. Soit *réel* : quelques rares modèles évoquent la possibilité de définir les grandeurs temporelles dans $\mathbb{R}_+^{0,\infty}$. Notons qu'alors, à notre connaissance, aucune technique de validation n'a été proposée.
- Temporisation des actions : elle se fait soit par l'ajout d'un opérateur temporel dédié («*opérateur*$\langle\cdots\rangle\ P$»), soit par une extension de l'opérateur de préfixe (préfixe d'un processus par une action : «$g\ \langle\cdots\rangle$ `;` P»), soit les deux.
- Hypothèses d'urgence des actions : une action est dite *urgente* lorsqu'elle *doit* se réaliser immédiatement, sans progression possible du temps, dès qu'elle est sensibilisée. Certains modèles présupposent implicitement que toutes les actions sont urgentes. La plupart associent l'urgence aux actions internes (l'action `i` ou une action intériorisée par `hide`). Certains modèles introduisent un mot clef spécifique pour déclarer urgentes des actions internes ou observables. D'autres, enfin, proposent des mécanismes pour relâcher l'urgence des actions internes sous certaines conditions.
- Opérateurs additionnels : certains langages proposent des facilités d'écriture pour spécifier des comportements réputés classiques par le biais d'opérateurs de haut niveau, qui toutefois n'introduisent pas véritablement de fonctionnalités nouvelles dans

le modèle (en d'autre terme, les comportements spécifiés pas ces opérateurs de haut niveau peuvent également être spécifiés au moyen d'opérateurs du modèle de base, mais de manière plus lourde).

	Domaine temporel	Temporisation	Urgence
TIC-LOTOS	discret	préfix et opérateur	toute action
LOTOS-T	discret ou dense	prefix	actions cachées
T-LOTOS	discret	opérateur	mot clef spécifique
U-LOTOS	discret	opérateur	mot clef spécifique
TLOTOS	discret	opérateur	relâchée
Time LOTOS	dense	opérateur	actions cachées
ET-LOTOS	dense	préfix et opérateur	actions cachées
TE-LOTOS	dense	préfix et opérateur	toute action
LOTOS/T	discret	préfix	mot clef spécifique
RT-LOTOS	dense	préfix et opérateur	actions cachées
E-LOTOS	dense	préfix et opérateur	actions cachées

FIG. 1.1 – *Comparaison des extensions temporelles à* LOTOS

La figure 1.1 reprend dans les grandes lignes quelques points de comparaison entre les différentes extensions temporelles à LOTOS listées ici.

i) TImed Calculus (QUEMADA)

Dans [QF87, QAdF89, QdFA93] les auteurs proposent une algèbre de processus appelée TImed Calculus (TIC), basée sur un domaine temporel discret. Le temps est introduit sur l'opérateur de préfixe : «at; P», où $t \in \mathbb{Z}$ et P est un processus TIC, désigne un processus qui réalise l'action a (implicitement urgente) après t unités de temps depuis la précédente action, puis se comporte comme P. Une extension appelée «opérateur de choix temporel» est disponible : «aT; P», où T désigne un intervalle de $\mathbb{Z}$, désigne un processus qui réalise l'action a après un délai choisi dans T. Le modèle propose également d'un opérateur «`Age(`t`,`P`)`» qui permet de consommer t unités de temps sur la première action de P.

Notons que cette approche associe les contraintes temporelles entre événements successifs dans une expression de préfixage. C'est pourquoi, il est impossible de faire une spécification de conditions temporelles pour des événements non successifs sans utiliser d'événements intermédiaires ou processus avec contraintes additionnelles [vHTZ89]. Cette approche ne permet pas non plus d'exprimer des contraintes temporelles globales et pour cela, lors d'un événement de synchronisation, le délai associé à une porte est défini par les contraintes temporelles locales des processus qui y participent [MBC+94].

ii) LOTOS-T (MIGUEL)

Dans [MFV92] les auteurs proposent une algèbre de processus appelée LOTOS-T, basée sur un domaine temporel qui doit être spécifié au moyen d'un type abstrait (comprenant au moins $>$, 0, et $+$), et qui donc, peut être discret ou dense. Ce modèle s'appuie sur le modèle TIC. Le temps est introduit par l'opérateur de préfixe : «a`{`t`};` P», où t est du temps et P est un processus LOTOS-T, qui désigne un processus réalisant l'action a après t unités de temps depuis la précédente action, puis se comporte comme P. L'urgence n'est plus implicitement systématique, mais devient attachée aux actions internes uniquement. Notons qu'il n'est plus possible de spécifier un intervalle temporel : le non-déterminisme temporel étant obtenu comme un cas particulier du non-déterminisme de l'opérateur de choix sur une variable libre de type temps :

```
choice t: nat [] [t] -> a{t}; B
```

ce qui, en pratique, réduit le domaine temporel à un domaine énumérable par l'opérateur de choix, donc vraisemblablement, non dense.

Notons par ailleurs que les auteurs proposent un modèle étendu, appelé LOTOS-TP, qui permet de spécifier également des comportements probabilisés.

iii) U-LOTOS/ T-LOTOS (BOLOGNESI)

Les auteurs [BL91a, BL91b] proposent deux algèbres de processus appelées U-LOTOS et T-LOTOS, basées sur un domaine temporel discret. Ces approches sont inspirées des réseaux de PETRI temporels de type MERLIN [Mer74, MF76] et de TCCS [MT90]

U-LOTOS introduit le temps par un nouvel opérateur de préfixe : «$(t)\,.\,P$» est le processus qui laisse s'écouler t unités de temps puis se transforme en P. L'urgence est spécifiée par l'opérateur `asap` (*as soon as possible*) : «`asap` G `in` P» rend urgente les portes de l'ensemble G dans P.

T-LOTOS propose également la construction «$(t)\,.\,P$», et généralise l'opérateur d'urgence des actions : «`timer` a`(`$t1$`,` $t2$`)` `in` P» est le processus qui se comporte comme P excepté que l'action a doit être exécutée dans l'intervalle temporel [$t1$,$t2$] dès qu'elle est sensibilisée. Le processus U-LOTOS «`asap a in P`» est équivalent au processus T-LOTOS «`timer a(0,0) in P`».

Le problème principal de la proposition T-LOTOS est relatif à l'occurrence d'une action observable. L'exécution d'une action interne peut être contrôlée par le processus lui-même, mais l'occurrence d'un événement observable (non interne) correspond à une *proposition d'interaction* faite à l'environnement, dont la participation à la synchronisation est requise pour l'occurrence de l'action. Pourtant, les opérateurs d'urgence forcent l'occurrence des actions observables à la fin de l'intervalle temporel, ce qui nous parait contre-intuitif.

iv) TLOTOS (LEDUC)

L'auteur [Led91] propose une algèbre de processus appelée TLOTOS, définie sur un domaine temporel discret, inspirée de ATP (*Algebra of Timed Processes* [NS94]).

Soit P et Q deux processus TLOTOS, et $d \in \mathbb{Z}$; le temps est introduit par trois opérateurs. L'opérateur *start delay* (ou opérateur de time-out), noté $\lfloor P \rfloor^d(Q)$, désigne le processus qui se comporte comme P à condition que P débute avant d unités de temps, ou autrement qui se comporte comme Q. L'opérateur *execution delay* (ou opérateur de watchdog), noté $\lceil P \rceil^d(Q)$, désigne le processus qui se comporte comme le processus P avant la $d^{ème}$ unités de temps, et comme Q ensuite. L'urgence implicite des actions est relâchée, localement à un processus, par l'opérateur *unbounded start delay*, noté $\lfloor P \rfloor^\omega$, et qui désigne le processus qui se comporte comme P excepté qu'il peut attendre un temps arbitrairement long avant de débuter. Lors de compositions, l'urgence ou non des actions est attachée à la notion d'actions cachées ou observables. La sémantique de ces trois opérateurs est identique à leurs correspondants du modèle ATP.

v) Timed LOTOS/ ET-LOTOS (LEDUC, LÉONARD)

Les auteurs de [LL92, LL93, LL97] proposent deux algèbres de processus appelées Timed LOTOS et ET-LOTOS, définies sur un domaine temporel dense.

Timed LOTOS [LL92] introduit le temps au moyen d'un opérateur de délai, noté $\Delta^{[d^-,d^+]}P$, qui désigne le processus qui laisse passer un temps choisi dans l'intervalle $[d^-,d^+]$, puis se comporte comme P (avec éventuellement $d^+ = \omega$ pour spécifier un intervalle non borné). Le modèle apporte également deux opérateurs qui s'appliquent aux portes d'un processus. L'opérateur *delay on interactions*, noté $\Delta^I_g P$ (où I est un intervalle temporel, g une porte (d'interactions), et P un processus), introduit un délai non-déterministe choisi dans I sur toutes les actions g de P. L'opérateur de *time-out*, noté $\lfloor P \rfloor^d_g(Q)$, spécifie que si le processus P propose une interaction sur la porte g sans succès pendant une période d, alors P est interrompu et remplacé par Q.

ET-LOTOS (pour Enhanced Timed LOTOS) [LL93, LL97] étend le modèle Timed LOTOS au niveau de l'opérateur de préfixe d'action par une offre limitée dans le temps et un mécanisme de capture de date : «g `@`t `{`d`};` P» spécifie le processus qui offre une interaction sur la porte g pendant au plus d unités de temps, enregistre le temps écoulé entre l'offre et l'occurrence de g dans la variable t, puis se comporte comme le processus P. Si, à l'instant d, l'action proposée ne s'est pas produite, alors ce processus devient le processus `stop`. L'urgence est attachée à la notion d'action cachée : une action interne est potentiellement urgente. Notons que l'offre limitée dans le temps peut s'appliquer à l'action interne `i`, mais que sa signification est tout autre : `i{`d`}` introduit un délai non-déterministe choisi dans $[0,d]$ avant de réaliser l'action interne `i`. L'opérateur de délai de Timed LOTOS $\Delta^{[d^-,d^+]}$ permettait de spécifier un délai non-déterministe choisi dans l'intervalle. L'opérateur de délai de ET-LOTOS Δ^d propose uniquement un délai déterministe, le non-déterminisme temporel étant introduit par ailleurs, au moyen de `i{`d`}`. Cela introduit une restriction du pouvoir d'expression du modèle qui sera expliqué dans un paragraphe suivant, lors d'une comparaison avec RT-LOTOS. Les opérateurs $\Delta^I_g P$ et $\lfloor P \rfloor^d_g(Q)$ sont également présents, mais apparaissent désormais comme des opérateurs de haut niveau pouvant être ré-écrits avec les opérateurs de bas niveau de ET-LOTOS.

Le formalisme ET-LOTOS a suscité de nombreux autres travaux. Dans [BDS95] on

peut trouver une sémantique dénotationnelle pour ET-LOTOS et dans [BD97, BK97] une proposition de sémantique de vrai parallélisme causal.

vi) TE-LOTOS (LEDUC, LÉONARD, QUEMADA, MIGUEL et al.)

Dans [LLdF+95], les auteurs proposent une unification de plusieurs travaux autour de LOTOS et du temps, appelée TE-LOTOS (pour *Time Extended* LOTOS), dans le cadre des activités de normalisation d'une version étendue de LOTOS.

Le modèle est défini sur un domaine temporel dense. Il propose un opérateur de retard déterministe, noté «`Wait(`d`);`P», qui retarde le processus P de d unités de temps. L'opérateur de préfixe d'action est étendu : «g`{`t `in` $d^-\ldots d^+$`}[`SP`];` P». Le terme SP est une expression booléenne ou une équation, et est appelé prédicat de sélection, qui conditionne l'occurrence de g. Les auteurs précisent que «t `in`» peut être omis ainsi que «`in` $d^-\ldots d^+$». Dans ce cas, $d^- = 0$, $d^+ = \infty$ si $g \neq$`i` et $d^+ = 0$ si $g =$`i`. De même, `[`SP`]` peut être omis et, dans ce cas, `[`SP`]=[true]`. La sémantique est la suivante : t est une variable temporelle libre permettant d'enregistrer le temps entre l'offre et l'occurrence de l'action g (cela remplace donc l'opérateur `@`). L'intervalle temporel $d^-\ldots d^+$ spécifie que l'occurrence de l'action g n'est possible qu'après d^- unités de temps et qu'avant d^+ unités de temps. Si, après d^+ unités de temps, l'action ne s'est pas produite, alors le processus se transforme en `stop` dans le cas où $g \neq$`i` : si $g =$`i` le temps ne peut plus progresser (ce comportement d'urgence est étendu aux actions cachées). Les auteurs proposent un modèle de bisimulation forte, ainsi que la dérivation de spécifications TE-LOTOS en un système de transitions étiquetées LTS (*Labelled Transition System*), qui permet l'exécution et la vérification formelle.

Une sémantique dénotationnelle, basée sur une notion d'observation temporisée, a été proposée dans [DBS95], sur un sous-ensemble (appelé *Timed Enhanced* LOTOS) de TE-LOTOS : l'opérateur de préfixe est alors restreint à «a`{`t `in` $d^-\ldots d^+$`};` P».

vii) LOTOS/T (NAKATA)

Les auteurs [NHT93, Nak97] proposent une algèbre de processus appelée LOTOS/T, basée sur un domaine temporel discret.

Les contraintes temporelles sont introduites au niveau de l'opérateur de préfixe : «a`[`$f(t,x)$`];` P», où t est une variable temporelle libre représentant le temps absolu, x est un vecteur de toutes les variables du système excepté t, et $f(t,x)$ est un prédicat de la logique du premier ordre défini sur un sous-ensemble de l'arithmétique de PRESBURGER, et P un processus LOTOS/T. L'action a est assujettie au prédicat P qui définit les conditions et les dates de son occurrence. Considérons par exemple :

$$\texttt{a[}2 \leqslant t \leqslant 3 \wedge x_0 = t\texttt{]; b[}t = x_0 + 3\texttt{]; stop}$$

Ce processus réalise l'action `a` entre les dates 2 et 3, puis réalise l'action `b` 3 unités de temps plus tard. L'assignation $x_0 = t$ permet d'enregistrer la date d'occurrence de `a`. L'opérateur «`asap` A `in` P» permet de spécifier comme urgentes les actions de l'ensemble A au sein du processus P.

Les auteurs proposent un modèle de bisimulation forte et faible pour les spécifications LOTOS/T, ainsi que la dérivation en un LTS.

Ce modèle permet de spécifier des contraintes très souples, mais laisse planer un doute sur la faisabilité d'une vérification, le doute portant sur la décidabilité et le risque d'explosion des LTS sous-jacents.

viii) RT-LOTOS (COURTIAT)

Les auteurs [CCS93, CSLO00] proposent une algèbre de processus appelée RT-LOTOS (pour *Real-Time* LOTOS), basée sur un domaine temporel dense.

Le temps est introduit par deux opérateurs de délai déterministe «`delay(`d`)` P» et non-déterministe «`latency(`l`)` P» qui spécifient un processus qui attend un délai d, ou choisit un délai aléatoire dans l'intervalle [0,l], avant de devenir le processus P. L'urgence est attachée à la notion d'action cachée : une action interne est potentiellement urgente. L'opérateur de latence a justement pour objet de relâcher dans un intervalle temporel le caractère urgent d'une action interne. L'opérateur de préfixe d'action est étendu : «g `@`x `{t};` P» désigne le processus qui offre une interaction sur la porte g pendant au plus t unités de temps, et enregistre le délai séparant l'offre de l'occurrence de g dans la variable temporelle libre x. Si l'action g ne s'est pas produite au bout de t unités de temps, alors si g est une action observable, le processus se transforme en `stop`, ou bien si g est une action cachée, le temps ne progresse plus dès que cette action est sensibilisée.

Le modèle RT-LOTOS qui sous-tend les travaux de recherche exposés dans ce document sera détaillé dans la section 1.2.2.

ix) E-LOTOS (ISO/IEC 15437:2001)

Les derniers travaux de normalisation ont abouti à la norme E-LOTOS (pour *Enhancements to* LOTOS) [ELo01]. Ce modèle étend la norme LOTOS [Lot88]. E-LOTOS apporte des facilités de programmation en terme de manipulation de données, de structuration en fonctions, d'utilisation d'exceptions. Du point de vue temporel, E-LOTOS apporte un opérateur de délai déterministe «`wait(`d`);`P», et un opérateur de préfixe d'action : «g`(`D`) @`T `[`SP`]` `;` P» spécifie le processus qui propose une synchronisation sur la porte g lorsque SP est vrai, échange les valeurs D de manière identique à Full-LOTOS, enregistre dans une variable (par exemple T =`'?t'`) le temps séparant l'offre et l'occurrence de l'action, ou l'impose (par exemple T =`'!5'`), puis se comporte comme le processus P. L'urgence est rattachée à la notion d'action cachée. E-LOTOS permet également l'urgence sur les actions observables à conditions qu'elles soient définies comme *exception*.

La combinaison de ces fonctionnalités permet de spécifier un intervalle temporel pour offrir une action (par exemple «`g @?t [1<t<4]`»). Il est également possible d'introduire du non-déterminisme temporel en combinant l'opérateur d'assignation non-déterministe de variable et l'opérateur de délai déterministe mais employé sur cette même variable (par exemple «`var t : time in ?t := any time [1<t<4]; wait(t) endvar;` P»). Le non-déterminisme temporel de E-LOTOS s'inscrit dans une démarche orientée *données*, c'est-à-

dire qu'un temps non-déterministe est vu comme n'importe quelle donnée non-déterministe. Il se démarque, sur ce point, du non-déterminisme temporel de RT-LOTOS par exemple, qui s'inscrit dans une démarche orientée *contrôle*, où la variable temporelle non-déterministe est une variable particulière introduite par un opérateur spécifique. Notons également que la combinaison de l'offre limitée dans le temps et le délai non déterministe diffère entre E-LOTOS et RT-LOTOS : dans l'expression RT-LOTOS «`latency(`l`)` g`{`t`}`» l'intervalle de latence et l'intervalle d'offre limitée dans le temps débutent au même instant, ce qui permet d'exprimer des violations temporelles potentielles dans le cas où $t < l$. Par contre, dans une construction E-LOTOS similaire, la contrainte d'offre d'une action dans un intervalle temporel n'entrera en vigueur qu'après que le délai non-déterministe se soit réalisé, et limite ainsi la spécification de violation temporelle potentielle.

1.2 Le langage RT-LOTOS

RT-LOTOS [CSLO00], extension temporelle à LOTOS [Lot88], est une algèbre de processus permettant d'exprimer des comportements contraints par le temps tout en héritant des caractéristiques de l'approche LOTOS (expression de la concurrence et de la synchronisation, choix du niveau d'abstraction, compositionnalité, ...). Cette technique de description formelle permet ainsi, au moyen de nouveaux opérateurs, d'exprimer les contraintes temporelles que les actions d'une spécification RT-LOTOS doivent respecter. Ces contraintes temporelles consistent essentiellement à retarder l'occurrence des actions (opérateur de délai), à exprimer le non-déterminisme temporel (opérateur de latence) et à limiter le temps pendant lequel des actions peuvent êtres offertes à leur environnement.

1.2.1 LOTOS

LOTOS (abréviation de *Language of Temporal Ordering Specification* [BB87]) est une technique de description formelle, promue au rang de norme ISO en 1988 [Lot88].

En s'appuyant sur les caractéristiques de base du langage, nous proposons ici une introduction brève à LOTOS. La sémantique formelle complète des opérateurs LOTOS est décrite en détail dans [BB87] et [Lot88], et ne sera donc pas rappelée dans ce mémoire. Dans [LFHH92], LOTOS est introduit par le biais d'une suite d'exemples de spécification très simples qui illustrent les caractéristiques les plus intéressantes du langage.

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

En ce sens, LOTOS implémente un paradigme de «boîte noire» rendant possible le

développement de spécifications abstraites et concises de systèmes complexes. À un certain niveau d'abstraction, il est possible d'exprimer les interactions d'un processus avec son environnement sans avoir à décrire la structure interne (ou implémentation) de ce processus.

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

i) Présentation informelle de Basic-LOTOS

Nous appelons *Basic*-LOTOS le sous-ensemble de LOTOS où les processus interagissent entre eux par synchronisation pure, sans échange de valeur. En Basic-LOTOS les actions sont identiques aux portes de synchronisation des processus. Les principaux opérateurs de Basic-LOTOS sont listés dans la figure 1.2.

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

FIG. 1.2 – *Principaux opérateurs de Basic*-LOTOS

Afin de donner une approche intuitive des opérateurs LOTOS, nous associons une assertion informelle à chaque configuration de la figure 1.3. Dans chaque schéma un processus $P[a,b,c,d]$ est composé de deux processus $P1[a,b]$ et $P2[c,d]$ (boites grisées) mis en relation avec différents opérateurs LOTOS et offrant chacun une synchronisation sur les portes a,b,c

ou d à leur environnement :

1. «$P1[a,b]$ ou $P2[c,d]$ suivant l'action qui se produira en premier» (choix)
2. «$P1[a,b]$ et $P2[c,d]$ s'exécutent en parallèle sans synchronisation» (parallélisme)
3. «$P1[a,b]$ et $P2[b,c]$ s'exécutent en parallèle sans synchronisation sauf pour la porte b sur laquelle les deux processus doivent se synchroniser» (synchronisation)
4. «$P1[a,b]$ et $P2[b,c]$ s'exécutent en parallèle sans synchronisation sauf pour la porte b sur laquelle les deux processus doivent se synchroniser; de plus b, n'est plus disponible pour une synchronisation potentielle avec les processus appartenant à l'environnement du processus P» (intériorisation)
5. «D'abord $P1[a,b]$ suivi de $P2[c,d]$ quand $P1$ sera terminé» (séquence)
6. «$P1[a,b]$ peut être interrompu à tout instant par $P2[c,d]$ avant sa terminaison» (préemption)

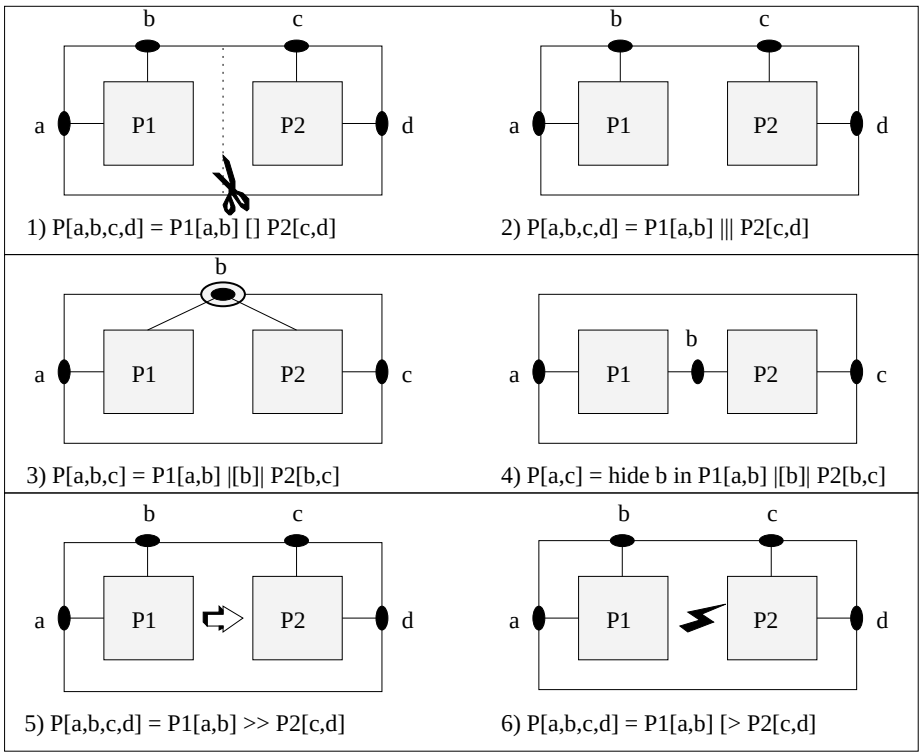


FIG. 1.3 – *Illustration des opérateurs de Basic*-LOTOS

ii) Syntaxe formelle de Basic-LOTOS

- Soit $\mathcal{P}$ l'ensemble des identifiants de processus
- Soit $X \in \mathcal{P}$.
- Soit $\mathcal{G}$ l'ensemble des portes définissables (i.e. les actions observables en Basic-LOTOS).

- Soient g, $g_1 \ldots g_n \in \mathcal{G}$.
- Soit L un sous-ensemble (pouvant être vide) quelconque de $\mathcal{G}$ noté $L = g_1, \ldots g_n$.
- Soit i l'action interne.

La syntaxe formelle de Basic-LOTOS est donnée dans la figure 1.4; la syntaxe de la déclaration d'un processus étant :

$$\textbf{process } X[g_1, \ldots g_n] := P \textbf{ endproc}$$

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

FIG. 1.4 – *Syntaxe de Basic-*LOTOS

Notons que ni l'opérateur `|||` ni l'opérateur `||` n'apparaissent explicitement dans la syntaxe. Ceux-ci correspondent à une notation simplifiée de l'opérateur de composition parallèle (`|||` = `|[`$\emptyset$`]|` et `||` = `|[`$\mathcal{G}$`]|`, $\mathcal{G}$ désignant l'ensemble des portes visibles des deux processus composés).

iii) Sémantique opérationnelle de Basic-LOTOS

La figure 1.5 présente les règles d'inférence de la sémantique opérationnelle de Basic-LOTOS. La sémantique opérationnelle est présentée dans le style SOS (*Structured Operational Semantics*) de PLOTKIN. Notons que les règles symétriques pour l'opérateur de choix et l'opérateur parallèle ont été omises pour des raisons de clarté et que les notations suivantes seront utilisées :

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

Ces règles sont les règles d'inférence standards de la sémantique opérationnelle de Basic-LOTOS. Des explications détaillées peuvent être trouvées dans [BB87].

iv) Présentation informelle de (Full) LOTOS

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

Les structures de données et les opérations associées sont définies au moyen du langage Act-One qui formalise la spécification de types abstraits de données : il définit les propriétés

$exit \xrightarrow{\delta} stop$
$g; P \xrightarrow{g} P \quad (g \in \mathcal{G})$
$i; P \xrightarrow{i} P$
$\dfrac{P \xrightarrow{g} P'}{P[]Q \xrightarrow{g} P'} \quad (g \in \mathcal{G}^{i,\delta})$
$\dfrac{P \xrightarrow{g} P' \qquad Q \xrightarrow{g} Q'}{P\|[L]\|Q \xrightarrow{g} P'\|[L]\|Q'} \quad (g \in L \cup \{\delta\})$ $\dfrac{P \xrightarrow{g} P'}{P\|[L]\|Q \xrightarrow{g} P'\|[L]\|Q} \quad (g \in \mathcal{G}^i \setminus L)$
$\dfrac{P \xrightarrow{g} P' \qquad (g \in \mathcal{G}^{i,\delta} \setminus L)}{hide\ L\ in\ P \xrightarrow{g} hide\ L\ in\ P'}$ $\dfrac{P \xrightarrow{g} P' \qquad (g \in L)}{hide\ L\ in\ P \xrightarrow{i} hide\ L\ in\ P'}$
$\dfrac{P \xrightarrow{g} P'}{P >> Q \xrightarrow{g} P' >> Q} \quad (g \in \mathcal{G}^i)$ $\dfrac{P \xrightarrow{\delta} P'}{P >> Q \xrightarrow{i} Q}$
$\dfrac{P \xrightarrow{g} P'}{P[> Q \xrightarrow{g} P'[> Q} \quad (g \in \mathcal{G}^i)$ $\dfrac{Q \xrightarrow{g} Q'}{P[> Q \xrightarrow{g} Q'} \quad (g \in \mathcal{G}^{i,\delta})$ $\dfrac{P \xrightarrow{\delta} P'}{P[> Q \xrightarrow{\delta} Q}$
$\dfrac{P_X[g_1/g'_1 \cdots g_n/g'_n] \xrightarrow{g} Q' \qquad X[g'_1 \cdots g'_n] := P_X}{X[g_1 \cdots g_n] \xrightarrow{g} Q'} \quad (g \in \mathcal{G}^{i,\delta})$ $\dfrac{P \xrightarrow{g} P'}{P\phi \xrightarrow{\phi(g)} P'\phi} \quad \begin{array}{l}(\phi = [g_1/g'_1 \cdots g_n/g'_n]) \\ (g \in \mathcal{G}^{i,\delta})\end{array}$

FIG. 1.5 – *Sémantique opérationnelle de Basic*-LOTOS

essentielles des données et les opérations qu'une implémentation correcte du type doit assurer [BB87].

Un type abstrait [EM85] est une structure qui n'est pas définie en terme d'implantation en mémoire ou par une explicitation de ses composantes, mais plutôt en termes d'opérations et de propriétés sémantiques. La spécification d'un type abstrait est indépendante de toute représentation de la (ou des) structure(s) en machine et de l'implantation des opérations associées. La spécification (*signature*) d'un type de donnée en LOTOS comporte deux parties : la structure logique et les opérations.

La spécification de la structure logique (*sort*) décrit une instance du type abstrait, les relations qui peuvent exister entre les éventuelles composantes de cette instance et les assertions invariantes qui décrivent les restrictions à apporter à ces relations.

La spécification des opérations (*opns*) décrit la syntaxe et la sémantique des primitives de manipulation du type abstrait en précisant l'interface de ces opérations.

Les valeurs en LOTOS peuvent être :

- échangées entre processus lors d'une synchronisation ;
- employées dans les prédicats de garde des processus;
- utilisées comme paramètres pour la définition des processus et pour l'instantiation de variables;
- associées à l'opérateur de «choix généralisé»;
- exportées lors d'une terminaison avec succès d'un processus.

Avec l'ajout des données dans les spécifications, les actions deviennent des entités qui adjoignent à une porte trois composantes de base, comme cela est illustré dans la figure 1.6 [LFHH92]. Une action peut :

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

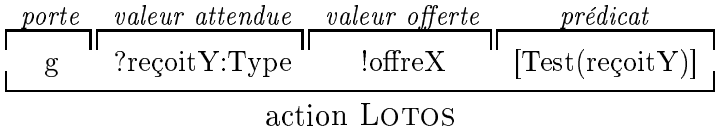


FIG. 1.6 – *Représentation d'une action en* LOTOS

L'usage des données implique quelques changements concernant l'interprétation des règles pour la synchronisation entre processus. La figure 1.7 illustre ces nouvelles règles en prenant l'exemple de deux processus P_1 et P_2 composés par une synchronisation sur la porte g (nous faisons l'hypothèse d'une fonction *valeur* qui renvoie la valeur d'une variable). La règle générale veut qu'après une synchronisation, les actions impliquées dans cette synchronisation doivent offrir des variables de *même type* et de *même valeur*.

P_1	P_2	Condition de synchronisation	Type d'interaction	Résultat
$g!v_1$	$g!v_2$	$valeur(v_1) = valeur(v_2)$	concordance des valeurs	synchronisation
$g!v$	$g?x:T$	$v \in T$	passage de valeur	après synchronisation, $x = valeur(v)$
$g?x:T$	$g?y:U$	$T = U$	choix aléatoire d'une valeur	après synchronisation, $x = y = v$ avec $v \in T$ (ou $v \in U$)

FIG. 1.7 – *Règles de synchronisation en* LOTOS *avec données*

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

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

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

La figure 1.8 liste les principaux opérateurs introduits dans LOTOS pour manipuler les données.[1] Dans ce tableau, x correspond à un nom de variable, $Type$ à un type de donnée et *pred* à une expression logique (prédicat de garde).

Opérateur	Notation	Description informelle
composition séquentielle	`exit(`$v_1, \ldots v_n$`)` `>>` `accept` $x_1 : T_1, \ldots x_n : T_n$ `in` P	Le processus qui a terminé avec succès transmet des valeurs $v_1, \ldots v_n$ au processus suivant qui peut les consulter à travers les variables $x_1 : T_1, \ldots x_n : T_n$ respectivement.
choix généralisé	`choice` $x : Type$ `[]` $P(x)$	Si P dépend des variables x, l'opérateur offre le choix entre les processus pour toutes les valeurs possibles de $x \in Type$.
déclaration de variable	`let` $x : Type = v$ `in` P	Réalise l'instanciation de la variable x avec la valeur v dans P.
prédicat de garde	`[`*pred*`] ->` P	Le processus aura le comportement de P si *pred* est vrai, sinon il devient `stop`.

FIG. 1.8 – *Principaux opérateurs* LOTOS *avec utilisation de données*

Finalement, les opérateurs ont l'ordre de priorité décroissante ci-dessous:

`;` `->` `[]` `|[`L`]|` `[>` `>>` `hide` `choice` `let`

1. Les deux premières constructions LOTOS ne sont pas supportés par l'outil `rtl` disponible pour la simulation et la vérification des spécifications RT-LOTOS, mais des constructions équivalentes sont utilisables.

v) Sémantique opérationnelle de (Full) LOTOS

La figure 1.9 présente de quelle manière la sémantique de Basic LOTOS est enrichie pour donner la sémantique opérationnelle de (Full) LOTOS. La plupart des règles peuvent être déduites facilement des règles sémantiques de Basic LOTOS, en remplaçant l'offre des portes de Basic LOTOS par celle de l'offre générale des portes, laquelle prend en compte l'échange de données.

$exit(v_1 \cdots v_n) \xrightarrow{\delta!<valeur(v_1)\cdots valeur(v_n)>} stop$
$g!v;P \xrightarrow{g!<valeur(v)>} P \quad (g \in \mathcal{G})$ $g?y:T;P \xrightarrow{g!<p>} let\ y:T=p\ in\ P \quad (g \in \mathcal{G})(p \in T)$
$\dfrac{P \xrightarrow{g} P'}{let\ x:T=v\ in\ P \xrightarrow{g} let\ x:T=v\ in\ P'} \qquad \dfrac{[v/x]\ P \longrightarrow P'}{let\ x:T=v\ in\ P \longrightarrow P'}$
$\dfrac{P \xrightarrow{g} P' \quad valeur(v)=true}{[v] \text{->} P \xrightarrow{g} P'}$
$\dfrac{P_X[g_1/g'_1 \cdots g_n/g'_n](valeur(v_1)/y_1 \cdots valeur(v_n)/y_n) \xrightarrow{g} Q' \qquad X[g'_1 \cdots g'_n](y_1:T_1 \cdots y_n:T_n) := P_X}{X[g_1 \cdots g_n](valeur(v_1) \cdots valeur(v_n)) \xrightarrow{g} Q'} \quad (g \in \mathcal{G}^{i,\delta})$

FIG. 1.9 – *Sémantique opérationnelle de (Full)* LOTOS

1.2.2 L'extension RT-LOTOS

En utilisant LOTOS, seul l'aspect *qualitatif* de l'ordonnancement des événements (c'est-à-dire les occurrences des actions) peut être exprimé mais sans aucune référence à l'aspect *quantitatif* des instants auxquels ces événements se produisent réellement. L'extension temporelle RT-LOTOS (*Real-Time* LOTOS) reprend les concepts et l'essentiel des opérateurs de LOTOS, et apporte de nouveaux opérateurs permettant d'exprimer des contraintes temporelles quantifiées.

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

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

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

- L'opérateur de délai (noté `delay(`d`)`) permet de retarder un processus d'une certaine quantité de temps d.
- L'opérateur de latence (noté `latency(`l`)`) permet de retarder un processus d'une certaine quantité de temps choisie de manière non-déterministe au sein de l'intervalle

de latence [0,l]. Notons qu'en fait, cet opérateur porte sur la ou les premières actions du processus auquel il est appliqué. Par ailleurs, il n'a d'effet que s'il porte sur une action interne, l'occurrence d'une action observable étant, en tout état de cause, soumise à la date de l'offre faite par l'environnement. En d'autres termes, l'opérateur de latence permet de relâcher la contrainte d'urgence d'une action interne. Il permet d'introduire de manière générale le non-déterminisme temporel.

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

Considérons par exemple le processus `Q = delay(`d`) latency(`l`) g{`t`}; P` qui combine ces trois opérateurs. La figure 1.10 représente sur une ligne temporelle les contraintes spécifiées : dans l'intervalle [0,d], aucune action n'est offerte; à une date choisie dans l'intervalle [d,d+l], l'action g est offerte; l'action g est offerte jusqu'à la date $d+t$, après laquelle elle n'est plus offerte à son environnement.

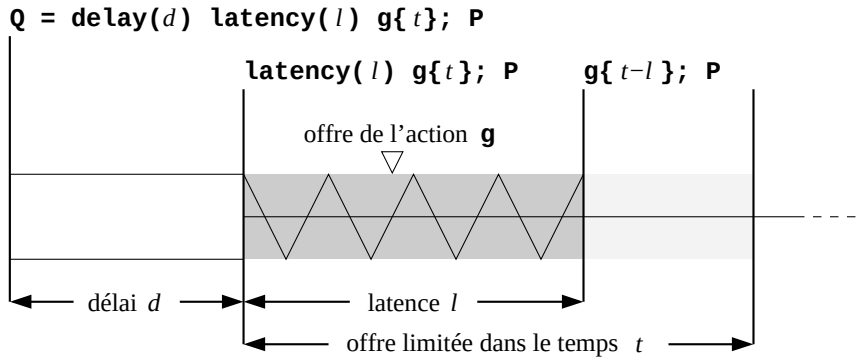


FIG. 1.10 – *Exemple de contraintes temporelles spécifiées avec* RT-LOTOS

En RT-LOTOS les actions observables ne sont par définition pas urgentes puisque leur occurrence dépend de l'environnement qui, de son côté, décide d'offrir ou non ces actions. Pour les actions internes (l'action interne par excellence i, l'action de terminaison δ, et les actions cachées avec `hide`) la situation est plus complexe : en l'absence de latence elles sont urgentes, par contre, en présence de latence leur urgence peut être relâchée dans un intervalle temporel.

i) Présentation informelle des opérateurs de RT-LOTOS

Les opérateurs temporels introduits par RT-LOTOS sont présentés de manière intuitive au moyen des graphiques décrits dans la figure 1.11, où les axes temporels sont supposés avoir pour origine l'instant où le processus associé devient *sensibilisé*. Ces schémas illustrent les moyens offerts par les opérateurs temporels pour décrire les dates d'*offre* et d'*occurrence* des actions *internes* (cachées à l'environnement du processus par l'opérateur *hide*) et *observables* (offertes à l'environnement et dépendant donc de lui).

Par convention nous avons représenté sur ces schémas les différents intervalles temporel de la manière suivante :

- une boite rectangulaire représente l'intervalle de délai, c'est-à-dire l'intervalle de temps pendant lequel il n'y a ni offre ni occurrence d'action ;
- une zone grisée représente l'intervalle de temps dans lequel à lieu l'occurrence de l'action ;
- un ressort représente l'intervalle de latence, c'est-à-dire l'intervalle de temps dans lequel est choisit une date pour l'offre de l'action; cette zone est également grisée (gris foncé) car l'occurrence de l'action peut éventuellement y avoir lieu ;
- un trait vertical épais représente l'instant à partir duquel l'action devient urgente (cas des schémas à gauche) ;
- un triangle blanc marque un instant possible pour l'offre de l'action ;
- un triangle noir marque un instant possible pour l'occurrence de l'action ; si le triangle noir est seul sur la ligne temporelle, alors l'offre et l'occurrence ont lieu au même instant.

1. La contrainte temporelle la plus simple consiste à retarder l'occurrence d'une action (observable ou interne) par une certaine quantité de temps. Dans le processus $P1$, l'occurrence possible de l'action a est retardée d'une durée d. En supposant que $\theta_{P1}(a)$ dénote la date à laquelle l'action a peut se produire, on a $\theta_{P1}(a) \in [d,\infty[$. En effet a est une action observable non urgente, l'occurrence effective de a dépend de la bonne volonté de son environnement.
2. Dans le processus $P2$, l'action a a été intériorisée et se comporte donc comme une action interne. Cette action est urgente puisqu'elle ne dépend plus de la disponibilité de son environnement. Donc, $\theta_{P2}(i_a) = d$, où i_a dénote l'action interne qui résulte de l'intériorisation de l'action a.
3. Dans le processus $P3$, une contrainte temporelle supplémentaire est associée à l'occurrence de a, qui, dès qu'elle est sensibilisée, peut être offerte à son environnement par le processus $P3$ pour une période limitée t. Nous avons donc $\theta_{P3}(a) \in [d,d+t]$. Si, pour une quelconque raison, l'action a ne peut se produire pendant cet intervalle de temps, une *violation temporelle* se produit conduisant à la transformation du processus $P3$ en `stop` (une extension complémentaire à RT-LOTOS permet de spécifier un mécanisme alternatif de traitement de violation temporelle, voir page 35).
4. Dans le processus $P4$, l'action a a été intériorisée et par conséquent $\theta_{P4}(i_a) = d$ est d'une certaine manière similaire à la situation décrite pour le processus $P2$.
5. Dans le processus $P5$, la situation présente la synchronisation de deux processus. Si l'action a peut se produire, elle se produira à un temps appartenant à l'intersection des intervalles temporels caractérisant les contraintes temporelles associées à chaque processus de $P5$, soit $P5\_1$ et $P5\_2$. D'où $\theta_{P5}(a) \in [\theta_{P5\_1}(x) + d1,\theta_{P5\_1}(x) + d1 + t1] \cap [\theta_{P5\_2}(y) + d2,\theta_{P5\_2}(y) + d2 + t2]$. En cas d'intervalles temporels disjoints, une violation temporelle se produit (l'action a ne peut se produire) pour laquelle l'environnement n'est pas responsable. Comme mentionné précédemment, même si les

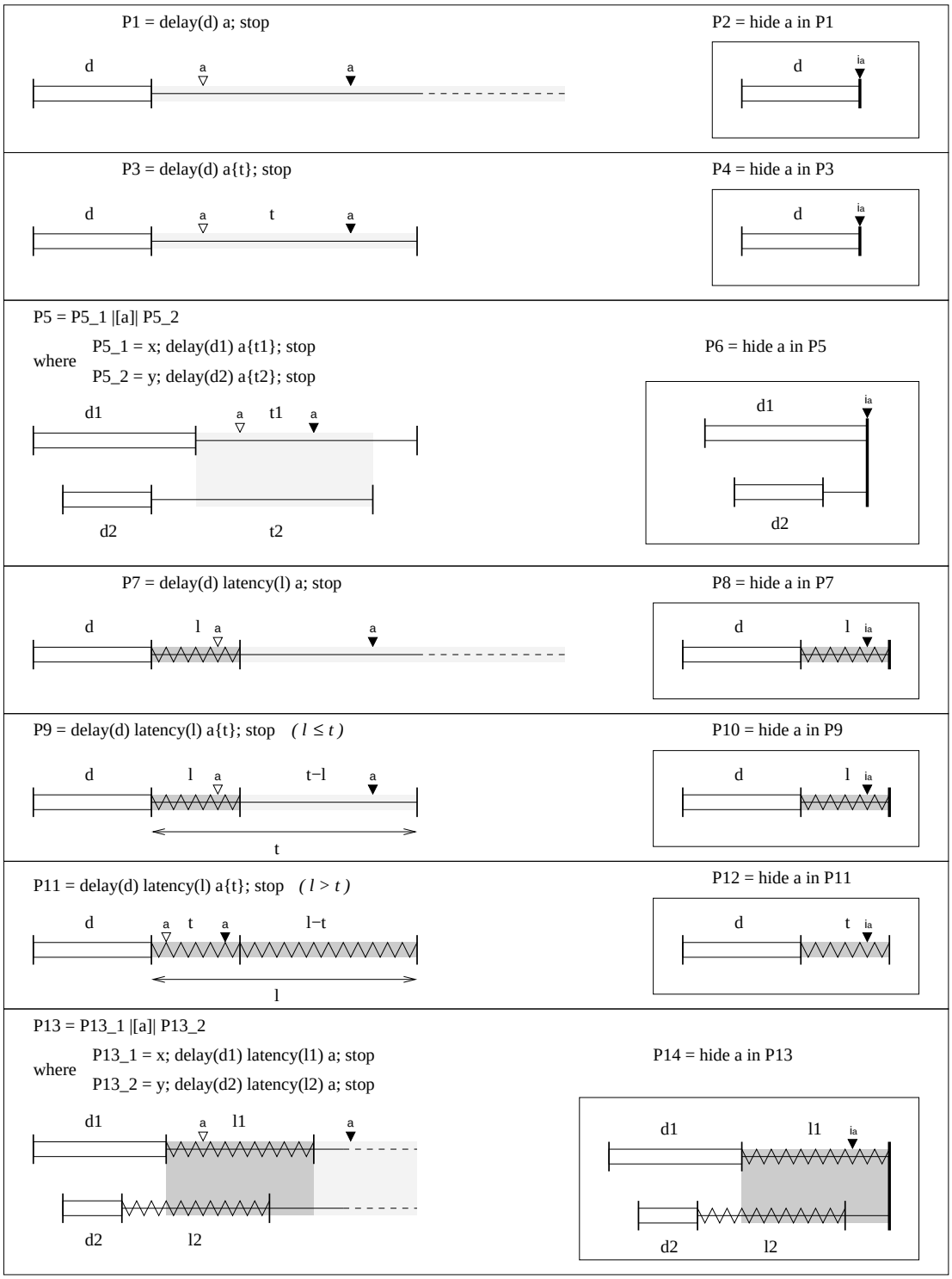


FIG. 1.11 – *Illustration des opérateurs* RT-LOTOS

intervalles temporels ne sont pas disjoints, une violation temporelle peut quand même se produire, si l'environnement n'accepte pas l'occurrence de l'action.

6. Dans le processus $P6$, l'action a a été intériorisée et par conséquent $\theta_{P6}(i_a) = max(\theta_{P5\_1}(x) + d1, \theta_{P5\_2}(y) + d2)$, si l'intersection des intervalles temporels est non vide.
7. Le processus $P7$ introduit l'opérateur de latence. Dans $P7$, l'action a est dans un premier temps retardée d'un temps d, ensuite éventuellement offerte à son environnement pendant une durée l puis, dans le cas où l'action a ne peut se produire, elle est continuellement offerte à son environnement. Comme pour le processus $P1$, on a $\theta_{P7}(a) \in [d,\infty[$, car il n'est pas possible de distinguer la raison pour laquelle une action observable ne s'est pas produite pendant l'intervalle de latence $[d,d+l[$ (cela pourrait être dû au processus lui-même qui n'a pas offert l'action, ou à l'environnement qui n'a pas accepté l'action).
8. En intériorisant l'action a dans le processus $P7$, on obtient le processus $P8$. Puisqu'il y a un non-déterminisme temporel dans l'offre de l'action a pendant l'intervalle de latence $[d,d+l]$, et comme ce non-déterminisme temporel ne dépend pas exclusivement de l'environnement, on a $\theta_{P8}(i_a) \in [d,d+l]$. Ceci illustre bien le fait que l'opérateur de latence a essentiellement pour objectif de relâcher au sein d'un intervalle temporel la condition d'urgence d'une action interne.
9. Le processus $P9$ décrit la combinaison des opérateurs de latence et de délai avec une offre de l'action a limitée pour une période t, où $l \leq t$. D'où $\theta_{P9}(a) \in [d,d+t]$ ce qui est identique à la situation présentée dans le processus $P3$.
10. En intériorisant l'action a dans le processus $P9$ on obtient le processus $P10$ où $\theta_{P10}(i_a) \in [d,d+l]$, exactement comme pour le processus $P8$ (ceci est une conséquence de l'hypothèse $l \leq t$).
11. Le processus $P11$ est similaire au processus $P9$ sauf que l'on suppose maintenant que $l > t$. Donc $\theta_{P11}(a) \in [d,d+l[$.
12. En intériorisant l'action a dans le processus $P11$ on obtient le processus $P12$ où $\theta_{P12}(i_a) \in [d,d+t[$. Notons que cet intervalle de temps est ouvert à droite, ce qui implique que l'action interne i_a peut ne pas se produire.
13. Finalement, les processus $P13$ et $P14$ présentent des situations plus complexes pour lesquelles on pourra vérifier que :
    - $\theta_{P13}(a) \geq max(\theta_{P13.1}(x) + d1, \theta_{P13.2}(y) + d2)$
    - $\theta_{P14}(i_a) \in [tmin,tmax]$ où :
        - $tmin = max(\theta_{P13.1}(x) + d1, \theta_{P13.2}(y) + d2)$
        - $tmax = max(\theta_{P13.1}(x) + d1 + l1, \theta_{P13.2}(y) + d2 + l2)$

En résumé, RT-LOTOS apporte plusieurs fonctionnalités temporelles :

- Extensions temporelles de base
    - retarder l'occurrence des actions observables et internes (voir l'opérateur de *délai*),

- exprimer le non-déterminisme temporel (voir l'opérateur de *latence*),
- limiter le temps pendant lequel une action observable est offerte à son environnement.

– Extensions temporelles complémentaires qui seront présentées page 35.

ii) Définition formelle de RT-LOTOS

Pour faciliter sa compréhension cette définition est divisée en deux parties :

– la définition formelle de Basic RT-LOTOS qui décrit les principaux opérateurs temporels de RT-LOTOS (délai, latence et offre limitée), mais ne permet pas d'échanges de valeurs lors de la synchronisation entre les processus.

– la définition formelle de (Full) RT-LOTOS qui inclut Basic RT-LOTOS, un opérateur complémentaire `@` utilisé pour connaître l'instant d'occurrence d'une action, ainsi que le mécanisme d'échange de valeurs lors de la synchronisation entre les processus.

Hypothèse et notations

Vis-à-vis du domaine temporel, l'unique hypothèse que nous faisons est que le temps est dénombrable, ce qui implique que le modèle sémantique sous-jacent est un Système de Transitions Étiquetées (LTS). De cette manière il est possible de prendre en compte aussi bien un domaine temporel discret (entiers naturels $\mathbb{N}$) qu'un domaine temporel dense (rationnels $\mathbb{Q}$).

Soit D le domaine temporel. D_0, D^∞ et D_0^∞ dénotent respectivement $D \cup \{0\}$, $D \cup \{+\infty\}$ et $D_0 \cup \{+\infty\}$.

Nous utiliserons les variables t et u,v sur respectivement l'ensemble D et D_0^∞.

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

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

– $P \overset{a}{\not\longrightarrow}$ signifie que le processus P ne peut réaliser l'action a

– $P \xrightarrow{t} P'$ signifie que le processus P ne peut exécuter aucune action pendant une période de t unités de temps et se comporte ensuite comme le processus P'.

iii) Syntaxe formelle de Basic RT-LOTOS

La syntaxe formelle de Basic RT-LOTOS est donnée par la figure 1.12; la syntaxe de la déclaration d'un processus étant :

$$\textbf{process } X[g_1, \ldots g_n] := P \textbf{ endproc}$$

Des syntaxes alternatives, à savoir, `delay(`u`)`, `latency(`v`)` et `delay(`u`,`v`)`, ont été introduites avec respectivement pour signification Δ^u, Ω^v et $\Delta^u\Omega^{v-u}$.

Notons que RT-LOTOS prend également en compte l'opérateur de non-déterminisme temporel `i{`u`}` introduit par ET-LOTOS. Ce point sera discuté dans la section 1.2.3.

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

FIG. 1.12 – *Syntaxe de Basic* RT-LOTOS

iv) Sémantique opérationnelle de Basic RT-LOTOS

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

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

Ces actions sémantiques présentent les caractéristiques d'urgence suivantes:

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

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

$$P \xrightarrow{i_s} \quad \Rightarrow \quad \forall t \neq 0,\ P \overset{t}{\not\longrightarrow}$$

L'effet principal de l'opérateur de latence de RT-LOTOS est donc de transformer, quand certaines conditions temporelles spécifiques sont remplies, les actions fortes (dérivées des actions syntaxiques) en actions sémantiques faibles. Notons que lorsque l'opérateur de latence Ω^t est appliqué à un processus P, la ou les *premières* actions offertes par P durant l'intervalle de latence défini par t, sont interprétées selon la sémantique des actions *faibles*.

Cet aspect apparaît dans la figure 1.13 qui donne la sémantique opérationnelle de Basic RT-LOTOS. Comme précédemment, pour plus de clarté, les règles d'inférence symétriques ont été omises et les conventions de notations suivantes sont utilisées:

- g_x (ainsi que g_y) représente g_s ou g_w, quand il est nécessaire d'expliciter dans les règles d'inférence le type (fort ou faible) des actions sémantiques considérées,
- g représente g_s ou g_w, quand il n'est pas nécessaire d'expliciter dans les règles d'inférence le type des actions sémantiques considérées.
- l'opérateur binaire $\wedge$ défini par $x \wedge y = \begin{cases} s & \text{si } (x = s) \wedge (y = s) \\ w & \text{sinon} \end{cases}$

Quelques explications sur le modèle sont présentées ci-dessous:

- Les règles $(1.a)$, $(2.a)$ et $(3.b)$ caractérisent les occurrences des actions sémantiques fortes associées respectivement à l'action de terminaison, à une action observable et à une action interne.

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

FIG. 1.13 – *Le modèle Basic* RT-LOTOS

- La règle (3.a) caractérise les occurrences d'une action interne faible, et permet d'introduire avec la règle (3.c) l'opérateur de délai non-déterministe de ET-LOTOS : `i{`t`}`.
- La règle (5.a) caractérise la synchronisation des actions g fortes et faibles; notons que la synchronisation d'une action forte avec une action faible conduit à une action faible.
- La règle (6.b), avec la règle (6.c), exprime que l'intériorisation des actions observables fortes conduit à l'action interne forte, i_s, qui est urgente; au contraire, intérioriser des actions observables faibles conduit à l'action interne faible, i_w, qui n'est pas urgente.
- La règle (7.b), avec la règle (7.c), caractérise l'urgence de la terminaison d'un processus, qui dépend de la nature faible ou forte de la terminaison.
- La règle (11.a) montre comment les actions observables faibles et fortes se transforment elles-même en actions faibles; ce qui est le principe de base de l'opérateur de latence; notons que, en considérant la règle (11.b), que l'opérateur de latence n'affecte pas les actions internes; la règle (11.c) montre que l'opérateur de latence n'a pas d'effet quand l'intervalle associé devient [0,0].
- Les règles (2.b) et (3.c), (10.b) et (10.c), (11.d) et (11.e) montrent les règles de progression du temps pour respectivement l'opérateur de préfixe, de délai et de latence. Les autres règles de progression du temps sont triviales.

Note : la sémantique de RT-LOTOS contient des règles avec des prémices négatives (par exemple $P \overset{g_s}{\not\longrightarrow}$ dans la règle 6.c). Comme cela est expliqué dans [Gro93],les prémices négatives doivent être utilisées prudemment car elles peuvent conduire à des "inconsistances" dans la sémantique, c'est-à-dire qu'aucun LTS ne peut être associé à certaines expressions de comportement. La sémantique du langage ET-LOTOS contient également des prémices négatives semblables. Il a été démontré dans [LL98b] que ces prémices négatives ne conduisent pas à des inconsistances en exposant comment un LTS peut être dérivé de toutes les expressions de comportement possibles. De manière similaire, il est possible de démontrer que les prémices négatives présentes dans la sémantique de RT-LOTOS ne conduisent pas à des inconsistances.

v) Introduction à (Full) RT-LOTOS

Soit $\mathcal{S}$ l'ensemble des *sortes*. Un ensemble typique (et minimal) de sortes est $\mathcal{S} = \{bool, nat, int, string, time\}$ où $bool = \{false,true\}$, $nat = \mathbb{Z}$, $int = \mathbb{N}$, $string$ est l'ensemble des chaîne de caractères affichables, et $time = D_0^\infty$.

Soit $\mathcal{V}$ l'ensemble des noms des *variables*. La fonction totale $sort : \mathcal{V} \rightarrow \mathcal{S}$ retourne pour chaque nom de variable la sorte $sort(v)$ de sa valeur. Dans la définition formelle qui suit, nous utilisons $x,y \in \mathcal{V}$, $v_b \in bool$, $v_i \in int$ et $v_s \in string$. Soit $\mathcal{O}$ l'ensemble des noms d'*opérations*, et $op \in \mathcal{O}$. Le domaine de chaque op est une liste d'éléments de $\mathcal{S}$ et son image est aussi un élément de $\mathcal{S}$. Notons que ces sortes sont assimilées ici à des objets mathématiques (comme des ensembles, des listes, ...) à la place des types abstraits Act-One

de (Full) LOTOS. Ceci a été motivé par deux raisons :

- des raisons théoriques visant à simplifier la théorie de vérification de (Full) RT-LOTOS,
- des raisons pragmatiques qui rendent l'utilisation des types concrets plus facile que celle des types abstraits.

En RT-LOTOS, seules les déclarations des variables (la signature des types abstraits) sont effectuées en utilisant la syntaxe de Act-One. La partie correspondant à la sémantique des types de données est, elle, implémentée en utilisant les langages C++ ou Java.

Un exemple de déclaration RT-LOTOS d'un type abstrait (ou sorte) *nat* (correspondant aux entiers naturels) est présenté dans la figure 1.14. Une opération est définie dans un type de manière similaire à celle d'objets mathématiques traditionnels. Par exemple, l'expression «`+ : nat,nat -> nat`» correspond à la définition de l'opération d'addition qui utilise deux opérandes de type `nat`, dont le résultat produit un autre `nat`. L'implémentation de cette opération est faite en C++ au moyen d'un objet qui possède une méthode qui prend deux entiers naturels et en retourne un autre, dont le résultat correspond à l'addition des deux derniers.

```
type nat is bool
  sorts nat
  opnt
  + : nat,nat -> nat
  - : nat,nat -> nat
  (...)
  > : nat,nat -> bool
endtype
```

FIG. 1.14 – *Exemple de déclaration d'un type naturel en* RT-LOTOS

Le formalisme Act-One n'est pas détaillé dans ce document car il n'est pas utilisé dans la définition de RT-LOTOS. Quelques exemples illustrant l'utilisation de Basic et de Full LOTOS peuvent êtres trouvés dans [BB87, LFHH92].

vi) Extensions temporelles complémentaires

Le modèle RT-LOTOS apporte des extensions temporelles autres que le délai fixe, la latence et l'offre limitée dans le temps. Ces extensions sont partiellement supportées par les outils actuellement disponibles pour RT-LOTOS, et par conséquent peu utilisées. C'est pourquoi ces apports du langage RT-LOTOS sont qualifiés d'*extensions temporelles complémentaires* et son présentées dans ce paragraphe.

◦ (Full) RT-LOTOS introduit un nouvel opérateur, noté @. Cet opérateur, associé à une action LOTOS, enregistre dans une variable le temps séparant l'instant auquel l'action est offerte de l'instant auquel elle se réalise. Cette variable peut alors être utilisée dans la suite du processus. Cet opérateur est tout à fait similaire à l'opérateur *time capture* de ET-LOTOS. Le fragment de code illustré dans la figure 1.15 montre un exemple d'utilisation de l'opérateur @ en RT-LOTOS. Il s'agit de mesurer du temps de réponse d'une interaction de l'utilisateur; si ce temps dépasse 5 secondes, un traitement particulier est réalisé.

```
(...)
 attend_utilisateur @t_entree;
 (
  [t_entree > 5] -> Traite_Erreur[...]
  []
  [t_entree <= 5] -> Traite_Entree[...]
 )
```

FIG. 1.15 – *Exemple d'utilisation de l'opérateur @*

Cet opérateur est exploitable pour une validation en terme de simulation mais n'est pas actuellement supporté par les outils de vérification formelle.

Une utilisation de cet opérateur est décrite dans la suite de ce mémoire, dans le paragraphe 3.1.5.

◦ Le langage RT-LOTOS propose également un mécanisme complexe d'exception en cas de violation temporelle [CdO94]. Ainsi, le processus «$g\{t,Q\};P$» offre l'action g pendant t unités de temps, puis, si la synchronisation à lieu dans ce laps de temps, se transforme en P, sinon en Q. La construction «$g\{t\};P$ [] $\Delta^t Q$» exprime un comportement assez similaire à celui-ci, excepté à la date t: l'écriture de «$g\{0,Q\};P$» assure que l'occurrence de g aura toujours la priorité sur les actions de Q; ce qui n'est pas le cas de «$g\{0\};P$ [] $\Delta^0 Q$». Ainsi, «$g\{0,Q\};P$» exprime que Q est possible seulement si l'environnement n'est pas près à se synchroniser sur g. Ce comportement a été obtenu par une extension du langage, en introduisant l'action spécifique π qui a une priorité inférieure aux autres actions, la sémantique opérationnelle étant enrichie de règles en ce sens. En fait, les propriétés apportées correspondent à des comportements très précis qui ont exceptionnellement un intérêt pratique uniquement dans certains cas "exceptionnels". C'est pourquoi ils n'ont jamais étés véritablement utilisés, ni implémentés. Ainsi, ce mécanisme d'exception et de reprise en cas de violation temporelle ne sera pas détaillé ici.

vii) Syntaxe formelle de (Full) RT-LOTOS

La syntaxe formelle de (Full) RT-LOTOS est donnée, dans la figure 1.16, par la syntaxe des expressions des valeurs (E), celle des offres des portes (O), et finalement celle des

expressions de comportements (P); la syntaxe de la déclaration d'un processus étant :

process $X[g_1, \ldots g_n](x_1 : s_1, \ldots x_m : s_m) := P$ **endproc**

$$
\begin{array}{lcl}
E & ::= & op(E_1, \ldots E_n) \mid x \mid v_b \mid v_i \mid v_s \\
O & ::= & \texttt{?}\ x\ \texttt{:}\ s \mid \texttt{!}\ E \\
P & ::= & \texttt{stop} \mid \texttt{exit} \mid X\ \texttt{[}\ L\ \texttt{]}\ \texttt{(}\ E_1, \ldots E_m\ \texttt{)} \\
 & \mid & \texttt{let}\ x\ \texttt{:}\ s\ \texttt{=}\ E\ \texttt{in}\ P \mid \texttt{hide}\ L\ \texttt{in}\ P \mid \Delta^u\ P \mid \Omega^u\ P \\
 & \mid & \texttt{i}\ \texttt{;}\ P \mid \mid \texttt{i}\ \texttt{\{}\ u\ \texttt{\}}\ \texttt{;}\ P \\
 & \mid & \texttt{i}\ \texttt{@}\ x\ \texttt{;}\ P \mid \texttt{i}\ \texttt{@}\ x\ \texttt{\{}\ u\ \texttt{\}}\ \texttt{;}\ P \\
 & \mid & g\ O_1 \cdots O_n\ \texttt{;}\ P \mid g\ \texttt{\{}\ u\ \texttt{\}}\ O_1 \cdots O_n\texttt{;}\ P \\
 & \mid & g\ \texttt{@}\ x\ \texttt{\{}\ u\ \texttt{\}}\ O_1 \cdots O_n\texttt{;}\ P \\
 & \mid & P\ \texttt{[]}\ P \mid P\ \texttt{|[}\ L\ \texttt{]|}\ P \mid P\ \texttt{>>}\ P \mid P\ \texttt{[>}\ P
\end{array}
$$

FIG. 1.16 – *Syntaxe de (Full)* RT-LOTOS

viii) Sémantique opérationnelle de (Full) RT-LOTOS

La sémantique de l'opérateur général de préfixage de (Full) RT-LOTOS, «g `@`x `{`u`}` $O_1 \cdots O_n$ `;` P», est définie par un opérateur auxiliaire, noté g `@(`x`,`v`)` `{`u`}` $O_1 \cdots O_n$ `;` P, où $v \in D^\infty$ qui représente le temps pendant lequel une action a été offerte avant d'être tirée. Par définition : «g `@`x `{`u`}` $O_1 \cdots O_n$ `;` $P \equiv g$ `@(`x`,0)` `{`u`}` $O_1 \cdots O_n$ `;` P»

La figure 1.17 montre comment la sémantique de Basic RT-LOTOS est enrichie pour donner la sémantique opérationnelle de (Full) RT-LOTOS. La plupart des règles sont très similaires à leurs homologues de (Full)-LOTOS. Les règles restantes peuvent être déduites facilement des règles sémantiques de Basic RT-LOTOS, en remplaçant l'offre des portes de Basic-LOTOS par celle de l'offre générale des portes qui prend en compte l'échange de données.

ix) Propriétés fondamentales

Le modèle RT-LOTOS satisfait les propriétés temporelles suivantes :

progression maximale : si $P \xrightarrow{i_s} P'$ pour un P' quelconque, alors, pour tout $t > 0$, $P \overset{t}{\not\longrightarrow}$;

déterminisme temporel : si $P \xrightarrow{t} P'$ et $P \xrightarrow{t} P''$, alors P' et P'' sont identiques ;

continuité temporelle : pour tout t et u, si $P \xrightarrow{t+u} P''$ alors il existe P' tel que $P \xrightarrow{t} P' \xrightarrow{u} P''$.

Notons toutefois que le modèle ne satisfait pas la propriété d'*additivité temporelle*, comme on peut le constater sur l'exemple suivant :

(2.*a*)	$g\{u\}!E;P \xrightarrow{g_s!<valeur(E)>} P \quad (g \in \mathcal{G})$
(2.*a'*)	$g\{u\}?y:s;P \xrightarrow{g_s!<p>} let\, y:s=p\, in\, P \quad (g \in \mathcal{G})(p \in s)$
(2.*b*)	$g\{u+t\};P \xrightarrow{t} g\{u\};P \quad (g \in \mathcal{G})$
(2.*c*)	$g\{0\};P \xrightarrow{t} stop \quad (g \in \mathcal{G})$
(2.*a*)	$g@(x,v)\{u\}!E;P \xrightarrow{g_s!<valeur(E)>} let\, x:time=v\, in\, P \quad (g \in \mathcal{G})$
(2.*a'*)	$g@(x,v)\{u\}?y:s;P \xrightarrow{g_s!<p>} let\, x:time=v, y:s=p\, in\, P \quad (g \in \mathcal{G})(p \in s)$
(2.*b*)	$g@(x,v)\{u+t\};P \xrightarrow{t} g@(x,v+t)\{u\};P \quad (g \in \mathcal{G})$
(2.*c*)	$g@(x,v)\{0\};P \xrightarrow{t} stop \quad (g \in \mathcal{G})$
(12.*a*)	$\dfrac{P \xrightarrow{g} P'}{let\, x:s=e\, in\, P \xrightarrow{g} let\, x:s=e\, in\, P'}$
(13.*a*)	$\dfrac{P \xrightarrow{g} P' \quad valeur(E)=true}{[E]->P \xrightarrow{g} P'}$
(13.*b*)	$\dfrac{P \xrightarrow{t} P' \quad valeur(E)=true}{[E]->P \xrightarrow{t} P'}$
(13.*c*)	$\dfrac{P \xrightarrow{t} P' \quad valeur(E)=false}{[E]->P \xrightarrow{t} stop}$

FIG. 1.17 – *Le modèle (Full)* RT-LOTOS

$$\Delta^i g; stop \xrightarrow{1}\xrightarrow{1} g; stop \quad \text{alors que} \quad \Delta^i g; stop \overset{2}{\not\longrightarrow} g; stop$$

D'autres propriétés du modèle ainsi que la définition de l'équivalence comportementale sont présentées dans [CdO94, CdO95].

1.2.3 Traitement du non-déterminisme temporel

RT-LOTOS s'appuie sur Timed LOTOS [LL92], le prédécesseur de ET-LOTOS [LL93, LL97], et c'est donc pourquoi RT-LOTOS et ET-LOTOS partagent de nombreux aspects communs au niveaux syntaxique et sémantique, notamment :

- L'hypothèse de non-urgence des actions observables (c'est-à-dire, l'occurrence d'une action observable ne peut pas être forcée par un processus, car elle ne peut dépendre que de l'environnement) et d'urgence des actions internes ;
- La capacité à retarder un comportement ou à retarder l'offre d'une action observable à son environnement ;
- La possibilité de mesurer l'instant auquel une action se produit ;
- La capacité à limiter le temps d'occurrence pendant lequel une action observable est offerte à son environnement. Ce point a été généralisé par le mécanisme de violation temporelle de RT-LOTOS.

Timed LOTOS proposait un opérateur de délai non-déterministe $\Delta^{[d^-,d^+]}$ dont la définition formelle conduisait à certains problèmes sémantiques délicats. ET-LOTOS et RT-LOTOS ont alors proposé, indépendamment, des solutions alternatives à l'expression du non-déterminisme temporel.

La solution proposée par ET-LOTOS consiste à retirer l'opérateur de délai non-déterministe de Timed LOTOS, et à ne conserver qu'un opérateur de délai déterministe classique Δ^d. Le non-déterminisme temporel est alors exprimé au moyen d'une variante de l'opérateur d'offre limitée dans le temps lorsqu'il s'applique à l'action interne `i` : la notation `i{`t`}` permet de relâcher l'urgence de l'action et d'introduire un délai non-déterministe choisi dans l'intervalle temporel [0,t] avant de réaliser l'action `i`.

La solution proposée par RT-LOTOS consiste à introduire un opérateur spécifique de *latence* Ω^t, noté `latency(`t`)`, qui exprime d'une manière générale le non-déterminisme, sans avoir à souffrir des effets de bord introduits par l'action interne (rappelons que le choix non-déterministe peut être résolu par l'action `i`).

Le processus ET-LOTOS «`i{`t`};` P» peut s'écrire en RT-LOTOS «`latency(`t`) exit >>` P». À l'inverse, le non-déterminisme temporel introduit en RT-LOTOS par «`latency(`t`)` P» ne peut pas être décrit en ET-LOTOS par «`i{t};` P» car il engendre une action interne `i`. Cela pose donc un problème lorsque l'on veut spécifier un comportement associant du non-déterminisme temporel avec un choix.

Pour illustrer la différence subtile entre les sémantiques de ces deux solutions, considérons la figure 1.18 qui donne le système de transitions étiquetées (LTS) associé à chaque exemple spécifié. Dans les LTS, le double cercle indique l'état initial, les transitions `t` indiquent une progression du temps, et les transitions `i(a)` et `i(b)` indiquent l'occurrence des actions internes `a` et `b`.

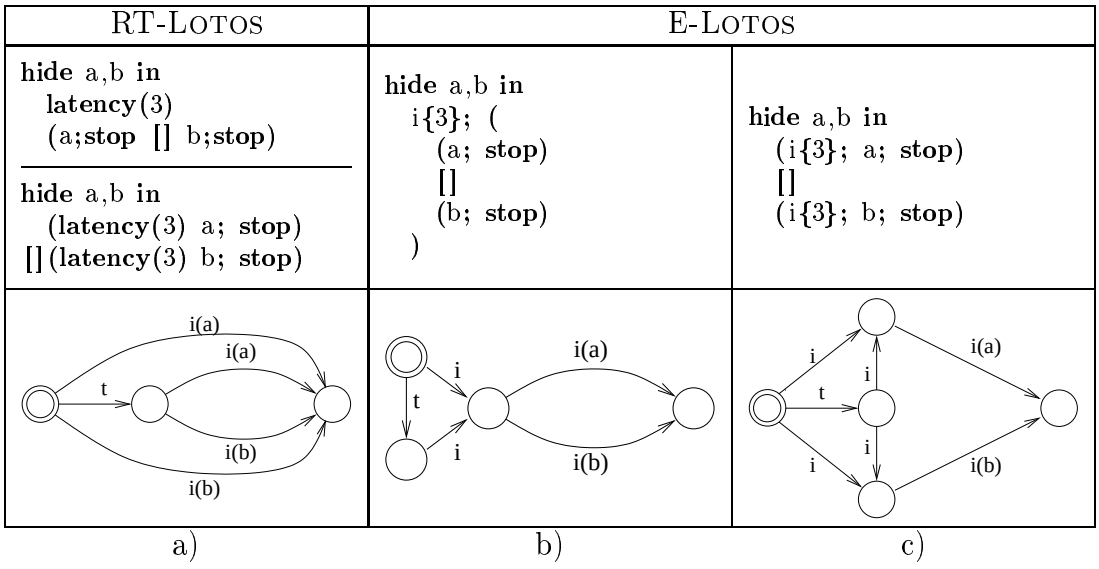


FIG. 1.18 – *Comparaison entre* `latency(t)` *et* `i{t}`

La figure 1.18a illustre l'utilisation de l'opérateur de latence, avec deux spécifications décrivant le même comportement (l'opérateur de latence est distributif vis-à-vis de l'opérateur de choix). Ce processus propose, après un délai non déterministe, deux actions a et b à son environnement qui devra décider laquelle choisir.

Le comportement exprimé dans la figure 1.18b est légèrement différent puisque le délai précédant les offres de `i(a)` ou `i(b)` n'est pas choisi indépendamment pour les deux actions, mais globalement, dans une étape précédente, par l'opérateur `i{t}`.

Pour assurer cette indépendance on pourrait tenter de distribuer le délai non déterministe sur chacune des branches du choix, comme exprimé dans la figure 1.18c. Il apparaît alors que le choix entre `a` et `b` n'est plus fait par l'environnement, mais localement, par l'occurrence de l'une ou l'autre des actions `i` placées sur chaque branche. Par conséquent, seule l'une des actions `a` ou `b` est offerte à l'environnement.

Ceci illustre un problème théorique plus général : il existe une bisimulation faible pour RT-LOTOS offrant les même propriétés de congruence que Basic LOTOS, ce qui n'est pas le cas pour ET-LOTOS (une preuve est proposée dans [CdO94]).

Par ailleurs, l'opérateur `i{`t`}` n'est pas très satisfaisant, d'une part car il introduit un particularisme dans la sémantique de l'opérateur d'offre limitée dans le temps «g`{`t`}`», et d'autre part car il gonfle artificiellement le nombre de transitions et d'états du LTS en introduisant ces transitions `i` non souhaitées. Un reproche similaire avait été formulé dans [LL97] à l'encontre de l'opérateur `latency(`t`)`, car il nécessite l'ajout d'un préfixe particulier pour distinguer les actions concernées ou non par la latence (attributs *weak* ou *strong*) dans le modèle sémantique (voir les figures 1.13 et 1.17). Nous avons éliminé cette critique en associant de manière explicite à l'automate temporel dérivé d'une spécification RT-LOTOS sur lequel s'effectuera l'analyse d'accessibilité conduisant au LTS associé, un domaine temporel d'urgence ne nécessitant plus la distinction entre actions urgentes et actions observables. Ce point sera abordé en détail dans le paragraphe 1.3.3.

1.3 Validation avec RT-LOTOS

En tant que langage formel, RT-LOTOS permet de décrire un système sans ambiguïté. La spécification peut être exploitée, à des fins de *validation*, à deux niveaux : en *simulation* et en *vérification*. La figure 1.19 illustre les diverses techniques de validation possibles sur une spécification RT-LOTOS.

L'outil `rtl` (*Real-Time* LOTOS *Laboratory*) a été réalisé par Roberto CRUZ DE OLIVEIRA. Les outils `dtasim` et `dta2kronos` son des contributions de cette thèse.

1.3.1 Simulation

La simulation permet de réaliser *une* exécution possible du système spécifié. Bien entendu, cela ne peut en aucun cas apporter la preuve formelle que le système est correct, mais cela offre tout de même à l'auteur de la spécification un certain degré de confiance dans le système spécifié. De plus, un utilisateur averti, aidé d'un environnement de travail adéquat, en menant des séries de simulations avec circonspection, peut déceler des cas d'erreur, qu'il pourra, dans un deuxième temps, identifier formellement.

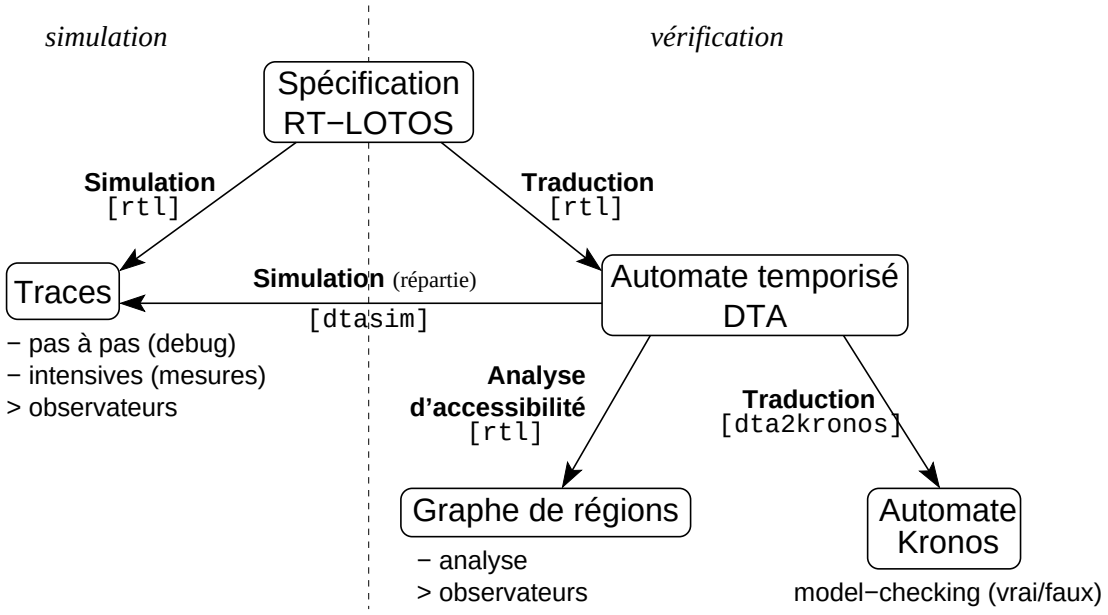


FIG. 1.19 – *Validation avec* RT-LOTOS

Dans la mesure où ce n'est généralement pas *une* simulation, mais un *ensemble de simulations*, qui permet de tirer des conclusions probantes, il convient de bien *paramétrer* la spécification, afin de pouvoir effectuer des séries de simulations en faisant varier des paramètres (en particulier des paramètres temporels).

La technique de simulation permet d'appréhender le comportement du système spécifié, soit de manière globale en effectuant des mesures (à différents niveaux de granularité), soit de manière plus localisée en utilisant conjointement des processus observateurs conçus pour repérer un dysfonctionnement particulier. On appelle observateur un processus que l'on va décrire en RT-LOTOS et composer avec la spécification du système; ce processus a pour vocation d'observer le système au regard d'une propriété précise à valider et de déclencher un signal `erreur` lorsque cette propriété est violée par le système. Ainsi, si dans une trace apparaît l'action `erreur`, on dispose alors d'un exemple de dysfonctionnement; cependant si cette action n'est pas présente, il n'est pas possible de conclure.

Considérons par exemple la figure 1.20 qui donne la spécification RT-LOTOS d'un médium de communication simple. Le système spécifié est composé de deux processus :

- `stream_sender` : un processus qui modélise l'émeteur et qui réalise périodiquement, avec une période paramétrée par la variable `period`, une action de synchronisation `iu_s` (pour *information unit sent*) ;
- `medium` : un processus qui modélise le médium de communication et qui réalise en boucle deux actions de synchronisations `m_in` et `m_out` espacées d'un délai variable choisit entre `dmin` et `dmax`, pour représenter le fait qu'un message entre dans le médium est en ressort au bout d'un temps plus ou moins long (délai de transport et gigue).

```
specification MEDIUM : noexit
 (...)
behaviour
 hide iu_s, iu_d in
 let period : nat = 30000 in

 stream_sender[iu_s](0,period)
 |[iu_s]|
 medium[iu_s,iu_d](14000,20000)

where

   process stream_sender [iu_s] (n : nat, period : nat) : noexit :=
     iu_s{0}!n; delay(period) stream_sender[iu_s] (n+1, period)
   endproc

   process medium [m_in,m_out] (dmin, dmax : nat) : noexit :=
     m_in?x:nat; delay(dmin,dmax)m_out!x;
     medium[m_in,m_out](dmin,dmax)
   endproc

endspec
```

FIG. 1.20 – *Exemple : médium de communication*

La figure 1.21 illustre différentes vues offertes par *une* simulation du médium de communication : la trace des événements, occurrences des actions sur des lignes temporelles, mesures de différentes caractéristiques du système. Le fichier de traces est généré par l'outil `rtl`. Sur la première colonne apparaît la date d'occurrence de l'action indiquée sur la deuxième colonne. Cette action peut être soit une progression temporelle, soit une action LOTOS, qui, à titre de documentation peut comporter plusieurs préfixes : `i(a)` indique que l'action `a` est une action interne (typiquement, elle a été intériorisée avec l'opérateur `hide`) et donc urgente, `w-a` (pour *weak*) indique que l'action `a` s'est produite à une date antérieure à sa zone d'urgence (typiquement, son urgence a été relâchée par un délai non déterministe). La troisième colonne du fichier de trace numérote les états de la simulation. Les deux autres diagrammes sont produits par des scripts (éventuellement définis par l'utilisateur) qui raffinent la trace de simulation pour en donner des vues que l'utilisateur juge plus pertinentes ou plus synthétiques.

Notons que les mesures exposées ici couvrent *une* seule simulation. Il est également possible, au moyen de scripts, d'effectuer des *séries* de simulations en faisant varier certains paramètres (par exemple la variable `period`), et de mesurer les comportements induits par ces paramètres. Par exemple, dans cette spécification, une valeur de `period` trop courte par rapport au délai de traversée du médium, met en défaut le système car, tel qu'il est modélisé, un seul message à la fois peut transiter dans le médium.

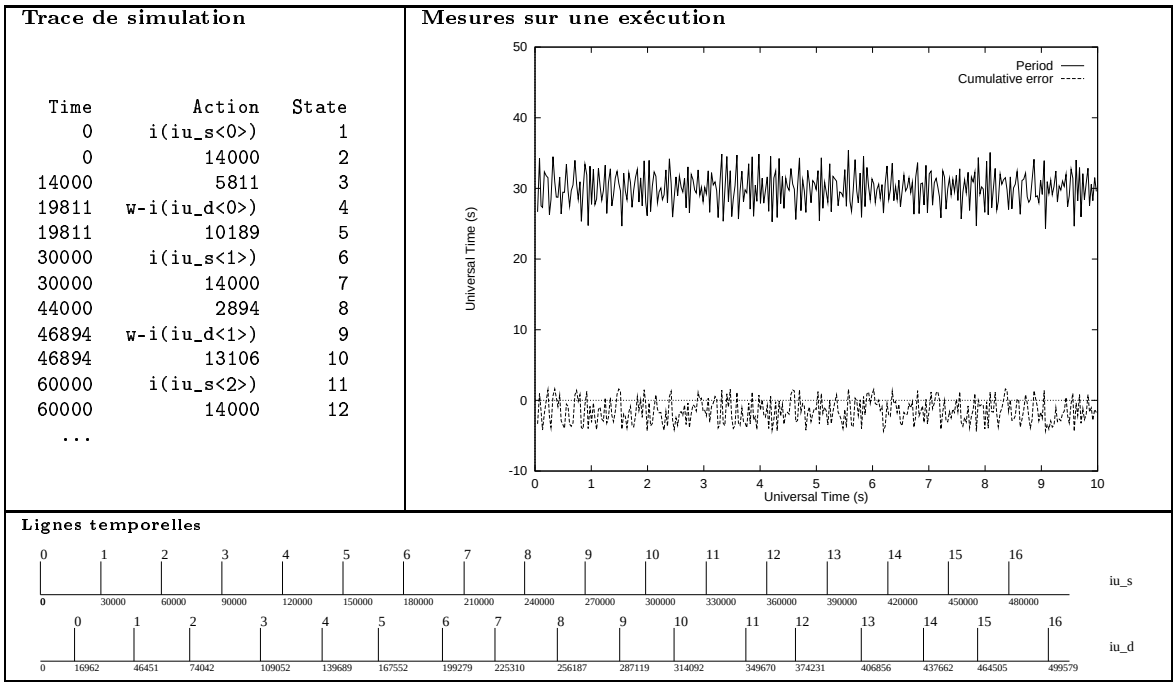


FIG. 1.21 – *Simulation du médium de communication*

L'outil `dtasim`

Le schéma de la figure 1.19 présente deux approches pour réaliser une trace de simulation : soit depuis la spécification RT-LOTOS par exécution de la sémantique opérationnelle, soit depuis un automate temporisé (`DTA`) dérivé de la spécification RT-LOTOS.

L'outil `rtl` permet d'effectuer la simulation depuis la spécification RT-LOTOS : l'arbre sémantique de la spécification est construit et interprété au fur et à mesure. Ces opérations sont coûteuses en temps machine. Par conséquent, nous avons développé une autre approche.

L'outil `rtl`, à des fins de vérification, permet de traduire la spécification dans un automate temporisé appelé `DTA` (voir paragraphe 1.3.2). L'idée a donc été d'exploiter cet automate pour réaliser de la simulation. Ainsi nous bénéficions en quelque sorte d'une compilation de la spécification.

Un outil, appelé `dtasim`, a été réalisé, pour implémenter la sémantique des automates de type `DTA` et produire une trace de simulation identique à celle produite par l'outil `rtl`. Un gain de performance notable à été observé : le rapport moyen des temps d'exécution entre une simulation par `dtasim` et par `rtl` est de l'ordre de 1 à 100 en faveur de `dtasim`.

Cette approche est donc très concluante, mais elle doit être modérée par les limites de `rtl` vis-à-vis de la génération du `DTA` essentiellement sur deux aspects. Tout d'abord, l'opérateur RT-LOTOS de capture temporelle `@` n'étant pas traduit dans le `DTA`, les spécifications qui l'emploient et qui peuvent être simulées par l'approche traditionnelle (exécution directe des actions sémantiques RT-LOTOS avec `rtl`), ne sont pas ainsi

simulables depuis le `DTA`. Ensuite, certaines spécifications non bornées peuvent occasionner une traduction en `DTA` qui ne se termine pas (typiquement, lorsqu'un processus est paramétré par une variable de type infini, tel que le processus `stream_sender` de la spécification précédente qui reboucle récursivement et numérote les paquets envoyés par un entier naturel), et ne sont pas, elles non plus, simulables depuis le `DTA`, alors qu'elles peuvent l'être par l'approche traditionnelle.

Par ailleurs, l'outil `dtasim` a donné lieu à une deuxième version permettant de réaliser des simulations réparties sur plusieurs machines [Cas99] : la spécification globale est alors découpée en plusieurs spécifications indépendantes, mais qui peuvent se synchroniser sur des portes d'interaction. Pour chacune d'elles, un `DTA` est engendré. Chaque `DTA` est simulé sur une machine dédiée. Les simulateurs se synchronisent entre eux par l'intermédiaire d'un *maître*. Cette approche a montré son intérêt dans le cas de systèmes nécessitants *peu* de synchronisation entre les simulateurs répartis.

1.3.2 Vérification formelle

La vérification formelle, telle qu'elle est mise en œuvre avec RT-LOTOS, consiste à explorer l'ensemble des *configurations* (ou *états globaux*) que le système spécifié peut atteindre depuis sa configuration initiale. Il s'agit d'une vérification formelle par *analyse d'accessibilité*.

Cette technique se déroule en deux temps : la spécification RT-LOTOS est compilée en un automate temporisé, puis une analyse d'accessibilité des configurations de cet automate temporisé est réalisée afin produire un *graphe minimal d'accessibilité*.

i) Automate temporisé dynamique (`DTA`)

L'outil `rtl` permet de traduire une spécification RT-LOTOS en un automate appelé `DTA` (pour *Dynamic Timed Automata*).

À chaque *état de contrôle* de cet automate est associé un ensemble de variables locales (à cet état) et à valeurs dans l'ensemble des rationnels non-négatifs ($\mathbb{Q}^+$). Ces variables sont appelées *horloges*. Les valeurs des variables-horloges croissent de manière continue et spontanément avec le passage du temps, car le temps peut s'écouler lorsque le système est dans un état de contrôle de l'automate. L'automate temporisé présenté ici est dit *dynamique* car le nombre d'horloges peut varier d'un état de contrôle à l'autre, par opposition à d'autres types d'automates temporisés définis avec un nombre fixe d'horloges qui sont alors globales à tous les états de contrôle.

Le système peut passer d'un état de contrôle à un autre au moyen de *transitions* atomiques (i.e. dont l'exécution ne consomme pas de temps). Chaque transition du `DTA` est étiquetée par :

1. Le nom de l'action LOTOS réalisée. S'il s'agit d'une action interne ce nom est préfixé par `i(·)`.

2. Le domaine de *sensibilisation*, noté K, précise pour quelles valeurs des horloges (de l'état de contrôle de départ de la transition), la transition est tirable. S'il n'y pas d'horloge, K est omis, et la transition est considérée comme sensibilisée en permanence.
3. Le domaine d'*urgence*, noté U, qui ne concerne que les actions potentiellement urgentes (c'est-à-dire les actions internes préfixées par `i(·)`), précise pour quelles valeurs des horloges le temps ne peut plus progresser (par conséquent, une des transitions *doit* être nécessairement tirée). S'il n'y pas d'horloge, U est omis, et dans ce cas, si l'action est interne, le temps ne progresse plus, si non, le temps peut progresser indéfiniment.
4. La liste (éventuellement vide) des horloges de l'état de contrôle d'arrivée devant être réinitialisées à 0, et notée C.
5. La liste (éventuellement vide) des horloges de l'état de contrôle de départ devant être recopiées dans les horloges de l'état de contrôle de l'état d'arrivée, et notée *theta*.

La figure 1.22 illustre le `DTA` construit à partir de la spécification RT-LOTOS donnée dans la figure 1.20 (en prenant la précaution de retirer le paramètre `n:nat` du processus `stream_sender`, sans quoi, la spécification étant non bornée, il ne serait pas possible de construire un `DTA` car `rtl` instancie les variables lors de la construction des états du `DTA`).

Représentation textuelle :

```
State 0: 1 clocks
State 1: 2 clocks
State 2: 1 clocks

3 states, 3 arcs
(0, i(iu_s) K={0<=c1<=0} U={0<=c1<=0} C={1,2}, 1)
(1, i(iu_d) K={14<=c2} U={20<=c2} theta={(1,1)}, 2)
(2, i(iu_s) K={30<=c1<=30} U={30<=c1<=30} C={1,2}, 1)
```

Représentation graphique :

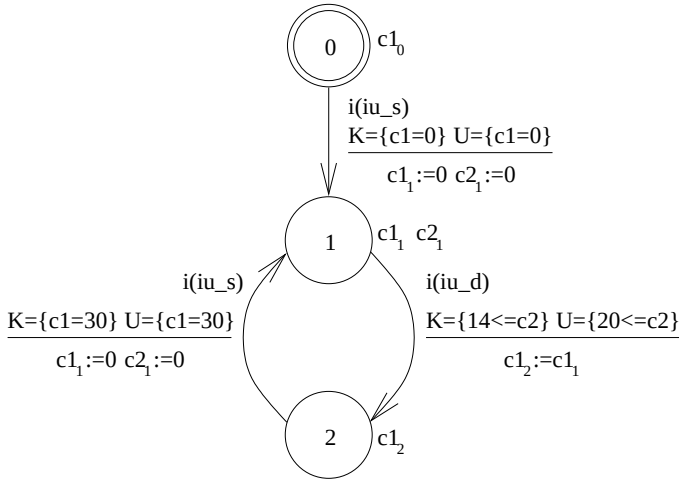


FIG. 1.22 – *Exemple de* `DTA`

Définition formelle du DTA

Soit L un ensemble d'étiquettes (ou *labels*).
Soit $D = \{t \in \mathbb{Q} \mid t > 0\}$ le domaine temporel, et $D_0 = D \cup \{0\}$ et $D_0^\infty = D_0 \cup \{\infty\}$.
Soit $C_{set} = \{c_i \mid i \in \mathbb{N}^+\}$ un ensemble indexé d'horloges (ou *clocks*).
Une condition temporelle est une conjonction d'inégalités de la forme $m \prec c_i \prec M$ où m,M sont des constantes de D_0^∞, et $\prec \in \{<, \leqslant\}$, et $c_i \in C_{set}$.
Soit $\nu \in D_0^n$ les valeurs des horloges c_i, et K une condition temporelle définie sur les horloges $\{c_i \mid i \leqslant n\}$. La notation $\nu \vDash K$ signifie que toutes les inégalités de K sont *vraies* lorsque l'on remplace les horloges c_i par leur valeur ν_i.

Définition 1 (Dynamic Timed Automaton)
Un automate temporisé dynamique (Dynamic Timed Automaton), *est un 4-tuple* $(S,Nclock,E,s_0)$, *où :*

- S *est un ensemble fini d'états de contrôle,*
- s_0 *est l'état de contrôle initial,*
- $Nclock : S \to \mathbb{N}$ *est une fonction qui associe le nombre d'horloges à chaque état de contrôle,*
- E *est un ensemble fini de transitions de la forme* (s,s',K,U,a,C,θ), *où*
  - $s,s' \in S$ *sont les états de contrôle de source et de destination de la transition,*
  - K *et* U *sont des conditions temporelles,*
  - $a \in L$ *est une étiquette d'action,*
  - $C \subseteq \{1,...,Nclock(s')\}$ *définit l'index des horloges de l'état de contrôle de destination qui doivent être ré-initialisées lorsque la transition est tirée,*
  - $\theta : \{1,...,Nclock(s)\} \to \{1,...,Nclock(s')\}$ *définit l'index des horloges de l'état contrôle de source qui doivent être recopiées dans les horloges de l'état contrôle de destination lorsque la transition est tirée.*

Un système de transitions étiquetées LTS(DTA) est associé à chaque automate temporisé dynamique DTA. Un état (s,ν) du LTS(DTA), appelé également une configuration, est entièrement décrit en spécifiant l'état de contrôle s du DTA, et les valeurs $\nu \in D_0^{Nclock(s)}$ de toutes les horloges définies dans cet état de contrôle. Les transitions du LTS(DTA) correspondent soit à une *transition explicite* qui représente le tir d'une transition du DTA, soit à une *transition implicite* qui représente le passage du temps dans un état de contrôle du DTA. La *transition explicite* est décrite par la règle du successeur par transition, et la *transition implicite* est décrite par la règle du successeur temporel.

Définition 2 (État initial du LTS(DTA))
L'état initial du $LTS(DTA)$ *est l'état* (s_0,ν_0) *où* $\nu_0 \in D_0^{Nclock(s_0)}$ *avec* $\forall i \in [1,Nclock(s_0)] : \nu_{0i} = 0$.

Définition 3 (Transition explicite du LTS(DTA))
Soit $(s,\nu) \in LTS(DTA)$ *et* $(s,s',K,U,a,C,\theta) \in E$ *une transition du DTA. Si* $\nu \vDash K$ *alors* $(s,\nu) \xrightarrow{a} (s',\nu')$, *et* $\forall i \in C : \nu'_i := 0$ *et* $\forall i \in [1,Nclock(s')], i \notin C : \nu'_i := \nu_{\theta^{-1}(i)}$.

Définition 4 (Transition implicite LTS(DTA))
Soit $(s,\nu) \in LTS(\texttt{DTA})$ *et* $\delta \in D$. *Soit* $isUrgent$ *un prédicat définit sur* L *qui précise si une action est urgente ou non sur son domaine* U. *Si* $\nu + \delta' \not\models K \cap U$ *pour chaque* $0 \leqslant \delta' \leqslant \delta$ *et pour chaque* $(s,s',K,U,a,C,\theta) \in E$ *avec* $isUrgent(a)$, *alors* $(s,\nu) \xrightarrow{\delta} (s,\nu+\delta)$.

Traduction de RT-LOTOS en DTA

Le cœur de l'algorithme de traduction des spécifications RT-LOTOS en automates temporisés DTA a été détaillé dans [CRdO95]. La technique employée est appelée *méthode par induction structurelle*. Plusieurs améliorations y ont étés apportés par la suite, telle que la suppression des actions *weak* (remplacées par l'explicitation des domaines d'urgence U), et l'optimisation du nombre d'horloges utilisées par chaque état de contrôle.

Intuitivement, l'algorithme consiste à enrichir l'arbre sémantique de la spécification RT-LOTOS avec des informations structurelles, puis à raffiner cette structure pour construire les transitions du DTA.

Dans un premier temps, l'algorithme va s'intéresser au sous-arbre induit par les compositions parallèle, et attribuer à chaque composant RT-LOTOS une référence, un couple d'entiers (n,m) qui peuvent être vus comme les coordonnées en profondeur et en largeur dans le sous-arbre induit par les compositions parallèles, et qui peut être attribué lors d'une exploration descendante de l'arbre sémantique. Ajoutons qu'à ce niveau, l'opérateur de préemption ([>) peut être vu comme une composition parallèle entre chacune des action du processus de gauche et le processus de droite. Cette référence permet de localiser la branche de composition parallèle à laquelle appartient chaque composant RT-LOTOS. Cette référence permet également de définir une *horloge virtuelle*, c'est-à-dire une horloge qui sera recopiée ou ré-initialisée suivant les besoins, et sur laquelle seront définies les diverses contraintes temporelles qui portent sur les composants de la branche de composition parallèle. À une profondeur donnée du sous-arbre des compositions parallèles, il y a autant d'horloges virtuelles que de branches de composition parallèle. Cet ensemble d'horloges, noté ϕ, est appelé le *déterminant* de la *configuration structurelle* englobant toutes ces compositions parallèles.

Dans un deuxième temps, l'algorithme construit un système de transitions entre ce qui est appelé des *configurations structurelles*, par une exploration remontante de l'arbre sémantique. Ce système de transitions comporte les informations structurelles suivantes :

- K, l'ensemble des contraintes temporelles de sensibilisation conditionnant l'occurrence de l'action g;
- g, l'étiquette de l'action de la transition;
- C, le déterminant de la configuration atteinte par la transition.

Lorsque l'algorithme rencontre l'opérateur de préfixe par une action «g`{`t`};` P», il initialise K avec une contrainte définie sur l'horloge virtuelle de la branche, et bornée par

l'offre limitée dans le temps définie sur l'action; il initialise également C avec le déterminant de P.

En remontant l'arbre sémantique, l'algorithme applique diverses modifications suivant l'opérateur RT-LOTOS qu'il rencontre. Nous précisons ici les opérations essentielles suivantes :

- Lorsque l'algorithme rencontre l'opérateur de composition parallèle, s'il y a une synchronisation sur g, il fait la conjonction des contraintes K, et l'union des déterminants C.
- Lorsque l'algorithme rencontre l'opérateur de délai Δ^t, il décale les contraintes de K de t unités de temps.
- Lorsque l'algorithme rencontre l'opérateur de latence Ω^t, les choses sont un peu plus compliquées. Dans sa première version, l'algorithme définissait des actions *weak* : la transition était alors dupliquée, l'une réalisant une action déclarée *weak* avec une condition K dont la borne supérieure était la latence t (borne exclue), et l'autre réalisant une action *strong* avec une condition K dont la borne inférieure était la latence t (borne incluse).

  Dans sa deuxième version, l'algorithme que nous proposons ici, ne duplique plus les actions concernées par la latence, mais il en conserve une seule, sans modifier la condition K, mais en introduisant un domaine d'urgence U (la sémantique du `DTA` a été, depuis, adaptée en conséquence) : U est défini de la même manière que la condition K dans le cas d'une action *strong*, c'est-à-dire en reprenant la condition K et en fixant sa borne inférieure à la latence t (borne incluse).

Notons que ce système de transitions s'appuie sur une structure arborescente (l'arbre sémantique), dont certaines branches sont repliées lors des instanciations de processus, et lors de l'utilisation de l'opérateur de séquence.

La méthode par induction structurelle détaillée dans [CRdO95] pour traduire les spécifications RT-LOTOS en automates temporisés `DTA` donne la sémantique opérationnelle des configurations structurelles. Il apparaît alors que cette sémantique opérationnelle calque la sémantique opérationnelle de RT-LOTOS et l'enrichit pour construire notamment la condition K. Ainsi, la règle qui traite l'opérateur de latence et qui définit deux actions *strong* et *weak* avec les domaines K décrits dans le paragraphe précédent, s'écrit :

$$\frac{\mathcal{P} \xrightarrow{K,g,C} \mathcal{P}'}{\Omega^u\mathcal{P} \xrightarrow{K',g_w,C} \mathcal{P}'},\ K' = K \otimes [0,u[\qquad \frac{\mathcal{P} \xrightarrow{K,g,C} \mathcal{P}'}{\Omega^u\mathcal{P} \xrightarrow{K',g_s,C} \mathcal{P}'},\ K' = K \otimes [u,\infty[$$

l'opérateur $\otimes$ réalisant la conjonction des domaines temporels.

Le paragraphe précédent décrit une amélioration du traitement de la latence pour supprimer la différenciation entre les actions *strong* et les actions *weak*. La formalisation de cette amélioration consiste à remplacer cette règle qui traite l'opérateur de latence dans la sémantique opérationnelle des configurations structurelles par une règle construisant le domaine d'urgence U. Cette nouvelle règle s'écrit :

$$\frac{\mathcal{P} \xrightarrow{K,g,C} \mathcal{P}'}{\Omega^u \mathcal{P} \xrightarrow{K,U,g,C} \mathcal{P}'}, U = K \otimes [u,\infty[$$

Il convient également d'enrichir la règle traitant du délai déterministe pour translater également le domaine U du délai spécifié, tout comme cela est fait pour le domaine K :

$$\frac{\mathcal{P} \xrightarrow{K,U,g,C} \mathcal{P}'}{\Delta^u \mathcal{P} \xrightarrow{K',U',g,C} \mathcal{P}'}, K' = shift(K,u), U' = shift(U,u)$$

Pour finir, l'algorithme raffine le système de transitions construit à l'étape précédente pour produire les transitions du `DTA`. Les états de contrôle du `DTA` sont identifiés par des entiers naturels quelconques, mais uniques. La fonction de réinitialisation `C` des horloges du `DTA` correspond directement à l'ensemble C construit. La fonction de recopie `theta` des horloges du `DTA` est définie par le déterminant ϕ de la configuration structurelle atteinte, privée de l'ensemble C. Toutes les horloges virtuelles présentes dans le déterminant de la configuration structurelle de départ de la transition sont indexées par des entiers naturels uniques et consécutifs. Une optimisation a été proposée à ce niveau : en effet, il apparaît que certaines horloges ainsi construites, ne sont jamais concernées par les contraintes temporelles de K et U, elles sont juste initialisées à 0 par les fonctions `C` et recopiées par les fonctions `theta`. Dès lors, elles n'ont aucune incidence sur le fonctionnement de l'automate. L'optimisation consiste donc à supprimer ces horloges inutilisées. En réduisant le nombre d'horloges, on accroît d'autant l'efficacité de l'analyse d'accessibilité.

La figure 1.23 illustre les deux sémantiques du `DTA` vis-à-vis de l'expression de la latence : à gauche, l'approche initiale formalisée dans [CRdO95] (l'idée était d'associer deux transitions lorsqu'il est question de la latence d'une action : une transition avec une action marquée *weak* au moyen du préfix «`w-`», et valide dans la région qui précède strictement la zone de latence, et une transition avec une action *strong*, i.e. sans le préfix «`w-`», et valide au-delà); à droite, l'approche présentée dans ce chapitre, qui ne duplique plus les transitions, mais les enrichit d'une contrainte temporelle d'urgence U.

La figure 1.24 illustre deux manières de générer le `DTA` depuis une spécification RT-LOTOS : à gauche, l'approche classique telle que formalisée dans [CRdO95] (l'idée est d'associer systématiquement une horloge à chaque composante parallèle de la spécification); à droite l'approche optimisée permettant de réduire le nombre d'horloges. Les comportements décrits par ces deux automates (ainsi que les graphes d'accessibilité) sont strictement identiques.

1.3.3 Analyse d'accessibilité

Les divers algorithmes de vérification des automates temporisés dépendent fortement du nombre d'horloges, avec une complexité en $O(n^3)$, où n est le nombre d'horloge [YL93]. C'est pourquoi les automates temporisés de type `DTA` sont particulièrement attractifs : le nombre d'horloges utilisées est réduit au minimum. Cela est dû, d'une part, à la structure dynamique de l'automate (les horloges sont locales à chaque état de contrôle, et en

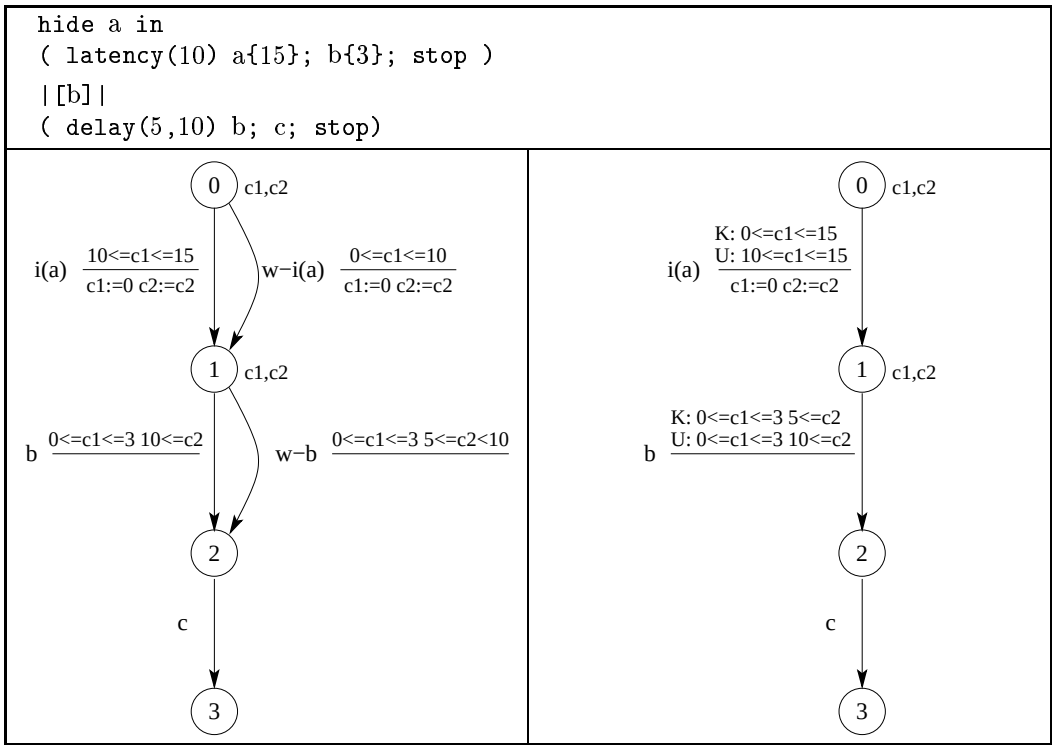


FIG. 1.23 – *L'urgence dans le* `DTA` *avec actions* weak *ou avec domaine temporel U*

nombre inférieur ou égal à celui des horloges globales), et d'autre part, à la technique de suppression des horloges inutilisées.

Un état global, ou une configuration, du système temporisé consiste en un état de contrôle (du `DTA` en l'occurrence) et une valuation des horloges associées à cet état de contrôle. Le domaine temporel étant dense, il y a une infinité de configurations. Une analyse finie d'un tel système requiert une partition de l'espace des configurations en un nombre fini de régions. Des algorithmes réalisant simultanément l'analyse d'accessibilité et la minimisation du système de transitions temporisé ont été proposés dans [BD91, ACH92, YL93]. Ce dernier algorithme a été adapté au `DTA` et implémenté dans l'outil `rtl`. Le graphe résultant est un *graphe minimal d'accessibilité* où :

- Un *nœud* (appelé également une classe) est défini par un état de contrôle et une région, représentée par un polyèdre convexe de dimension égale au nombre d'horloges de l'état de contrôle. Les configurations appartenant à une même classe ont les mêmes propriétés d'accessibilité, dans le sens où elles ne peuvent être distinguées en terme de séquences d'occurrence d'actions futures. Ainsi, une classe correspond à une représentation finie d'un nombre infini de configurations. Dans ce mémoire nous parlerons indistinctement de classe (de configurations) et de régions (d'horloges, associées à un état de contrôle).
- Un *arc* indique soit l'occurrence d'une action discrète RT-LOTOS, soit une progression

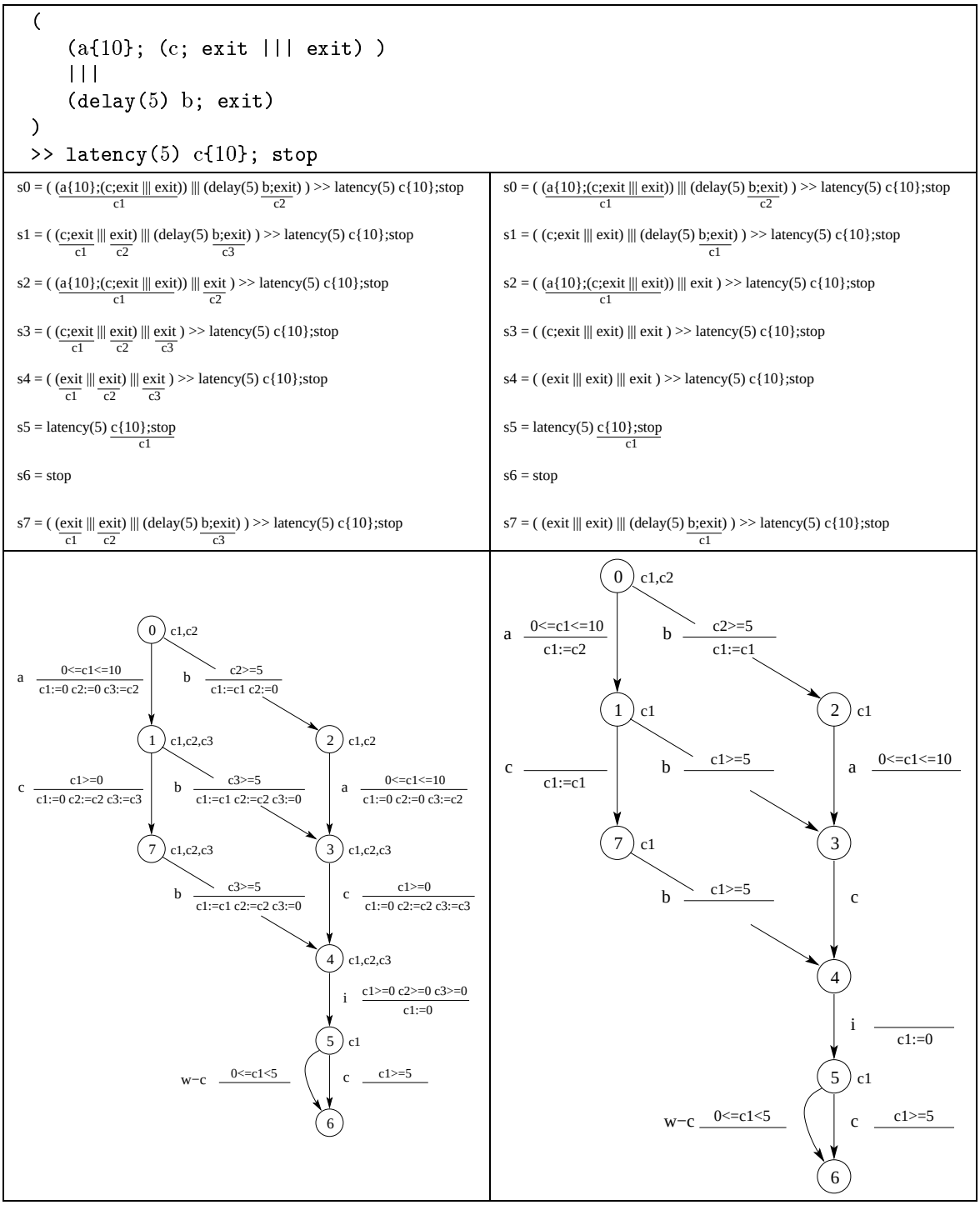


FIG. 1.24 – *Optimisation des horloges du* `DTA`

temporelle.

Cette approche, appliquée à la spécification RT-LOTOS dont la traduction en DTA est donnée par la figure 1.22, conduit au graphe d'accessibilité représenté par la figure 1.25. Comme attendu, ce graphe est très simple, et contient seulement 6 classes associées aux 3 états de contrôle accessibles du DTA. Notons que différentes classes peuvent être construites sur un même état de contrôle du DTA (3 classes sont associées à l'état de contrôle 1). Dans le cas général, des états de contrôle du DTA non accessibles peuvent exister, ce qui signifie qu'il n'y a pas de classe associée dans le graphe d'accessibilité final. Si le temps progresse à l'intérieur d'une classe, cette classe contient un nombre infini d'éléments (s,ν), car le domaine temporel est dense. Si une classe est marquée urgente (i.e. le temps ne peut pas progresser à l'intérieur de celle-ci), alors la classe a seulement un nombre fini d'éléments correspondant au nombre d'arcs entrant dans cette classe.

La représentation textuelle de la figure 1.25a montre le fichier généré par `rtl`. La figure 1.25b illustre la même information sous la forme d'un graphe, et les régions associées à l'état de contrôle 1 du DTA sont détaillées dans la figure 1.25c. Sur ce le graphe d'accessibilité, les notations suivantes ont été adoptées :

- x-$(n\ m)$ identifie la classe associée à l'état de contrôle x du DTA, et $(n\ m)$ caractérise les valeurs des horloges d'une configuration accessible appartenant à cette classe ; cette configuration accessible est appelée la *configuration représentative* de la classe et est choisie par l'alorithme de construction du graphe d'accessibilité pour identifer la classe.
- x-$(n\ m)$ dans un cercle grisé représente une classe urgente.
- Les nœuds regroupés dans une même boite rectangulaire correspondent à des classe appartenant à un même état de contrôle du DTA.
- Une transition étiquetée par `t` représente une progression temporelle. La valeur effective de cette durée dépend de la configuration de la classe à l'origine de la transition, et par conséquent, ne peut pas être explicitée globalement au niveau de la transition entre classes.

Une conséquence de l'algorithme de minimisation implémenté dans `rtl` (adapté de [YL93]), est que toutes les configurations d'une classe ne sont pas nécessairement accessibles. Une configuration au moins est effectivement accessible. Par exemple, considérons la classe `1-(0 0)` de la figure 1.25c. Seules les configurations appartenant à la première bissectrice de la région sont effectivement accessibles. L'algorithme de minimisation proposé dans [YL93] permet de considérer des régions plus grandes que celles requises du point de vue de l'accessibilité stricte, et ainsi de réduire le nombre de régions par rapport à d'autres algorithmes comme ceux proposés dans [BD91, ACH92].

i) Du DTA au graphe minimal d'accessibilité

Soit DTA $= (S,Nclock,E,s_0)$ un automate temporisé dynamique. Une partition de l'état de contrôle $s \in S$ est un ensemble $\pi_s = \{\{s\} \times Z_i \mid i \in I \text{ fini}\}$ où $\{Z_i \mid i \in I\}$ est une partition de $D_0^{Nclock(s)}$. Nous considérons que chaque élément Z_i est un polyèdre convexe

a) Représentation Textuelle

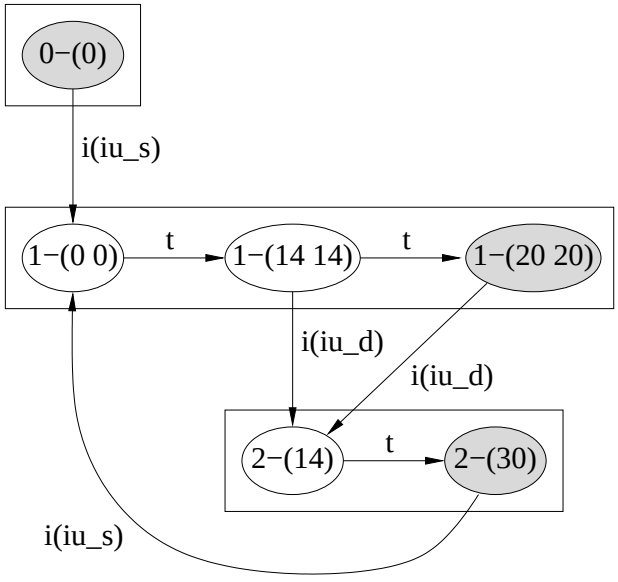
Définition des régions

```
0-(0) URG c1=0
1-(20 20) URG 0<=c1<=30 c2>=20 c2-c1>-10
1-(14 14) 0<=c1<30 14<=c2<20 -10<c2-c1<20
1-(0 0) 0<=c1<=24 0<=c2<14 -10<c2-c1<14
2-(30) URG c1=30
2-(14) 0<=c1<30
```

Définition du graphe

```
( 0-(0), i(iu_s), 1-(0 0) )
( 1-(20 20), i(iu_d), 2-(14) )
( 1-(14 14), i(iu_d), 2-(14) )
( 1-(14 14), t, 1-(20 20) )
( 1-(0 0), t, 1-(14 14) )
( 2-(30), i(iu_s), 1-(0 0) )
( 2-(14), t, 2-(30) )
st0=1 st1=3 st2=2
6 classes
3 reachable DTA states
1cl(2st) 2cl(1st)
```

b) Graphe d'accessibilité



c) Régions de l'état de contrôle 1

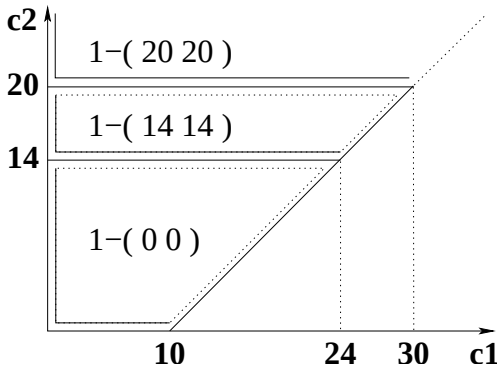


Fig. 1.25 – *Exemple de graphe minimal d'accessibilité*

décrit par une conjonction finie d'inégalités comparant une horloge ou la différence de deux horloges à deux constantes entières. Une partition π du DTA est une union des partitions π_s de chaque état de contrôle $s \in S$.

Considérons la configuration $(s,\nu) \in B = \{s\} \times Z_i$, et la «trajectoire temporelle» $(s,\nu+\delta)$ lorsque δ croit de 0 à ∞. Si la trajectoire sort de B, soit alors $C = \{s\} \times Z_j \in \pi_s$ la première classe atteinte par la trajectoire. La classe C est appelée alors le *successeur temporel immédiat* de la configuration (s,ν).

Une partition π induit un système de transitions *quotient* $LTS(\mathtt{DTA}/\pi)$ dont les noeuds sont les classes de π. Un arc étiqueté par $a \in L$ de la classe B à la classe C est défini si, pour des configurations (s,ν) de B, nous avons $(s,\nu) \xrightarrow{a} (s',\nu') \in LTS(\mathtt{DTA})$. Un arc étiqueté par t de la classe B à la classe C est défini si des configurations de B ont C comme *successeur temporel immédiat*. Un arc $B \xrightarrow{a} C$ est dit *stable* si tous les états de B ont une transition par a vers un état de C. De même, un arc $B \xrightarrow{\mathtt{t}} C$ est dit *stable* si tous les états de B ont la classe C comme successeur temporel immédiat. Une classe B est stable si tous ces arcs sont stables. Une partition π est stable si toutes ses classes sont stables.

Une partition π non stable peut être raffinée pour construire une partition stable brute π'. Cette partition stable induit un système de transitions *minimisé* $LTS(\mathtt{DTA}/\pi')$. Typiquement, nous nous intéressons exclusivement aux états du $LTS(\mathtt{DTA})$ accessibles depuis l'état initial s_0. Les classes de π' contenant ces états là sont justement les classes du $LTS(\mathtt{DTA}/\pi')$ accessibles depuis la classe initiale (i.e. celle contenant s_0). Le sous-graphe du $LTS(\mathtt{DTA}/\pi')$ induit par ces nœuds est appelé le *graphe minimal d'accessibilité*.

ii) Vérification à la volée

Un autre avantage du DTA est que, les horloges étant locales à chaque état de contrôle, il est possible de réaliser l'analyse d'accessibilité au fur et à mesure de la construction du DTA. On parle alors de vérification *à la volée* des spécifications RT-LOTOS.

```
hide a, b in

P1[a](1)
|[a]|
P2[a,b](0)

P1[a](per:time) = delay(per) a; P1[a](per)

P2[a,b](n:nat) = ( a; P2[a,b](n+1) )
                 []
                 [n>0] -> ( b; P2[a,b](n-1) )
```

FIG. 1.26 – *Spécification vérifiable uniquement* à la volée

La spécification RT-LOTOS illustrée dans la figure 1.26 est un cas pathologique qui ne peut être analysé qu'à la volée. En effet, l'analyse d'accessibilité va guider la construction

du `DTA` : les états du `DTA` qui ne sont pas accessibles ne sont pas développés. Dans le cas présent, le processus `P2` est paramétré par une variable entière. L'algorithme de génération du `DTA` devrait a priori explorer tout l'espace d'états et chercher à construire toutes les instances possibles du processus `P2` (i.e. pour chaque valeur possible de `n:nat`), ce qui est naturellement impossible car cet espace est de taille infinie. Par contre, en réalisant l'analyse à la volée, seuls les états accessibles depuis l'état initial sont explorés. En l'occurrence, énumérer les différentes valeurs prises par `n` est sans intérêt, seules deux situations sont pertinentes : lorsque $n = 0$, et lorsque $n > 0$. La technique de vérification à la volée de spécification RT-LOTOS est détaillée dans [CRdO95].

iii) Utilisation de l'analyse d'accessibilité

Le graphe minimal d'accessibilité peut être exploité de plusieurs manières :

- L'auteur de la spécification RT-LOTOS peut éventuellement ne rechercher que l'occurrence d'une action ou d'une série d'actions caractéristiques de la validité du système spécifié. Une lecture directe du graphe minimal d'accessibilité (aidée éventuellement par des outils de parcours de graphes, de projection des actions, de minimisation, etc.) peut lui permettre de conclure. Cette technique peut être étendue avec l'emploi d'observateurs.
- L'approche par observateurs est fréquemment utilisée en association avec l'analyse d'accessibilité. Un observateur est un processus composé avec la spécification, qui ne doit pas interférer avec elle, mais uniquement observer son comportement vis-à-vis d'une propriété à vérifier et déclencher une action `erreur` lorsqu'il détecte une violation de la propriété. La propriété est *vraie* si l'action `erreur` n'est pas accessible depuis l'état initial de la spécification composée, en d'autres termes, si elle n'est pas présente dans le graphe minimal d'accessibilité associé.
- Le graphe minimal d'accessibilité exprime toutes les configurations accessibles du système. Il est très riche en informations temporelles et logiques sur le comportement du système. Le chapitre 2 exposera une technique permettant d'exploiter ces informations, d'effectuer des calculs des durées entre actions, et au final, de construire un automate temporisé d'ordonnancement permettant d'implémenter le système spécifié.

1.3.4 Model-checking

Le contrôle de modèle (*model-checking*) est une technique de vérification automatique de systèmes à états finis. Les algorithmes de model-checking déterminent, par une analyse issue de la théorie des graphes, les états qui satisfont une formule dans un espace d'état [CES86, MOSS99]. Dans le cas considéré ici, la technique qui consiste à prouver des formules de logique temporelle sur un automate temporisé [AD94]. Cette technique possède deux avantages : la construction du graphe d'accessibilité n'est plus exhaustive mais partielle, guidée par la formule à vérifier ; par ailleurs, le model-checking est une méthode possédant un grand pouvoir d'expression, et qui a été améliorée au cours des dernières années. La

logique temporelle quantitative TCTL [ACD90, HNSY94] a été introduite pour exprimer les propriétés représentant des aspects temporels. TCTL étend notamment les modalités $\exists U$ (il existe une exécution) et $\forall U$ (pour toute exécution) de CTL [EC82, CES86] tout en faisant référence de manière explicite au temps.

Les algorithmes du model-checking déterminent les états qui satisfont une formule de logique temporelle en utilisant une analyse basée sur la théorie des graphes de l'espace d'états. Pour remédier au problème de l'explosion d'états, cette technique a été étendue en utilisant une approche basée sur la représentation symbolique des états.

Le paradigme de vérification adopté et implémenté dans la majorité des outils supportant la technique du model-checking consiste à modéliser le système par un automate temporisé, exprimer la propriété par une formule de la logique temporelle quantitative TCTL, et vérifier la validité de la formule sur l'automate temporisé.

L'approche choisie pour RT-LOTOS n'a pas été de re-développer un model-checker spécifique, mais de réutiliser des outils existants, en l'occurrence l'outil `Kronos` [HNSY94, Kro] (développé à l'IMAG) qui a été l'un des premiers outils a avoir implémenté la technique du model-checking; il a été utilisé avec succès dans la vérification de plusieurs systèmes temps-réel. L'approche présentée ici permet donc de tirer parti du pouvoir d'expression du langage RT-LOTOS : l'automate temporisé étant généré automatiquement avec l'outil `rtl`, il n'est plus nécessaire d'énumérer à la main, ni le nombre d'états du système, ni les différentes transitions entre ces différents états, et de bénéficier du model-checking (certaines propriétés temporelles de vivacité ne peuvent pas en effet être exprimées à l'aide d'observateurs).

L'architecture de `Kronos` est donnée dans la figure 1.27. La propriété temporelle exprimée en TCTL et l'automate temporisé sont des entrées pour ce model-checker qui évalue sa véracité et qui peut fournir la trace des séquences d'états menant à la violation de la propriété.

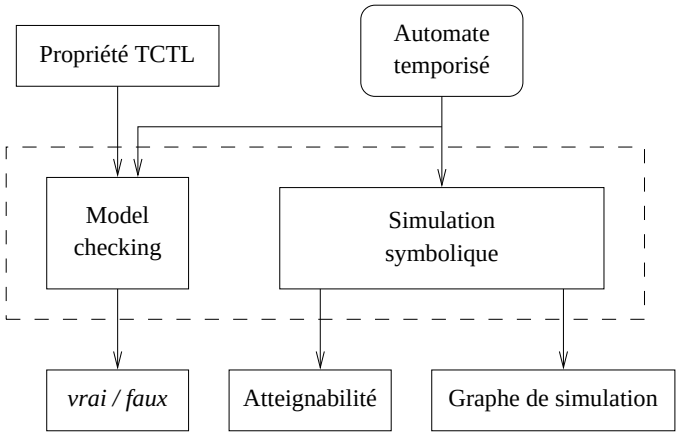


FIG. 1.27 – *Architecture générale de l'outil* `Kronos`

Le principe consiste à traduire les automates temporisés de type `DTA` en automates temporisés de type `Kronos`, et notés TA ici. Un outil `dta2kronos` a été conçu dans ce sens.

Ces deux types d'automates se distinguent principalement sur deux points : la localisation des horloges, et l'expression de l'urgence.

- Dans le modèle `Kronos`, les horloges sont globales au système, alors que dans le modèle `DTA` elles sont locales à chaque état de contrôle. Il faut donc réécrire les conditions temporelles des transitions du `DTA` en fonction d'un ensemble d'horloges globales (il s'agit principalement de tenir à jour une table de correspondance entre les horloges locales du `DTA` et les horloges globales `Kronos` en suivant les indications des fonctions `theta` de recopies d'horloges du `DTA`).
- Dans le modèle `Kronos`, l'urgence est caractérisée par une condition temporelle appelée *invariant* attachée à chaque état de contrôle de l'automate qui doit rester satisfaite lors d'une progression du temps à l'intérieur de l'état de contrôle. Dans le modèle `DTA`, l'urgence est caractérisée par une condition temporelle U placée sur chaque transition. Intuitivement, l'invariant est borné par la conjonction des domaines U.

Algorithme de traduction

Soit un `DTA` $(S,Nclock,E,s_0)$. Dans la syntaxe du `DTA`, les notations suivantes ont été utilisées :
Pour toute transition (s,s',K,U,a,C,θ) du système de transitions E, si K et U ne figurent pas dans la transition, alors ils sont considérés comme étant égaux tous les deux à $[0,\infty[$.

Par simplification, le domaine $[0,\infty[$ va être noté $TRUE$ car la valeur d'une horloge est toujours positive. Autrement dit, la proposition «*la valeur d'une horloge est positive*» est toujours vraie.

Avant de présenter l'algorithme de traduction, nous allons d'abord définir un opérateur appelé $Pred$ sur un domaine temporel A.

Définition 5 (Pred(A))
$Pred(A)$ est défini comme étant le domaine temporel qui précède le domaine A, inclus les frontières gauches de A. Autrement dit, c'est le domaine dont les valeurs d'horloges sont inférieures ou égales à celles du domaine A.

Exemples :

- $Pred([1,5]) = [0,1]$
- $Pred(\{1 \leqslant c_1 \leqslant 3,\ 4 \leqslant c_2\}) = \{c_1 \leqslant 1,\ c_2 \leqslant 4\}$ et qui peut être interprété dans ce cas comme une disjonction des contraintes qui le composent. C'est-à-dire $c_1 \leqslant 1 \vee c_2 \leqslant 4$.

La traduction d'un `DTA` $= (S,Nclock,E,s_0)$ en automate temporisé $TA = (S^{TA},X,E^{TA},I,s_0^{TA})$ équivalent se fait en appliquant l'algorithme suivant :

1. L'ensemble des états de contrôle $S^{TA} = S$
2. L'état de contrôle initial $s_0^{TA} = s_0$
3. L'ensemble des horloges est $X = \{c_i \mid 1 \leqslant i \leqslant N\}$ où $N = \max\{Nclock(s) \mid s \in S\}$.

4. Pour toute transition (s,s',K,U,a,C,θ) de E, on construit la transition $(s^{TA},L,\psi,\rho,s'^{TA})$ de l'automate temporisé de cette façon :
   (a) L'états de contrôle de source $s^{TA} = s$
   (b) L'états de contrôle de destination $s'^{TA} = s'$
   (c) L'étiquette d'action $L = \{a\}$
   (d) La condition de sensibilisation de la transition $\psi = K$
   (e) La valuation ρ (copie et initialisation des horloges) est construite de la manière suivante :

$$\rho : X \rightarrow X^*$$
$$\rho(x) = \begin{cases} \theta^{-1}(x) & si\ x \in Im(\theta) \\ 0 & sinon\ (c'est - \acute{a} - dire\ si\ x \in X - Im(\theta)) \end{cases}$$

5. Pour tout état s de l'ensemble d'états S^{TA}, l'invariant $I(s)$ est calculé par l'algorithme suivant :
   (a) Pour toute transition étiquetée par a et partant de l'état s faire :
   Si $isUrgent(a)$ alors :
   Si U est non vide alors $I(s) \leftarrow Pred(U)$
   Sinon $I(s) \leftarrow TRUE$
   Sinon $I(s) \leftarrow TRUE$
   (b) L'invariant global associé à l'état s est la conjonction de tous les invariants calculés pour chaque transition a de L.

Le `DTA` sera transformé en automate temporisé équivalent. L'équivalence est ici une bissimulation forte. Cette équivalence préservera les propriétés temporelles à vérifier. Le tableau 1.28 illustre quelques exemples élémentaires de transformation d'automates.

Lors de la construction de l'automate temporisé TA associé au `DTA`, certains étapes sont évidentes, d'autres non. Ainsi, l'association : entre les ensembles d'états de contrôle (1) des deux automates, entre les états de contrôle initiaux respectifs (2), entre les états de contrôle sources et destinations (4.a) et (4.b), entre les étiquettes d'actions (4.c), et entre les contraintes de sensibilisation des transitions (4.d) ont été triviales. Par contre, les constructions (3), (4.e), et (5) nécessitent d'être justifiées; voir l'annexe A.

Mise en œuvre : Formalisation d'un croisement ferroviaire

Cet exemple décrit un contrôleur automatique qui ouvre et ferme une barrière à un croisement de chemin de fer. Il a été présenté initialement dans [ACHWT92]. La spécification en langage naturel reprise ici est tirée de [KLK00].

Le système correspond à la composition de trois composants : *Train*, *Gate*, et *Controller*, qui s'exécutent en parallèle et se synchronisent par les événements : `approach`, `leave`, `lower` et `down`.

- Lorsqu'un train est en approche du croisement, *Train* envoie un signal `approach` au *Controller* et entre dans le croisement au moins 300 secondes plus tard. Quand

Comportement RT-LOTOS et DTA associé	État 0 de l'automate temporisé généré
a; stop 2 states, 1 arcs (0, a, 1)	invar: TRUE trans: TRUE => a; C1:=0; goto 1
hide a in a; stop 2 states, 1 arcs (0, i(a), 1)	invar: C1=0 trans: TRUE => a; C1:=0; goto 1
delay(3) a; stop 2 states, 1 arcs (0, a K={3<=c1} U={3<=c1}, 1)	invar: TRUE trans: C1>=3 => a; C1:=0; goto 1
hide a in delay(3) a; stop 2 states, 1 arcs (0, i(a) K=3<=c1 U=3<=c1, 1)	invar: C1<=3 trans: C1>=3 => a; C1:=0; goto 1
delay(5,10) a{6}; stop 2 states, 1 arcs (0, a K={5<=c1<=11} U={10<=c1<=11}, 1)	invar: TRUE trans: C1>=5 and C1<=11 => a; C1:=0; goto 1
hide a in delay(5,10) a{6}; stop 2 states, 1 arcs (0, i(a) K={5<=c1<=11} U={10<=c1<=11}, 1)	invar: C1<=10 trans: C1>=5 and C1<=11 => a; C1:=0; goto 1
hide a in (delay(4) a2; stop) |[a]| (delay(5,10) a6; stop) 2 states, 1 arcs (0, i(a) K={4<=c1<=6,5<=c2<=11} U={4<=c1<=6,10<=c2<=11}, 1)	invar: C1<=4 or C2<=10 trans: C1>=4 and C1<=6 and C2>=5 and C2<=11 => a; C1:=0, C2:=0; goto 1

FIG. 1.28 – *Exemples de calcul d'invariant pour quelques comportements* RT-LOTOS

un train quitte le croisement, *Train* envoie un signal `leave` au *Controller*. Le signal `leave` est envoyé dans les 500 secondes qui suivent le signal `approach`.

- *Controller* envoie un signal `lower` à *Gate* exactement 100 secondes après le signal `approach` et envoie un signal `raise` dans les 100 secondes qui suivent `exit`.
- *Gate* répond par `down` au signal `lower` dans les 100 secondes, et répond par `up` au signal `raise` entre 100 et 200 secondes.

Chaque entité (*Train*, *Gate*, et *Controller*) peut être représentée par un processus RT-LOTOS dont le comportement implémente les contraintes énoncées précédemment. Il y a alors deux approches possibles : soit réaliser la synchronisation de ces trois entités avec `Kronos` après avoir spécifié séparément ces trois processus RT-LOTOS et engendré les trois automates temporisés correspondants, ou bien décrire en RT-LOTOS la composition de ces trois processus puis engendrer l'automate temporisé global. La figure 1.29 donne la spécifi-

```
specification Railroad_Crossing : exit

type natural is boolean
  sorts nat
  opns + : nat,nat->nat
       - : nat,nat->nat
endtype

behaviour
 hide approach, leave, lower, raise, into, outto, down, up in

  ( Train[approach,leave,into,outto]
    |[approach,leave]|
    Controller[approach,lower,leave,raise] )
  |[lower,raise]|
  Gate[lower,raise,down,up]

where

  process Train[approach,leave,into,outto] : noexit :=
    approach;
    ( ( into; outto; leave; exit )
      |[into,outto,leave]|
      ( ( delay(300,500) into{200}; exit )
        |||
        ( delay(300,500) outto{200}; exit )
        |||
        ( delay(300,500) leave{200}; exit ) ) )
    >> Train[approach,leave,into,outto]
  endproc

  process Controller[approach,lower,leave,raise] : noexit :=
    approach; delay(100) lower;
    leave; latency(100) raise{100};
    Controller[approach,lower,leave,raise]
  endproc

  process Gate[lower,raise,down,up] : noexit :=
    lower; latency(100) down{100};
    raise; delay(100,200) up{100};
    Gate[lower,raise,down,up]
  endproc

endspec
```

FIG. 1.29 – *Spécification* RT-LOTOS *du croisement ferroviaire*

```
#states 4
#trans 4
#clocks C1 C2 C3
#sync approach leave

state: 0
prop: STATE0
invar: C1=0 or C2=0 or C3=0
trans:
TRUE => approach;
  C1:=0, C2:=0, C3:=0; goto 1

state: 1
prop: STATE1
invar: C1<=500
trans:
C1>=300 and C1<=500 => into;
  C1:=C2, C2:=C3, C3:=0; goto 2

state: 2
prop: In_Gate
invar: C1<=500
trans:
C1>=300 and C1<=500 => outto;
  C1:=C2, C2:=0, C3:=0; goto 3

state: 3
prop: STATE3
invar: C1<=500
trans:
C1>=300 and C1<=500 => leave;
  C1:=0, C2:=0, C3:=0; goto 0
```

```
#states 4
#trans 4
#clocks C4
#sync approach lower leave raise

state: 0
prop: STATE0
invar: C4=0
trans:
TRUE => approach; C4:=0; goto 1

state: 1
prop: STATE1
invar: C4<=100
trans:
C4>=100 => lower; C4:=0; goto 2

state: 2
prop: STATE2
invar: C4=0
trans:
TRUE => leave; C4:=0; goto 3

state: 3
prop: STATE3
invar: C4<=100
trans:
C4<=100 => raise; C4:=0; goto 0
```

```
#states 4
#trans 4
#clocks C5
#sync lower raise

state: 0
prop: STATE0
invar: C5=0
trans:
TRUE => lower; C5:=0; goto 1

state: 1
prop: STATE1
invar: C5<=100
trans:
C5<=100 => down; C5:=0; goto 2

state: 2
prop: Gate_Down
invar: C5=0
trans:
TRUE => raise; C5:=0; goto 3

state: 3
prop: STATE3
invar: C5<=200
trans:
C5>=100 and C5<=200 => up;
      C5:=0; goto 0
```

Fig. 1.30 – *Automates temporels issus des processus* Train, Gate, *et* Controller

cation RT-Lotos globale de ce système. La figure 1.30 présente les automates temporisés engendrés séparément pour les processus *Train*, *Gate* et *Controller*, leur synchronisation étant réalisée ensuite par l'outil `Kronos`.

Il est alors possible d'utiliser l'outil `Kronos` pour vérifier que ce système respecte bien certaines propriétés. Par exemple, nous voulons vérifier que «*lorsqu'un train est dans le croisement, la barrière est baissée.*» Cela s'exprime par la formule Tctl suivante :

$$init \Rightarrow \forall\Box(In\_Gate \Rightarrow Gate\_Down)$$

Le model-checker `Kronos` nous assure que c'est bien le cas.

1.4 Expériences avec RT-Lotos

RT-Lotos a été utilisé au sein du projet CESAME (projet CNET-CNRS sur la conception formelle de réseaux haut débit multimédia) pour la spécification et l'évaluation de protocoles de synchronisation multimédia. Des résultats caractérisant la qualité de service de différents mécanismes de synchronisation ont été obtenus par simulation intensive via `rtl`. Parmi ces mécanismes nous pouvons citer des protocoles de synchronisation intra-flux et inter-flux [CdCCdO96].

RT-Lotos a été utilisé dans le cadre de deux conventions de recherche établies avec la DER de EDF pour l'évaluation d'une fonction critique d'un système de contrôle-commande en centrale. L'application a consisté à réaliser la rétro-ingénierie de cette fonction critique (un système de monitoring redondé mis en œuvre par deux calculateurs interconnectés par un réseau à haut débit), à faire une spécification RT-Lotos détaillée des mécanismes de

tolérance aux fautes mis en œuvre et à valider cette spécification. Différentes simulations de la spécification complète, ainsi que des vérifications formelles de quelques sous-ensembles de cette spécification, ont été réalisées sur des jeux de paramètres temporels, conduisant à l'identification de certains scénarios anormaux, dont certains ont amené à des modifications du système analysé [ACdOP97].

RT-LOTOS est actuellement utilisé pour formaliser le processus de conception de documents multimédia interactifs de manière à valider (simuler et/ou vérifier de manière exhaustive) la cohérence temporelle de ces documents [CdO96]. L'approche développée consiste, partant d'un modèle auteur de haut niveau, à engendrer de manière automatique une spécification RT-LOTOS décrivant la dynamique des contraintes temporelles, puis à utiliser l'analyse d'accessibilité de `rtl` pour valider la cohérence du document et générer un automate d'ordonnancement permettant de réaliser la présentation du document à un utilisateur [SLC01].

1.5 Conclusion

Ce chapitre a proposé un éventail de méthodes formelles pour la spécification et la validation de systèmes temps-réel : les extensions temporisées des réseaux de PETRI, les automates temporisés, et les algèbres de processus. Parmi les algèbres de processus, nous nous sommes particulièrement arrêtés sur le langage LOTOS et les extensions temporisées qui ont vu le jour. Dans ce cadre, nous avons présenté le langage de spécification formel RT-LOTOS, extension temporisée à l'algèbre de processus LOTOS permettant de quantifier des contraintes temporelles. Parmi les divers aspects abordés, un point notable de l'approche RT-LOTOS est l'opérateur de *latence* qui offre une expression et un traitement judicieux du non-déterminisme temporel.

Ce chapitre a également exposé les technique mises en œuvre pour la validation de spécifications RT-LOTOS, en terme de simulation et de vérification formelle par construction d'un graphe d'accessibilité et par *model-checking*. L'approche traditionnelle consiste, pour la simulation, à exécuter directement les actions sémantiques de la spécification RT-LOTOS pour en obtenir une trace d'exécution, et pour la vérification formelle, à traduire la spécification RT-LOTOS en un automate temporisé de type `DTA`, puis à construire un graphe de régions des configurations accessibles de cet automate.

Nous avons présenté trois contributions concernant l'automate temporisé `DTA`. Dans le but de mieux traiter la latence RT-LOTOS, nous avons proposé une modification de la sémantique du `DTA` ainsi que de l'algorithme qui l'engendre afin de ne plus faire apparaître de distinction entre les action *strong* et les actions *weak* et de réduire ainsi la taille du `DTA`. Nous avons également proposé des techniques centrées sur le `DTA`, d'une part pour réaliser des simulations rapides d'une spécification RT-LOTOS pouvant être traduite en un `DTA`, et d'autre part pour s'interfacer avec le *model-checker* `Kronos`.

Ce chapitre a retracé le contexte de nos travaux : la spécification des systèmes sujets à des contraintes temporelles et logiques au moyen de la technique de description formelle RT-LOTOS, et la validation de ces systèmes soit par simulation, soit par vérification for-

melle au moyen de l'analyse d'accessibilité et au moyen du model-checking. La vérification par model-checking est particulièrement ciblée et épurée, c'est-à-dire que cette technique analyse un système au regard d'une question formulée en logique temporelle et répond *oui* ou *non* (éventuellement avec un contre-exemple à l'appui). Par contre, la vérification par construction d'un graphe de régions requiert un travail supplémentaire pour analyser la réponse obtenue : bien souvent il s'agit simplement de rechercher des configurations dans le graphe d'accessibilité, mais il est également possible d'extraire bien d'autres informations du graphe d'accessibilité. Le chapitre suivant propose une réflexion dans ce sens, avec comme objectif de définir la cohérence d'un système temps-réel, et de proposer l'ordonnancement dans le temps de ses actions.

2

Ordonnancement temporel de systèmes cohérents

Nous traitons dans ce chapitre de la problématique de l'ordonnancement de systèmes temps-réel. La première section décrit les besoins que nous avons en matière d'ordonnancement et pointe des travaux qui apportent des éléments de réponse. La deuxième section présente formellement la solution retenue. La troisième section présente un algorithme pour l'engendrer de manière automatique. La quatrième section présente les moyens mis en oeuvre pour extraire les comportements cohérents d'un système. La cinquième section montre dans quels contextes nous pouvons employer l'ordonnancement tel que nous le concevons. Enfin, la dernière section conclut le chapitre.

2.1 Synthèse d'automates d'ordonnancement

La problématique de l'ordonnancement peut être appréhendée sous différentes vues : la vue spécifique de l'ordonnancement de tâches dans un contexte système d'exploitation ou matériel, ou alors la vue générale de l'ordonnancement d'actions élémentaires, dans un contexte quelconque.

L'ordonnancement de tâches a trait classiquement à la commutation entres tâches, entre flux d'exécution, décidée et réalisée par une entité spécifique d'ordonnancement; cette entité devant alors assurer certaines propriétés comme par exemple, une priorité entre les tâches, l'échéance des tâches, une politique d'ordonnancement, etc. Des travaux théoriques [Feh99, Die97, Die01, AGS00, KMH00, KMH01] étudient ce problème pour assurer une vérification formelle des propriétés du système à ordonnancer. L'approche consiste alors à spécifier dans un langage formel approprié au temps-réel : (i) les tâches à ordonnancer, (ii) des contraintes spécifiques au système, (iii) un automate qui va implémenter l'entité d'ordonnancement et

sa politique. Des techniques de vérification formelle sont alors employées sur la spécification globale.

Nous avons choisi d'aborder, au contraire, le problème de l'ordonnancement d'actions dans un cadre générique : le système et ses contraintes temporelles sont spécifiées dans un langage formel (RT-LOTOS); cette spécification est vérifiée formellement; les comportements indésirables (du point de vue du contexte dans lequel la méthode est instanciée) et exhibés par la vérification formelle, peuvent êtres supprimés; et finalement nous proposons un automate qui réalise l'ordonnancement des actions dans le temps du système validé.

2.1.1 Motivation pour un automate d'ordonnancement

Le langage RT-LOTOS permet de spécifier, à un haut niveau d'abstraction, un système contraint par le temps sous la forme d'une composition de contraintes temporelles (voir le chapitre 1). Depuis cette spécification, nous sommes à même :

- de prouver formellement (en vérifiant des formules de logique temporelle (sous-section 1.3.4) ou en construisant le graphe des configurations accessibles (sous-section 1.3.3)) que le système satisfait les propriétés requises;
- d'exhiber l'ensemble des comportements *cohérents* possible du système (i.e. les comportements jugés satisfaisants au regard de la définition retenue pour la cohérence pour un domaine d'application particulier, voir section 2.4).

Fort de ces résultats, une fois la validité de la spécification du système assurée, apparaît le besoin de dériver un objet exprimant de manière opérationnelle les comportements cohérents du système étudié, ou, plus simplement, les suites d'actions réalisables par le système, ainsi que les relations logiques et temporelles entre ces actions.

Par exemple, dans le domaine multimédia, après avoir décrit un document et prouvé sa cohérence temporelle et logique (voir [SSC99, SC00, SC01, SLC01]), nous souhaitons réaliser sa présentation (voir section 2.5). Nous avons donc besoin ici d'un objet décrivant les comportements cohérents de la présentation du document et qui soit directement utilisable par un logiciel de présentation de document multimédia.

Parmi les objets existants dans le monde informatique, l'automate paraît particulièrement adapté pour exprimer simplement une suite d'actions à réaliser. L'automate requis doit pouvoir exprimer globalement les contraintes temporelles que le système doit satisfaire. Il doit exprimer de manière aussi simple que possible :

- l'évolution du temps : une seule horloge marque la progression du temps;
- les relations de dépendance logique et temporelle entre actions : les conditions de tirs des actions portant explicitement sur les dates de tirs des actions éventuellement tirées précédemment.

De plus, nous souhaitons tirer partie de l'analyse de validité effectuée auparavant : l'automate doit être construit à partir des résultats de l'analyse d'accessibilité.

Un tel automate est alors appelé un *automate d'ordonnancement.*

2.1.2 Adéquation des automates temporels existants

Un certain nombre de travaux proposent de nouveaux types d'automates temporels pour exprimer plus simplement des relations temporelles entre actions.

Dans [KL96], les auteurs emploient la notion de distance temporelle entre les transitions pour tenter de maîtriser l'explosion du nombre d'états lors de l'exploration de l'espace d'états. Ils proposent ainsi un algorithme d'accessibilité sur ce modèle.

Pour éviter l'explosion de l'espace d'états qui peut être provoquée simplement par l'ajout d'horloges aux états, l'algorithme emploie d'abord l'*équivalence historique* (le modèle proposé ajoute aux états des informations temporelles sur le passé des transitions) puis la bisimulation forte des transitions pour regrouper les états en classes d'équivalence. Dans cette approche, les états équivalents possèdent des actions observables identiques bien que les transitions entre les états puissent se produire à des dates différentes. L'algorithme enrichit alors l'espace d'état résultant avec des relations de synchronisation qui décrivent des distances temporelles entre les dates de tir.

Le modèle proposé apporte une solution pragmatique proche du type d'automate que nous recherchons : les relations temporelles entre les tirs des actions sont explicites. Les conditions de tir des actions reposent notamment sur l'historique des actions tirées par le système.

Les automates de type *automates à enregistrement d'événements* (event-clock automata) [AFH94], présentent également des concepts attrayants. Ces automates associent à chaque événement une horloge qui enregistre la date de sa dernière occurrence. La classe des automates à enregistrement d'événements est, d'une part, assez expressive pour modéliser les systèmes synchronisés (finis) de transitions et, d'autre part, déterministe (au sens des automates temporisés [AD91]). Les auteurs présentent également une traduction des systèmes temporels de transitions en automates à enregistrement d'événements, ainsi qu'un algorithme pour vérifier l'équivalence de leurs comportements.

Le concept qui retient notre attention ici est le fait de disposer d'horloges qui mémorisent la date de tir de chaque action (occurrence d'un événement). Cependant, une restriction apparaît au niveau des automates à enregistrement d'événements : les conditions temporelles exprimées sur les enregistrements des dates de tir ne permettent de les comparer qu'à des valeurs constantes, il aurait été intéressant de pouvoir les comparer aux horloges qui enregistrent les dates des événements passés.

Les auteurs de [VHJ96] proposent une méthode systématique pour raffiner une spécification décrite par des formules *Calculs de Durée* (DC formulas) en une composition d'*automates temps-réel communicants*. Les formules DC sont d'abord étendues et dérivées en *automates de planification*, puis, à partir de ces automates de planification sont synthétisés des automates temps-réel communicants respectant un ensemble de contraintes requises.

L'aspect qui retient notre attention ici est le fait que les automates temps-réel com-

municants sont des automates temporisés à une seule horloge qui est remise à zéro à chaque transition. Cette hypothèse structurelle simplifie beaucoup la manipulation de ce type d'automates. Il convient toutefois d'être prudent, car cette hypothèse introduite ainsi sur l'automate sans ajouter de fonctionnalités de recopie ou de mémorisation des états de l'horloge, permet difficilement d'exprimer des contraintes de dépendance entre les dates des événements.

2.1.3 Problématique de la synthèse d'automates

La problématique de la synthèse d'automates d'ordonnancement a fait également l'objet de plusieurs études.

En se restreignant à la spécification de certains types de contraintes temporelles dites *implémentables*, l'auteur de [Die97, Die01] propose un algorithme pour synthétiser un ordonnanceur. Le critère de *formules implémentables* tient au fait que l'objectif de ces travaux est de programmer des contrôleurs logiques (PLC) de systèmes électroniques. Par raffinements successifs de l'ensemble des *formules implémentables* spécifiés, l'auteur produit un automate qui peut être implanté dans un PLC.

Contrairement à cette approche, nous ne souhaitons pas devoir faire d'hypothèses particulières sur la nature des contraintes temporelles spécifiées. Par contre nous souhaitons nous appuyer sur l'analyse d'accessibilité pour filtrer les comportements indésirables selon le domaine d'application choisi.

Une autre approche de synthèse d'un contrôleur, proposée initialement dans [AMP94] puis étendue dans [AMPS98], vise, en s'appuyant sur une notion de jeu temps-réel, à restreindre la relation de transition d'un automate temporel de manière à ce que les comportements résultants satisfassent certaines propriétés. L'algorithme repose essentiellement sur la formulation d'une part, de la notion de jeu temps réel, et d'autre part, de la stratégie gagnante.

Cette approche résout sous une formulation novatrice, la question de l'accessibilité des configurations satisfaisant des propriétés, ou en d'autres termes, selon le point de vue des auteurs, la stratégie gagnante.

Là encore notre approche se distinguera dans la mesure où nous souhaitons bénéficier des résultats de l'analyse d'accessibilité, et ne pas avoir à nous préoccuper de la validité du système lors de la synthèse de l'automate d'ordonnancement.

Finalement, dans [AGS00], les auteurs proposent une méthode pour construire un *système ordonnancé* satisfaisant un ensemble de contraintes temporelles ainsi qu'une politique d'ordonnancement.

Les auteurs caractérisent les politiques d'ordonnancement par des contraintes de sûreté et simplifient le processus de synthèse en appliquant un principe de composition de contraintes. Ce travail vise à établir un lien entre la théorie de l'ordonnancement de processus d'une part et la spécification et l'analyse de systèmes temporels d'autre part.

Notre approche est différente au sens que notre seul objectif est de synthétiser un automate temporel d'ordonnancement caractérisant l'ensemble des comportements cohérents de notre spécification de haut niveau exprimée en RT-LOTOS. La politique d'ordonnancement à appliquer n'est pas explicitée au niveau de l'automate temporel d'ordonnancement, mais au niveau de l'application qui le met en œuvre (par exemple un navigateur pour un document multimédia [SLC01]).

L'objectif ici est donc d'élaborer un modèle d'automate temporel qui relie tous les points intéressants mis en avant par les différents travaux évoqués précédemment, tant au niveau de son pouvoir d'expression que dans l'élaboration d'une technique de synthèse.

2.2 Une proposition d'automate d'ordonnancement : le TLSA

Les automates temporisés [AD91] ont été proposés pour modéliser des systèmes temps-réel à états finis. Un automate temporel se caractérise ainsi par un ensemble fini d'états de contrôle et un nombre fini d'horloges. Toutes les horloges évoluent au même rythme et mesurent la quantité de temps qui s'est écoulée depuis qu'elles ont été initialisées. Chaque transition du système peut remettre à zéro certaines horloges, et comporte une condition de sensibilisation exprimée sous la forme d'une contrainte sur les valeurs des horloges.

Nous proposons ici un nouveau type d'automate temporisé pour l'ordonnancement: le *Time Labelled Scheduling Automaton*[1], ou TLSA en abrégé. Un TLSA présente certaines caractéristiques spécifiques qui le différencient des automates temporels classiques. Un TLSA a autant "d'horloges" que d'états de contrôle définis dans l'automate. Nous appellerons ces "horloges" du TLSA des `temporisateurs` pour les différencier des horloges traditionnelles. Chaque temporisateur mesure le temps passé par l'automate dans l'état de contrôle associé. Aucune fonction explicite ne définit quand un temporisateur doit être initialisé. En effet, le temporisateur associé à un état de contrôle est remis à zéro quand l'automate entre dans cet état de contrôle, et sa valeur courante est gelée quand l'automate en sort.[2] Deux conditions temporelles sont associées à chaque transition d'un TLSA :

1. la fenêtre de tir (dénotée W) définit l'intervalle temporel pendant lequel la transition peut être tirée. Elle prend la forme d'une inégalité à satisfaire par le temporisateur associé à l'état de contrôle en entrée de la transition.
2. la condition de sensibilisation (dénotée K) définit les contraintes temporelles à satisfaire pour tirer la transition. Elle prend la forme d'une conjonction d'inégalités à

1. Ces travaux ont été publiés dans [LC02a, LC02b].

2. Pour cette raison, nous avons dit précédemment qu'un TLSA est un modèle à une seule horloge, puisque, dans n'importe quel état de contrôle, il y a un et un seul temporisateur actif: à un instant donné, cette horloge unique est alors identifiée au temporisateur actif.

satisfaire par un sous-ensemble des temporisateurs, à l'exclusion du temporisateur associé à l'état de contrôle en entrée de la transition.

Note : W, la fenêtre temporelle de tir, concerne uniquement le temporisateur courant. Intuitivement, W caractérise le temps minimum et maximum pendant lequel le système doit rester dans l'état de contrôle courant. Quant à K, la condition de sensibilisation, elle concerne uniquement les temporisateurs passés et caractérise ainsi les transitions sensibilisées en fonction du comportement passé du système.

2.2.1 Définition formelle d'un TLSA

Soit L un ensemble d'étiquettes. Soit $D = \{t \in \mathbb{Q} \mid t > 0\}$ le domaine temporel, $D_0 = D \cup \{0\}$ et $D_0^\infty = D_0 \cup \{\infty\}$.

Soit $\perp$ la notation de la valeur "indéfinie", et $D_{0\perp} = D_0 \cup \{\perp\}$. Par définition, nous posons $\perp + \delta = \perp$, $\perp - \delta = \perp$ pour tout $\delta \in D_0^\infty$.

Soit $T_{set} = \{t_i | i \in [0,n-1]\}$ un ensemble de *temporisateurs*, avec $t_i \in D_{0\perp}$. Dans ce contexte, une condition temporelle est une conjonction d'inégalités de la forme $m \prec t_i \prec M$ où m,M sont des expressions linéaires de constantes (dans D_0^∞) *et de temporisateurs* t_j ($t_j \in T_{set}$, $j \neq i$), avec $\prec \in \{<, \leqslant\}$.

Soit $\mu = (\mu^0, \ldots, \mu^{n-1}) \in D_{0\perp}^n$ la valeur des temporisateurs t_i avec $i \in [0,n-1]$, et $\mathcal{K}$ une condition temporelle définie sur un sous-ensemble de temporisateurs t_i. La notation $\mu \vDash \mathcal{K}$ indique que toutes les inégalités de $\mathcal{K}$ sont *vraies* en remplaçant chaque temporisateur t_i par sa valeur μ^i.

Définition 6 (Time Labelled Scheduling Automaton)
Un Time Labelled Scheduling Automaton est un triplet (S,E,s_0) *où :*

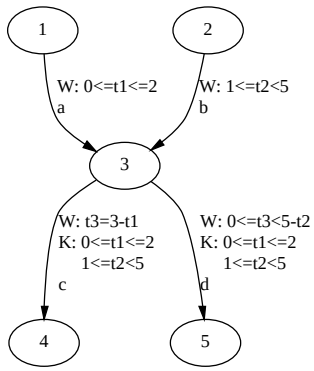
- $S = \{s_0, \ldots, s_{n-1}\}$ *est un ensemble fini d'états de contrôle;*
- $T_{set} = \{t_i | i \in [0,n-1]\}$ *un ensemble de temporisateurs;*
- E *est un ensemble fini de transitions de la forme* (s_i,s_j,W,K,a)*, où* $s_i,s_j \in S$ *sont les états de contrôle en entrée et en sortie de la transition,* W *et* K *sont des conditions temporelles,* $a \in L$ *est une action;*
- s_0 *est l'état de contrôle initial.*

La *fenêtre de tir* W d'une transition (s_i,s_j,W,K,a) est une condition temporelle définie comme une inégalité de la forme $m \prec t_i \prec M$ où m,M sont des expressions linéaires de constantes et de temporisateurs t_k ($t_k \in T_{set}$ $k \neq i$). Si m ou M est égal à $\perp$ en remplaçant chaque temporisateur t_k par sa valeur μ^k, nous posons alors $\mu \nvDash W$.

La *condition de sensibilisation* K d'une transition (s_i,s_j,W,K,a) est une condition temporelle définie comme une conjonction d'inégalités de la forme $\mathcal{I}_k = (m \prec t_k \prec M)$, où $k \neq i$, et m,M sont des expressions linéaires de constantes et de temporisateurs t_l ($t_l \in T_{set}$, $l \neq k$ et $l \neq i$). Si t_k est égal à $\perp$, nous posons alors $\mu \vDash \mathcal{I}_k$; en outre, si m ou M est égal à $\perp$ en remplaçant chaque temporisateur t_l par sa valeur μ^l, nous posons $\mu \vDash \mathcal{I}_k$.

Lorsqu'un temporisateur t_i a pour valeur $\mu^i = \perp$, cela signifie que le système n'est pas passé par l'état i, mais a emprunté un autre chemin de l'automate pour atteindre l'état courant. Pour cette raison, lorsque que la valeur $\perp$ apparaît dans une inégalité composant la condition K (conditions portant sur l'historique des tirs), il n'y a pas eu d'histoire pouvant induire une restriction sur les dates de tir. L'inégalité en question est alors ignorée en posant $\mu \vDash \mathcal{I}_k$. Par contre, lorsque que la valeur $\perp$ apparaît dans l'une des bornes de la condition W (la fenêtre de tir de la transition), cela signifie que cette transition dépend de l'histoire d'un tir qui ne s'est pas produit. Réaliser cette transition n'aurait pas de sens, c'est pourquoi nous l'interdisons en posant alors $\mu \nvDash W$.

Ces règles sont illustrées dans la figure 2.1.



Supposant que nous soyons dans l'état de contrôle 1, la transition $1 \xrightarrow{a} 3$ est tirée dans la fenêtre temporelle $0 <= t_1 <= 2$. Venant de l'état de contrôle 1, la transition $3 \xrightarrow{d} 5$ n'est pas sensibilisée, parce que $t_2 = \perp$, et donc W est faux (voir la définition de la fenêtre de tir). Dans la condition de sensibilisation K de la transition $3 \xrightarrow{c} 4$, la première inégalité est satisfaite ($0 <= t_1 <= 2$), aussi bien que la seconde, puisque $t_2 = \perp$ (voir la définition de la condition de sensibilisation). La transition $3 \xrightarrow{c} 4$ sera alors tirée à la date $t_3 = 3 - t_1$.

FIG. 2.1 – *Illustration des règles de tir*

Un système temporel de transitions étiqueté $LTS(TLSA)$ est associé à chaque $TLSA$. Un état (s,μ) de $LTS(TLSA)$, également appelé une configuration, est complètement décrit par l'état de contrôle s du TLSA et les valeurs $\mu \in D^n_{0\perp}$ de tous les temporisateurs de l'automate. Les transitions de $LTS(TLSA)$ correspondent aux transitions explicites du TLSA, (celles qui caractérisent l'occurrence d'une action), et aux transitions implicites (celles qui représentent le passage du temps).

Définition 7 (État initial de *LTS(TLSA)*)
L'état initial de $LTS(TLSA)$ *est la configuration* (s_0,μ_0), *où* $\mu_0 \in D^n_{0\perp}$ *avec* $\mu_0 = (0,\perp,\ldots,\perp)$.

Définition 8 (Transition explicite de *LTS(TLSA)*)
Soit $(s,\mu) \in LTS(TLSA)$ *et* $(s,s',W,K,a) \in E$ *une transition du* $TLSA$. *Si* $\mu \vDash W$ *et* $\mu \vDash K$, *alors* $(s,\mu) \xrightarrow{a} (s',F(s,s',\mu))$,

où $F : S \times S \times D_{0\perp}^n \rightarrow D_{0\perp}^n$ *est la fonction*[3] *définie par :*

$$F(s_i,s_j,\mu) = \mu' \quad avec \quad \begin{cases} \mu'^j = 0 & \\ \mu'^k = \perp & \forall k \neq i \mid (s_k,s_j,W',K',a') \in E \\ \mu'^l = \mu^l & \forall l \in [0,n-1], l \neq j, l \neq k \end{cases}$$

Le rôle de la fonction F est de se prémunir d'éventuels conflits d'historique de tir en cas de rebouclage du TLSA. En effet, en cas de rebouclage et en l'absence de la fonction F, une action tirée lors d'un précédent tour de boucle mais n'appartenant pas au flux d'exécution courant, pourrait influer sur les futures conditions W et K.

Définition 9 (Transition implicite de *LTS(TLSA)*)
Soit $(s,\mu) \in LTS(TLSA)$ *et* $\delta \in D$. *Si* $act(s,G(s,\mu,\delta))$, *alors* $(s,\mu) \xrightarrow{\delta} (s,G(s,\mu,\delta))$,
où $G : S \times D_{0\perp}^n \times D \rightarrow D_{0\perp}^n$ *est la fonction définie par :*

$$G(s_i,\mu,\delta) = \mu' \quad avec \quad \begin{cases} \mu'^i = \mu^i + \delta & \\ \mu'^k = \mu^k & \forall k \neq i \in [0,n-1] \end{cases}$$

sachant que $act : S \times D_{0\perp}^n \rightarrow \{true,false\}$ est un prédicat indiquant si dans une configuration donnée il existe au moins une transition sensibilisée (i.e. qui satisfait sa fenêtre de tir et sa condition de sensibilisation), autorisant ainsi une progression temporelle. Ce prédicat est défini par:

$$act(s_i,\mu) = \bigvee_{\forall (s_i,s_j,W,K,a) \in E} (\mu \vDash W) \wedge (\mu \vDash K)$$

Ainsi, le temps peut progresser à l'intérieur d'un état tant que le temporisateur courant n'a pas atteint la valeur maximale des bornes supérieures des transitions actives (définies par le prédicat *act*). Lorsque le temps atteint cette valeur, il ne peut plus progresser, et une transition doit être choisie et nécessairement tirée. Ceci caractérise les propriétés d'urgence du TLSA, sans recourir à des invariants comme le font d'autres formalismes [HNSY94].

2.2.2 Propriétés du formalisme TLSA

Le TLSA présente certains aspects séduisants :

1. Au niveau structurel :
   (a) l'échéancier des actions est établi en fonction d'une horloge unique qui, à un instant donné, pointe sur le temporisateur actif ;
   (b) les relations de dépendance entre actions, et les conditions de tir des transitions font référence explicitement aux dates des tirs éventuels des transitions passées.

3. Dans l'exemple de la figure 2.1, lors de la transition $1 \xrightarrow{a} 3$, le temporisateur t_2 est réinitialisé à $\perp$ par la fonction F.

Ces deux points rendent les TLSA attrayants pour une implémentation directe. Notamment, le calcul de tir d'une transition est assez immédiat puisque quand le système entre dans un état de contrôle, les bornes des conditions W et de K de toutes les transitions sortantes peuvent immédiatement être évaluées par substitution de variables et des opérations arithmétiques simples. Ainsi, pendant que le temps progresse, le temporisateur courant doit juste être comparé à quelques valeurs fixes.

2. Au niveau de l'expression de l'ensemble des comportements d'un système :
   Le TLSA offre l'avantage de n'exprimer que les configurations accessibles du système spécifié. En cela il se distingue d'un graphe minimal d'accessibilité (à partir duquel il peut être synthétisé) puisque dans ce dernier chaque région est un surensemble contenant par définition au moins une configuration accessible, mais aussi éventuellement d'autres configurations non accessibles (ce choix est explicité dans la sous-section 1.3.3).

   Néanmoins les travaux de recherche ayant trait à la vérification des systèmes temporels ont préalablement fait naître d'autres objets formels exprimant l'ensemble des configurations accessibles d'un système de manière exhaustive et *exclusive*, c'est-à-dire que sont représentés tous les comportements accessibles et uniquement les comportements accessibles. Notamment, les graphes d'états accessibles et les graphes de classes d'états accessibles classiques présentés dans [BD91] et étendus dans [Ber01] ont ces propriétés. Ces graphes sont le produit de ce qui est appelé *analyse d'accessibilité directe* par opposition aux graphes minimaux d'accessibilité.

   Le TLSA est synthétisé à partir d'un graphe de régions. Que ce graphe soit un graphe minimal d'accessibilité ou bien un graphe d'accessibilité directe n'a pas d'incidence sur la technique de synthèse du TLSA. Cependant, par construction, le nombre d'états du TLSA est directement lié au nombre de nœuds du graphe de régions dont il est issu. Ainsi, un TLSA synthétisé à partir d'un graphe d'accessibilité direct comporte a priori plus d'état qu'un TLSA synthétisé à partir d'un graphe minimal d'accessibilité. Nous gardons ainsi le bénéfice de la minimisation en synthétisant le TLSA à partir du un graphe minimal d'accessibilité.

   Par ailleurs, un TLSA raffine le graphe de régions à partir duquel il est synthétisé et n'exprime que les configurations accessibles. Puisqu'un graphe minimal d'accessibilité comporte des configurations non accessibles, ce travail de raffinage est profitable; il est par contre sans intérêt si le TLSA est synthétisé à partir d'un graphe d'accessibilité directe qui ne comporte déjà que des configurations accessibles. Dans ce cas là le TLSA ne fait qu'exprimer ces informations sous une autre forme, une forme qui a malgré tout des aspects structurels séduisants.

3. Au niveau de la spécification des durées séparant les actions :
   Il apparaît également que le TLSA permet d'expliciter les durées séparant les actions. Cela apporte une réponse simple et immédiate à certains types de questions sur un système temporisé. Un exemple illustrant cette fonctionnalité est présenté dans le paragraphe suivant.

Mise en évidence du calcul des durées par le TLSA

L'exemple qui suit est tiré de [DMP91] :

John va travailler soit en voiture (30-40 minutes), soit en autobus (au moins 60 minutes). Fred va travailler soit en voiture (20-30 minutes), soit en métro (40-50 minutes). Aujourd'hui John est parti de la maison entre 7:10 et 7:20, et Fred est arrivé au travail entre 8:00 et 8:10. Nous savons également que John est arrivé au travail environ 10-20 minutes après que Fred ait quitté son domicile.

Nous souhaitons répondre aux questions suivantes :

1. est-ce que l'information décrite dans ce scénario est cohérente ?
2. est-il possible que John ait pris l'autobus, et Fred le métro ?
3. quelles sont les dates possibles auxquelles Fred a du partir de son domicile ?

Les contraintes évoquées dans le scénario précédent peuvent être spécifiées en RT-LOTOS (voir figure 2.2).

Nous pouvons calculer le graphe minimal d'accessibilité de cette spécification (246 classes, 418 transitions). Pour répondre à la première question, nous pouvons rechercher dans ce graphe s'il existe des chemins partant de l'état initial et passant à la fois par l'action `john_at_work` et par l'action `fred_at_work`. De tels chemins existent; la réponse est donc *oui : le scénario, tel qu'il est décrit, est possible.* Une autre manière de procéder est de composer (i.e. synchroniser selon le schéma de la figure 2.4) cette spécification avec un *processus observateur* qui réalise les deux actions `john_at_work` et `fred_at_work` sans imposer d'ordre entre leurs occurrences, suivi d'une action spécifique `observ1` (voir figure 2.3.a). Nous calculons le graphe minimal d'accessibilité de cette nouvelle spécification, et nous notons sur ce graphe la présence de l'action `observ1`. Cela signifie que la situation testée par l'observateur se produit bien; la réponse à la première question est bien *oui.*

Nous pouvons procéder de manière identique pour répondre à la deuxième question : soit rechercher dans le graphe minimal d'accessibilité de la spécification de la figure 2.2 des chemins passant par les actions `john_at_work`, `fred_at_work`, `john_by_bus`, et `fred_by_metro`; ou alors composer cette spécification avec un nouveau processus observateur (voir figure 2.3.b) testant cette situation et déclenchant l'action `observ2`. Cette étude, par l'une ou l'autre des méthodes, prouve que la réponse à la deuxième question est *non.*

Par contre, nous ne pouvons apporter de réponse à la troisième question en travaillant directement sur le graphe minimal d'accessibilité. Il faut une opération supplémentaire qui va être réalisée lors de la synthèse du TLSA associé. Le graphe minimal d'accessibilité de la spécification de la figure 2.2 nous donne toutes les configurations accessibles depuis l'état initial, c'est-à-dire celles qui, en respectant les contraintes temporelles décrites dans le scénario, aboutissent à une situation où John et Fred sont au travail, et celles qui ne peuvent y aboutir, à savoir, les situations de blocage. Nous allons donc élaguer du graphe minimal d'accessibilité toutes les branches ne conduisant pas à une situation où John et Fred sont au travail (la section 2.4 décrit les techniques et outils pour réaliser cette opération). Ensuite nous allons synthétiser le TLSA sur ce nouveau graphe minimal d'accessibilité. La lecture de la deuxième transition de ce graphe (voir figure 2.5) nous donne la solution : Fred quitte

```
specification Dechter : exit

type natural is sorts nat
  opns +   : nat,nat -> nat
       -   : nat,nat -> nat
endtype

behaviour
  (* Note  7:10=430 7:20=440 8:00=480 8:10=490 infinity=10000 *)
  hide john_left_home, john_by_car, john_by_bus, john_at_work,
       fred_left_home, fred_by_car, fred_by_carpool, fred_at_work in

  (
    ( delay(430,440) john_left_home{10};
      ( delay(30,40) john_by_car{10}; john_at_work{0}; exit )
      []
      ( delay(60,10000) john_by_bus; john_at_work{0}; exit )
    )
    |||
    (
      ( latency(10000) fred_left_home;
        ( delay(20,30) fred_by_car{10}; fred_at_work{0}; exit )
        []
        ( delay(40,50) fred_by_carpool{10}; fred_at_work{0}; exit )
      )
      |[fred_at_work]|
      ( delay(480,490) fred_at_work{10}; exit )
    )
  )
  |[fred_left_home,john_at_work]|
  ( fred_left_home; delay(10,20) john_at_work{10}; exit )
endspec
```

FIG. 2.2 – *Spécification* RT-LOTOS *du problème de Dechter*

```
(
  ( john_at_work; exit )
  |||
  ( fred_at_work; exit )
)
>> ( observ1; exit )
```

a)

```
(
  ( john_by_bus; john_at_work; exit )
  |||
  ( fred_by_metro; fred_at_work; exit )
)
>> ( observ2; exit )
```

b)

FIG. 2.3 – *Exemples d'observateur*

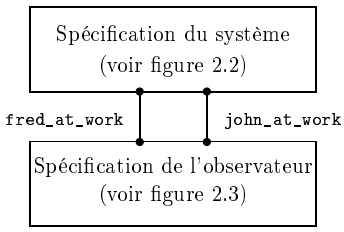


FIG. 2.4 – *Mise en place d'un observateur*

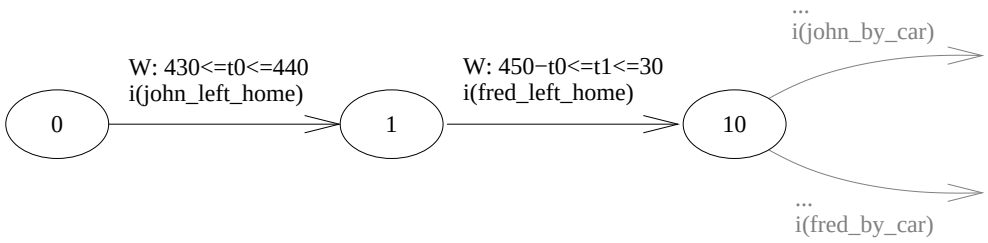


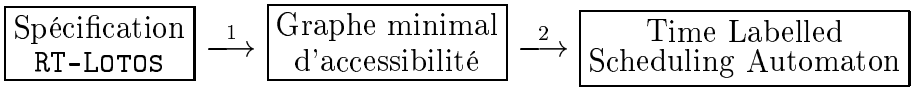
FIG. 2.5 – *TLSA partiel issu du problème de Dechter*

son domicile au plus tôt à 7:30, et au plus tard 30 minutes après que John ait quitté le sien.

Cette faculté du TLSA pour le calcul des durées sera exploitée plus amplement dans un contexte de présentation de documents multimédia (voir section 2.5).

2.3 Un algorithme pour synthétiser un TLSA

Nous présentons ici un algorithme pour synthétiser un TLSA à partir d'un graphe minimal d'accessibilité, lui-même dérivé d'une spécification RT-LOTOS.



L'opération indiquée par la flèche 1 est la technique de vérification de spécifications RT-LOTOS exposée en 1.3.2. L'opération indiquée par la flèche 2 est l'opération de génération d'un TLSA exposée ici.

La figure 2.7 expose le principe de l'algorithme de synthèse d'un TLSA. Il considère en entrée un graphe minimal d'accessibilité. Il comporte trois étapes principales qui conduisent au TLSA (voir la figure 2.6).

Le point délicat de l'algorithme consiste à rechercher pour chaque région du graphe minimal d'accessibilité *la configuration accessible au plus tôt*, sous-entendu, en partant de

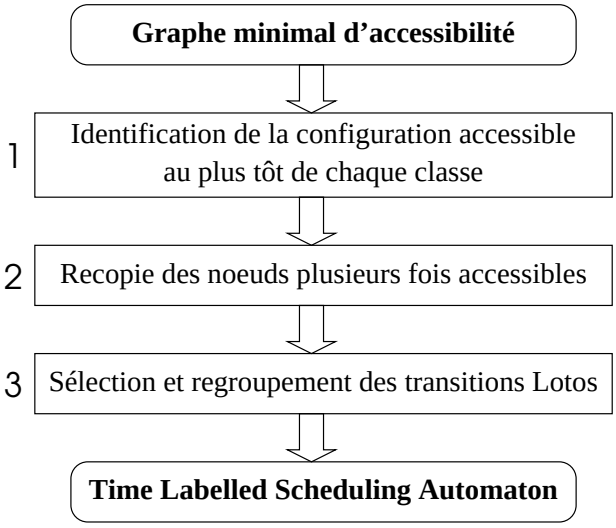


FIG. 2.6 – *Vue d'ensemble de l'algorithme*

Initialiser à 0 les horloges de la configuration accessible au plus tôt de la région initiale.

Pour chaque transition du graphe minimal d'accessibilité :

1
- Évaluer la configuration accessible au plus tôt de la région atteinte.
- Définir la valuation temporelle de la transition comme le délai séparant la configuration accessible au plus tôt de la région de départ de celle de la région d'arrivée.
- Ajouter la valuation temporelle à l'historique des transitions sortant du nœud atteint.

2
- **Si** la région atteinte a déjà une configuration accessible au plus tôt différente **Alors :**
  - Recopier la région atteinte
- **Fi si**

Fin de boucle

Pour chaque transition discrètes LOTOS :

3
- Définir la fenêtre de tir W par la valuation temporelle de la transition augmentée des valuations des transitions temporelles réalisées à l'intérieur de l'état de contrôle qui est à l'origine de la transition.
- Définir la condition de tir K en fonction de l'historique de la transition.

Fin de boucle

Extraire les transitions LOTOS et renommer les nœuds de départ et d'arrivée par le numéro de l'état de contrôle correspondant.

Regrouper les transitions en fonction de W et K.

FIG. 2.7 – *Principe de l'algorithme de synthèse d'un TLSA*

la configuration initiale. Par définition, chaque région du graphe minimal d'accessibilité comporte au moins *une* configuration accessible et éventuellement plusieurs.

Ces configurations sont caractérisées par des équations sur les horloges de l'état du système. Les horloges doivent appartenir à la région considérée, et de plus, elles sont paramétrées par les dates des tirs passés des transitions discrètes LOTOS et temporelles réalisés par le système au cours du temps. Aussi, l'algorithme bâtit pour chaque transition un *historique* des transitions qu'il a évaluées lorsqu'il a cheminé pour atteindre la transition considérée.

Soit $DTA = (S,Nclock,E,s_0)$ un DTA et $RG = LTS(DTA/\pi')$ le graphe minimal d'accessibilité associé, dérivés d'une spécification RT-LOTOS (voir la section 1.3.2 pour les détails).

Intuitivement la configuration accessible au plus tôt d'une classe du RG peut être définie comme étant la première configuration de la classe que le système peut atteindre (à partir de la configuration initiale du système) avant n'importe quelle progression du temps à l'intérieur de cette classe.

Définition 10 (Configuration accessible au plus tôt d'une classe)
Cette définition est récursive. Soient A, B des classes de RG, S_0 la classe initiale de RG ($(s_0,\nu_0) \in S_0$), et la transition $A \to B \in RG$.

- *(s_0,ν_0) est la configuration accessible au plus tôt de la classe S_0.*
- *Soit $(s,\nu) \in A$ la configuration accessible au plus tôt de la classe A, où les valeurs de ν sont exprimées par des expressions linéaires de constantes (définies dans D_0^∞) et de temporisateurs (dont les valeurs sont définies dans $D_{0\perp}$) :*
  - *si $A \xrightarrow{t} B$, alors :*
    Soit δ_m la valeur temporelle minimale telle que $(s,\nu+\delta_m) \in B$; δ_m est exprimée par une expression linéaire de constantes et de temporisateurs. Alors, $(s,\nu+\delta_m)$ est accessible et c'est, en outre, la configuration accessible au plus tôt de la classe B. δ_m est appelée la valuation temporelle de la transition t. Notons que les classes A et B sont dans ce cas associées au même état de contrôle du DTA, à savoir l'état de contrôle s.
  - *si $A \xrightarrow{a} B$, alors :*
    Soit (s,s',K,U,a,C,θ) une transition du DTA, où s' est l'état de contrôle de B. Soit δ_M la valeur temporelle maximale telle que $(s,\nu+\delta_M) \in A$; δ_M est exprimée par une expression linéaire de constantes et de temporisateurs; notons que si A est urgente alors $\delta_M = 0$. Soit δ le temps passé dans toutes les classes associées à l'état de contrôle s avant d'atteindre la configuration accessible au plus tôt de la classe A. Soit le temporisateur t_s défini par $\delta \leqslant t_s \leqslant \delta+\delta_M$. Alors $(s',\nu') \in B$ est la configuration accessible au plus tôt de la classe B, où $\nu'^i = 0 \quad \forall i \in C$, et $\nu'^i = \nu^{\theta^{-1}(i)} + t_s - \delta \quad \forall i \in [1,Nclock(s')]$, $i \notin C$.

2.3.1 Initialisation de l'algorithme

La configuration accessible au plus tôt initiale du système est l'état de contrôle "0" du DTA avec toutes ses horloges à 0.

L'algorithme est initialisé avec le calcul du temps de tir des transitions de sortie de la première région du cluster de l'état 0 du DTA (c'est celle qui n'a aucune transition d'entrée, ou encore celle qui a pour représentant (0 0 ...0)).

Le premier point accessible de cette région est P_0 qui a pour ses N horloges $p_{0_1} = 0$, $p_{0_2} = 0$, ..., $p_{0_N} = 0$.

2.3.2 Identification de la configuration accessible au plus tôt de chaque classe

L'objectif est de déterminer, dans chaque région du graphe minimal d'accessibilité, les configurations accessibles depuis l'état initial (par définition, chaque région du graphe minimal d'accessibilité comporte au moins *une* configuration accessible et éventuellement plusieurs).

Ces configurations sont caractérisées par des équations sur les horloges de la région. Les paramètres des équations sont les dates de tir des actions précédentes (i.e. actions présentes sur le ou les chemins joignant la région initiale à la région étudiée).

Par la suite nous appellerons *cluster de régions* l'ensemble des régions du graphe minimal d'accessibilité qui correspondent à un même état de contrôle du DTA. Du fait de la nature du graphe minimal d'accessibilité, les régions appartenant à un même cluster sont liées entre elles uniquement par des transitions temporelles. Les transitions qui sortent d'un cluster pour atteindre les régions d'un autre cluster sont des transitions discrètes LOTOS.

i) Évaluation des transitions du graphe minimal d'accessiblité : cas général d'une itération

Considérons une région A comprenant $N = Nclock(s)$ horloges c_1, c_2, ..., c_x. Cette région est un polyèdre convexe de dimension N délimité par des bornes sur chaque horloge $m^1 \prec c^1 \prec M^1$, $m^2 \prec c^2 \prec M^2$, ..., $m^N \prec c^N \prec M^N$ avec $\prec \in \{<, \leqslant\}$; et par $N \times (N-1)$ inéquations linéaires sur les horloges deux à deux $m'^1 \prec c^2 - c^1 \prec M'^1$, ...

Le temps peut progresser à l'intérieur d'une région, mais pour sortir de la région A et arriver dans la région B le système doit réaliser soit a) une transition temporelle t, soit b) une transition discrète LOTOS.

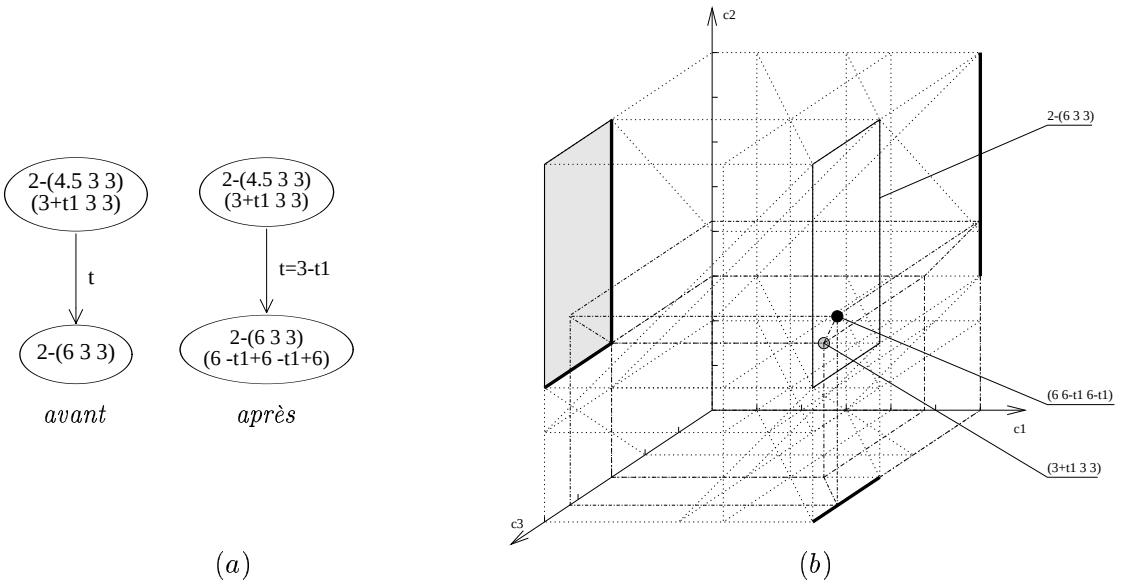
D'une manière générale, l'algorithme consiste à déterminer, à partir de la configuration accessible au plus tôt de la région de départ, le temps de tir de chaque transition sortant de cette région, et à déterminer les horloges de la configuration accessible au plus tôt de chacune des régions atteintes par ces transitions.

L'algorithme 2.1 indique comment s'agencent ces opérations : i) évaluation des transitions temporelle et discrètes, ii) recopie des régions accessibles plusieurs fois, iii) construcion des historiques des tirs passés, et iv) conditions d'itération de l'algorithme.

Algorithme 2.1 Recherche des configurations accessibles au plus tôt

Notons *capt* la configuration accessible au plus tôt d'une région.
Notons $une\_transition.pere$ le nœud père de $une\_transition$.
Notons $une\_transition.fils$ le nœud fils de $une\_transition$.
$0-(0\ldots0).capt = (0\ldots0)$
$push(FIFO, fils(0-(0\ldots0)))$
while il n'y a pas eu de réplication **do**
  while $\neg vide(FIFO)$ **do**
    $une\_transition = pop(FIFO)$
    if $\neg une\_transition.deja\_traitee$ **then**
      if $une\_transition.pere.capt \neq \emptyset$ **then**
⌜ *i)* Soit $d = 0$ la somme des valuations des transitions `t` du cluster.
Soit $capt\_dest : configuration$.
a) → **if** $une\_transition.action$ est une transition temporelle **then**
Appliquer l'algorithme 2.2 qui produit $capt\_dest$ pour une transition `t`.
b) → **else** {$une\_transition.action$ est une transition discrète LOTOS }
⌞ **Appliquer l'algorithme 2.3** qui produit $capt\_dest$ pour une transition LOTOS.
end if
[*ii)* **Appliquer l'algorithme 2.4** qui réalise les éventuelles réplications.
$une\_transition.deja\_fait = true$
[*iii)* Ajouter $une\_transition.valuation\_temporelle$ à $une\_transition.fils.historique$.
⌜ *iv)* **if** tous les pères de $une\_transition.fils$ sont déjà faits ou ont $theta = \emptyset$ **then**
Ajouter dans $FIFO$ les transitions filles du nœud $une\_transition.fils$.
end if
end if
end if
end while
Forcer la réplication des nœuds qui ont un *capt* mais possédant une transition entrante non traitée. {Cas typique d'un rebouclage.}
if $empty(FIFO)$ **then**
Rechercher des états possédant un *capt*, ayant toutes ses transitions d'entrée déjà traitées, mais des transitions de sorties non traitées.
Ajouter ces éventuelles transitions dans $FIFO$.
⌞ **end if**
end while

a) Traitement des transitions implicites (étiquetées par t)



(a) (b)

FIG. 2.8 – *Traitement d'une transition* t

Une transition étiquetée par t de A à B signifie qu'il y a une progression de temps pour atteindre la classe B. Soit $a = (s,(\nu_a{}^1,\ldots,\nu_a{}^N))$ la configuration accessible au plus tôt de la classe A déterminée à l'étape précédente, avec $N = Nclock(s)$. Soit $b = (s,(\nu_b{}^1,\ldots,\nu_b{}^N))$ la configuration accessible au plus tôt de la classe B que nous alors déterminer maintenant. La configuration b peut être définie en déterminant la *valeur minimale* δ_m, tel que $\nu_b = (\nu_a{}^1 + \delta_m, \ldots, \nu_a{}^N + \delta_m)$ appartienne à la région de la classe B (c.-à-d. qu'elle satisfait le système d'inéquations bornant cette région). Puisque les horloges évoluent uniformément depuis la configuration a, nous ajoutons δ_m aux horloges de a pour définir les horloges de b.

La borne inférieure de ce système d'inéquations (qui peut être vu comme une conjonction de contraintes) est le temps qui doit s'écouler pour que le système atteigne une configuration incluse dans la classe B. La valeur de δ_m (constante ou expression linéaire) est appelée *valuation temporelle* de la transition t.

Note: δ_m peut être défini par valeurs supérieures (la borne inférieure du système d'inéquations est exclue). Le traitement d'une telle situation est explicité page 82.

Exemple: comme illustré dans la figure 2.8a, nous partons de $a = (2,(3 + t1,3,3))$ et $A =$ "2-(4.5 3 3)". Le graphe minimal d'accessibilité nous précise que la région de la

classe $B =$ `"2-(6 3 3)"` est bornée par

$$\begin{cases} c_1 = 6 \\ 3 \leqslant c_2 < 8 \\ 3 \leqslant c_3 < 5 \\ -3 \leqslant c_2 - c_1 < 2 \\ -3 \leqslant c_3 - c_1 < -1 \\ -5 < c_3 - c_2 < 2 \end{cases}$$

$$\text{Avec} \quad \begin{cases} c_1 = {\nu_b}^1 = {\nu_a}^1 + \delta_m = 3 + t1 + \delta_m \\ c_2 = {\nu_b}^2 = {\nu_a}^2 + \delta_m = 3 + \delta_m \\ c_3 = {\nu_b}^3 = {\nu_a}^3 + \delta_m = 3 + \delta_m \end{cases} \quad \text{la solution est} \quad \delta_m = 3 - t1$$

Ainsi, la configuration accessible au plus tôt de la classe B est $b = (2,(6,6-t1,6-t1))$, et la valuation temporelle de la transition $A \xrightarrow{t} B$ est égale à $3 - t1$ (en d'autres termes, partant de la configuration $a = (2,(3+t1,3,3))$, le temps doit évoluer d'au moins $\delta_m = 3-t1$ pour atteindre la région B).

Algorithme 2.2 Traitement des transitions temporelles

Soit $i \in \mathbb{N}$, quelconque, unique.
Construction du système d'équations à résoudre :
- ajouter la variable z_i au *capt* de la région du nœud père,
- remplacer dans les équations de la région du nœud fils.

Résoudre le système et extraire un encadrement e de z_i.
if la borne inférieure de e est large **then**
  $une\_transition.valuation\_temporelle = e.inf$
  Ajouter $e.inf$ à d
  $capt\_dest : configuration = une\_transition.pere.capt + e.inf$
else {La borne inférieure de e est stricte.}
  $une\_transition.valuation\_temporelle = z_i$ avec z_i encadré par e
  Ajouter z_i à d.
  $capt\_dest : configuration = une\_transition.pere.capt + z_i$
end if

∘ Gestion des bornes de `t` par valeurs supérieures

Une petite difficulté apparaît dans le traitement d'une action `t` lorsque la borne inférieure de la conjonction d'inéquations bornant t est exclue.

Lorsque nous déterminons cette conjonction nous obtenons généralement : $min \boxed{\leqslant} t \prec max$. Nous écrivons alors que le temps minimal nécessaire pour atteindre la région B est $t = min$. Cette valeur min apparaîtra alors directement dans le calcul des δ qui suivent.

La difficulté apparaît donc lorsque nous avons $min \boxed{<} t \prec max$. Nous ne pouvons pas poser simplement, comme précédemment, $t = min$. Il faut décrire le fait que le temps minimal nécessaire pour atteindre la région B soit strictement supérieur à min. Pour cela nous introduisons une variable intermédiaire z et nous posons $t = z$ avec $min < z \prec max$. Nous pouvons ainsi utiliser cette variable z dans le calcul des δ.

Exemple : évaluons la transition de la figure 2.9.

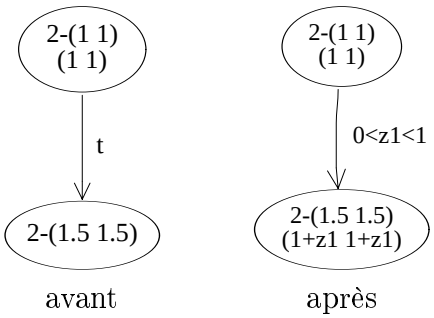


FIG. 2.9 – *Transition* `t` *: valuation strictement supérieure à ...*

Partant de la configuration `(1 1)`, nous cherchons à atteindre la région `2-(1.5 1.5)` définie par :

$$1 < c1 < 2 \quad 0 \leqslant c2 < 96 \quad -2 < c2 - c1 < 94$$

Posons :

$$c1 = 1 + t \quad c2 = 1 + t \qquad t \textit{ étant l'inconnue}$$

En remplaçant nous obtenons :

$$1 < 1 + t < 2 \quad 0 \leqslant 1 + t < 96 \quad -2 < 0 < 94$$

soit :

$$0 < t < 1 \quad -1 \leqslant 1 + t < 95$$

La conjonction des inéquations sur t donne :

$$0 < t < 1$$

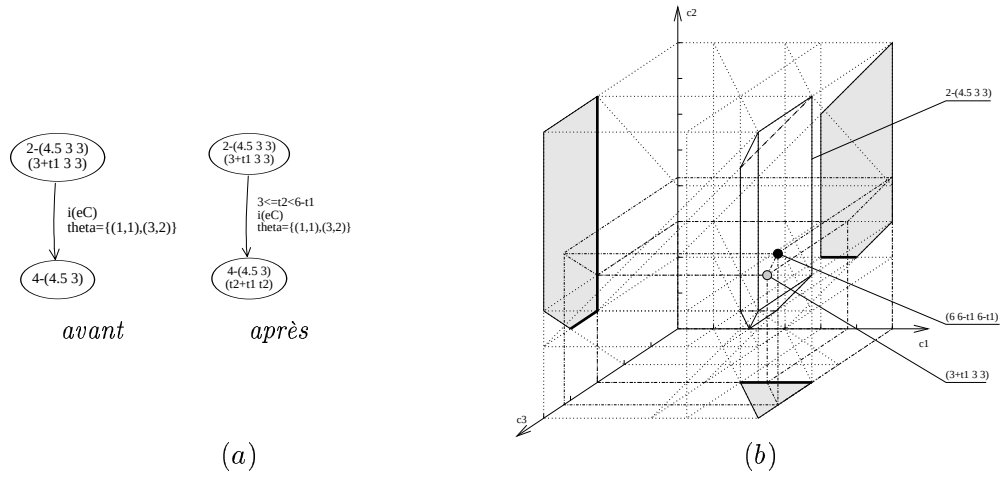
Nous avons donc évalué que, partant de la configuration `(1 1)`, il faut que le temps évolue d'un délai *strictement supérieur* à 0. Nous posons donc $z1 = t$, nous pourrons alors utiliser la variable $z1$ par la suite en retenant le fait que $0 < z1 < 1$.

b) Traitement des transitions explicites (étiquetées par une action LOTOS)

À chaque état de contrôle n du `DTA` est associée une variable temporelle t_n, appelée *temporisateur* caractérisant le temps passé à l'intérieur de l'état, et donc du cluster. Ainsi,

toutes les transitions sortant d'un cluster (transitions discrètes LOTOS) sont associées à cette même variable t_n.

Avant de quitter une région par une action LOTOS, le temps peut progresser. Il s'agit donc ici d'évaluer le temps maximum qui peut s'écouler à l'intérieur de la région A considérée à partir de la configuration accessible au plus tôt a avant de tirer l'action LOTOS g et d'atteindre une configuration de la région B (qui correspond à un nouvel état de contrôle du `DTA`).



(a) (b)

FIG. 2.10 – *Traitement d'une transition* LOTOS

Soit $a = (s,(\nu_a{}^1,\ldots,\nu_a{}^N)$ la configuration accessible au plus tôt de la classe A déterminée à l'étape précédente, avec $N = Nclock(s)$. Une transition étiquetée par une action LOTOS de A à B signifie que le temps peut progresser dans la classe A (à moins que A ne soit une classe urgente) avant d'atteindre la classe B par l'occurrence de l'action LOTOS. Par conséquent, nous recherchons la *valeur maximale* de δ_M, tel que $(\nu_a{}^1 + \delta_M, \ldots, \nu_a{}^N + \delta_M)$ appartienne toujours à la région de la classe A et vérifie le système d'inéquations bornant cette région. Nous savons donc que nous pouvons sortir de A par une action LOTOS après un délai d compris entre 0 et δ_M (la borne δ_M peut être incluse ou non).

Exemple : dans la figure 2.10a, nous avons $a = (2,(3 + t1,3,3))$ et $A =$ `"2-(4.5 3 3)"`. Le graphe minimal d'accessibilité nous précise que la région A de la figure 2.10b est bornée par

$$\begin{cases} 4 < c1 < 6 \\ 3 \leqslant c_2 < 8 \\ 3 \leqslant c_3 < 5 \\ -3 < c_2 - c_1 < 2 \\ -3 < c_3 - c_1 < -1 \\ -5 < c_3 - c_2 < 2 \end{cases}$$

$$\text{avec} \quad \begin{cases} c_1 = {\nu_a}^1 + \delta_M = 3 + t1 + \delta_M \\ c_2 = {\nu_a}^2 + \delta_M = 3 + \delta_M \\ c_3 = {\nu_a}^3 + \delta_M = 3 + \delta_M \end{cases} \qquad \text{la solution est} \quad \delta_M \stackrel{\leq}{=} 3 - t1$$

ce qui signifie que δ_M est la valeur maximale inférieure à $3 - t1$.

Les valeurs des horloge de la configuration a_s correspondant à l'instant où le système quitte A sont donc $a_s^1 = a^1 + d$, $a_s^2 = a^2 + d$, ..., $a_s^N = a^N + d$

Il faut ajouter au délai d le temps σ que le système a passé dans le cluster de A pour obtenir la fenêtre temporelle du tir de l'action LOTOS. Ce temps est la somme des transitions t que le système a tirées à l'intérieur du cluster pour arriver à a.

Par conséquent, l'action LOTOS peut être tirée depuis la classe A après une progression du temps t_s comprise entre σ et $\sigma + \delta_M$. Il s'agit de la fenêtre de tir de cette action LOTOS.

Il faut garder à l'esprit que la date de tir t_s est choisie par le système.

Exemple : le système doit tirer deux transitions t entre les classes associées à l'état de contrôle 2 avant d'atteindre la classe A; puis $\sigma = (3 - t1) + (t1) = 3$, qui caractérise le temps passé dans toutes les classes associées à l'état de contrôle 2 avant d'atteindre la classe A.

De plus, le système peut rester $3 \leqslant t2 < 3 - t1$ unités de temps dans les classes associées à l'état de contrôle 2 avant de tirer l'action `i(eC)`. Ceci définit la fenêtre de tir de cette transition.

Quand le système quitte la classe A, les valeurs des horloges sont :

$$\begin{cases} c1 = {\nu_a}^1 - \sigma + t_s \\ \vdots \\ cN = {\nu_a}^N - \sigma + t_s \end{cases}$$

En appliquant les fonctions C (remise à zéro des horloges) et θ (recopie des valeurs des horloges) (voir la définition formelle du `DTA` page 46) de cette transition, nous obtenons b, la configuration accessible au plus tôt de la classe B.

Exemple : la configuration accessible au plus tôt de la classe $4 - (4.5\ 3)$ est : $(4,(t2 + t1,t2))$.

○ Traitement des régions urgentes

Le temps ne peut progresser dans une région urgente par définition de l'urgence. En conséquence il ne peut y avoir de transition `t` sortant d'une région urgente, l'urgence apparaît comme un cas particulier du calcul du temps de tir d'une action LOTOS.

Le traitement de ce cas consiste uniquement à fixer $d = 0$ pour le calcul de t_s, de a_s et de b (d étant le temps passé par le système dans la région avant de tirer une action LOTOS, tel qu'il a été défini précédemment).

Algorithme 2.3 Traitement des transitions LOTOS

Soit n le numéro de l'état de contrôle du cluster.
if le $une\_transition.fils$ est un état urgent **then**
Soit $\delta = 0$
$une\_transition.valuation\_temporelle = 0$
$une\_transition.W = d$
else {Le $une\_transition.fils$ n'est pas un état urgent}
Construction du système d'équations à résoudre :
- ajouter la variable δ au *capt* de la région père,
- remplacer dans les équations de la région du père.
Résoudre le système et extraire un encadrement e de δ.
$une\_transition.valuation\_temporelle = \delta$ avec δ encadré par e
$une\_transition.W = c_n$ encadré par $e + d$ (i.e. $c_n = \delta + d$)
end if
$capt\_dest : configuration$
$= appliquer\_C\_theta(une\_transition, une\_transition.pere.capt + \delta)$

ii) Recopie des nœuds accessibles plusieurs fois

Quelques classes (exemple la classe $4-(6\ 5)$ dans la figure 2.11) sont accessibles par deux transitions ou plus. Dans ce cas, l'algorithme peut déterminer des configurations accessibles au plus tôt distinctes. En conséquence, les nœuds du graphe minimal d'accessibilité associés à ces classes doivent être recopiés autant de fois qu'il y a de configurations accessibles au plus tôt différentes , et ce, pour que chaque réplication de région ne conserve qu'une unique configuration accessible au plus tôt.

Lors de cette réplication, les transitions sortantes sont elles aussi répliquées. Nous ne conservons pour chaque réplication de région que les transitions entrantes qui arrivent sur l'unique configuration accessible au plus tôt.

Ce cas de figure apparaît notamment en présence de cycles: pour chaque classe concernée par un cycle, l'algorithme trouve une première configuration accessible au plus tôt lors du premier passage, puis il en trouve une deuxième correspondant aux passages suivants qui peut éventuellement avoir à prendre en compte des temporisateurs affectés dans le cycle.

Exemple : évaluons les transitions de la figure 2.11.

L'évaluation de la transition `( 2-(2 2) (2 2), t, 3-(2 0 0) )` donne pour la région `3-(2 0 0)` la configuration accessible au plus tôt : `(t2 0 0)` avec $2 \leqslant t2 < 3$. Or, lorsque nous évaluons la transition `( 3-(1.5 0 0) (t2 0 0), t, 3-(2 0 0) )` nous trouvons que le système atteint la région `3-(2 0 0)` par la configuration `(2 2-t2 2-t2)` avec $1 + z1 \leqslant t2 < 2$.

Il faut donc créer deux régions `3-(2 0 0)`, l'une, succédant à la région `2-(2 2)` et

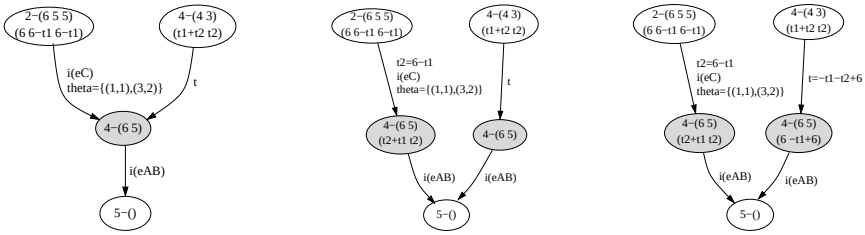


FIG. 2.11 – *Recopie d'un nœud*

ayant pour configuration accessible au plus tôt : `(t2 0 0)`, l'autre, succédant à la région `3-(1.5 0 0)` et ayant pour configuration accessible au plus tôt : `(2 2-t2 2-t2)`.

Algorithme 2.4 Réplication d'un nœud

if $une\_transition.fils.capt \neq \emptyset \wedge une\_transition.fils.capt \neq capt\_dest$ **then**
  Répliquer $une\_transition.fils$.
  Sur ce nouveau nœud, positionner $capt = capt\_dest$.
  Recopie des transitions sortantes.
  Positionner le nœud d'arrivé de $une\_transition$ sur ce nouveau nœud.
else {le fils n'a pas de *capt* ou bien il est égal à $capt\_dest$}
  $une\_transition.fils.capt = capt\_dest$
end if

iii) Report des fenêtres de tir sur l'état d'arrivé

Les fenêtres de tir et les configurations accessibles au plus tôt sont paramétrées par les temps des tirs précédemment réalisés. Pour déterminer si les configurations accessibles au plus tôt sont différentes entre elles et répliquer la région, il est nécessaire de connaître les variables qui paramètrent ces configurations accessibles au plus tôt.

C'est pourquoi nous associons à chaque région la liste des temps des tirs nécessaires pour l'atteindre. Ainsi, si une région peut être atteinte par deux transitions (ou plus), lorsque nous déterminons la configuration accessible au plus tôt en empruntant la seconde transition, nous pouvons tester si cette configuration accessible au plus tôt est la même que celle évaluée pour la première transition au regard des informations (inéquations) que nous avons sur chaque variable qui paramètre les horloges de ces configurations.

Lorsque nous évaluons une transition, il faut donc d'une part reporter les informations de la région de départ sur la région d'arrivée, et d'autre part y ajouter le temps de tir de la transition considérée.

Cependant, dans le cas d'un automate qui reboucle, lorsque nous évaluons une transition LOTOS, nous pouvons nous retrouver avec l'information concernant le temps de tir de cette transition datant du tour précédent. Il faut donc prendre soin de supprimer cette information de l'historique avant d'y ajouter le temps de tir nouvellement calculé.

Une petite difficulté apparaît lorsque plusieurs transitions atteignent alors une même région (après avoir vérifié que nous arrivons bien sur la même configuration accessible au plus tôt) : il faut alors réaliser l'union des fenêtres de tir pour chaque variable.

iv) Évaluation des transitions du graphe minimal d'accessibilité : condition d'itération

Nous avons vu comment réaliser *une* itération de l'algorithme; voyons maintenant sous quelles conditions passer d'un pas d'itération à l'autre.

À chaque région est associé un historique des fenêtres des tirs valides pour l'atteindre (voir iii)). Cet historique n'est complet que si nous avons traité toutes les transitions atteignant la région considérée. Nous ne pouvons donc traiter les transitions de sortie de la région que lorsque *toutes* les transitions d'entrée ont été traitées.

Donc, après le traitement d'une transition, nous envisageons le traitement des transitions de sortie de la région atteinte que si toutes les transitions d'entrée de cette région d'arrivé ont été traitées.

Cette condition n'est pas suffisante pour les automates qui rebouclent : lors du premier passage, la région sur laquelle l'automate reboucle aura forcément une transition d'entrée non traitée (au moins la transition par laquelle le système reboucle sur la région). Par conséquent, si nous respectons cette condition strictement, toute la partie du graphe qui découle d'une telle région ne pourra pas être développée. Nous allons donc assouplir cette condition et considérer que pour les transitions LOTOS qui n'ont pas de fonction `theta` (recopie des horloges au niveau du `DTA`), nous pouvons développer les transitions sortant de la région atteinte. Ceci se justifie par le fait que, puisqu'il n'y a pas de recopie d'horloge, nous sommes certains que le passé de la région (qui n'a pas encore été totalement étudié puisqu'il reste des transitions entrantes non encore traitées), ne va pas influencer l'avenir.

Une question se pose avec les régions nouvellement répliquées : à une étape donnée de l'algorithme, sommes-nous certains, lorsque nous traitons la dernière transition d'entrée d'une région répliquée, qu'une nouvelle transition ne va pas s'y rajouter au cours d'une réplication future, et fausser l'historique des tirs de cette région? La réponse est NON : ceci est garanti par le fait que nous explorons le graphe *en largeur d'abord* (les transitions à traiter sont placées dans une liste de type FIFO, c'est-à-dire une avec politique *premier arrivé, premier servi*). Les transitions atteignant une région répliquée sont les redirections des transitions qui atteignaient la région d'origine. Par conséquent, ces transitions sont toutes à un même niveau de profondeur de l'algorithme qui les explore en largeur d'abord. Et l'historique de la région répliquée ne dépend donc que de ces transitions. Nous sommes

donc certains que, au moment où nous traitons les transitions sortant de la région répliquée, toutes les transitions entrantes auront été traitées.

Le seul risque est de placer dans la file les transitions sortant d'une région répliquée autant de fois qu'il y a de transitions entrantes. Pour l'algorithme, cela ne pose pas de problème de traiter à nouveau une transition déjà traitée, mais pour optimiser le temps de traitement, il vaut mieux faire attention à ne pas placer dans la file une transition déjà traitée.

2.3.3 Sélection et regroupement des transitions LOTOS

Nous venons d'enrichir le graphe minimal d'accessibilité en exhibant pour chaque région sa configuration accessible au plus tôt, et en indiquant sur chaque transition, sa valuation temporelle (i.e. la progression temporelle séparant les deux configurations accessibles au plus tôt qu'elle relie).

L'objectif, maintenant, est de construire un automate temporisé où sont spécifiés les fenêtre des tirs et les conditions de tir des actions LOTOS, pour engendrer ainsi finalement le TLSA.

Ce processus comporte deux étapes :

- extraire du graphe précédent les transitions correspondant aux actions LOTOS, et renommer les états avec les noms des états du DTA,
- regrouper les transitions qui ont des fenêtres de tir contiguës.

i) Extraction des transitions LOTOS

1. Sur chaque transition LOTOS du graphe précédent, nous remplaçons les variables z par leurs valeurs dans les bornes de la fenêtre de tir ainsi que dans l'historique des fenêtres de tirs. Il faut alors changer les inégalités en inégalités strictes.
2. Nous extrayons les transitions LOTOS en renommant le père et le fils de chaque transition avec le numéro de l'état de contrôle du DTA correspondant. La fenêtre de tir de l'action est notée W.
3. Nous reportons sur chaque transition l'historique des tirs du nœud père : cet historique décrit dans quelles conditions la transition décrite par l'arc pourra être tirée (on peut maintenant en retirer les informations sur les variables z, elles ne serviront plus). Cet historique peut être vu comme la condition de sensibilisation de l'action, et est noté K.

A l'issue de ces trois opérations nous obtenons (figure 2.12) un graphe temporisé comprenant autant d'états que le graphe d'accessibilité a de clusters (et numérotés avec les états de contrôle du DTA) et autant de transitions que de transitions LOTOS dans le graphe minimal d'accessibilité.

A chaque état est associé *un* temporisateur comptant le temps passé à l'intérieur de l'état (voir sous-section 2.2.1).

Chaque transition décrit une action LOTOS dont le tir est conditionné par un historique des tirs précédents K et indique la fenêtre temporelle portant sur l'horloge de l'état pendant laquelle le tir est valide W.

Par convention, sur les transtitions du `TLSA`, la première inéquation correspond à W et les suivantes à K.

ii) Regroupement des transitions

Dans ce nouveau graphe, il y a autant de transitions que d'actions LOTOS dans le graphe d'accessibilité, c'est-à-dire "beaucoup". Nous allons tenter ici de regrouper au maximum ces transitions pour fournir une information condensée.

○ Approche mathématique

Les conditions temporelles W et K d'une transition peuvent êtres vues comme la description des faces d'un polyèdre convexe (faces incluses ou non), de dimension le nombre de temporisateurs impliqués dans W et K. Une propriété de W et K est que la variable désignant le temporisateur courant n'apparaît que dans W.

Si nous considérons chaque ensemble de transitions ayant le même état père, le même état fils, et la même action LOTOS, chacun de ces ensembles décrit (par leurs conditions W et K) une union de polyèdres. Malheureusement, par construction du `TLSA`, ces polyèdres peuvent se recouvrir partiellement, voir complètement. Ces transitions contiennent alors des informations redondantes. L'objectif ici est donc de proposer une représentation condensée de ces informations.

En d'autres termes, nous recherchons une écriture minimale de cette union de polyèdres. Cette union doit être écrite par un ensemble de transitions de type `TLSA`. Il faut donc trouver une partition de cette union de polyèdres qui soit un ensemble (minimal) de polyèdres convexes recouvrant totalement cette union et permettant l'écriture de la condition W (la variable désignant le temporisateur courant n'apparaît que dans W).

Pour réaliser cela, nous bénéficions de la librairie de manipulation de polyèdres *Parama Polyhedra Library* [PPL]. Cette librairie mathématique permet de définir des polyèdres convexes dans l'espace des rationnels, bornés éventuellement par des inégalités strictes, et offre une grande variété d'opérations sur ces polyèdres, notamment l'union convexe de polyèdres. Pour ces raisons, cette librairie correspond bien à nos besoins. À l'heure où nous écrivons ces lignes, les fonctionnalités de manipulation d'inéquations strictes de cette bibliothèque sont encore en phase de test.

○ Approche pragmatique

Sans raisonner nécessairement sur des polyèdres, nous pouvons procéder à ces regroupements en manipulant simplement ces inéquations, en y appliquant les transformations classiques (substitution de variables, simplifications, etc.). Ces manipulations sont guidées par une heuristique qui vise à atteindre un regroupement optimal des transitions. Notons que l'heuristique proposé ici, bien que "relativement intuitive" (elle a pu être implémentée

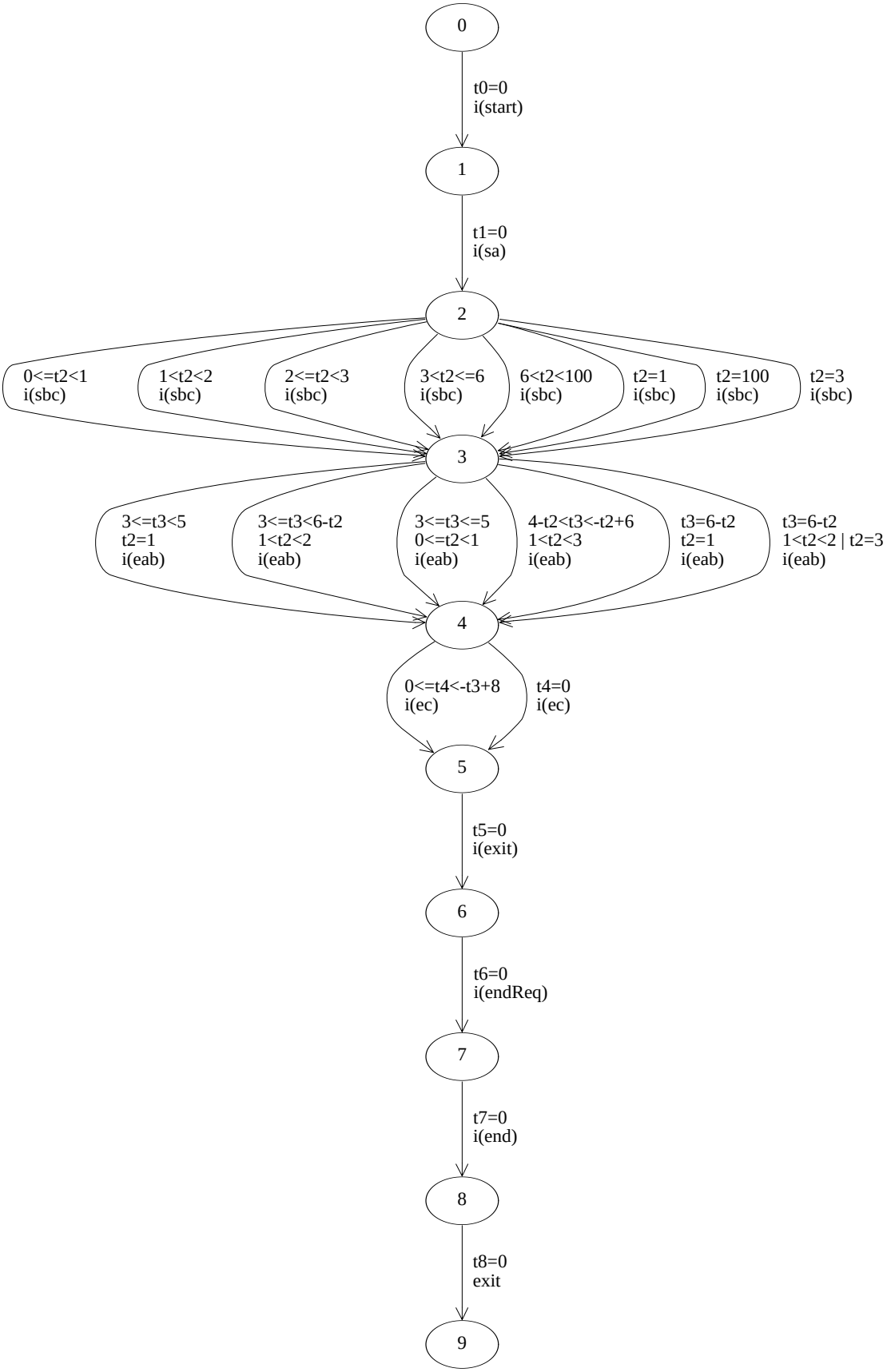


FIG. 2.12 – *Graphe des configurations accessibles au plus tôt : TLSA avant regroupement des transitions*

de manière ad hoc), n'atteint pas le regroupement optimal dans certains cas complexes (grand nombre de variables, grand nombre de transitions).

Chaque transition est comparée avec toutes les autres transitions identiques du point de vue LOTOS (même état père, même état fils, même action LOTOS). S'il y a moyen de regrouper cette transition avec une ou plusieurs transitions, nous réalisons les regroupements, puis nous supprimons cette transition.

Les transitions peuvent se regrouper lorsque :

a) des transitions ont les mêmes conditions de tir K et des fenêtres de tir W adjacentes;

b) des transitions ont les mêmes fenêtres de tir W (et nous regroupons les conditions de tir K);

c) des transitions sont incluses dans une autre.

a) Regroupement suivant les conditions de tir

Nous tentons de regrouper les transitions qui ont les mêmes conditions de tir et des fenêtres de tir adjacentes.

Après avoir vérifié que les transitions ont des historiques de tirs présentant des expressions littéralement équivalentes il faut voir si les deux fenêtres de tir peuvent se regrouper pour donner une expression littérale simplifiée. Si c'est le cas, nous réalisons le regroupement (voir figure 2.13).

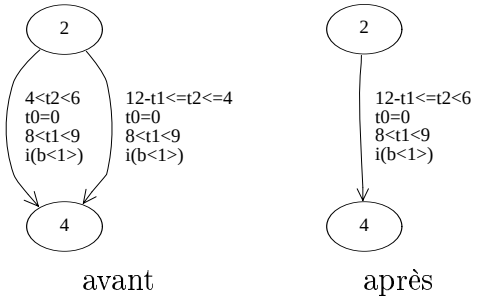


FIG. 2.13 – *Fenêtres de tir contiguës*

b) Regroupement suivant les fenêtres de tir

Nous tentons de regrouper les actions qui ont la même fenêtre de tir.

b.i) Deux fenêtres de tir W son égales si :

- leurs expressions littérales sont identiques (voir figure 2.14),
- elles sont équivalentes après avoir remplacé les variables dont nous connaissons la valeur exacte indifféremment dans l'un ou l'autre des deux historiques (i.e. nous avons $t_x = \cdots$ et non pas $\cdots \prec t_x \prec \cdots$ dans l'historique de la première transition ou dans l'historique de la seconde transition).

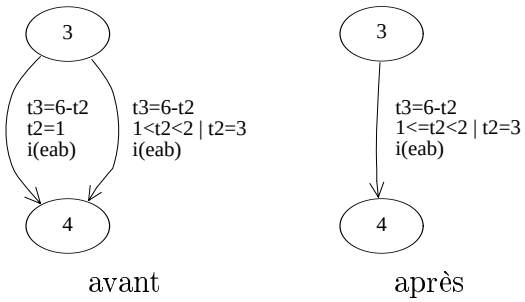


FIG. 2.14 – *Fenêtres de tir ayant la même expression*

Un problème se pose dans la deuxième situation : puisque les deux fenêtres de tir n'ont pas la même expression, bien qu'elles soient identiques compte tenu de la valeur des variables, quelle expression donner au regroupement des deux ? Il faut une expression de la fenêtre de tir la plus générale possible :

- si une expression, comportait des variables et pas l'autre, nous la gardons (voir figure 2.15)
- si non (les deux expressions comportent des variables), nous composons une nouvelle fenêtre de tir suivant la formule :

$$NouvelleBorne = 2 \times BorneArc1 - BorneArc2$$

après avoir remplacé les variables dans $BorneArc1$ par leur valeur exacte tirée de l'historique de la première transition, et celles de $BorneArc2$ par leur valeur exacte tirée de l'historique de la seconde transition (voir figure 2.16).

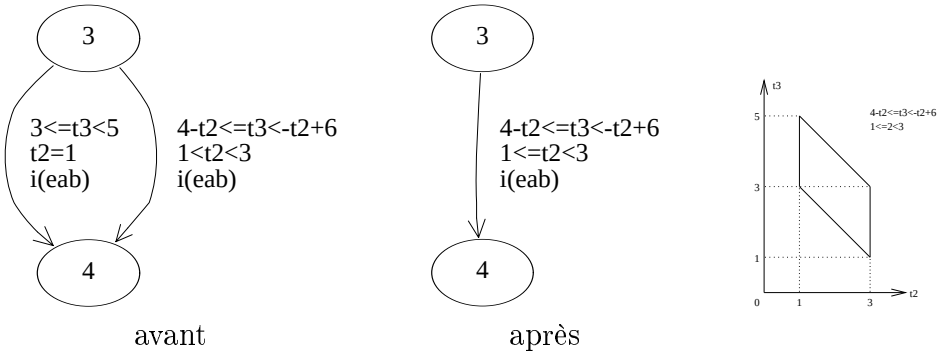


FIG. 2.15 – *Fenêtre de tir ayant des expressions différentes mais dont une seule a des variables*

b.ii) Regrouper les conditions de tir K :
Cela consiste tout simplement à écrire leur disjonction, ou autrement dit, à écrire l'union des polyèdres décrits par les conditions K de chacun des transitions.

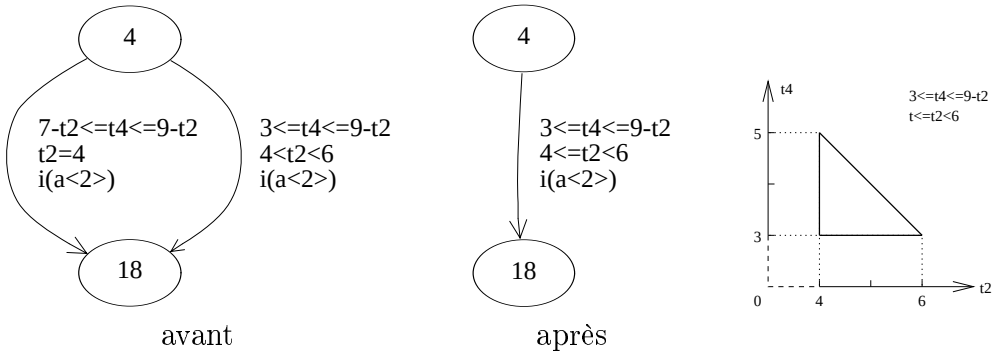


FIG. 2.16 – *Fenêtre de tir ayant des expressions différentes et dont chacune a des variables*

Cependant, nous nous abstiendrons de réaliser ce regroupement si le polyèdre résultant n'est pas convexe.

c) Regroupement de transitions incluses

Une transition est dite *incluse* dans un autre lorsque le polyèdre décrit par W et K de la première transition est inclus dans le polyèdre décrit par W et K de la seconde.

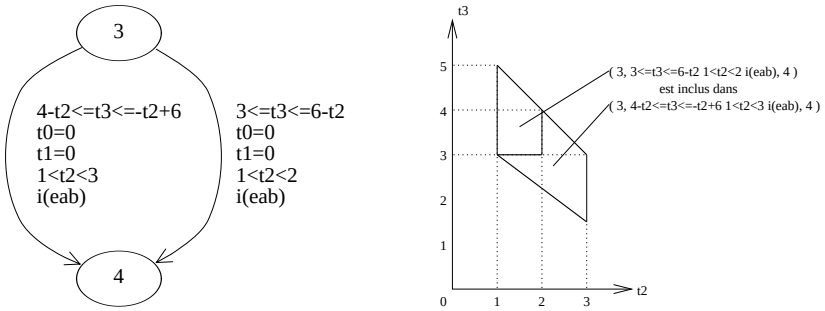


FIG. 2.17 – *Transitions incluses*

d) Réitérer les regroupements

À l'issue des trois phases de regroupement en fonction des conditions de tir, il peut être judicieux de retenter de regrouper en fonction des fenêtres de tir. Et réciproquement. En fait, nous réitérons alternativement ces deux phases de regroupement jusqu'à ce que nous obtenions un graphe stable.

e) Simplifier l'historique

Certaines informations ne sont plus pertinentes dans l'historique des tirs : l'historique des tirs permet de choisir quelle transition tirer pour sortir d'un état donné. Si la condition portant sur un tir passé est la même sur toutes les transitions de sortie d'un état, alors

elle ne rentrera pas en ligne de compte pour choisir l'un ou l'autre des transitions. Nous pouvons donc supprimer cette information dans tous les historiques des transitions de sortie de l'état (voir figure 2.18).

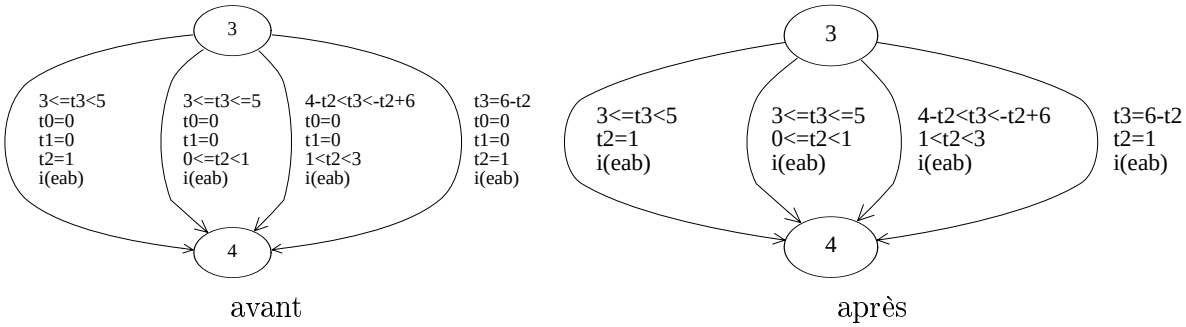


FIG. 2.18 – *Simplifier l'historique des tirs*

Cette étape diminue considérablement les calculs lors des deux phases de regroupement précédentes. Il est intéressant de la réaliser à chaque itération de regroupement.

2.3.4 Découpage des états

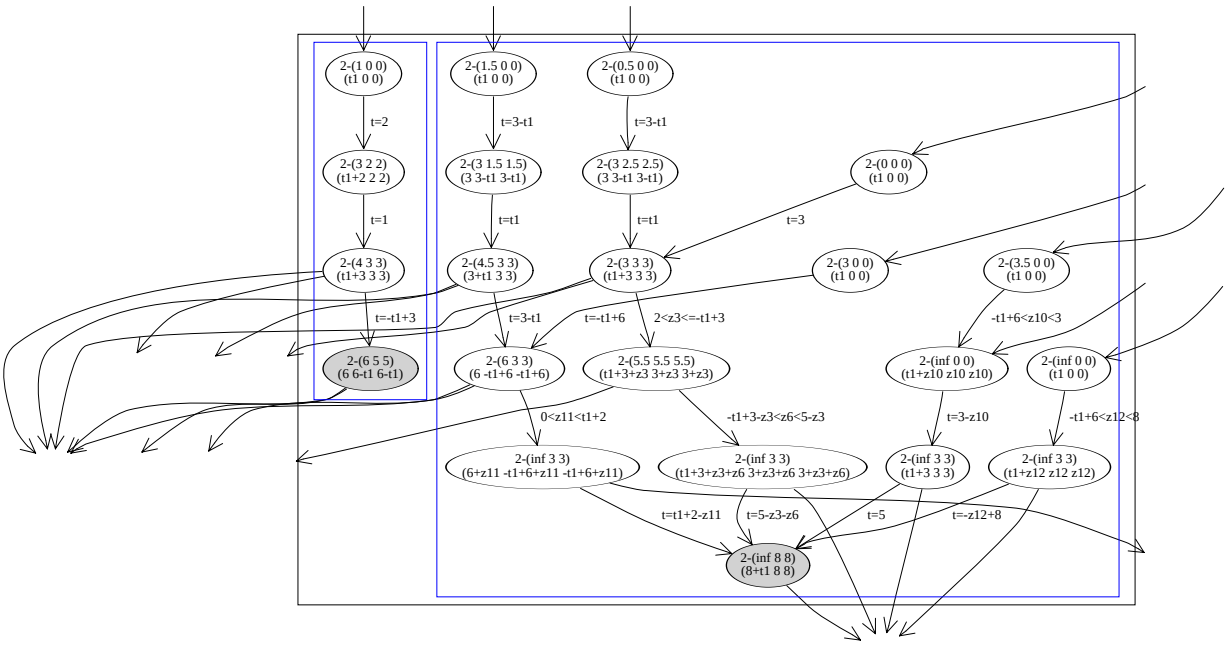
Nous étudions ici une variante possible de l'algorithme. Nous nous interrogeons sur un compromis possible entre la notion d'état et la notion de conditions temporelles sur les transitions qui, d'une certaine manière, induisent de tester un état du système. Dans certains cas, il serait même possible de réduire la taille des équations des conditions W et K en partitionnant les états.

En effet, il peut arriver qu'un cluster de régions soit composé en fait de plusieurs sous-graphes connexes de transitions `t`. Les transitions LOTOS menant et partant de ces sous-graphes connexes n'ont pas de lien causal entre elles. Il est donc inutile de composer des conditions W et K qui prennent en compte l'ensemble de ces comportements. En séparant ces comportements distincts dans le `TLSA` nous pouvons espérer obtenir des conditions W et K plus simples.

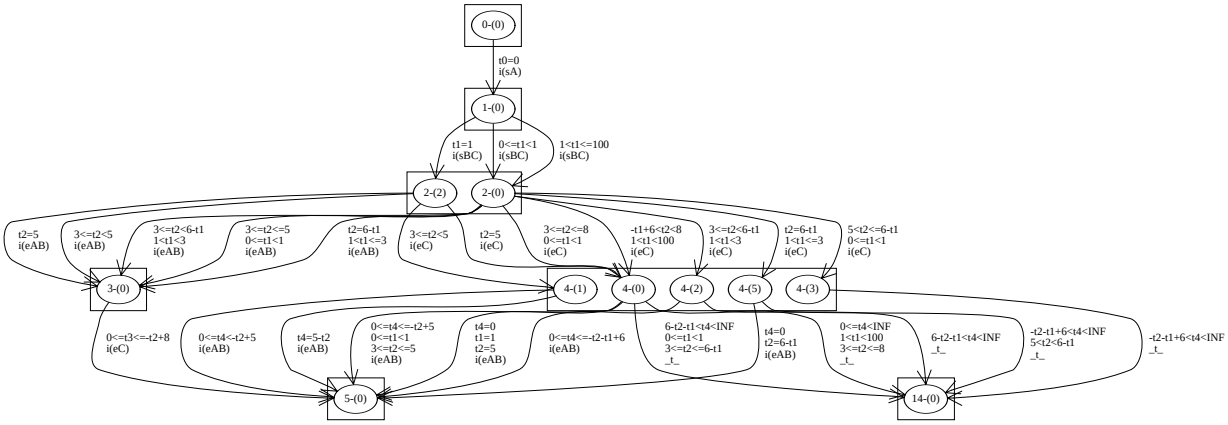
Dans ces circonstances, lors de la phase d'extraction des transitions LOTOS (page 89) et de renommage des nœuds par le numéro de l'état de contrôle du `DTA`, au lieu de regrouper toutes les régions du cluster en un seul état de `TLSA`, nous allons découper le cluster en autant de sous-graphes connexes, et former autant d'états de `TLSA`. Les opérations de regroupement devraient en être simplifiées.

L'intérêt est donc d'essayer de limiter le nombre d'équations en augmentant un peu le nombre d'états.

Un réel gain a pu être observé sur certaines spécifications courtes, mais il est difficile de le dire dans le cas général, le risque étant que l'opération de regroupement (page 90) se passe moins bien (par exemple dans le cas de la figure 2.19).



Découpage du cluster de régions de l'état de contrôle 2



TLSA avec nœuds découpés

FIG. 2.19 – *Découpage des états de contrôle*

2.3.5 Pré-traitement du graphe minimal d'accessibilité

Pour diverses raisons techniques, le graphe minimal d'accessibilité tel qu'il est produit l'outil `rtl` n'est pas directement exploitable par l'algorithme de synthèse du `TLSA`. Divers traitements préalables sont requis. Ils sont détaillés en annexe B.

2.4 Élimination des comportements non cohérents

Nous avons vu dans la section 2.2 qu'il était possible de tirer partie des résultats d'accessibilité pour synthétiser un automate d'ordonnancement. Notamment, nous proposons de retirer du graphe minimal d'accessibilité les comportements incohérents.

Nous exposons ici les moyens techniques à mettre en œuvre pour réaliser cette opération sur un graphe minimal d'accessibilité en vue de synthétiser ultérieurement un `TLSA` cohérent.

2.4.1 Démarche du filtrage des comportements non cohérents

L'analyse d'accessibilité permet d'exhiber les configurations accessibles d'un système. Parmi ces configurations accessibles, certaines peuvent être jugées *non cohérente* compte tenu du domaine d'application.

Par exemple, dans le domaine des documents multimédia interactifs, une configuration de la présentation du document est dite incohérente s'il n'est pas possible d'atteindre, depuis cette configuration, la fin de la présentation du document (voir la section 2.5). D'une manière générale, il appartient à l'utilisateur de la méthodologie de définir ce qu'est, pour lui et compte tenu du domaine d'application, une configuration non cohérente.

Ensuite, puisque nous disposons de l'ensemble des configuration accessibles, nous pouvons filtrer les configurations accessibles non cohérentes et ne conserver que les configurations accessibles cohérentes pour synthétiser le `TLSA`. La démarche consiste à parcourir le graphe minimal d'accessibilité et à supprimer de ce graphe les nœuds correspondant aux configurations jugées non cohérentes ainsi que les transitions qui y conduisent.

2.4.2 Suppression des configurations indésirables

Nous proposons un outil, l'outil `dtacut`, dans le but de faciliter la suppression des transitions indésirables d'un graphe minimal d'accessibilité.

L'outil `dtacut` est un filtre qui prend en entrée un graphe minimal d'accessibilité, qui prend en argument des expressions régulières désignant l'étiquette des transitions ou des nœuds à rechercher.

Les transitions ainsi indiquées, ou les transitions reliant les nœuds indiqués, sont marquées. Ensuite l'algorithme explore le graphe, depuis ces transitions, en avant, puis en arrière pour marquer récursivement les autres transitions qui partent des transitions indiquées, ou qui y mènent.

L'outil `dtacut` a alors deux comportements possibles : soit il ne garde que les transitions qui sont à la fois marquées en avant *et* en arrière (situation utile dans le cas de graphes qui rebouclent), soit il garde les transitions qui sont marquées en avant *ou* en arrière.

L'utilisateur a donc pour charge de combiner judicieusement les options de `dtacut`, voir de combiner plusieurs appels en série à `dtacut`, pour supprimer les transitions qu'il juge indésirables.

Lorsque nous supprimons des transitions et des nœuds, nous risquons de perdre certaines informations temporelles intéressantes : considérons une région qui possède une transition LOTOS marquée convenablement (c'est-à-dire que nous allons garder) et également une transition `t` vers une région (du même état de contrôle, forcément), d'où ne part aucune transition marquée. Cette deuxième région et cette transition `t` devraient être supprimés. Cela dit, si nous faisons cela, nous perdons l'information comme quoi le système a le droit de faire progresser le temps dans l'état de contrôle et évoluer dans cette deuxième région. En quelque sorte, nous réduisons l'invariant. Pour éviter cela, il faut garder cette région et cette transition `t`.

L'outil `dtacut` propose donc de garder de manière arbitraire les transitions `t` ainsi que les régions qu'elles atteignent (nœuds fils) lorsque les nœuds pères sont marqués.

2.4.3 Insertion de transitions particulières `_t_`

Par définition du `TLSA`, le système a le droit de rester dans un état de l'automate jusqu'à ce que le temps atteigne la borne supérieure de la dernière transition tirable.

Un souci intervient dans la synthèse du `TLSA` lors de l'extraction des transitions LOTOS : considérons un état de contrôle (ou cluster de régions), dans lequel il existe des *régions-puits* (i.e.: nœuds ne comportant aucune transition de sortie). Lors de l'extraction des transitions LOTOS, nous allons perdre l'information comme quoi le système a le droit de faire évoluer le temps à l'intérieur de ces régions puits, c'est-à-dire que l'automate `TLSA` va borner l'évolution du temps dans cet état de contrôle à la borne supérieure de la dernière transition tirable. Dans ces circonstances nous excluons du `TLSA` des comportements qui avaient été jugés cohérents.

La parade consiste alors à ajouter artificiellement, dans le graphe minimal d'accessibilité et avant la synthèse du `TLSA`, des transitions LOTOS portant l'étiquette `_t_` (par exemple) depuis ces régions puits vers des nœuds à région vide créés eux aussi artificiellement.

De cette manière, le `TLSA` résultant disposera au niveau de l'état de contrôle incriminé, d'une transition particulière `_t_` dont la borne supérieure correspond au temps maximum que le système peut rester dans cet état. Nous gardons ainsi l'information temporelle qui aurait pu nous faire défaut au début, et ce, sans changer la sémantique du `TLSA`.

Cette opération préliminaire à la synthèse est réalisée par l'outil `rg2gcp`. Il est judicieux de l'utiliser conjointement avec la fonctionnalité de `dtacut` qui garde les transitions `t`.

2.5 Utilisation du TLSA

Un TLSA est un modèle d'implémentation qui exprime au moyen d'une seule horloge les contraintes temporelles qu'un système doit satisfaire. Nous disposons d'un moyen pour synthétiser un TLSA à partir d'un graphe minimal d'accessibilité, sachant que ce graphe d'accessibilité est lui-même engendré à partir d'une spécification de haut niveau exprimant la composition d'un ensemble de contraintes temporelles. Nous montrons également comment un graphe d'accessibilité peut être facilement corrigé avant de synthétiser un TLSA dans le but d'éliminer tous les comportements potentiellement incohérents du système modélisé (voir section 2.4). Le modèle TLSA est actuellement utilisé pour l'ordonnancement de la présentation de documents multimédia interactifs et nous proposons dans cette section une illustration simple dans ce domaine.

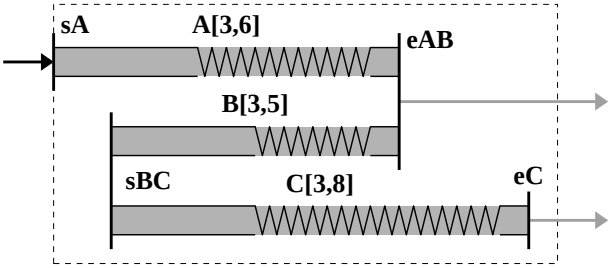
2.5.1 Présentation d'un cas d'étude

Cet exemple illustre le cheminement depuis la spécification jusqu'à l'ordonnancement d'un document multimédia interactif. Dans ce cadre, une méthode formelle est employée pour décrire les contraintes temporelles caractérisant la présentation de chaque média composant un document, les interactions avec l'utilisateur, ainsi que les contraintes globales de synchronisation entre les médias. La spécification formelle, dérivée d'un modèle d'édition de haut niveau, consiste essentiellement en une composition de processus (RT-LOTOS) simples caractérisant des contraintes temporelles élémentaires. L'analyse d'accessibilité est alors exécutée sur la spécification RT-LOTOS pour vérifier sa cohérence temporelle. Différentes variantes de cette propriété de cohérence temporelle ont été définies, selon que les actions menant aux états potentiellement bloquants sont contrôlables ou non (voir [SLC01, SSC99] pour plus de détails).

Considérons l'exemple de la figure 2.20a. L'auteur de ce scénario multimédia souhaite présenter trois médias appelés respectivement A, B et C. Les durées de présentation de ces médias sont définies respectivement par les intervalles temporels suivants : [3,6], [3,5] et [3,8]. Ceci signifie que, par exemple, la présentation du média A doit durer au moins 3 secondes et au plus 6 secondes. Ainsi, du point de vue de l'auteur, n'importe quelle durée de présentation du média est acceptable du moment qu'elle appartient à l'intervalle temporel spécifié.

L'auteur exprime en outre les contraintes de synchronisation globales suivantes :

1. les présentations des médias A et B doivent se terminer simultanément;
2. les présentations des médias B et C doivent commencer simultanément;
3. le début du scénario multimédia est déterminé par le début de A, et sa terminaison est déterminée par la fin de A et de B, ou par la fin de C.



(*a*)

```
specification example : exit
type natural is ... endtype
behavior
 hide sA, sBC, eAB, eC  in
 ( (
   (sA; delay(3,6) eAB{3}; stop)
   |||
   (sBC; delay(3,8) eC{5}; stop)
  )
  |[sBC, eAB]|
  (sBC; delay(3,5) eAB{2}; stop)
 )
 |[sA, sBC]|
 (sA; latency(inf) sBC; stop)
endspec
```

Note :

$delay(d1,d2) \equiv delay(d1)latency(d2 - d1)$

$inf \equiv +\infty$

(*b*)

FIG. 2.20 – (*a*) *Scénario multimédia* (*b*) *Spécification* RT-LOTOS

2.5.2 Mise en oeuvre de la méthodologie

La spécification RT-LOTOS correspondante est présentée dans la figure 2.20b, et le graphe (minimal) d'accessibilité associé, obtenu par l'outil `rtl`, est présenté dans la figure 2.21a.

Le graphe minimal d'accessibilité permet d'analyser la cohérence temporelle du scénario multimédia. Nous disons que ce scénario est *potentiellement cohérent* s'il existe au moins un chemin commençant par `i(sA)` (l'action `i(sA)` caractérise le début de la présentation du média A, et par conséquent, le début de la présentation du scénario) et menant à la fin de la présentation du scénario (l'occurrence des actions `i(eAB)` ou `i(eC)`). D'après la figure 2.21a, le scénario est effectivement potentiellement cohérent.

En étudiant les contraintes de haut niveau, nous pouvons noter que le début de la présentation de B et de C est resté non spécifié par rapport au début de A. Sur un exemple aussi simple, il est facile de se rendre compte que si le début des médias B et C est

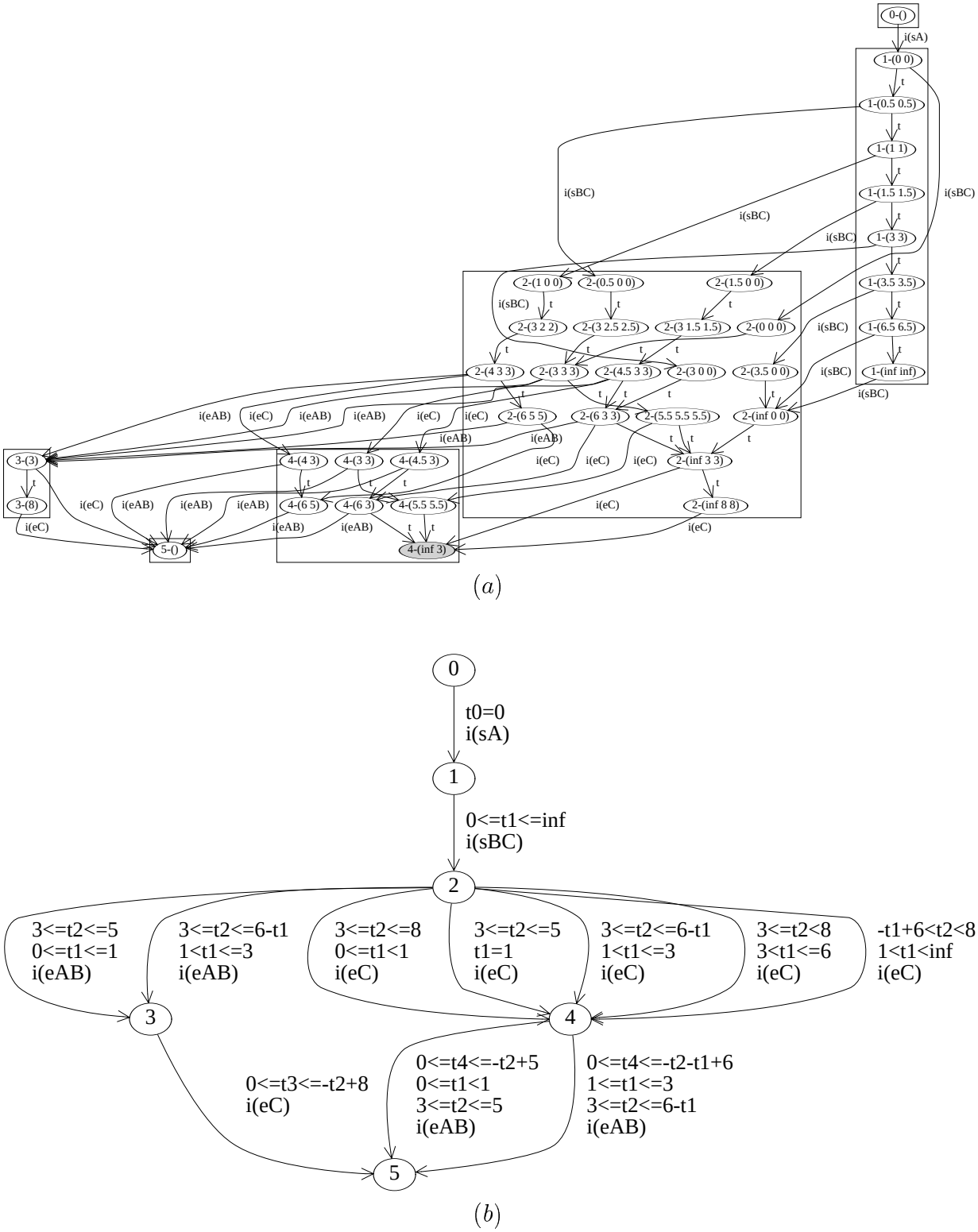


FIG. 2.21 – (*a*) *Graphe minimal d'accessibilité* (*b*) *TLSA associé*

déclenché trop tard par rapport au début de A, certaines contraintes temporelles ne seront jamais satisfaites (par exemple, la contrainte que A et B doivent finir simultanément). Ceci caractérise des comportements potentiellement incohérents dans la présentation du scénario multimédia; ils sont représentés dans le graphe d'accessibilité par des chemins partant de la région initiale et conduisant à des régions de configurations bloquantes (voir, par exemple, la région notée *4-(inf 3)* en bas de la figure 2.21a.).

La figure 2.21b illustre le `TLSA` dérivé du graphe d'accessibilité de la figure 2.21a en appliquant l'algorithme présenté dans la section 2.3. Ce `TLSA` ne peut pas être utilisé tel quel pour ordonnancer la présentation du scénario multimédia car il comporte les mêmes comportements incohérents que le graphe d'accessibilité dont il est dérivé. Considérons, par exemple, le comportement où les actions `i(sA)`, `i(sBC)`, `i(eC)` se produisent respectivement aux dates $t_0 = 0$, $t_1 = 5$ and $t_2 = 6$, et qui mène le scénario dans l'état 4 de la figure 2.21b. Nous remarquons que dans cet état il n'y a aucune transition sensibilisée, puisque les deux conditions de sensibilisation (voir la sémantique formelle du `TLSA` dans la sous-section 2.2.1) sont $0 \leqslant t_1 < 1$ ou $1 \leqslant t_1 \leqslant 3$ alors que $t_1 = 5$.

Les configurations bloquantes n'étant pas désirables, nous pouvons les éliminer en supprimant du graphe d'accessibilité tous les chemins partant de la région initiale et conduisant à ces régions de configurations bloquantes (voir le paragraphe 2.4.2).

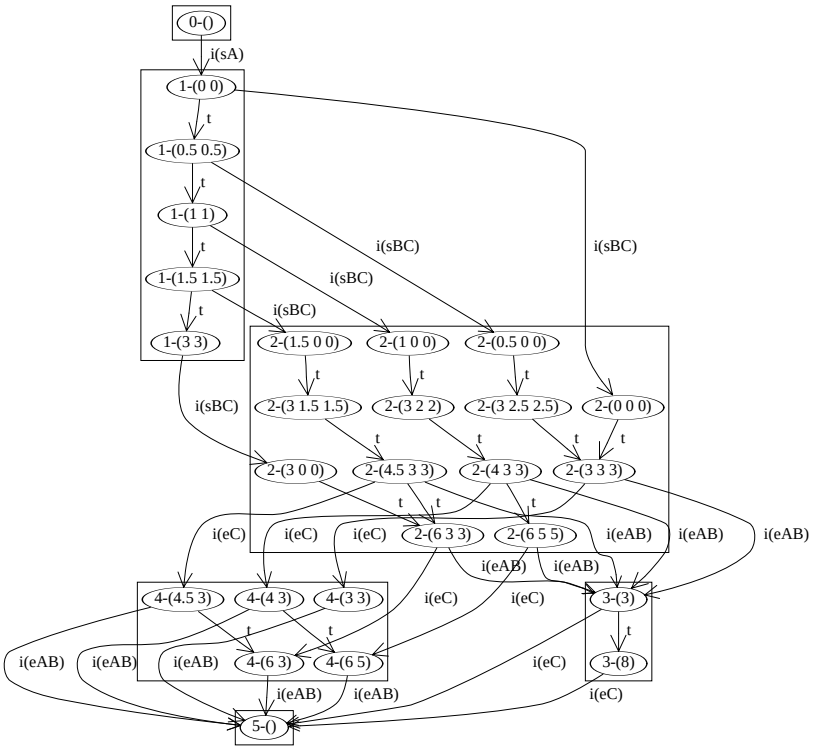
Cette correction ne pose pas de problème particulier si les actions étiquetant un chemin incohérent du graphe d'accessibilité sont toutes contrôlables (i.e. des actions dont l'occurrence est indépendante de l'environnement extérieur) (voir [SC00] pour une définition de la contrôlabilité). Supposons dans notre exemple, que ces actions soient contrôlables, il est alors possible de supprimer tous les chemins incohérents et de produire un graphe d'accessibilité corrigé qui soit cohérent (voir la figure 2.22a).

La figure 2.22b illustre le `TLSA` dérivé du graphe d'accessibilité de la figure 2.22a, tel qu'il est produit par l'outil `rg2tlsa`. Ce `TLSA` permet de réaliser un ordonnancement cohérent de la présentation du scénario multimédia, pour lequel la présentation des médias B et C doit commencer au plus tard 3 secondes après le début de A (voir la condition de tir $0 \leqslant t_1 \leqslant 3$ définie pour la transition de l'état 1 vers l'état 2). Cette information apparaît explicitement au niveau d'une transition du `TLSA`, alors qu'elle reste cachée dans la caractérisation des états du graphe d'accessibilité.

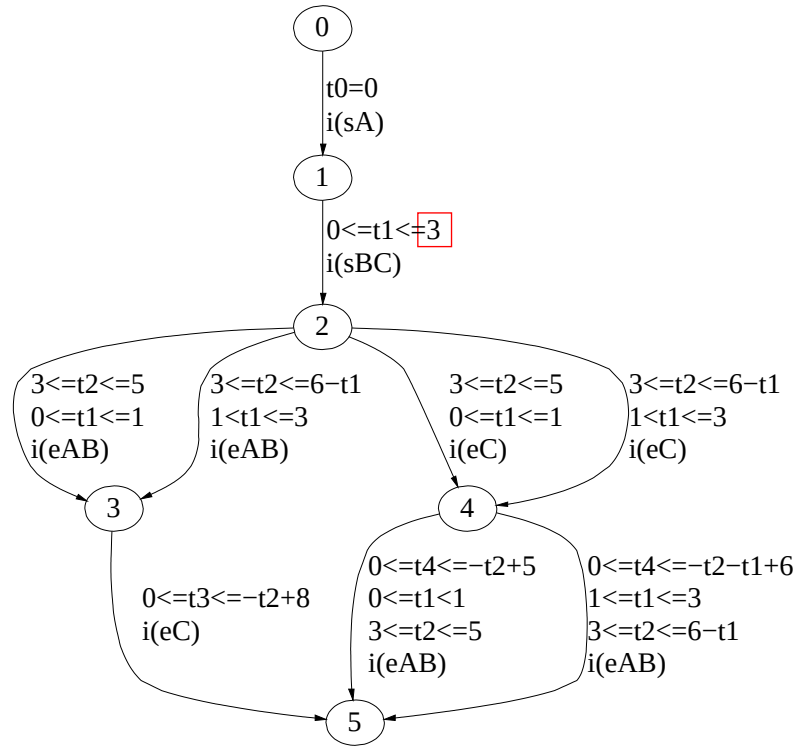
2.5.3 Bilan de la méthodologie

L'approche proposée comprend ainsi trois étapes principales:

1. Nous utilisons RT-LOTOS comme langage de spécification de haut niveau pour exprimer une composition de contraintes temporelles;
2. Nous engendrons, par l'outil `rtl`, le graphe d'accessibilité de la spécification RT-LOTOS et supprimons du graphe d'accessibilité tous les comportements incohérents pour obtenir un graphe d'accessibilité *cohérent*;
3. Nous réalisons la synthèse du `TLSA` à partir du graphe d'accessibilité cohérent.



(a)



(b)

FIG. 2.22 – (a) *Graphe minimal d'accessibilité consistant* (b) *TLSA associé*

Le document multimédia peut alors être présenté sur l'ordinateur de l'utilisateur à l'aide d'un navigateur dédié au TLSA (voir [SC01]). Un tel navigateur a été réalisé.

Ainsi, le TLSA peut être employé à des fins d'ordonnancement, dans le sens où c'est un modèle d'exécution qui définit les intervalles temporels pendant lesquels les actions doivent se produire pour satisfaire la composition initiale des contraintes temporelles. Par contre, pour réaliser *une exécution* du système à partir du TLSA (comme c'est le cas lorsque nous présentons un document multimédia à un utilisateur), il faut se définir également une *politique d'ordonnancement*, c'est-à-dire un ensemble de critères permettant de choisir, dans les plages temporelles exprimées par le TLSA, la date effective à laquelle réaliser une action. Cette politique d'ordonnancement dépend du domaine d'application. Ainsi, dans le cas des documents multimédia, une politique d'ordonnancement à été définie dans [SC01].

2.6 Conclusion

Nous avons présenté et formalisé une nouvelle variante d'automate temporel, appelée Time Labelled Scheduling Automata (ou TLSA en abrégé). Le TLSA est un modèle opérationnel, prévu pour exprimer globalement les contraintes temporelles qu'un système doit satisfaire. Nous avons présenté un algorithme pour synthétiser un TLSA à partir d'un graphe minimal d'accessibilité : cette approche propose de raffiner le graphe des régions, d'en élaguer certaines branches jugées non souhaitables, d'extraire les dates de tir possible des actions, et de présenter ces informations sous la forme d'un TLSA.

Un TLSA exprime tous les comportements cohérents souhaitables quand il est synthétisé à partir d'un graphe minimal d'accessibilité (lui-même dérivé d'une spécification RT-LOTOS), duquel tous les chemins incohérents (ceux menant à une configuration bloquante) ont été enlevés. Il décrit de manière simple comment ordonnancer les actions.

Dans le cadre de la présentation de documents multimédia, un outil qui *joue* le TLSA a été écrit en Java [SC01]. Cette expérience laisse penser qu'il est relativement aisé de réaliser d'autres outils exécutant un TLSA et adaptés à des domaines d'application particuliers.

Le premier chapitre de ce mémoire avait présenté diverses méthodes formelles pour la spécification et la vérification des systèmes temps-réel, et plus particulièrement la technique RT-LOTOS. Ce chapitre a présenté une approche pour mettre en œuvre et exploiter la vérification par analyse d'accessibilité et la construction d'un graphe de régions. Dans ce contexte, la rigueur de l'écriture de la spécification initiale, sur laquelle toutes ces techniques formelles se déroulent, reste un point critique. Le chapitre suivant revient sur ce travail qui doit être fait en amont et propose une approche permettant de faire le lien entre les techniques de description semi-formelle et les techniques de description formelle avec comme objectif d'assister le concepteur lors de la phase initiale de spécification.

3

Méthodologie liant formel et semi-formel

Ce chapitre propose une méthodologie permettant de lier une approche formelle aux cycles traditionnels de conception et de développement de produits informatiques.

La première section décrit une approche intégrant `UML` *et* RT-LOTOS. *La seconde section expose les algorithmes de traduction qui prennent part dans la méthodologie proposée. La troisième section expose comment enrichir la méthodologie et décrit des extensions réalisées. Enfin, la dernière section conclut le chapitre.*

3.1 Un profil temps-réel formel pour `UML`

La maîtrise tant des concepts que des techniques de spécification et de vérification formelle, est une tâche réputée ardue et fastidieuse. C'est pourquoi il apparaît primordial de l'abstraire autant que possible du cycle de production d'objets informatiques, et ainsi de soulager de cette tâche l'utilisateur final.

L'approche développée dans [AdSSL$^+$01, dSSAL$^+$01, dSSAL$^+$02] se propose de donner une sémantique formelle à un profil temps-réel, et d'insérer, dans un cycle de production s'appuyant sur une méthodologie `UML` [UML01], une phase de vérification opérant grâce au formalisme RT-LOTOS.

Normalisé à l'OMG (*Object Management Group*), `UML` est un langage fédérateur des meilleurs pratiques du génie logiciel et des technologies objet en particulier. En effet, UML est le produit de la fusion des méthodes OMT (*Object Modeling Technique*, de Jim Rumbaugh), BOOCH'93 (de Grady Booch), et OOSE (*Object Oriented Software Engineering*,

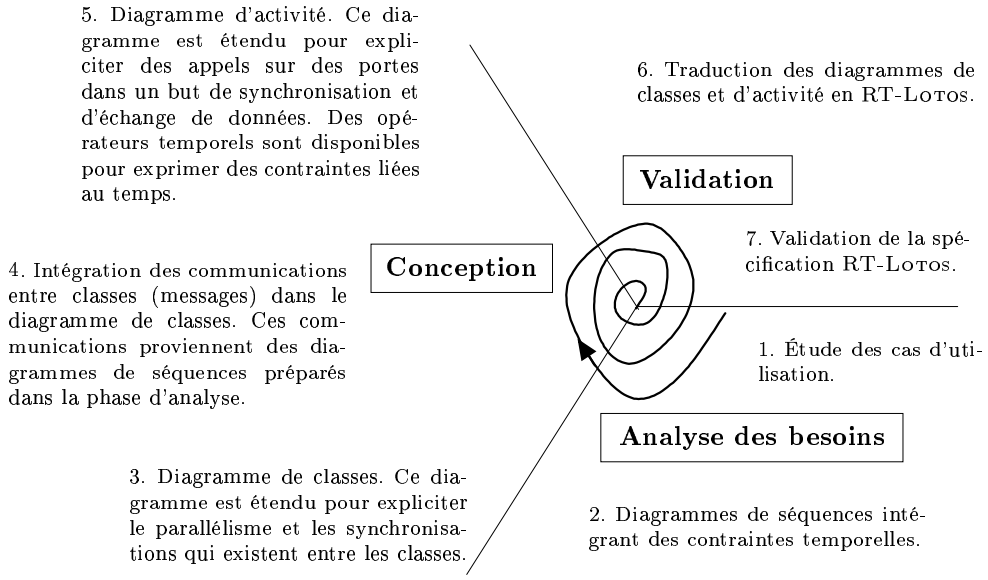


FIG. 3.1 – *Une méthodologie* UML *intégrant la validation* RT-LOTOS

de Ivar Jacobson). Cette approche est aujourd'hui parmi les plus reconnues des méthodologies objets. Elle permet d'intégrer différents processus du développement, de la conception à la réalisation.

UML définit 9 diagrammes pour représenter les différents points de vue d'une modélisation :

- Les diagrammes de cas d'utilisation : représentation des fonctions du système du point de vue de l'utilisateur.
- Les diagrammes de séquences : représentation temporelle des objets et de leurs interactions.
- Les diagrammes de classes : représentation de la structure statique en terme de classes et de relations.
- Les diagrammes de collaboration : représentation spatiale des objets, des liens et des interactions.
- Les diagrammes d'objets : représentation des objets et de leurs relations qui correspond à un diagramme de collaboration simplifié, sans représentation des envois de message.
- les diagrammes d'activité : représentation du comportement d'une opération en terme d'actions.
- Les diagrammes d'états-transitions : représentation du comportement d'une classe en terme d'états.
- Les diagrammes de déploiement : représentation du déploiement des composants sur les dispositifs matériels.

L'approche proposée dans ce chapitre prend sa place dans une phase de validation au sein de la méthodologie UML, comme illustré dans la figure 3.1.

L'approche TURTLE (pour *Timed UML and* RT-LOTOS *Environment*) propose un profil[1] UML qui donne une sémantique formelle aux associations entre classes, définit des opérateurs temporels, et ajoute des facilités de validation des contraintes logiques et temporelles. La sémantique formelle de ce profil est donnée par la traduction d'une spécification TURTLE dans une spécification formelle RT-LOTOS, ce qui permet de conduire des simulations du modèle et une vérification formelle par la construction de graphes d'accessibilité.

Il y a ici la volonté d'apporter un service de validation (simulation, vérification formelle) auprès de l'utilisateur, sans lui imposer l'apprentissage d'une méthode formelle et de RT-LOTOS en particulier.

3.1.1 Les projets UML temps-réel

UML fournit un cadre opérationnel pour les solutions orientées objet. Dans sa version 1.4, UML couvre la plupart des vues nécessaires pour construire une solution raisonnable à un problème donné. Cependant, plusieurs aspects du développement de systèmes temps-réel exigent un support additionnel. En ajoutant les vues complémentaires et des notations, UML peut être naturellement prolongé pour mieux définir et concevoir les systèmes temps-réel.

Plusieurs propositions ont était faites dans ce sens, d'une part sur un plan pragmatique, par les professionnels du développement logiciel, et d'autre part sur un plan théorique dans le cadre de travaux de recherche.

i) Les outils industriels

Plusieurs fabricants d'outils logiciels sont en compétition pour offrir des solutions en matière d'UML temps-réel sous la forme d'une notation augmentée d'une méthodologie :

- *Rose RT* de *Rational* implémente *UML-RT*, une amélioration d'UML avec des concepts du langage ROOM [SR98];
- *Rhapsody* d'*I-Logix* utilise autant que possible les constructions natives d'UML 1.4 [Dou99];
- *TAU* de *Telelogic* emploie UML en tant qu'entrée pour SDL [Bjo00];
- *Real-Time Studio* d'*Artisan Software* a son propre opérateur temporel [RTS99];
- *Esterel Studio* d'*Esterel-technologies* combine UML et le langage synchrone Esterel [EST].

Les premiers outils dans la liste ci-dessus s'appuient sur un paradigme d'asynchronisme, le cinquième étant synchrone. Tous proposent également des opérateurs temporels limités à des temporisateurs armés avec une durée fixe. Les diagrammes comportementaux manquent d'opérateurs temporels natifs exprimant des intervalles temporels et des actions limitées dans le temps. Néanmoins, il existe parfois des solutions, mais celles-ci demeurent orientées

1. Un profil UML spécialise le méta-modèle UML dans un méta-modèle spécifique consacré à un domaine donné d'application [TG01]. Un profil peut contenir certains éléments du méta-modèle de référence, des mécanismes d'extension, une description de la sémantique du profil, des notations additionnelles, et des règles pour la traduction, la validation et la présentation du modèle.

vers la génération de code pour une cible et un logiciel d'exploitation spécifique. Il n'est alors pas possible de mener à bien la validation a priori de modèles UML.

ii) Les travaux de recherche

De nombreux travaux ont été effectués pour donner à UML une sémantique précise [BF98, Bru99, ECM+99] et pour relier UML à une technique de description formelle, telle que les systèmes de transitions étiquetées [JJP98, LG00], les réseaux de PETRI [DP98], la méthode Z [DdB01], les langages synchrones [APFR01], PVS [Tra00], ou E-LOTOS [CM00].

À la différence de [DP98], le profil présenté ici demeure conforme à UML 1.4 et lui adjoint des concepts empruntés à la technique de description formelle RT-LOTOS. Il s'agit d'un langage asynchrone, ce qui différencie cette approche de [APFR01]. Tout comme [DdB01, Tra00, CM00], le procédé de traduction de notre profil UML étendu vers RT-LOTOS donne une sémantique formelle à ce profil. Notons que [DdB01, Tra00, CM00] proposent une définition formelle de concepts UML tels que la création dynamique d'objets, l'échange de messages, la composition de composants, etc., mais n'introduient pas de nouveaux opérateurs pour décrire des contraintes temporelles.

3.1.2 L'approche TURTLE

Le profil UML proposé ici fournit des facilités destinées à améliorer la description logique et temps-réel des relations entre classes, et les comportements internes de ces classes. Compatible avec UML 1.4, le profil TURTLE [AdSSL+01, dSSAL+01] étend deux des diagrammes de la notation normalisée à l'OMG [UML01] : d'une part, les *diagrammes de classes* pour la structuration du logiciel (architecture *statique*, voir iii)) et, d'autre part, les *diagrammes d'activité* pour la définition des comportements internes aux classes (*dynamique* du système, voir iv)).

Cela se traduit, d'une part, au niveau du diagramme de classes, par l'introduction d'un nouveau stéréotype[2] UML au méta-modèle UML et, d'autre part, au niveau de la description comportementale des classes (diagramme d'activité), par l'introduction, entre autres, de trois opérateurs temporels.

Les diagrammes de classes sont modifiés de manière à rendre explicite l'expression de parallélisme et de la synchronisation entre les classes, par opposition au parallélisme "naturel" et donc implicite entre objets d'un modèle UML basé sur la norme [UML01]. Deux types de classes cohabitent au sein des diagrammes de classes du profil TURTLE : des classes dites "normales", et des classes stéréotypées *Tclass*. Quatre relations (association, agrégation, composition et héritage) permettent d'établir des liens entre les classes. Ces dernières peuvent être, au choix du concepteur, actives ou passives. Une classe active est une classe dont les instances possèdent leur propre flux d'exécution (ou processus léger) et peuvent ainsi déclencher une interaction.

2. Un stéréotype est un ajout indirect au méta-modèle UML. Le stéréotype TURTLE ainsi que les types abstraits sont identifiés graphiquement par un symbole TURTLE situé dans le coin supérieur droit de la classe considérée.

Les diagrammes d'activité sont étendus avec un retard déterministe, un retard non déterministe et une offre limitée dans le temps. Notre choix s'est porté sur une utilisation des diagrammes d'activité au lieu des diagrammes d'états (*Statecharts*) pour décrire les comportements internes des classes. Notons que ces deux types de diagrammes sont supportés par les outils UML. Des travaux sont prévus pour apporter le même type de fonctionnalité (spécification de comportements temporels) au diagrammes d'états.

La traduction vers le langage RT-LOTOS donne une sémantique formelle à ce profil et permet d'exploiter l'outil `rtl` (*Real-Time* LOTOS *Laboratory*, voir 1.3) à des fins de validation de diagrammes TURTLE, par simulation et analyse d'accessibilité, et ainsi de confronter la modélisation à d'éventuelles erreurs logiques et temporelles.

L'objectif de l'approche TURTLE n'est pas de fournir, en s'appuyant sur UML, une syntaxe graphique à RT-LOTOS comme cela a été fait pour LOTOS [GLo89, BNT94] et pour E-LOTOS [GA98]. Au contraire, il s'agit bien de fournir une sémantique formelle à UML en se proposant d'exprimer les comportements temporels d'objets UML en RT-LOTOS.

i) Le type abstrait *Gate*

En plus des procédés classiques dont disposent habituellement les classes pour communiquer (appel de méthode, modification d'attributs publics, etc.), les classes stéréotypées *Tclass* peuvent communiquer via des portes (élément de type *Gate*). Ainsi nous introduisons le type abstrait *Gate* (figure 3.2a). Par défaut, une *Gate* permet des échanges bidirectionnels d'information. Deux types abstraits sont ajoutés pour rendre ces échanges unidirectionnels (figure 3.2b) : nous faisons une distinction entre les portes sur lesquelles des données peuvent être émises (*OutGate*) et les portes sur lesquelles des données peuvent être reçues (*InGate*). Les autres techniques de communications entre classes (variables partagées, appels de méthodes, etc.) ne sont pas supportés par le profil TURTLE et doivent être explicités au moyens de *Gate*. Dans la suite de ce chapitre, nous utilisons l'expression « une *Tclass* réalise un appel sur une *Gate* `g` » pour exprimer qu'une *Tclass* désire communiquer sur la porte `g` de type *Gate*.

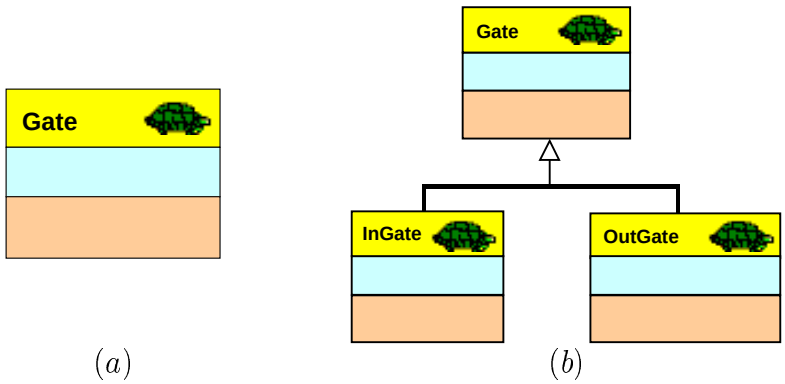


FIG. 3.2 – *Le type abstrait* Gate

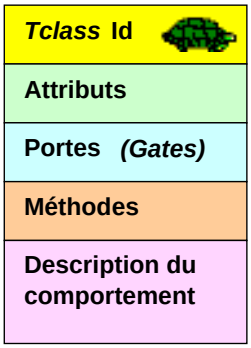
ii) Le stéréotype *Tclass*

La figure 3.3 décrit la structure générale d'un stéréotype *Tclass* : les attributs de type *Gate* y sont séparés des autres attributs. Les propriétés principales des *Tclass* sont les suivantes :

Prop.1 Tous les attributs, sauf ceux de type *Gate*, doivent être déclarés privés (-) ou protégés (#).

Prop.2 Toutes les méthodes, sauf les constructeurs, doivent être déclarées privées (-) ou protégées (#). Les constructeurs peuvent être déclarés publics.

Prop.3 Un diagramme d'activité doit accompagner chaque *Tclass* et décrire ainsi son comportement. Ce diagramme peut utiliser tout attribut ou classe déclarés ou hérités dans la Tclass.



- ☐ Identificateur de la classe de stéréotype *Tclass*.
- ☐ Liste d'attributs, excepté les attributs de type *Gate*.
- ☐ Liste d'attributs de type *Gate*. Ils peuvent être déclarés publics (+), privés (-), ou protégés (#).
- ☐ Liste de méthodes, incluant un constructeur.
- ☐ Diagramme d'activité qui peut faire référence aux attributs, portes et méthodes précédemment définis, ou hérités.

FIG. 3.3 – *Structure d'une classe stéréotypée* Tclass

iii) Diagramme de classes

Un diagramme de classes UML est une représentation graphique d'un ensemble de classes interconnectées par des relations, telles que des associations. Le profil TURTLE donne une sémantique formelle aux relations d'association. Pour cela, nous définissons le type abstrait *Composer*, qui ne doit pas être utilisé directement : une association entre deux *Tclasses* est attribuée par d'autres types héritiers de *Composer*. Ces types sont : *Parallel*, *Synchro*, *Sequence* et *Preemption*. Deux d'entre eux sont représentés dans la figure 3.4.

Toute association entre deux *Tclasses* doit avoir une sémantique bien définie et ainsi n'être attribuée que par une seule classe associative de type *Composer*. Les classes suivantes héritent de *Composer* :

Parallel Les deux *Tclass* mises en relation par une association à laquelle est attribué cet opérateur sont exécutées en parallèle et sans synchronisation. Les deux *Tclass* doivent être des classes actives[3].

3. Une classe est en activité si elle représente un flux d'exécution du système [Dou99].

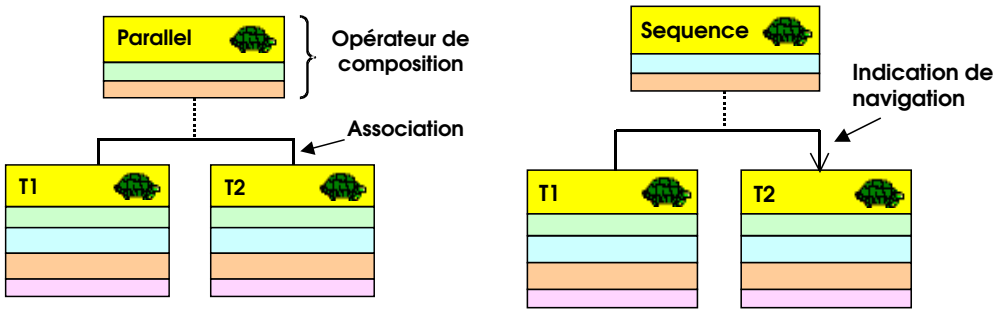


FIG. 3.4 – *Opérateurs de composition héritant de* Composer

Synchro Les deux *Tclass* mises en relation par une association à laquelle est attribué cet opérateur réalisent des synchronisations entre elles dans deux flux d'exécution séparés. Cette synchronisation peut donner lieu à un échange de données dont le format est précisé lors de l'appel, dans le diagramme d'activité. Si l'association entre les deux *Tclass* comporte un sens de navigation, alors l'échange de données ne peut se réaliser que dans le sens indiqué par la navigation. Deux *Tclass* doivent se synchroniser sur deux *Gates*, qui doivent être listées dans une formule OCL (*Object Constraint Language*). Par exemple, supposons que les portes `g1` et `g2`, de type *Gate* et de la *Tclass* `T1`, se synchronisent respectivement avec les portes `g3` et `g4` de type *Gate* de la *Tclass* `T2`. Dans ce cas, la formule OCL qui accompagne la relation d'association entre `T1` et `T2` doit être `{T1.g1 = T2.g3 and T1.g2 = T2.g4}`. À chaque fois que `T1` réalise une action sur `g1`, elle doit attendre que la classe `T2` réalise une action sur `g3` et réciproquement. Lorsque l'action est enfin réalisée par les deux classes, l'échange de données a lieu, et les deux classes poursuivent leurs activités respectives.

Sequence Les deux *Tclass*, mises en relation par une association à laquelle est attribué cet opérateur, sont exécutées l'une après l'autre, dans le sens donné par la navigation de l'association. Dans la relation `T1` *Sequence* `T2`, `T1` doit se terminer[4] avant que `T2` ne débute. Les deux *Tclasses* doivent être des classes actives.

Preemption La *Tclass* désignée par la navigation d'une association attribuée par l'opérateur *Preemption* peut interrompre l'autre *Tclass* à tout moment. Dans la pratique, `T1` *Preemption* `T2` signifie que `T2` peut interrompre `T1`, c'est-à-dire que `T1` est tuée et `T2` devient alors active.

Priorité entre *Composers* Pour éviter tout risque de confusion qui pourrait apparaître lors de l'utilisation conjointe de ces association entre *Tclasses* nous avons défini des priorités syntaxiques. L'ordre des priorités que nous avons choisi est inspiré par notre expérience et par ce qui nous semble "naturel" d'écrire lorsque nous concevons un système.

$$Preemption > Sequence > Synchro > Parallel$$

4. On dit qu'une *Tclass* se termine lorsque toutes les activités associées à cette classe ont atteint leur point de terminaison.

Syntaxe abstraite des *Composers* Les règles de grammaire représentées dans la figure 3.5 indiquent de quelle manière les différents *Composers* s'assemblent.

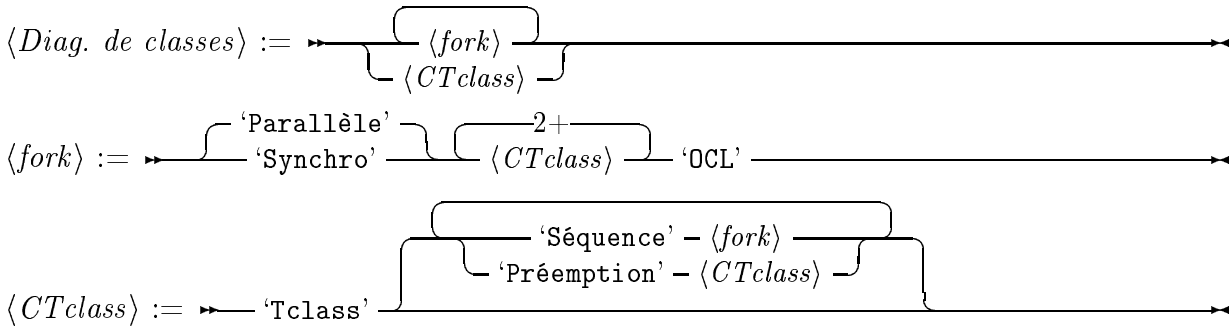


FIG. 3.5 – *Règles de composition des diagrammes de classes* ***TURTLE***

iv) Diagrammes d'activité

Les symboles natifs des diagrammes d'activité `UML` sont supportés, mais les opérations d'appel ne sont pas traduites en RT-LOTOS, puisque deux *Tclasses* peuvent communiquer en se synchronisant sur des *Gates*. La figure 3.6 liste les symboles non temporels utilisables dans les diagrammes d'activité `TURTLE` (notés *AD*).

Nous avons recensé les besoins suivants en terme d'expression du temps :

- Expression d'un intervalle temporel pendant lequel la réalisation d'une action est attendue.
- Expression d'un système de déroutement lorsqu'une action ne s'est pas réalisée dans l'intervalle voulu.
- Expression du non-déterminisme temporel, à savoir le fait qu'une action réalisée par une activité se produise après un délai variable, non connu a priori.

Pour répondre à ces besoins, nous nous sommes inspirés des opérateurs temporels proposés par RT-LOTOS :

- L'intervalle temporel pendant lequel la réalisation d'une action est attendue est exprimé en composant l'opérateur de délai et l'opérateur d'offre limitée dans le temps.
- Le mécanisme de déroutement est réalisé au niveau de l'offre limitée dans le temps.
- Le non-déterminisme temporel est exprimé par l'opérateur de latence qui décrit un délai non déterminé mais borné.

La figure 3.7 présente les trois symboles retenus pour les opérateurs temporels. S'y ajoutent un opérateur de délai applicable à un intervalle temporel, cumulant ainsi les effets d'une durée fixe égale à la borne inférieure de l'intervalle et d'une latence égale à la différence entre les bornes supérieure et inférieure de l'intervalle.

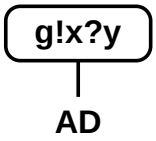
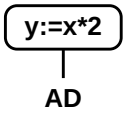
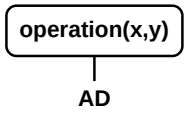
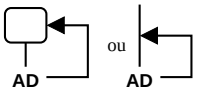
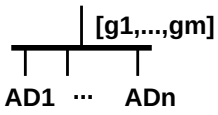
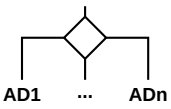
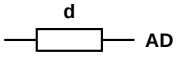
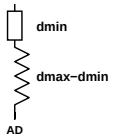
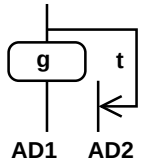
Litteral	Activité TURTLE	Description
Début		Début du diagramme d'activité.
Action		Appel (ou offre de synchronisation) sur la *Gate* g avec échange possible de valeurs. *AD* est ensuite interprété.
Assignation		Assignation d'une valeur à un attribut. *AD* est ensuite interprété.
Appel		Appel d'une méthode *statique* (i.e. sans effets de bords sur l'activité en cours). *AD* est ensuite interprété.
Boucle		Structure de boucle. *AD* est ensuite interprété à chaque fois que l'interpréteur entre dans la boucle.
Parallèle		Synchronisation sur les *Gates* g1,... gm entre les n sous-activités décrites par $AD_1, \ldots AD_n$ qui s'exécutent en parallèle. La liste des *Gates* est éventuellement vide. À des fins de documentation, les sous-activités peuvent être séparées par des tirets verticaux *(swinlanes)*.
Choix		Choix parmi les $AD_1, \ldots AD_n$ sous-activités pour lesquelles la *garde* est *valide* et qui sont prêtes à s'exécuter.
Jonction		La terminaison conjointe des n sous-activités AD_1, $\ldots AD_n$ est suivie par l'exécution en parallèle des m sous-activités *AD'1*, *...AD'm*. Ces dernières ont éventuellement la possibilité de se synchroniser sur k *Gates* g1, ... gk.
Stop		Termination de l'activité.

FIG. 3.6 – *Opérateurs non temporels des diagrammes d'activité* TURTLE

Litteral	Activité TURTLE	Description
Délai		Retard déterministe. AD est interprété après d unités de temps.
Latence		Retard non déterministe. AD est interprété après au plus t unités de temps.
Intervalle temporel		Délai non déterministe. AD est interprété après au moins $dmin$ et au plus $dmax$ unités de temps.
Offre limitée		Offre limitée dans le temps. La synchronisation sur la *Gate* g est offerte pendant au plus t unités de temps. Si la synchronisation se réalise, AD_1 est interprété, sinon AD_2 est interprété.*

*Lorsque le symbole de *latence* est suivi du symbole d'*offre limitée* dans une activité, le délai de latence et la limite d'offre débutent au même instant.

FIG. 3.7 – *Opérateurs temporels des diagrammes d'activité* ***TURTLE***

Syntaxe abstraite des diagrammes d'activité Les règles de grammaire représentées dans la figure 3.8 indiquent de quelle manière les différents symboles précédents s'assemblent dans les diagrammes d'activité TURTLE.

3.1.3 Mise en œuvre de TURTLE

Le profil TURTLE a été développé pour valider les modélisations de système temps-réel vis-à-vis d'erreurs de conception, et plus particulièrement d'incohérences de synchronisation. La figure 3.9 dépeint ce processus de validation. Les classes TURTLE et leurs relations de composition sont extraites du diagramme de classes, sauvées dans un fichier XMI (*XML Metadat Interchange*, format d'échange de données entres outils UML normalisé à l'OMG), et converties en spécification RT-LOTOS pour être validées à l'aide de l'outil rtl. Les systèmes peuvent être vérifiés en utilisant les techniques d'analyse d'accessibilité, ou bien validés en simulation (exploration partielle du comportement du système).

i) Traduction

La traduction des diagrammes TURTLE se passe en deux étapes :

1. Traduction des *Tclasses*.
   A chaque *Tclass* correspond un processus RT-LOTOS qui porte le même identificateur, dont les portes observables sont les *Gates* déclarées publiques, dont les portes intériorisées sont les *Gates* déclarées privées ou protégées, dont les paramètres sont les attributs et dont la spécification RT-LOTOS est donnée par traduction du diagramme d'activité de la *Tclass*.

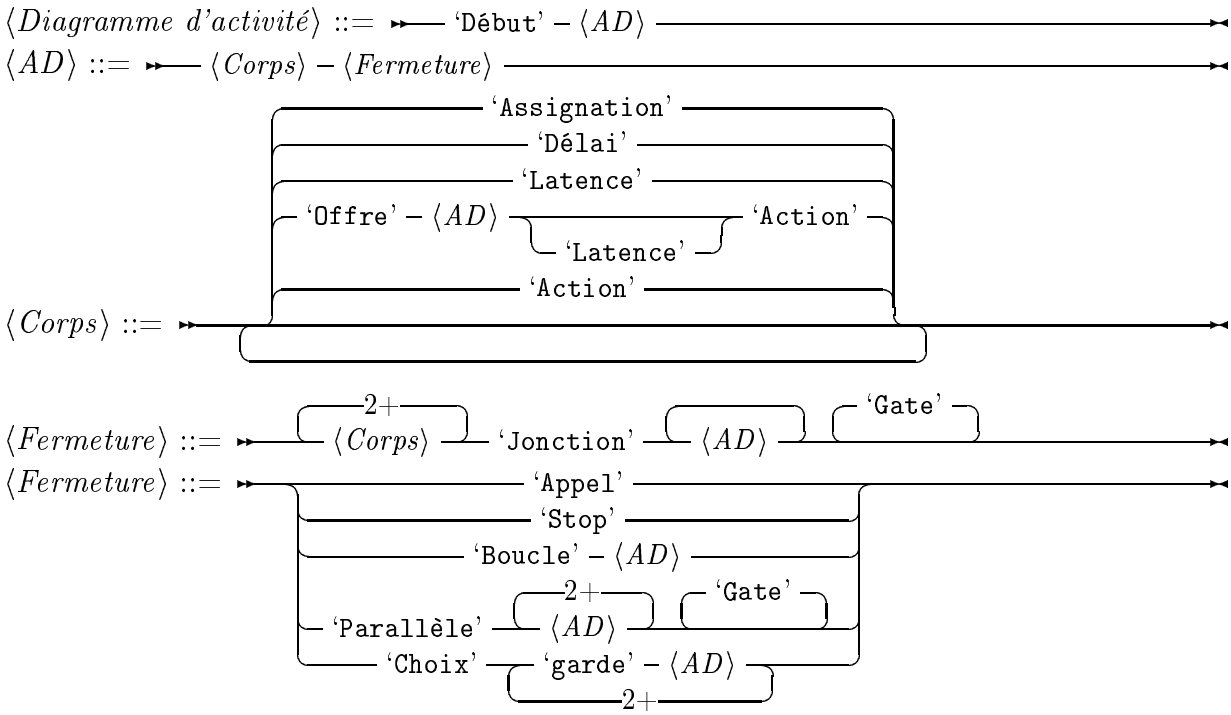


FIG. 3.8 – *Règles de composition des symboles d'activité TURTLE*

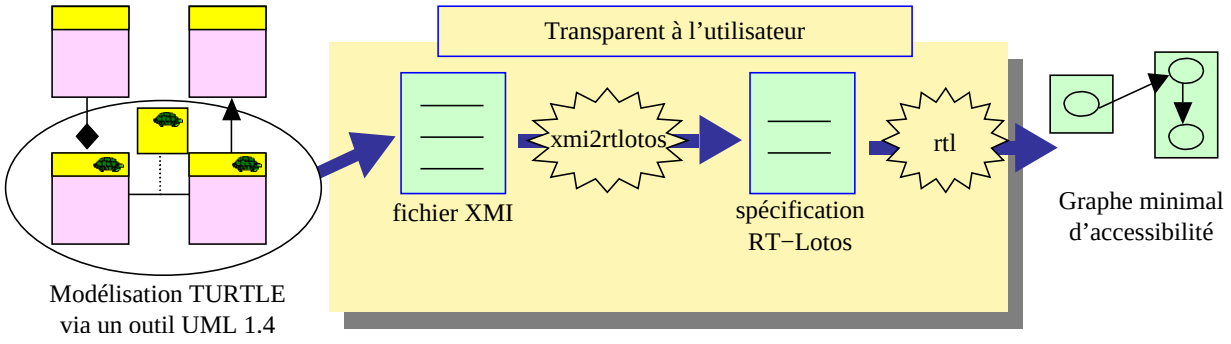


FIG. 3.9 – *De la modélisation TURTLE à la vérification*

La figure 3.10 donne la traduction en RT-LOTOS de chacun des symboles d'activité TURTLE présentés figure 3.6 et figure 3.7. Dans cette figure, $\mathcal{T}$(AD) dénote la traduction de la sous-activité AD qui suit le symbole.

2. Traduction des *Composers* entre *Tclasses*.

   Les *Composers* vont se traduire pour l'essentiel par des compositions entre les processus RT-LOTOS construits précédemment, avec parfois une réécriture de certains processus RT-LOTOS ainsi que des renommages de portes. Le détail des algorithmes de traduction des *Composers* est donné dans la section 3.2. Les principales étapes sont les suivantes :

   (a) Traduction des *Tclasses* à l'origine de relation de *Preemption*.

   (b) Traduction des *Tclasses* à l'origine de *Sequence*.

   (c) Traduction des *Tclasses* qui ne sont à l'origine ni de *Preemption* ni de *Sequence*.

   (d) Réduction (et réécriture) des *Tclasses* à l'origine à la fois de *Sequence* et de *Preemption*.

   (e) Traduction des sous-ensembles connexes de *Tclasses* liées par des relations *Parallel* et *Synchro* (renommage des portes).

   (f) Traduction des *Tclasses* actives.

ii) Vérification

La vérification que nous proposons avec la méthodologie TURTLE s'appuie sur une analyse du graphe minimal d'accessibilité engendré à partir de la spécification RT-LOTOS issue de la traduction de la spécification TURTLE.

Les techniques de validations offertes sur une spécification RT-LOTOS ont été décrites dans la section 1.3. Concernant TURTLE, nous proposons d'exploiter le graphe minimal d'accessibilité à plusieurs niveaux :

Lecture directe. Si le graphe est de taille raisonnable, il peut être étudié par une personne qui décide si le fonctionnement du système est acceptable ou non, soit visuellement, soit en s'aidant de techniques et d'outils de manipulation de systèmes de transitions étiquetées (projection, équivalences, etc.)

Observateur. Si l'auteur souhaite valider certaines propriétés spécifiques sur son système, il compose sa spécification avec une *Tclass* observateur qui peut soit adopter le comportement nominal du système vis-à-vis de cette propriété, soit déclencher une action `erreur`. Si l'auteur ne voit pas apparaître d'occurrence de l'action `erreur` dans le graphe minimal d'accessibilité de la spécification globale, il a alors la preuve formelle que son système vérifie la propriété décrite par l'observateur.

Confrontation aux diagrammes de séquences. Lors de l'ébauche de son système, l'auteur a défini des diagrammes de séquences. Ces diagrammes permettent de décrire des scénarios, de représenter une ou plusieurs traces d'exécution du système, en plaçant sur un chronogramme les interactions entre les composants du système et en ignorant les détails internes propres à chacun des participants. Moyennant un

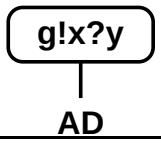
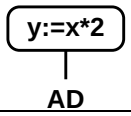
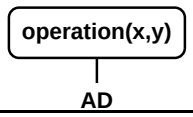
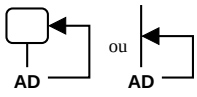
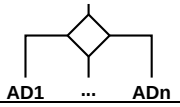
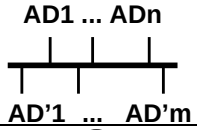
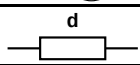
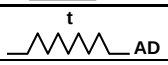
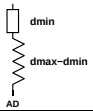
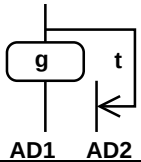
Activité TURTLE	Traduction
	$\mathcal{T}$(AD)
	g !x ?y:nat ; $\mathcal{T}$(AD)
	let y : YType = x*2 in $\mathcal{T}$(AD)
	$\mathcal{T}$(AD) (Appel non traduit)
	process LabelX[g1,...,gn]:noexit:= $\mathcal{T}$(AD) >> LabelX[g1,...gn] endproc
	$\mathcal{T}$(AD1) |[g1,...,gm]| ...|[g1,...,gm]| $\mathcal{T}$(ADn)
	[c1] -> $\mathcal{T}$(AD1) [] ...[] [cn] -> $\mathcal{T}$(ADn)
	($\mathcal{T}$(AD1) ||| ... ||| $\mathcal{T}$(ADn)) >> ($\mathcal{T}$(AD'1) |[g1,...,gk]| ...|[g1,...,gk]| $\mathcal{T}$(AD'm))
	exit
	delay(d) $\mathcal{T}$(AD)
	latency(t) $\mathcal{T}$(AD)
	delay(dmin,dmax) $\mathcal{T}$(AD)
	g{t, $\mathcal{T}$(AD2)}; $\mathcal{T}$(AD1)

FIG. 3.10 – *Traduction* RT-LOTOS *des activités* *TURTLE*

outillage simple (parcours de graphe), il est relativement aisé de vérifier que de telles traces appartiennent bien, au graphe minimal d'accessibilité du système spécifié avec TURTLE. Ainsi, l'auteur peut confronter sa spécification aux séquences d'exécution qu'il envisageait au départ.

3.1.4 Étude de cas

La méthodologie TURTLE a été mise en oeuvre sur des problèmes issus du monde industriel [AdSSSL02, Apv02], notamment dans le cadre de la reconfiguration dynamique de systèmes embarqués à bord de satellites. Nous l'illustrons ici sur un exemple académique bien connu : la machine à café.

Bien que simple, cette machine à café nous permet de mettre en évidence l'utilisation du stéréotypes *Tclass*, des types abstraits et des trois opérateurs temporels introduits dans TURTLE.

i) Conception

Nous considérons ici une première version d'une machine qui distribue thé et café après insertion de deux pièces de monnaie identiques (figure 3.11). Un retard excessif dans l'introduction de la deuxième pièce ou dans le choix de la boisson entraînera le remboursement de la (ou des) pièce(s) déjà inséré(es). Ces deux situations sont modélisées par des offres limitées dans le temps (`coinDelay` et `buttonDelay`). Des délais matérialisent le temps de réaction du bouton et le temps minimum de préparation d'une boisson. Ce temps de préparation peut varier, ce qui explique la présence de latences. Une latence est placée avant le `push` pour modéliser un comportement humain aléatoire. Notons enfin les contraintes ajoutées aux associations entre classes pour désigner les *Gates* impliquées dans chacune des synchronisations.

ii) Validation

La spécification RT-LOTOS de cette spécification TURTLE a été engendrée (voir annexe C). L'outil `rtl` en a donné le graphe minimal d'accessibilité et permis d'identifier les situations de blocage et les différents états accessibles.

La figure 3.12 présente un extrait du graphe minimal d'accessibilité de la spécification présentée dans la figure 3.11. Considérons la séquence d'actions qui correspond à un utilisateur désirant un thé. Supposons que cet utilisateur insère deux pièces mais attende trop longtemps. L'offre de synchronisation sur `tea` ou `coffee` expire, ce qui a pour effet d'éjecter les deux pièces. L'utilisateur appuie un bref instant plus tard sur le bouton *tea* (*Tclass* `Button`, *Gate* `push`). L'offre de synchronisation ne peut plus avoir lieu. L'utilisateur reprend ses pièces, pensant que la machine est en panne. Supposons qu'un deuxième utilisateur désirant du café arrive et insère deux pièces. L'offre de synchronisation sur `tea` n'ayant pas expiré (offre illimitée), un thé lui est instantanément servi.

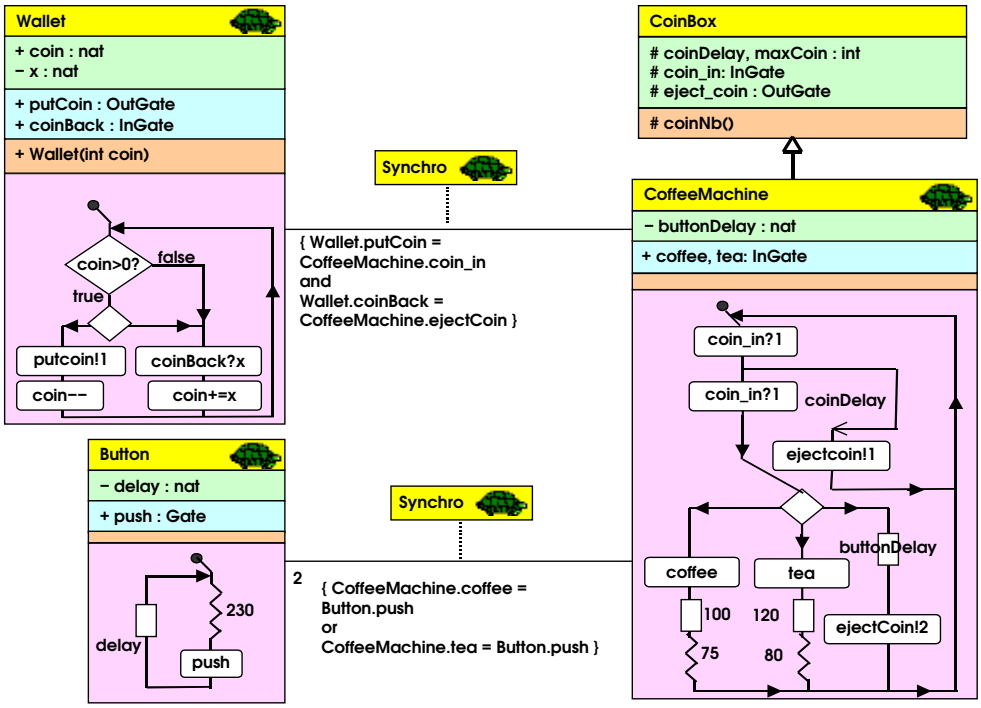


FIG. 3.11 – *Diagramme de classes n° 1 de la machine à café*

Pour corriger ce problème, nous proposons une deuxième version de la machine à café et modifions le diagramme selon la figure 3.11. Une synchronisation supplémentaire permet à la machine d'activer les boutons qui ne peuvent alors être actionnés que pendant un temps limité (offre sur `push` limitée à 40). Cette activation de bouton implique un délai de traitement dans le distributeur (délai de 50).

Cette modification du modèle appelle une nouvelle validation. La figure 3.14a présente le graphe d'accessibilité de la machine à café de la figure 3.13. Un état de contrôle (rectangle) est partitionné en classes de configurations équivalentes. Les conditions pour quitter une classe sont les suivantes : soit le temps s'est écoulé (transition `t`), soit une synchronisation a eu lieu sur une porte LOTOS. Considérons le graphe d'accessibilité de la 3.14a. Quitter l'état initial (état 0) requiert une synchronisation sur la porte `putCoin`. Dans l'état de contrôle 12, aucune synchronisation ne peut avoir lieu à partir du premier état : un changement d'état correspond exclusivement à une progression du temps (transition `t`). Par contre, quand l'offre sur la *Gate* `tea` ou `coffee` expire (délai `buttonDelay`), alors la synchronisation sur `coinBack` permet de passer de l'état 12 à l'état 15. Une valeur 2 est alors échangée.

Ce graphe met en évidence l'impossibilité pour un utilisateur d'obtenir du thé (ou du café). En effet, le délai d'activation du bouton (`push`) expire avant que la machine ne soit prête à délivrer du café ou du thé. Si ce délai passe de 40 à 60, le graphe de la figure 3.14b montre qu'il est alors possible d'avoir du thé ou du café (il s'agit d'une portion du graphe minimal d'accessibilité).

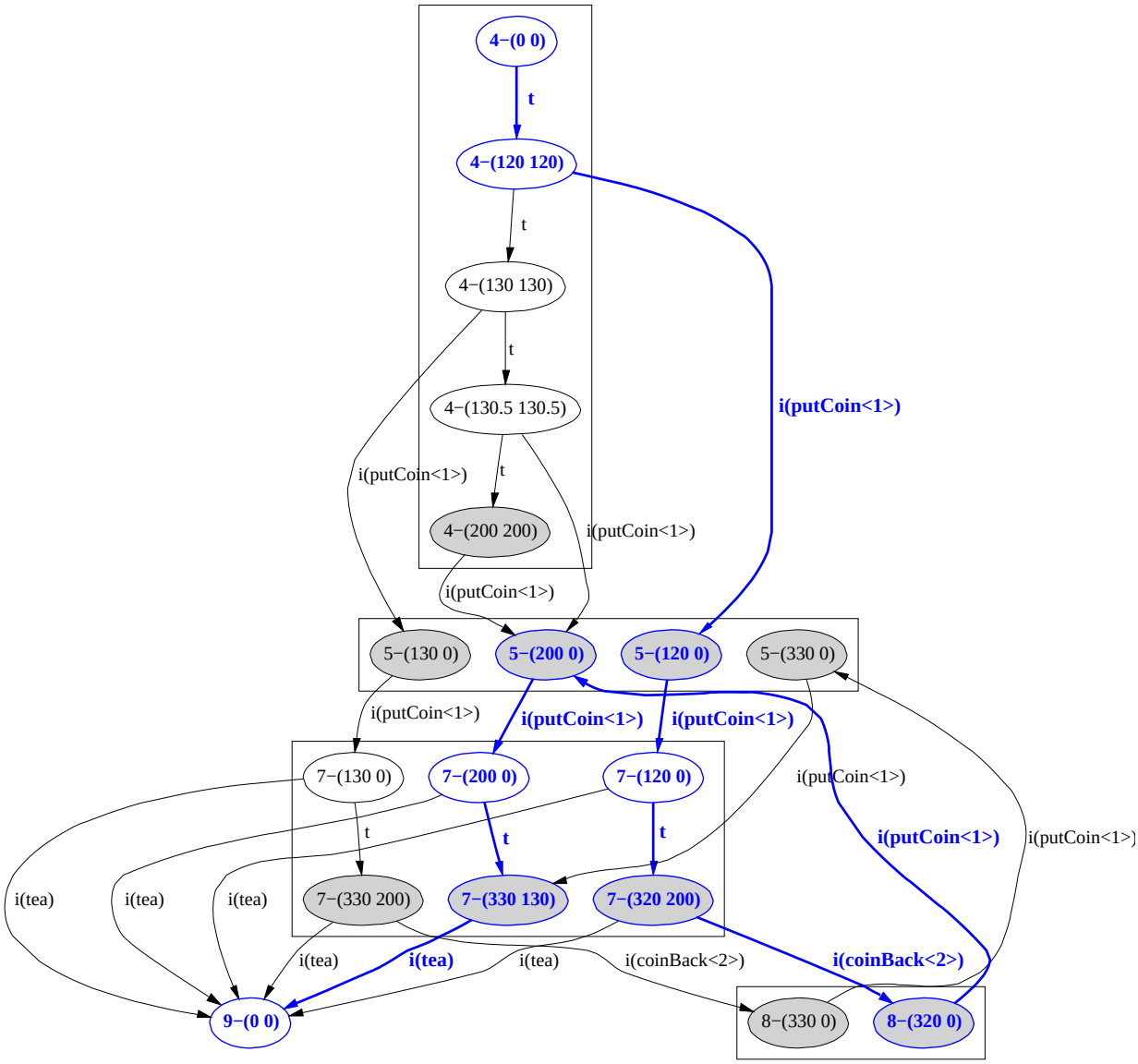


FIG. 3.12 – *Extrait du graphe minimal d'accessibilité de la machine à café n° 1*

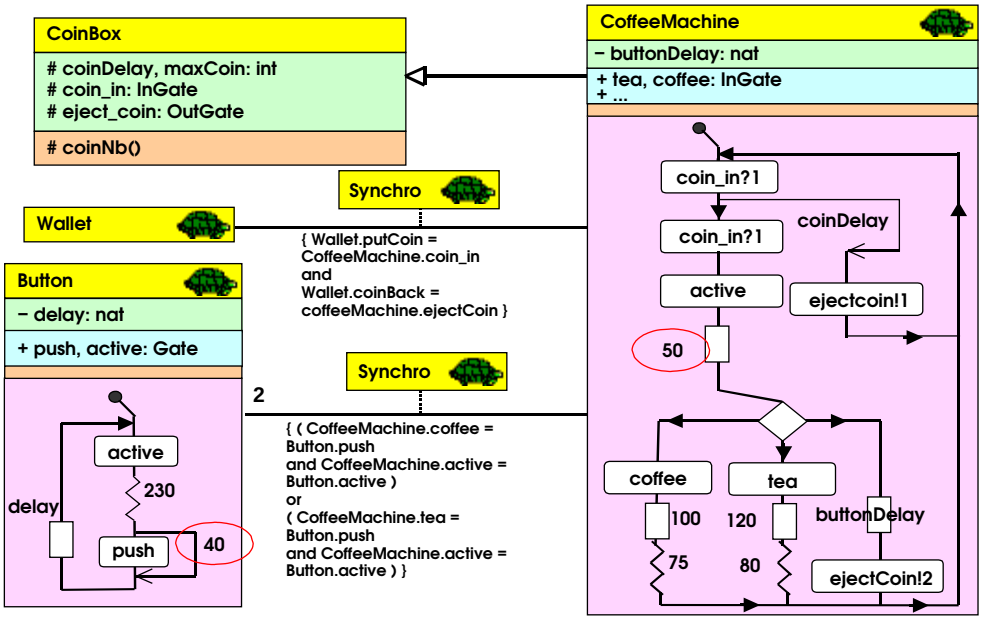


FIG. 3.13 – *Diagramme de classes n° 2 de la machine à café*

3.1.5 Les limites de la méthodologie

La méthodologie TURTLE est très attrayante mais rencontre néanmoins les limites récurrentes liées à l'expression de systèmes temporels, notamment par l'algèbre de processus RT-LOTOS, et à leur vérification.

i) Maîtrise de l'explosion combinatoire

La difficulté majeure est le risque d'explosion combinatoire : un nombre important d'actions avec un grand nombre de composantes parallèles produit un entrelacement important entres ces actions ce qui augmente le nombre d'états et de transitions de manière exponentielle. Le nombre d'horloges utilisées pour exprimer les contraintes temporelles du système est également un facteur d'explosion combinatoire.

Il est possible de contenir ce phénomène à deux niveaux : lors de la modélisation, et lors de vérification.

- La vérification s'appuie sur la construction du graphe minimal d'accessibilité. Dans ce graphe, un entrelacement prend la forme d'un motif en *losange*. Il existe des travaux pour appréhender le problème de l'explosion combinatoire et améliorer les performances de l'analyse, notamment les *graphes à pas couvrant* [VAM96, VM97], qui tendent à résorber ces structures en *losange*; les *ordres partiels* [Pel98, God96, Val93, GW93, BJLY98], qui exploitent l'indépendance entre les actions et définissent des séquences d'actions équivalentes; et les *symétries* [God99, EJP97]. L'idée générale est de définir des classes d'états

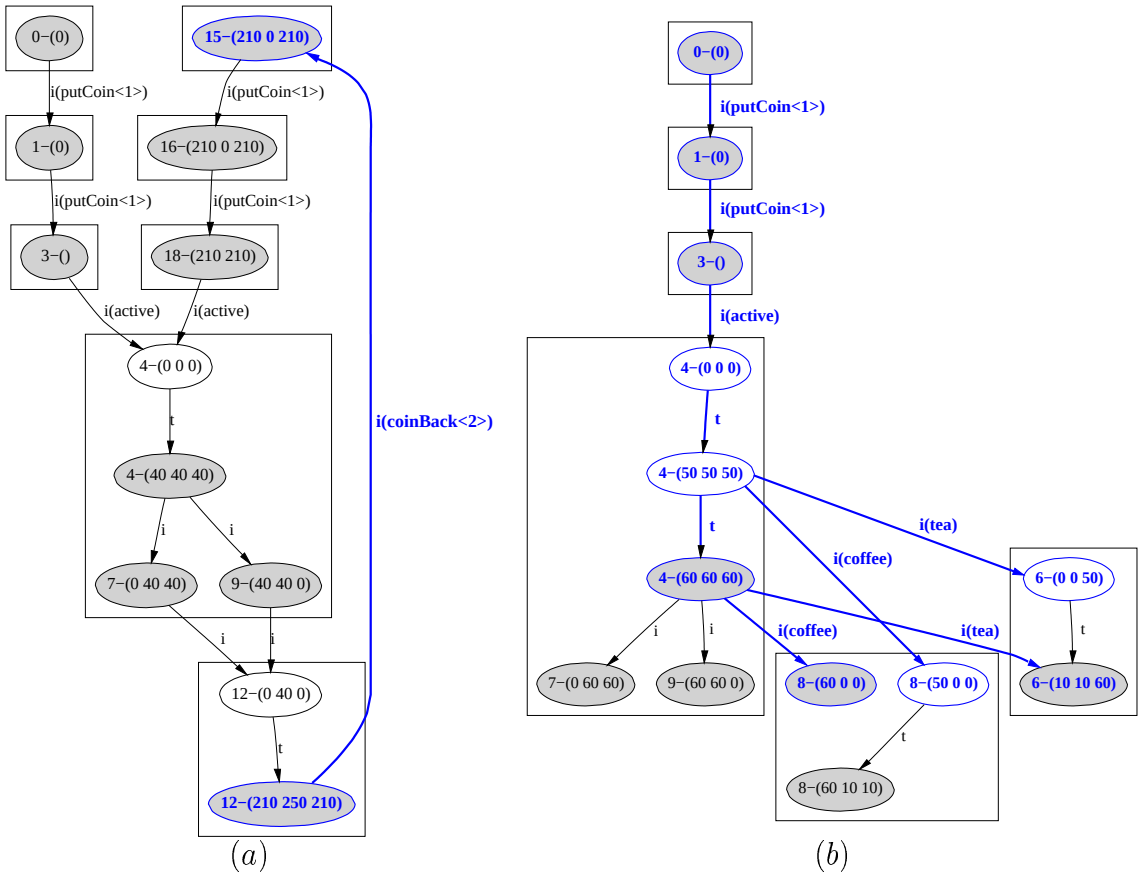


FIG. 3.14 – *Graphe minimal d'accessibilité de la machine à café n° 2 avec une offre sur les boutons limitée à 40 (graphe (a)) puis à 60 (graphe (b))*

équivalents et de n'explorer qu'un état par classe.

Toutefois, actuellement, ces techniques ne sont pas implémentées dans les outils de vérification disponibles pour RT-LOTOS.

◦ La modélisation s'appuie sur la spécification en RT-LOTOS, produite ici par la traduction de la spécification TURTLE. En adoptant un *style de spécification* RT-LOTOS adéquat, nous pouvons limiter l'explosion combinatoire.

Tout d'abord en ayant recours à des *actions composites*. Une action composite est une action LOTOS classique mais qui a une interprétation particulière pour l'auteur de la spécification : elle regroupe en elle-même plusieurs *actions logiques* qui doivent, a priori, se produire au même instant. Ainsi, l'occurrence d'une action LOTOS composite va signifier pour l'auteur l'occurrence à ce même instant des divers actions logiques qui la composent. Cela évite de voir apparaître l'entrelacement qui se serait produit si l'auteur avait choisi de modéliser chaque action logique par une action LOTOS indépendante.

Ensuite, lorsque l'auteur connaît a priori le séquencement des actions qu'il spécifie, il peut tout simplement composer sa spécification avec un processus qui décrit ce séquencement d'actions. De cette manière il peut contrôler l'entrelacement des actions.

Toutefois, actuellement, ces constructions ad-hoc ne sont pas mises en œuvre par les algorithmes de traduction liés à TURTLE.

ii) Implémentation de l'opérateur de mémoire temporelle

Le langage RT-LOTOS propose un opérateur de mémoire temporelle sur les actions (voir page 36). Il est noté `@` : par exemple `a@x` signifie que la variable `x` prendra pour valeur le temps écoulé entre la date à laquelle l'action `a` est offerte et la date de son occurrence. Cet opérateur permet des constructions très souples et très puissantes. Il sera notamment utilisé dans l'extension au profil TURTLE présenté page 147.

Le handicap majeur est que, pour l'instant, cet opérateur n'est pas supporté par l'outil de vérification `rtl` : il est utilisable en simulation, mais pas en analyse d'accessibilité. Cela tient au fait que la valeur du temps collectée par cet opérateur peut conditionner la suite du comportement du processus. Dans l'exemple précédent, la variable `x` pourrait être utilisée dans un délai, un choix, etc., dans la suite de la spécification. Dans une telle situation, l'algorithme de calcul du graphe minimal d'accessibilité doit a priori explorer tout l'espace des valeurs temporelles $\mathbb{Q}$ que cette variable peut prendre.

Une solution pour palier ce problème a été proposée dans [Her97]. L'auteur définit des *automates temporisés avec temporisateurs* qui incluent une fonction définie pour chaque horloge de chaque état qui indique si l'horloge croît avec le temps ou non (on parle alors de *temporisateur* et non plus d'horloge). Cette fonction peut être paramétrée lors de chaque transition. Lorsque l'auteur traduit l'opérateur ET-LOTOS de capture temporelle `a@x`, il ajoute à l'automate un nouveau temporisateur associé à l'action `a`. Ce temporisateur est initialisé à zéro lorsque le système entre dans l'état correspondant à l'offre de l'action `a`, et il croit avec le temps. Puis, lorsque le système réalise l'action `a`, ce temporisateur voit sa fonction d'accroissement positionnée à zéro. Sa valeur pourra ensuite être testée dans les

gardes des transitions et les invariants des états. Une telle approche n'a pas été implémentée dans l'outil `rtl`.

3.2 Algorithmes pour traduire les diagrammes de classes TURTLE

Nous présentons ici le détail des algorithmes évoqués dans la section 3.1.3 permettant de traduire en RT-LOTOS les *Composers* présents dans les diagrammes de classes `TURTLE`. Le diagramme d'activité de chaque *Tclass* a été traduit lors d'une phase précédente qui a engendré pour chacune d'elle un processus RT-LOTOS. Il s'agit maintenant de composer entre eux ces processus selon les relations décrites par les *Composers* du diagramme de classes. L'annexe C présent la spécification RT-LOTOS engendrée en appliquant ces algorithmes à la spécification `TURTLE` de la machine à café présentée dans la figure 3.11.

3.2.1 Définitions préliminaires

Les algorithmes donnés dans les sections suivantes manipulent un certain nombre d'entités. Nous les définissons ici.

Définition 11 (Ensemble de *Tclasses*)
$\mathcal{T} = \{T_i \ / \ i \in [1,n_T]\}$

Définition 12 (Tclass)
$T = \langle Label, GateList, traduction\_AD, traduction\_CD \rangle$
Label, traduction_AD et traduction_CD sont des chaînes de caractères; GateList est un ensemble de $gate\_ref$*, avec* $gate\_ref = \langle process\_name, gate\_name, call\_position \rangle$

Définition 13 (Ensemble de Relations (*Composers*))
$\mathcal{R} = \{R_i \ / \ i \in [1,n_R]\}$

Définition 14 (Relation (*Composer*))
$R = \langle T, type, OCL, T' \rangle$ *avec* $T, T' \in \mathcal{T}$

Définition 15 (Type d'une relation)
$type \in \{Parallel, Synchro, Sequence, Preemption\}$

Définition 16 (Formule OCL)
La formule OCL définit un ensemble d'associations $\langle gate\_ref, gate\_ref' \rangle$

Nous avons également besoin des fonctions suivantes :

- $LabelOf(Tclass)$ retourne la chaîne *label* de $Tclass$ qui deviendra le nom du processus RT-LOTOS décrit par la chaîne $traduction\_AD$
- $NameOf(Tclass)$ retourne la chaîne $"\texttt{p}" + label$ qui deviendra le nom du processus RT-LOTOS décrit par la chaîne $traduction\_CD$

Excepté dans la sous-section iv) nous ne modifierons pas les processus RT-LOTOS qui décrivent les diagrammes d'activité des *Tclasses*. Ils ont été engendrés dans une phase précédente : le tableau de traduction 3.10 de la sous-section 3.1.3 nous a permis de construire la chaîne $traduction\_AD$. Ils portent comme identificateur $LabelOf(Tclass)$.

Par contre, le but des algorithmes présentés ici est de construire une spécification RT-LOTOS qui compose, d'après le diagramme de classes, ces processus qui décrivent les activités des *Tclasses*. Pour cela, nous sommes amenés à créer de nouveaux processus qui composent entre eux les processus $LabelOf(Tclass)$ (typiquement pour décrire les *Composers Sequence* et *Preemption*). Ils seront décrits par la chaîne $traduction\_CD$ que nous allons construire. Ils porteront comme identificateur $NameOf(Tclass)$ de la *Tclass* à l'origine des relations de *Sequences* et de *Preemption*. Ensuite, ces nouveaux processus seront composés entre selon les relations de *Parallel* et de *Synchro* du diagramme de classes.

Aussi, ces algorithmes n'effectuent aucune vérification de conformité vis-à-vis du profil TURTLE défini en 3.1.2 et présupposent qu'il est parfaitement respecté.

Par ailleurs, il est important de bien garder présent à l'esprit la différence entre une *classe* et une *instance de classe*. Considérons l'exemple de la figure 3.15 : une *Tclass* A peut être préemptée par deux *Tclasses* B et C, avec de plus, une séquence de B à C, et pour finir les *Tclasses* A et C définies actives au démarrage. Supposons qu'à un moment donné, B préempte A. Alors, à la terminaison de B il y aura deux instances actives de la *Tclass* C : celle activée au démarrage et celle activée en séquence de B.

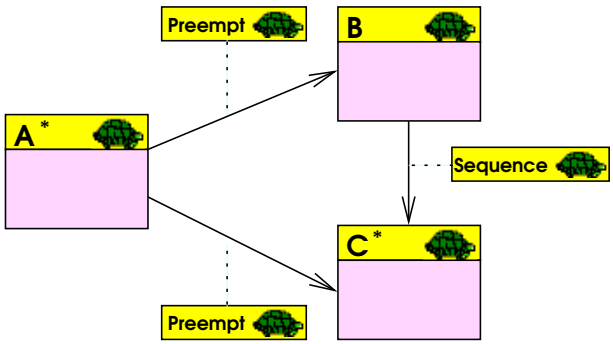


FIG. 3.15 – *Instance de classes*

Cet exemple illustre le fait que l'aspect dynamique du système repose sur la notion d'instance de classe, alors que le diagramme de classes ne donne qu'une vue statique du système : son architecture. L'auteur de la spécification du système définit quelles sont les classes instanciées à l'initialisation du système au moyen du diagramme d'objets. Puis, au cours de la vie du système, des classes seront instanciées au grès des compositions par des relations de *Sequence* et de *Preemption*. Ce comportement se traduit naturellement dans la spécification RT-LOTOS par l'instanciation des processus traduisant les *Tclasses*.

3.2.2 Traduction des *Tclasses* à l'origine de relations de *Preemption*

Nous traitons ici des *Tclasses* qui sont à l'origine de relations de *Preemption* et qui ne sont pas à l'origine de relations de *Sequence*.

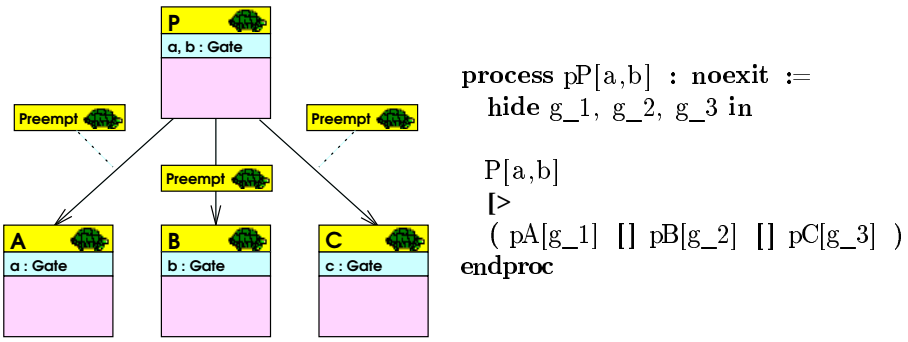


FIG. 3.16 – Tclass *à l'origine de plusieurs relations de* Preemption

Comme l'illustre la figure 3.16, une *Tclass* peut être à l'origine de plusieurs relations de *Preemption*. Cela se traduit, au niveau RT-LOTOS, par le fait que le processus issu de la *Tclass* préemptible est mis en relation par l'opérateur *disrupt* (noté [>), avec une composition par une relation de choix non déterministe (notée []) de tous les processus RT-LOTOS issus des *Tclasses* pointées par les relations de *Preemption*.

Nous définissons un nouveau processus RT-LOTOS englobant cette construction. Il porte l'identificateur $NameOf(Tclass)$ de la *Tclass* à l'origine des relations de *Preemption*.

De plus, pour éviter toute éventuelle ambiguïté sur les noms des portes LOTOS nous allons effectuer un renommage des portes.

Ces opérations sont réalisées par l'algorithme 3.1.

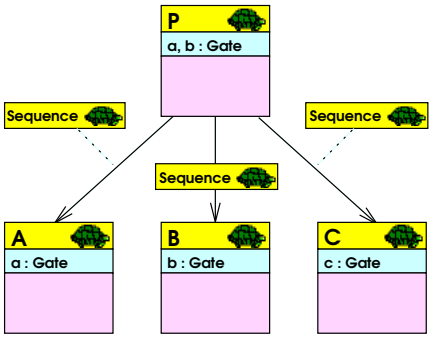
3.2.3 Traduction des *Tclasses* à l'origine de relations de *Sequence*

Nous traitons ici des *Tclasses* qui sont l'origine de relations de *Sequence* et qui ne sont pas à l'origine de relations de *Preemption*.

Comme l'illustre la figure 3.17, une *Tclass* peut être à l'origine de plusieurs relations de *Sequence*. Cela signifie que lorsque cette *Tclass* atteint son point de terminaison, les *Tclasses* pointées par les relations de *Sequence* deviennent toutes actives avec un parallélisme implicite entre elles. Cela se traduit, au niveau RT-LOTOS, par le fait que le processus issu de la *Tclass* de tête est mis en relation par l'opérateur de séquence LOTOS (noté >>), avec une composition par une relation de parallélisme total (notée |||) entre tous les processus RT-LOTOS issus des *Tclasses* pointées par les relations de *Sequence*. Cette construction de parallélisme total est réalisée en appelant la procédure $TranslatePS$ prévue pour traduire les *Composers*, *Parallel* et *Synchro* (voir la section 3.2.6). De même, si

Algorithme 3.1 Traduction des *Tclasses* à l'origine de relations de *Preemption*

for all $T_i \in \mathcal{T}$ **do**
  let $S : String$
  let $\mathcal{T}' := \{T_j \ / \ T_j \in \mathcal{T} \ \wedge \ \langle T_i, Preemption, OCL, T_j \rangle \in \mathcal{R}\}$
  if $\mathcal{T}' \neq \emptyset$ **then**
    let $n\_gate : Natural := \sum_{T_j \in \mathcal{T}'} NbGate(T_j)$
    if $n\_gate > 0$ **then**
      $S := "$`hide g_0,...,g_(`n_gate`) in`$"$
    end if
    let $st\_gates : String := ""$
    if $GatesOf(T_i) \neq \emptyset$ **then**
      $st\_gates := "[" + GatesOf(T_i) + "]"$
    end if
    $S \mathrel{+}= LabelOf(T_i) + st\_gates + "$`[> (`$"$
    let $cpt : Natural := 0$
    for all $T_j \in \mathcal{T}'$ **do**
      $S \mathrel{+}= NameOf(T_j) + "$`[g_(`$cpt$`),...,g_(`$cpt + NbGate(T_j)$`)]`$"$
      if $T_j \neq LastOf(\mathcal{T}')$ **then**
        $S \mathrel{+}= "$`[]`$"$
      end if
      $cpt \mathrel{+}= NbGate(T_j)$
    end for
    $S \mathrel{+}= ")"$
    $T_i.traduction\_CD := "$`process `$" + NameOf(T_i) + st\_gates +$
      $"$`: noexit :=`$" + S + "$`endproc`$"$
  end if
end for




```
process pP[a,b] : noexit :=
  hide g_1, g_2, g_3 in

  P[a,b]
  >>
  ( pA[g_1] ||| pB[g_2] ||| pC[g_3] )
endproc
```

FIG. 3.17 – Tclass *à l'origine de plusieurs relations de* Sequence

des *Tclasses* pointées par les relations de *Sequence* sont associées entre elles par des relations *Synchro*, la construction de parallélisme avec synchronisation est également réalisée par la procédure $TranslatePS$.

Comme précédemment, nous définissons un nouveau processus RT-LOTOS englobant cette construction. Il porte l'identificateur $NameOf(Tclass)$ de la *Tclass* à l'origine des *Sequences*.

De plus, pour éviter toute éventuelle ambiguïté sur les noms des portes LOTOS nous allons effectuer un renommage des portes. Ce renommage nous est également fourni par la procédure $TranslatePS$.

Ces opérations sont réalisées par l'algorithme 3.2.

Algorithme 3.2 Traduction des *Tclasses* à l'origine de relations de *Sequence*

for all $T_i \in \mathcal{T}$ **do**
  let $S : String$
  let $\mathcal{T}' := \{T_j \ / \ T_j \in \mathcal{T} \ \wedge \ \langle T_i, Sequence, OCL, T_j \rangle \in \mathcal{R}\}$
  if $\mathcal{T}' \neq \emptyset$ **then**
    let $st\_psi, st\_hide : String$
    $TranslatePS(\mathcal{T}', \mathcal{R}, \text{out } st\_psi, \text{out } st\_hide)$
    let $st\_gates : String := ""$
    if $GatesOf(T_i) \neq \emptyset$ **then**
      $st\_gates := "[" + GatesOf(T_i) + "]"$
    end if
    $T_i.traduction\_CD := "\texttt{process } " + NameOf(T_i) + st\_gates +$
      $" : \texttt{ noexit } :=" + st\_hide + st\_psi + "\texttt{endproc}"$
  end if
end for

3.2.4 Traduction des *Tclasses* qui ne sont à pas l'origine de relations de *Preemption* ou de *Sequence*

Pour les *Tclasses* qui ne participent pas à des relations de *Preemption* ou de *Sequence* ainsi que celles qui sont pointées par des relation de *Preemption* ou de *Sequence*, nous construisons simplement un processus RT-LOTOS portant l'identificateur $NameOf(Tclass)$ qui se contente d'appeler le processus RT-LOTOS portant l'identificateur $LabelOf(Tclass)$. Ceci est réalisé par l'algorithme 3.3.

Algorithme 3.3 Traduction des *Tclasses* qui ne sont à pas l'origine de relations de *Preemption* ou de *Sequence*

for all $T_i \in \mathcal{T}$ **do**
  let $S : String$
  let $\mathcal{T}' := \{T_j \ / \ T_j \in \mathcal{T} \ \wedge \ \langle T_i, type, T_j \rangle \in \mathcal{R} \ \wedge \ type \notin \{Sequence, Preemption\}\}$
  if $\mathcal{T}' \neq \emptyset$ **then**
    let $st\_gates : String := ""$
    if $GatesOf(T_i) \neq \emptyset$ **then**
      $st\_gates := "[" + GatesOf(T_i) + "]"$
    end if
    $T_i.traduction\_CD := "\texttt{process } " + NameOf(T_i) + st\_gates +$
      $" : \texttt{ noexit } :=" + LabelOf(T_i) + st\_gates + "\texttt{endproc}"$
  end if
end for

Cette construction peut sembler superflue, ou du moins, optimisable. Cependant nous avons choisi de la conserver car elle permet un déroulement homogène des algorithmes de traduction, et facilite la re-lecture et la compréhension du code RT-LOTOS engendré.

3.2.5 Réduction des *Tclasses* à l'origine à la fois de relations de *Sequence* et de *Preemption*

i) Impossibilité d'une écriture directe RT-LOTOS

Considérons par exemple un diagramme de classes dans lequel une *Tclass* A est à l'origine à la fois d'une relation de *Preemption* par B et d'une relation de *Sequence* vers C (voir la figure 3.18).

Nous ne pouvons écrire directement en RT-LOTOS (`A [> B) >> C` car cela signifie que si B prend la main sur A, il sera malgré tout suivi par C.

Nous ne pouvons pas non plus écrire (`A >> C) [> B` car cela signifie que lorsque C prend la suite de A, il peut être lui aussi préempté par B.

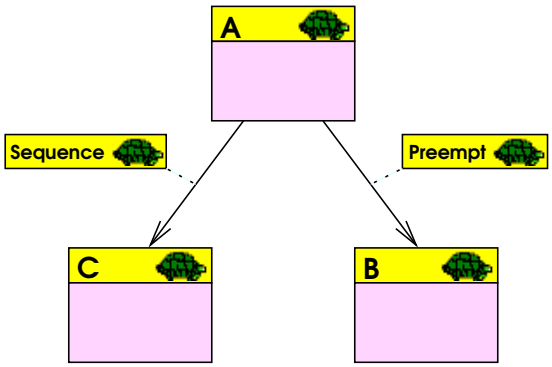


FIG. 3.18 – Tclasses *à l'origine à la fois de relations de* Sequence *et de* Preemption

Ces comportements ne sont pas spécifiés dans le diagramme de classes : soit A est préempté par B soit il est suivi en séquence par C, mais il n'y a aucune relation de composition entre B et C. Il faut une construction RT-LOTOS qui exprime cela.

ii) Élaboration d'une proposition

Poursuivons sur ce même exemple. L'étape exposée section 3.2.3 nous a permis de construire un processus exprimant la séquence pA `:=` `(`A `>>` C`)`. Nous allons ré-écrire ce processus de manière à intégrer une préemption par B de A uniquement (i.e. pas de préemption possible de C par B).

Il nous faut revenir à la définition première de l'opérateur LOTOS `disrupt` : cet opérateur signifie qu'au niveau de chacune des actions du processus préempté le système a le choix entre cette action et le processus qui préempte. Nous allons donc réécrire le `disrupt` par un choix.

Par exemple, supposons que A`:=(a; exit)`. Alors `(`A `[>` B`)` se réécrit en : `(` `(` `a;` `(exit []` B`)` `)` `[]` B `)`. Maintenant, si nous voulons exprimer qu'à la terminaison de A (et uniquement de A) le système poursuit en séquence par C, nous écrirons : `(` `(` `a;` `(exit >>` C`)` `[]` B `)` `)` `[]` B `)`.

Il y a malgré tout une nouvelle difficulté pour mettre en place cette solution : le processus à l'origine des relations de *Sequence* et de *Preemption* (le processus A dans notre exemple) peut parfois s'écrire avec plusieurs `exit`. Typiquement, il y a un `exit` pour chacune des branches des opérateurs de choix et des opérateurs de parallélisme (avec ou sans synchronisation). Dans le cas des branches d'un choix, nous pouvons écrire la séquence derrière chacun des `exit` puisqu'il n'y aura qu'une seule branche sélectionnée. Par contre, pour les branches d'une composition parallèle nous ne pouvons pas, sans quoi nous lancerions en parallèle autant de fois les processus pointés par les relations de *Sequence* qu'il y a de branches. Ce qui n'est pas correct.

Une manière élégante pour pallier ce problème est de réécrire également l'opérateur de séquence LOTOS. En effet, composer en séquence deux processus revient à synchroniser le démarrage du processus de droite avec la terminaison du processus de gauche; ces deux

processus étant en réalité composés en parallèle avec une synchronisation sur l'action de terminaison LOTOS qui est notée δ. Cette action de terminaison provient de `exit`, le processus de terminaison LOTOS. En effet : `exit := (`δ`; stop)`.

Nous allons donc introduire une nouvelle porte `delta` et remplacer les `exit` de notre processus à l'origine des relations de *Sequence* et de *Preemption* (le processus A dans notre exemple) par `(delta; stop)`; il convient également de réécrire toutes les compositions parallèles (avec ou sans synchronisation) à l'intérieur de ce processus pour ajouter une synchronisation sur `delta`. Ce nouveau processus sera composé en parallèle avec les processus pointés par les relations de *Sequence*, avec une synchronisation sur `delta`; ces derniers débutant leur activité par l'action `delta`. De cette manière, nous évitons la confusion entre la terminaison (par `(delta; stop)`) du processus à l'origine des relations de *Sequence* et de *Preemption*, et la terminaison (par `exit`) de l'éventuel processus qui pourrait le préempter, sans quoi le système risquerait de partir en séquence à sa suite : dans notre exemple, A se termine par `(delta; stop)` et B se termine par `exit`; ainsi, puisque la séquence a été redéfinie autour de l'action `delta` et non plus autour de l'action δ, alors C partira bien en séquence après A uniquement si celui-ci n'a pas été préempté par B.

iii) Démarche globale de l'algorithme

Algorithme 3.4 Fonction $String\ Make\_Pre(T, \mathcal{T}, \mathcal{R})$

let $\mathcal{T}' := \{T_j \ / \ T \in \mathcal{T} \ \wedge \ \langle T, Preemption, OCL, T_j \rangle \in \mathcal{R}\}$
if $\mathcal{T}' \neq \emptyset$ **then**
  let $S : String$
  let $n\_gate : Natural := \sum_{T_j \in \mathcal{T}'} NbGate(T_j)$
  if $n\_gate > 0$ **then**
    $S := "$`hide g_0,...,g_(`n_gate`) in`$"$
  end if
  let $cpt : Natural := 0$
  for all $T_j \in \mathcal{T}'$ **do**
    $S \mathrel{+}= NameOf(T_j) + "$`[g_(`$cpt$`),...,g_(`$cpt + NbGate(T_j)$`)]`$"$
    if $T_j \neq LastOf(\mathcal{T}')$ **then**
      $S \mathrel{+}= "$`[]`$"$
    end if
    $cpt \mathrel{+}= NbGate(T_j)$
  end for
  return $"$`process `$+ NameOf(T) + "$`_Preempt`$" + "$`: noexit :=`$" +$
    $S + "$`endproc`$"$
end if

Algorithme 3.5 Fonction *String Make_Seq*(T, $\mathcal{T}$, $\mathcal{R}$)

let S : *String*
let $\mathcal{T}' := \{T_j \ / \ T_j \in \mathcal{T} \ \wedge \ \langle T, Sequence, OCL, T_j \rangle \in \mathcal{R}\}$
if $\mathcal{T}' \neq \emptyset$ **then**
  let *st_psi*,*st_hide* : *String*
  TranslatePS($\mathcal{T}'$,$\mathcal{R}$,out *st_psi*,out *st_hide*)
  return "`process` " + *NameOf*(T) + "`_Sequence [delta]`" +
    " : `noexit` :=" + *st_hide* + "`delta; (`" + *st_psi* + "`) endproc`"
end if

1. Nous reprenons la démarche de l'algorithme exposé section 3.2.2 pour construire un processus RT-LOTOS qui compose l'ensemble des processus issus des *Tclasses* pointées par les relations de *Preemption* par un choix non déterministe. Ce processus porte l'identificateur *NameOf*(*Tclass*) + "`_Preempt`". Ces opérations sont réalisées par la fonction *Make_Pre* décrite par l'algorithme 3.4.
2. Nous reprenons la démarche de l'algorithme exposé section 3.2.3 pour construire un processus RT-LOTOS qui compose en parallèle l'ensemble des processus issus des *Tclasses* pointées par les relations de *Sequence*. Ce processus porte l'identificateur *NameOf*(*Tclass*) + "`_Sequence`". Ces opérations sont réalisées par la fonction *Make_Seq* décrite par l'algorithme 3.5.
3. Nous réécrivons le processus *LabelOf*(*Tclass*) issu de la *Tclass* à l'origine des relations de *Sequence* et de *Preemption* (que nous appellerons *processus de tête*) de manière à exprimer le choix non déterministe entre chacune des actions qu'il réalise et le processus *NameOf*(*Tclass*) + "`_Preempt`" définit à l'étape 1. Ceci traduit les *Composers* de *Preemption*. Cette opération de réécriture est réalisée par la procédure *Rewrite* décrite dans la sous-section iv) page 133.
4. Nous réécrivons également le processus *LabelOf*(*Tclass*), pour d'une part substituer tous les `exit` qui le composent par `(delta; stop)`, et d'autre part pour réécrire toutes ses compositions parallèle (avec ou sans synchronisation) afin de leur ajouter une synchronisation sur `delta`. De plus, nous ajoutons la porte `delta` à la liste des portes du processus *LabelOf*(*Tclass*). Ces opérations sont réalisées également par la procédure *Rewrite*.
5. Nous construisons maintenant le processus RT-LOTOS qui va porter l'identificateur *NameOF*(*Tclass*) et qui va réaliser la déclaration et l'intériorisation de la porte `delta` par le mot clef `hide`, suivi de la composition du processus *LabelOf*(*Tclass*) et du processus `( delta;` *NameOf*(*Tclass*) + "`_Sequence`"`)` par une synchronisation sur `delta`.

Les étapes de cette démarche sont mises en œuvre par l'algorithme 3.6.

Algorithme 3.6 Réduction des *Tclasses* à l'origine de relations de *Sequence* et de *Preemption*

for all $T_i \in \mathcal{T}$ **do**
  let $\mathcal{R}_{Pre} := \{R \ / \ R \in \mathcal{R} \ \wedge \ R = \langle T_i, Preemption, OCL, T_j \rangle, \forall T_j \in \mathcal{T}\}$
  let $\mathcal{R}_{Seq} := \{R \ / \ R \in \mathcal{R} \ \wedge \ R = \langle T_i, Sequence, OCL, T_j \rangle, \forall T_j \in \mathcal{T}\}$
  if $\mathcal{R}_{Seq} \neq \emptyset \ \wedge \ \mathcal{R}_{Pre} \neq \emptyset$ **then**
    let $S_{Pre} : String := Make\_Pre(T_i, \mathcal{T}, \mathcal{R}_{Pre})$
    let $S_{Seq} : String := Make\_Sec(T_i, \mathcal{T}, \mathcal{R}_{Seq})$
    $T_i.traduction\_AD := Rewrite(T_i.traduction\_AD, "\texttt{delta}", NameOf(T_i) + "\texttt{\_Preempt}")$
    let $st\_gates : String := ""$
    if $GatesOf(T_i) \neq \emptyset$ **then**
      $st\_gates := "[" + GatesOf(T_i) + "]"$
    end if
    $T_i.traduction\_CD := "\texttt{process }" + NameOf(T_i) + st\_gates +$
      $"\texttt{ : noexit :=}" + "\texttt{hide delta in}" +$
      $LabelOf(T_i) + st\_gates + "\texttt{|[delta]|}" +$
      $NameOf(T_i) + "\texttt{\_Sequence}" +$
      $"\texttt{where}" + S_{Pre} + S_{Seq} + "\texttt{endproc}"$
    $\mathcal{R} := \mathcal{R} \setminus (\mathcal{R}_{Pre} \cup \mathcal{R}_{Seq})$
  end if
end for

iv) Réécriture du processus de tête

L'algorithme présenté ici est un peu particulier dans la mesure où il se propose de réécrire un processus issu de la traduction des diagrammes d'activité pour exprimer des comportements décrits dans les diagrammes de classes. Cette singularité est inhérente au fait qu'il n'y a pas de correspondance une à une systématique entre les éléments TURTLE et la traduction de leurs comportements temporels en RT-LOTOS : TURTLE n'est pas une syntaxe graphique de RT-LOTOS.

La réécriture de code consiste pour l'essentiel en une analyse syntaxique et grammaticale du code d'origine qui s'accompagne d'opérations de recopie et de modification des éléments de codes lus.

La figure 3.19 donne les règles de grammaire de l'analyseur syntaxique qui réécrit le processus selon nos critères. Les actions sémantiques associées à ces règles correspondent implicitement à une recopie des éléments lus (elles ont étés omises sur la figure pour en faciliter la lecture), exceptés en certains points où elles sont indiquées par des accolades : c'est là que le code est modifié selon nos critères.

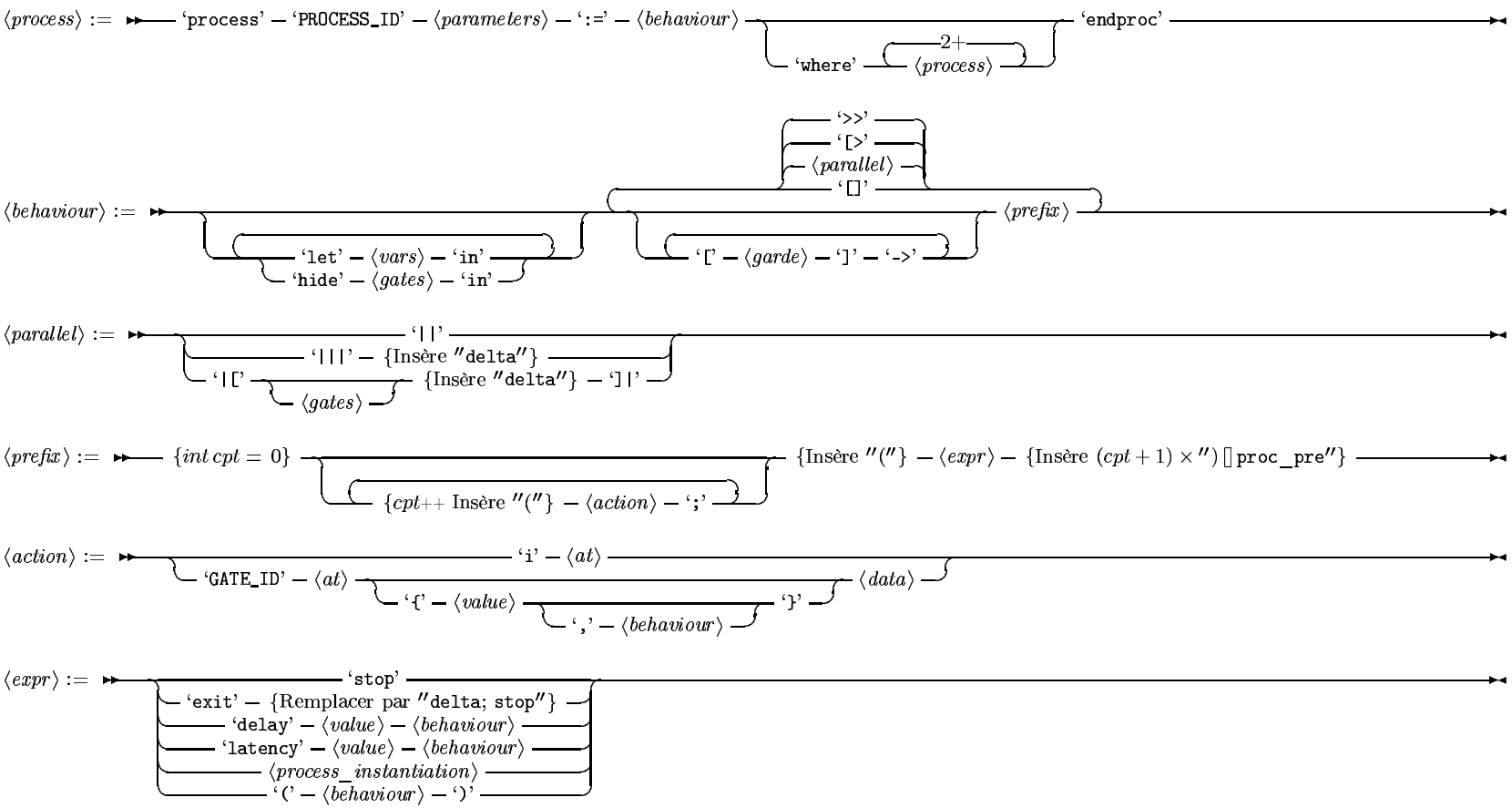


FIG. 3.19 – *Fonction String Rewrite*(*process* : *String*, *delta* : *String*, *proc_pre* : *String*)

v) Proposition d'optimisation

M^r Hubert GARAVEL, membre du jury de cette thèse, a proposé une construction alternative pour exprimer en RT-LOTOS un diagramme de classes dans lequel une *Tclass* A est à l'origine à la fois d'une relation de *Preemption* par B et d'une relation de *Sequence* vers C (voir la figure 3.18 page 130) : `(`A `[> (`B `>> stop) ) >>` C

L'idée est que l'expression "`>> stop`" permet d'intercepter les `exit` du processus B, et donc d'éviter que le processus C s'exécute en séquence avec la terminaison de B.

La fonction *Make_Pre* (voir algorithme 3.4) qui construit le processus $NameOf(Tclass) + "$`_Preempt`$"$, c'est à dire le processus qui englobe les processus pointés par les relations de *Preemption*, n'est pas modifiée. Par contre la fonction *Make_Seq* (voir algorithme 3.5) qui construit le processus $NameOf(Tclass) + "$`_Sequence`$"$ c'est à dire le processus qui englobe les processus pointés par les relations de *Sequence*, est modifiée car l'action `delta` introduite dans les paragraphes précédents n'est plus utilisée ici (voir algorithme 3.7). De même, l'algorithme 3.6 mis en place pour réduire et réécire les *Tclasses* à l'orignie de relations de *Sequence* et de *Preemption* est remplacé par l'algorithme 3.8 qui implémente la construction alternative proposée dans ce paragraphe.

Algorithme 3.7 Amélioration de la fonction *String Make_Seq*(T, $\mathcal{T}$, $\mathcal{R}$)

```
let S : String
let T' := {T_j / T_j ∈ T ∧ ⟨T,Sequence,OCL,T_j⟩ ∈ R}
if T' ≠ ∅ then
  let st_psi,st_hide : String
  TranslatePS(T',R,out st_psi,out st_hide)
  return "process " + NameOf(T) + "_Sequence" + " : noexit :=" +
                                                  st_psi + "endproc"
end if
```

3.2.6 Traduction des ensembles de *Tclass* en relation de *Parallel* et *Synchro*

Ces deux types de *Composers* (entrelacement total, parallélisme avec points de synchronisation), se traduisent en RT-LOTOS suivant la même démarche, par une composition en parallèle, avec ou non, synchronisation sur des portes.

La procédure *TranslatePS*($\mathcal{T}$, $\mathcal{R}$, *st_psi* : *out String*, *st_hide* : *out String*) est employée à plusieurs moments dans la traduction. Elle prend en argument un ensemble de *Tclasses* qui sont en *Parallel* ou *Synchro*, ainsi qu'un ensemble de *Composers* (incluant ceux cités précédemment). Deux chaînes de caractères sont produites : l'une est la traduction RT-LOTOS des *Composers* (composition en parallèle avec ou sans synchronisation des processus issus des *Tclasses*), et l'autre est la déclaration des portes intériorisées et renommées (notées `hide`).

Algorithme 3.8 Amélioration de la réduction des *Tclasses* à l'origine de relations de *Sequence* et de *Preemption*

for all $T_i \in \mathcal{T}$ **do**
  let $\mathcal{R}_{Pre} := \{R \ / \ R \in \mathcal{R} \ \wedge \ R = \langle T_i, Preemption, OCL, T_j \rangle, \forall T_j \in \mathcal{T}\}$
  let $\mathcal{R}_{Seq} := \{R \ / \ R \in \mathcal{R} \ \wedge \ R = \langle T_i, Sequence, OCL, T_j \rangle, \forall T_j \in \mathcal{T}\}$
  if $\mathcal{R}_{Seq} \neq \emptyset \ \wedge \ \mathcal{R}_{Pre} \neq \emptyset$ **then**
    let $S_{Pre} : String := Make\_Pre(T_i, \mathcal{T}, \mathcal{R}_{Pre})$
    let $S_{Seq} : String := Make\_Sec(T_i, \mathcal{T}, \mathcal{R}_{Seq})$
    let $st\_gates : String := ""$
    if $GatesOf(T_i) \neq \emptyset$ **then**
      $st\_gates := "[" + GatesOf(T_i) + "]"$
    end if
    $T_i.traduction\_CD :=$ "`process` " $+ NameOf(T_i) + st\_gates +$
      ": `noexit` :=" $+$
      "(" $+ LabelOf(T_i) + st\_gates +$ "([> (" $+$
      $NameOf(T_i) +$ "`_Preempt`" $+$ " >> `stop`)) " $+$
      " >> " $+ NameOf(T_i) +$ "`_Sequence`" $+$
      "`where`" $+ S_{Pre} + S_{Seq} +$ "`endproc`"
    $\mathcal{R} := \mathcal{R} \setminus (\mathcal{R}_{Pre} \cup \mathcal{R}_{Seq})$
  end if
end for

i) Définitions préliminaires

Nous allons tout d'abord introduire de nouvelles structures de données qui nous aideront à renommer les portes avant de réaliser la synchronisation entre les processus RT-LOTOS.

Soit *RenameGateSet* un ensemble de *RenameGate*.

Soit $RenameGate = \langle label, set(gate\_ref) \rangle$.

Soit *label* une chaîne (*String*).

Soit la fonction $FindNameGate(RenameGateSet, gate\_ref)$ qui retourne le *label* correspondant à $gate\_ref$.

Soit la fonction $Exists(RenameGateSet, gate\_ref)$ qui retourne *vrai* ou $faux$ suivant la présence ou non de $gate\_ref$ dans *RenameGateSet*.

Soit la procédure $Add(RenameGateSet, gate\_ref, gate\_ref')$ qui ajoute $gate\_ref'$ dans l'ensemble où $gate\_ref$ est présent ($gate\_ref$ et $gate\_ref'$ se réfèrent au même *label*).

Soit la procédure $Create(RenameGateSet, String, gate\_ref)$ qui crée un nouvel ensemble pour le label *String*, et contenant l'élément $gate\_ref$.

Algorithme 3.9 Construire l'ensemble de renommage des portes

```
let cpt : Natural := 0
let Rename : RenameGateSet := ∅
for all ⟨T,Synchro,OCL,T'⟩ ∈ R do
  if T,T' ∈ T then
    for all ⟨g_ref,g_ref'⟩ ∈ OCL do
      if Exists(Rename,g_ref) then
        Add(Rename,g_ref,g_ref')
      else
        Create(Rename,"g_(cpt++ )")
      end if
    end for
  end if
end for
for all T ∈ T do
  for all g_ref ∈ GateList(T) do
    if ¬Exists(Rename,g_ref) then
      Create(Rename,"g_(cpt++ )",g_ref)
    end if
  end for
end for
return Rename
```

Soit la fonction $InitRename(\mathcal{T}, \mathcal{R})$ qui prend en argument un ensemble de *Tclasses* qui participent à des relations de *Parallel* ou *Synchro*, ainsi qu'un ensemble de relations de *Composers* (incluant ceux cités précédemment), et qui construit un *RenameGateSet*. Cette fonction parcourt les formules OCL présentes dans $\mathcal{R}$. Elle y lit les égalités entre les noms des *Gates* et construit cette structure qui regroupe ensemble tous les noms désignant la même *Gate*. En d'autres termes, elle identifie, dans les formules OCL, les sous-ensembles connexes de *Gates* et définit pour chaque sous-ensemble un nom de porte LOTOS. Cette fonction est décrite par l'algorithme 3.9.

ii) La procédure *TranslatePS*

Le fonctionnement de la procédure $TranslatePS(\mathcal{T}, \mathcal{R}, st\_psi : out\ String, st\_hide : out\ String)$ évoquée précédemment est donné par l'algorithme 3.10.

Algorithme 3.10 Traduction des relations *Parallel* et *Synchro*

let $Rename : RenameGateSet := InitRename(\mathcal{T},\mathcal{R})$
let $\mathcal{T}' := \{T^{first}\}$ with $T^{first} \in \mathcal{T}$
$\mathcal{T} := \mathcal{T} \setminus \{T^{first}\}$
let $parenthesis\_cpt : Natural := 1$
let $st\_psi := "(" + NameOf(T^{first}) + "["$
for all $g\_ref \in GateList(T^{first})$ **do**
  $st\_psi \mathrel{+}= FindNameGate(Rename,g\_ref)$
end for
$st\_psi \mathrel{+}= "]"$
while $\mathcal{T} \neq \emptyset$ **do**
  let $T \in \mathcal{T}$
  $\mathcal{T} := \mathcal{T} \setminus \{T\}$
  let $\mathcal{R}' = \{R \ / \ \langle T,Synchro,OCL,T'\rangle \ \wedge \ T' \in \mathcal{T}'\}$
    $\uplus \{R \ / \ \langle T',Synchro,OCL,T\rangle \ \wedge \ T' \in \mathcal{T}'\}$
  if $\mathcal{R}' = \emptyset$ **then**
    $st\_psi \mathrel{+}= "|||"$
  else
    let $g\_set : set\ of\ String \ := \emptyset$
    while $\mathcal{R}' \neq \emptyset$ **do**
      if $\exists \langle T,type,OCL,T'\rangle \in \mathcal{R}'$ **then**
        let $R = \langle T,type,OCL,T'\rangle \in \mathcal{R}'$
        for all $\langle g\_ref,g\_ref'\rangle \in OCL$ **do**
          $g\_set \oplus= FindNameGate(Rename,g\_ref)$
        end for
      else
        let $R = \langle T',type,OCL,T\rangle \in \mathcal{R}'$
        for all $\langle g\_ref',g\_ref\rangle \in OCL$ **do**
          $g\_set \oplus= FindNameGate(Rename,g\_ref)$
        end for
      end if
    end while
    ../..

```
        ../..
        st_psi += "|["
        for all g : String ∈ g_set do
          st_psi += g + ","
        end for
        st_psi += "]|"
        parenthesis_cpt++
        st_psi += "(" + NameOf(T) + "["
        for all g_ref ∈ GateList(T) do
          st_psi += FindNameGate(Rename,g_ref)
        end for
      end if
    end while
    while parenthesis_cpt > 0 do
      st_psi += ")"
      parenthesis_cpt--
    end while
    st_hide := "hide "
    for ⟨label,set(gate_ref)⟩ ∈ Rename do
      st_hide += label
    end for
    st_hide += " in"
```

Avec :

- L'opérateur $\oplus=$ ajoute de manière unique un élément dans un ensemble.
- L'opérateur $\uplus$ réalise l'union de deux ensembles (i.e. sans dupliquer les éléments identiques présents dans les deux ensembles).

3.2.7 Traduction des *Tclasses* actives

Lorsque l'auteur bâtit le diagramme de classes, il indique également quelles sont les classes qui sont activées au démarrage. Les *Tclasses* sélectionnées sont soit explicitement composées entre elles par les *Composers Parallel* ou *Synchro*, ou bien n'ont pas de relation de composition et sont alors implicitement en *Parallel*.

La traduction de l'ensemble des *Tclasses* actives se fait simplement par appel à la procédure $TranslatePS$ si ce n'est que le paramètre $\mathcal{T}$ peut contenir plusieurs instances d'une même *Tclass* (car nous gérons également la notion d'ordre de multiplicité des classes UML).

Le résultat de cet appel à la procédure *TranslatePS* devient alors la clause `behaviour` de la spécification RT-LOTOS engendrée. Tous les autres processus RT-LOTOS générés par les différents étapes de la traduction sont placés dans la clause `where` de la spécification.

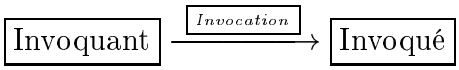
3.3 Extensions par domaine d'application

Par nature, `UML` propose des mécanismes pour spécialiser et étendre des profils existants en vue de répondre à un besoin spécifique d'un domaine d'application particulier. Nous utilisons cette faculté et proposons ici quelques extensions possibles bâties sur `TURTLE`.

Pour définir une extension au profil `TURTLE`, nous introduisons de nouveaux types d'association entre classes, de nouveaux symboles d'activité. Nous devons dès lors définir des règles de composition pour employer ces nouveaux éléments, typiquement au moyen d'une grammaire. De plus, nous précisons la sémantique de ces nouveaux éléments en explicitant leur traduction en `TURTLE` natif.

3.3.1 Invocation de classes

Assez rapidement nous rencontrons le besoin de modéliser un appel de méthode, c'est-à-dire, le fait qu'une classe insère dans son propre flux d'exécution des activités d'autre classes. Pour cela nous définissons un profil `TURTLE` étendu qui introduit le *Composer Invocation*.



Contrairement à l'opérateur *Synchro* qui caractérise une synchronisation entre deux flux d'exécution distincts, *Invocation* dénote une synchronisation qui, comme pour un appel de méthode dans le paradigme objet, est exécutée dans le flux d'exécution de l'appelant.

La figure 3.20 indique de quelle manière nous autorisons la composition de *Tclasses* par le *Composer Invocation* au sein du diagramme de classes. Par ailleurs nous donnons à *Invocation* une priorité égale à celle de *Synchro*.

Considérons deux *Tclass*, `T1` et `T2`, mises en relation par une association dirigée de `T1` à `T2` et attribuée par la classe associative *Invocation*. Nous disons alors que `T2` peut-être invoquée par `T1`. Tout comme dans la relation de synchronisation, une porte de chaque *Tclass* doit être impliquée dans chaque invocation. Soit `g1` (respectivement `g2`) une porte de `T1` (respectivement `T2`). Considérons que la formule OCL suivante est associée à la relation : `{T1.g1 = T2.g2}`. Alors, quand `T1` réalise un appel sur `g1`, elle doit attendre que `T2` réalise un appel sur `g2`. Quand `T2` réalise cet appel, les données sont échangées uniquement dans le sens de la navigation (c.à.d. de `T1` à `T2`), en quelque sorte pour passer des paramètres à `T2`. `T1` est bloquée jusqu'à ce que `T2` réalise de nouveau un appel sur `g2`. Dans ce deuxième cas, un échange de données peut avoir lieu dans le sens contraire à celui indiqué par la navigation, en quelque sorte pour renvoyer à `T1` la(les) valeur(s) de retour de l'appel. Cette invocation est similaire à l'appel de méthode du paradigme objet. Ainsi, le code de `T2` qui est invoqué par `T1` est exécuté dans le flux d'exécution de `T1`.

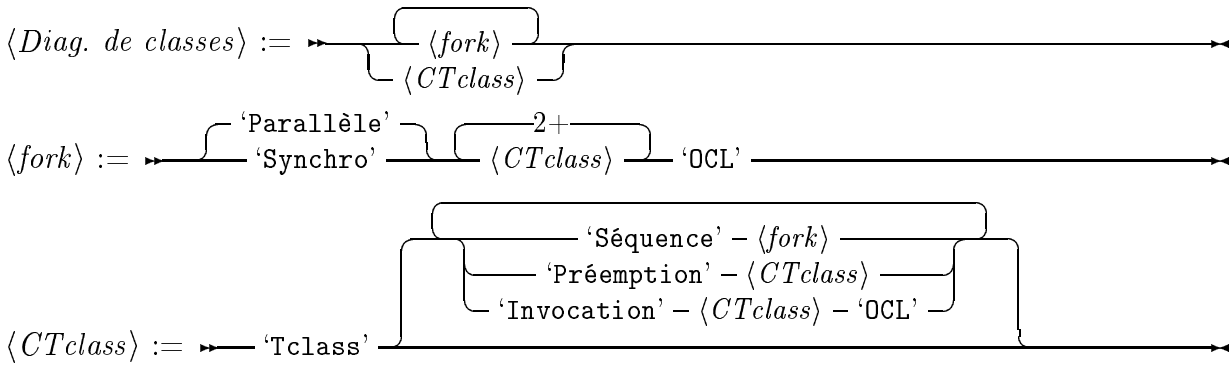


FIG. 3.20 – *Règles de composition de l'*Invocation

Un exemple d'utilisation de *Invocation* est proposé dans la figure 3.21.

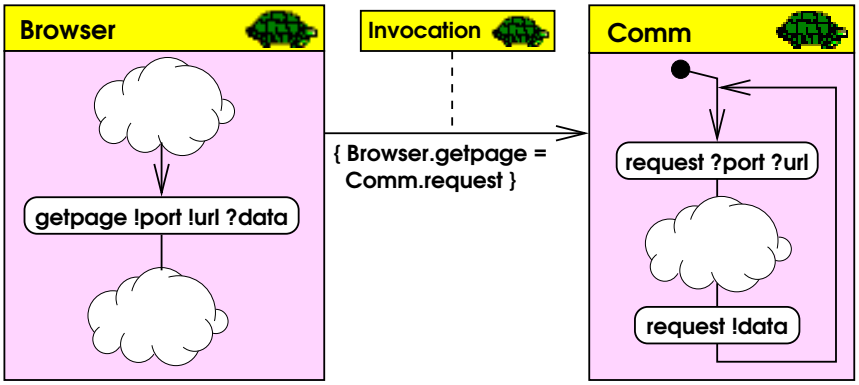


FIG. 3.21 – *Exemple d'utilisation de* Invocation

Nous traduisons le *Composer Invocation* en TURTLE natif directement par le *Composer Synchro* : les deux *Tclasses* invoquante et invoquée sont mises en relation par une *Synchro* sur la porte désignée par la formule OCL de l'*Invocation*.

Au niveau des diagrammes d'activité nous spécialisons le symbole d'appel sur une porte, d'une part côté invoquant, d'autre part coté invoqué. La porte spécialisée est la porte désignée par la formule OCL de l'*Invocation*.

- La *Tclass* invoquante réalise une invocation en faisant un appel sur la porte désignée par l'*Invocation*. Les valeurs qu'elle passe en paramètre sont préfixées, lors de l'appel sur cette porte, par le symbole "!". Les valeurs de retour qu'elle attend de l'*Invocation* sont préfixées, lors de l'appel sur cette même porte, par le symbole "?". Nous traduisons cela en TURTLE natif par deux appels de synchronisation successifs sur cette même porte : le premier avec les valeurs préfixées par "!", le deuxième avec les valeurs préfixées par "?".

– La *Tclass* invoquée attend une invocation en effectuant un appel sur la porte désignée par l'*Invocation*. Les paramètres acceptés sont préfixés par le symbole "?" sur cette porte (ils correspondent à ceux préfixés par "!" coté invoquant). La *Tclass* invoquée conclut son activité par un deuxième appel sur la porte désignée par l'*Invocation*. Les valeurs passées en retour sont préfixées par le symbole "!" sur cette porte (ils correspondent à ceux préfixés par "?" coté invoquant). Nous traduisons cela en TURTLE natif directement, sans transformation.

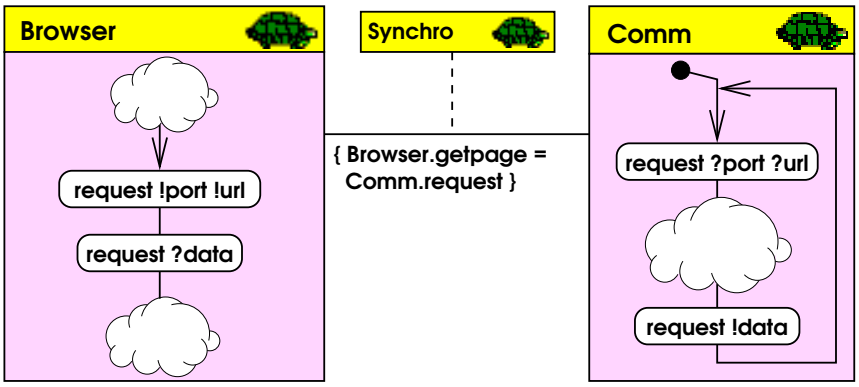


FIG. 3.22 – *Traduction de* Invocation *en* TURTLE *natif*

La figure 3.22 illustre la traduction en TURTLE natif l'exemple proposé par la figure 3.21.

3.3.2 Suspension/reprise d'activité

Lorsque nous modélisons des systèmes dans le domaine du temps-réel nous rencontrons souvent le besoin d'exprimer le fait qu'une tâche puisse être suspendue dans le temps, et reprise par la suite, par une autre classe (typiquement, le déroutement sur la réception d'un signal, ou un gestionnaire de tâches comme schématisé dans la figure 3.23).

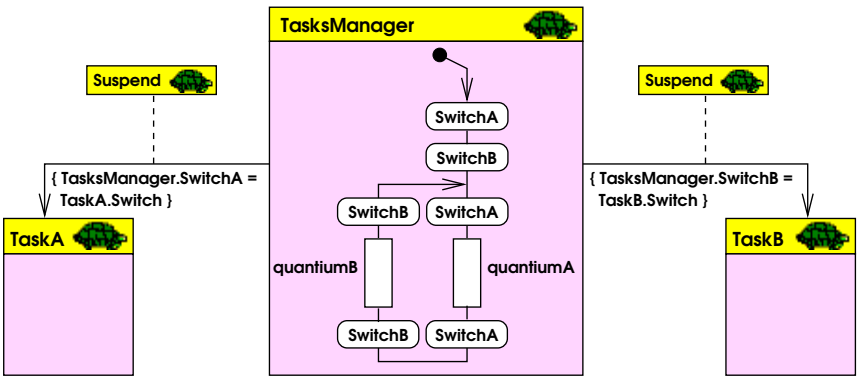
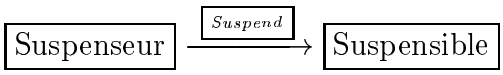


FIG. 3.23 – *Exemple d'utilisation de* Suspend

i) L'association *Suspend*

Le *Composer Preemption* dont nous disposons au niveau de `TURTLE` natif ne permet pas de reprise : la tâche préemptée est détruite et remplacée par la seconde. Pour pallier cette difficulté, nous spécialisons `TURTLE` et définissons un profil pour le domaine temps-réel système qui introduit un *Composer Suspend* qui permet de suspendre une tâche et de la reprendre. De plus, nous proposons de nouveaux opérateurs temporels suspensibles. Notons par ailleurs que nous avons choisi pour *Suspend* une association fléchée dans le sens opposé à ce que serait une association de *Preemption*.



La figure 3.24 indique de quelle manière nous autorisons la composition de *Tclasses* par le *Composer Suspend* au sein du diagramme de classes. Par ailleurs nous donnons à *Suspend* une priorité égale à celle de *Synchro*.

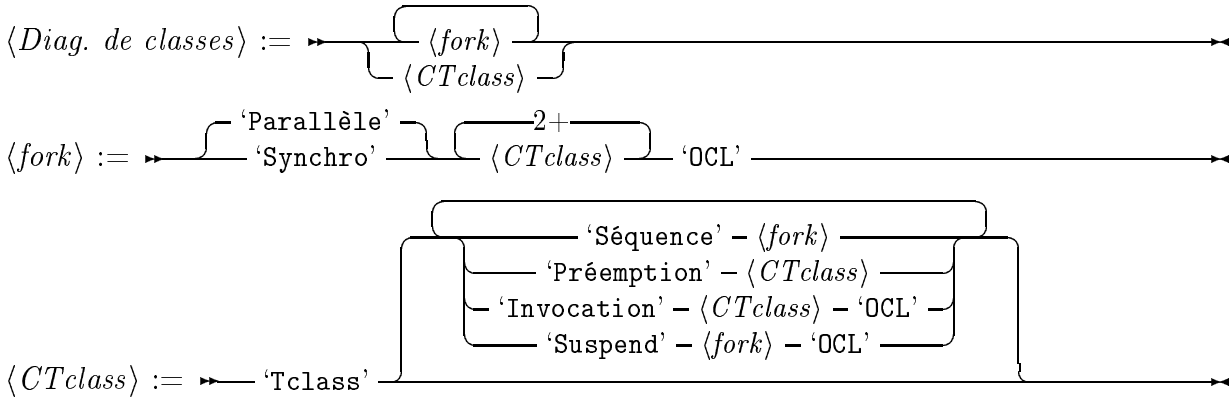


FIG. 3.24 – *Règles de composition de* Suspend

Considérons deux *Tclass*, `T1` et `T2`, mises en relation par une association, dirigée de `T1` à `T2`, et attribuée par la classe associative *Suspend*. Nous disons alors que `T2` peut-être suspendue et réactivée par `T1`. Une porte `s` (*signal*) de `T1` doit être impliquée dans chaque *Suspend*. Elle est précisée dans la formule OCL associée à la relation. Lorsque `T1` réalise un appel sur `s`, elle suspend l'activité de `T2`. Par la suite, elle peut réaliser un nouvel appel sur `s` pour réactiver `T2`. `T1` peut suspendre et réactiver `T2` autant de fois que nécessaire.

ii) Des opérateurs temporels suspensibles

Nous conservons la sémantique des trois opérateurs temporels `TURTLE` natif qui expriment des contraintes sur la progression du temps *universel*, c'est-à-dire un temps dont la progression est continue (qui ne peut être suspendu) et identique pour tous les composants (par opposition à des composants qui mesurent le temps au moyens d'horloges potentiellement affectées par des dérives ou de la gigue).

Par contre nous avons besoin d'introduire de nouveaux opérateurs temporels pour exprimer des contraintes sur des durées que l'on peut suspendre et reprendre (typiquement pour modéliser un calcul ou une opération qui consomme un temps donné, mais qui peut être interrompu et être repris ultérieurement). Ces opérateurs sont : un délai suspensible, une latence suspensible, et une offre limitée dans le temps suspensible. Nous reprenons les symboles originaux sur lesquels nous ajoutons un petit sablier pour indiquer que le temps peut être suspendu, comme représenté dans la figure 3.25

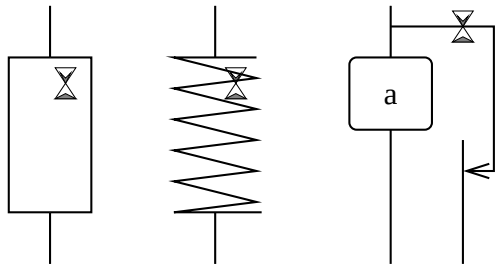


FIG. 3.25 – *Opérateurs temporels suspensibles*

Ces nouveaux symboles d'activité obéissent aux mêmes règles de composition que leurs homologues non-suspensibles. Ces règles de composition ont été définies figure 3.8 page 115.

3.3.3 Traduction du profil temps-réel système en TURTLE

En 3.3.2 nous présentons une extension du profil TURTLE dédiée aux applications du domaine temps-réel système.

La sémantique de ce nouveau profil est donnée par sa traduction en TURTLE natif. Nous présentons ici cette traduction qui consiste en une réécriture des *Tclasses* utilisant le profil étendu en de nouvelles *Tclasses* respectant le profil TURTLE natif.

Pour simplifier les explications fournies ici, nous faisons l'hypothèse que la porte précisée par la formule OCL associée à la relation de *Suspend* est `s` (pour *signal*).

i) Traduction de l'association *Suspend*

L'association *Suspend* se traduit par une association de *Synchro* entre la *Tclass* qui ordonne la suspension/reprise et les *Tclasses* suspensibles. La porte de synchronisation est la porte `s` impliquée dans l'association *Suspend*.

De plus, les *Tclasses* suspensibles sont outillées de la manière suivante. Les *Tclasses* suspensibles sont mises en relation de *Synchro* sur la porte `s` avec une *Tclass* qui a pour objet de compter le nombre d'appels réalisés sur `s` et d'indiquer, à la demande, si ce nombre est pair ou impair. Cette construction est représentée dans la figure 3.26.

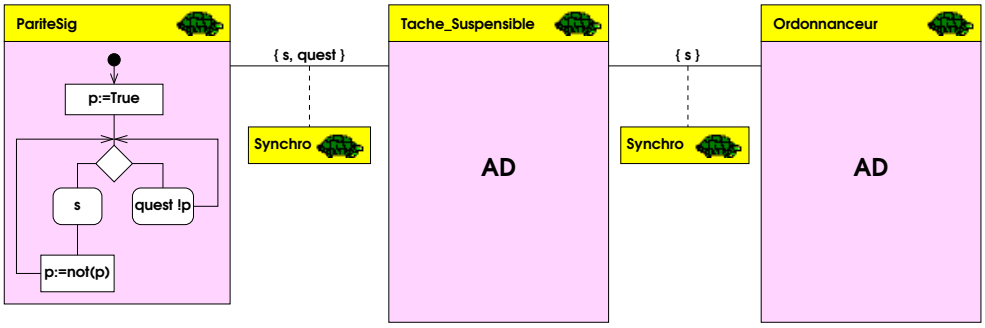


FIG. 3.26 – *Outillage des* Tclasses *suspensibles*

ii) Traduction des opérateurs suspensibles

D'une manière générale, chaque opérateur où le temps peut être suspendu, doit être précédé d'un test sur la parité des appels sur la porte d'interruption, et ce, pour attendre éventuellement d'avoir à nouveau la main afin de débuter l'opérateur suspensible, comme indiqué dans la figure 3.27.

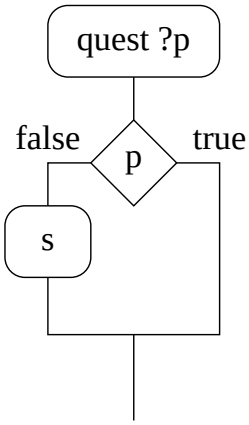


FIG. 3.27 – *Outillage des opérateurs suspensibles : tester la parité des interruptions*

La traduction spécifique de chacun des opérateurs suspensible est détaillé par la suite.

iii) Traduction d'un appel suspensible sur une *Gate*

Au sein d'une *Tclass* suspensible, tous les appels sur des *Gate* sont a priori suspensibles; en effet, ils modélisent les actions (visibles) réalisées par les *Tclass*.

Lorsqu'une activité effectue un appel sur une *Gate*, le temps peut s'écouler avant que la synchronisation sur cette *Gate* se réalise. Mais lorsqu'elle se réalise, elle se réalise de manière atomique, en un temps nul. C'est donc le temps d'attente de synchronisation qui est suspensible lors d'un appel sur une porte. Nous traduisons ce comportement par un choix entre l'offre sur la *Gate*, et l'offre sur `s`. Si une interruption (action `s`) survient

avant la synchronisation sur la *Gate*, alors nous interrompons l'offre sur cette *Gate*, et nous la reprendrons au prochain signal par l'action `s`. La transformation qui produit ce comportement est décrite figure 3.28.

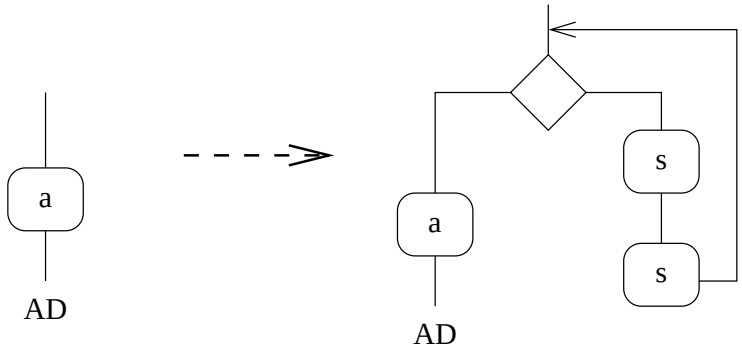


FIG. 3.28 – *Appel suspensible sur une* Gate

iv) Où placer le test de parité par rapport à l'opérateur de choix ?

La question se pose lorsque nous trouvons un opérateur non suspensible en tête de l'une des branches d'un choix, alors que d'autres branches débutent par un opérateur suspensible (celles-ci nécessite un test sur la parité des interruptions).

Effectivement, le fait de placer une action `quest?p` en tête d'une branche d'un choix va forcer le choix dans cette direction (l'action est immédiatement tirable).

Cas simple où nous n'avons que des synchronisation suspensibles sur chacune des branches du choix. La figure 3.29 décrit une transformation possible pour traduire ce cas.

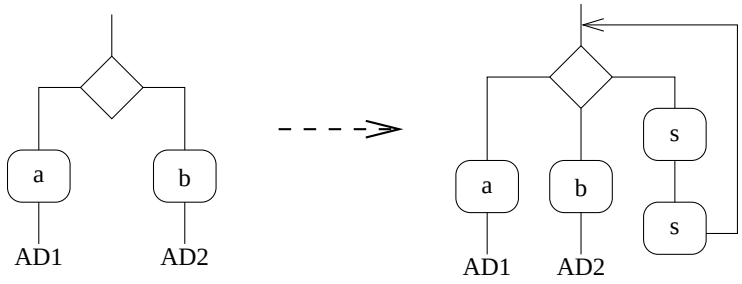


FIG. 3.29 – *Transformation simple du choix dans les* Tclasses *suspensibles*

Transformation dans le cas général Le principe est le suivant : nous traduisons le choix par une composition parallèle avec l'une des branches qui tue l'autre lorsque la synchronisation a été réalisée, et réciproquement (utilisation d'actions urgentes `k`n, pour *kill*).

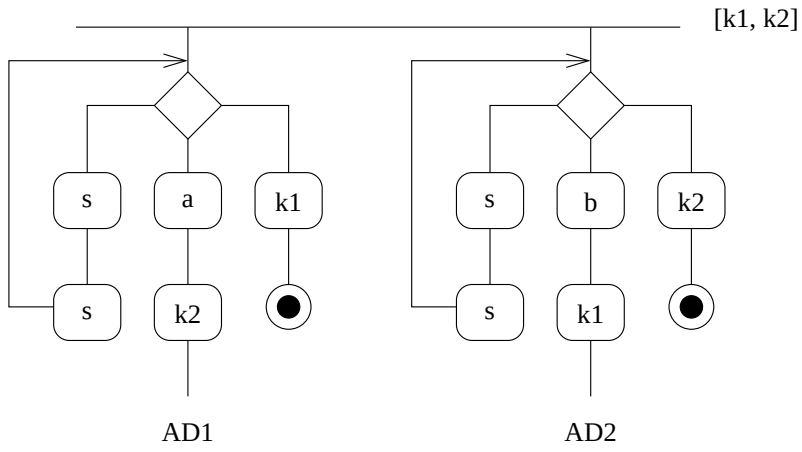


FIG. 3.30 – *Transformation générale du choix dans les* Tclasses *suspensibles*

Les transformations proposées plus bas présenteront une variante particulière dans le cas où elles sont placées en tête d'un choix.

v) Traduction d'un délai suspensible

Nous traitons ici de l'opérateur général `delay(dmin,dmax)`. La transformation s'appuie essentiellement sur l'opérateur de mémoire temporelle RT-LOTOS *at* (noté `@`[5]). L'idée consiste à proposer un choix entre l'activité débutant par `delay(dmin,dmax)` et l'action `s` précédée d'un `delay(dmax)`. De plus, nous mesurons le temps qui s'écoule entre le début du choix (sensibilisation de l'action `s`) et l'occurrence de l'action `s` (si elle se produit), de manière à ce que, lorsque l'activité reprend, nous puissions proposer un `delay(dmin,dmax)` diminué du temps déjà écoulé.

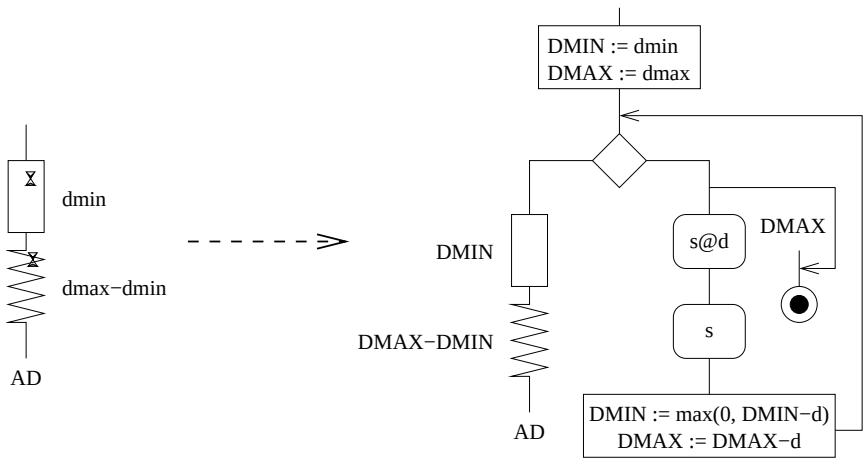


FIG. 3.31 – *Transformation simple d'un délai suspensible*

5. Cet opérateur a été discuté en sous-section 3.1.5.

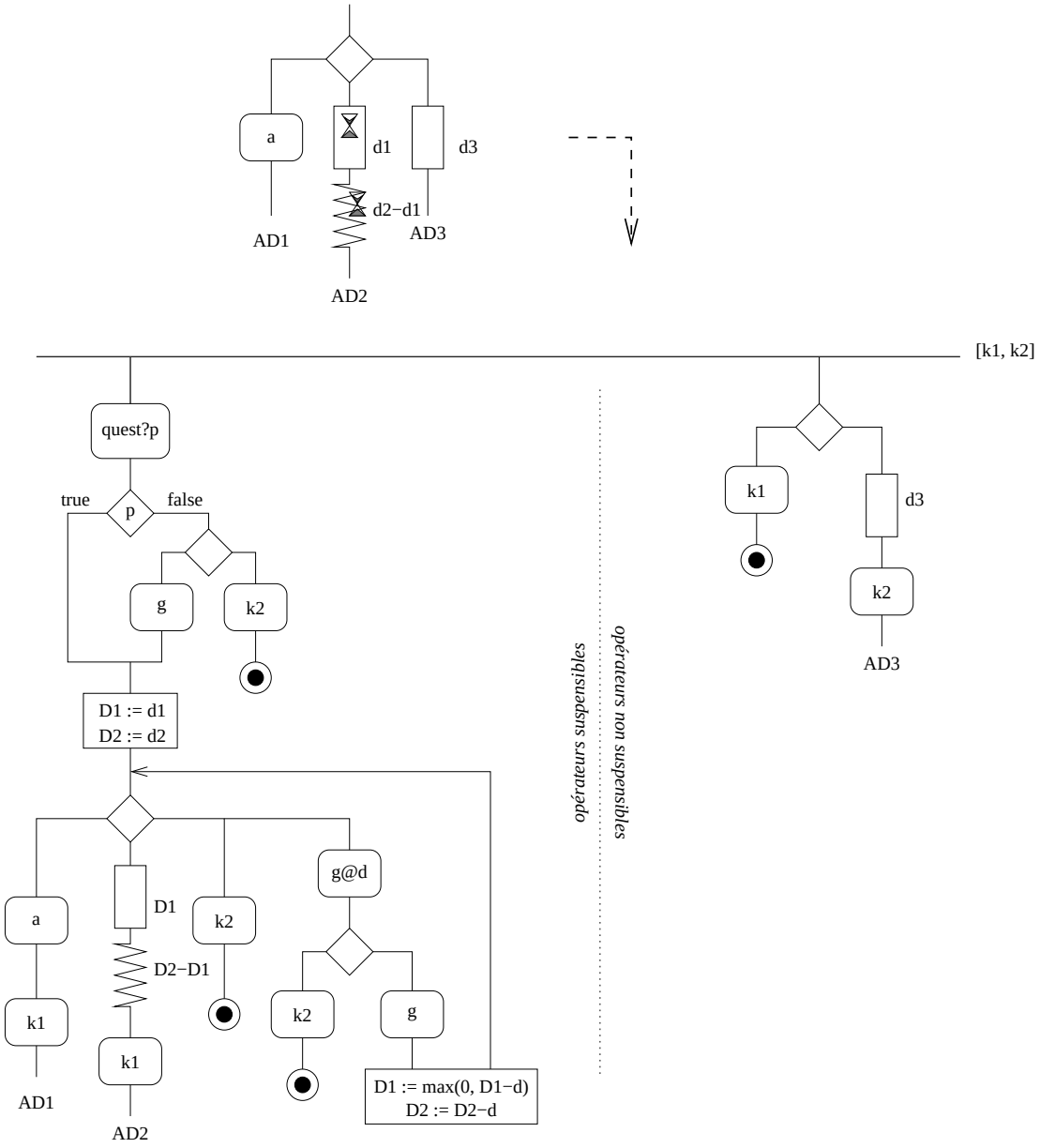


FIG. 3.32 – *Transformation générale d'un délai suspensibles*

Comme précédemment nous proposons une transformation dans le cas simple (figure 3.31), ainsi qu'une transformation dans le cas d'un choix (figure 3.32).

N'oublions pas que la traduction de cet opérateur suspensible, doit être précédée du test sur la parité de l'occurrence de l'action `s`. Dans le cas simple, nous insérons au préalable le motif indiqué figure 3.27. Dans le cas d'un choix, la transformation proposée intègre ce test.

vi) Traduction d'une offre limitée suspensible

L'idée est similaire à celle proposée dans le cas d'un délai suspensible : nous mesurons le temps écoulé entre le début de l'offre et l'occurrence de l'action d'interruption `s`, pour diminuer d'autant l'offre temporelle lorsque l'activité reprend. Cette construction est exposée figure 3.33.

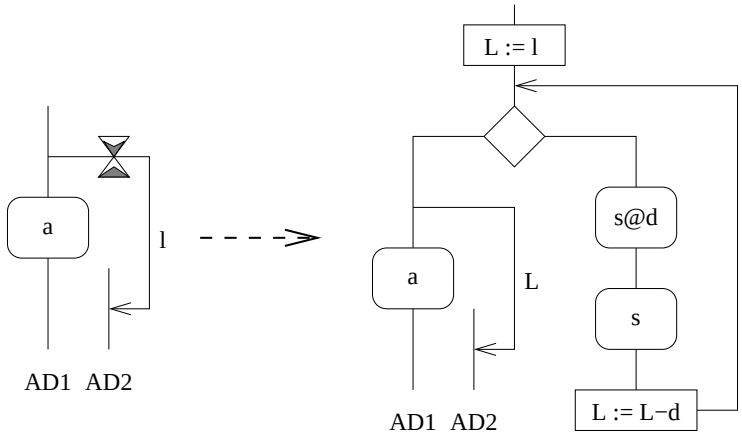


FIG. 3.33 – *Traduction d'une offre limitée suspensible*

N'oublions pas que la traduction de cet opérateur suspensible, doit être précédée du test sur la parité de l'occurrence de l'action `s`. Dans le cas simple nous insérons au préalable le motif indiqué figure 3.27. Dans le cas d'un choix, la transformation proposée intègre ce test.

Cas d'un choix

Dans le cas où une offre limitée suspensible est placée en tête d'une activité correspondant à une branche d'un choix, nous sommes confrontés aux difficultés habituelles. L'idée consiste à réécrire l'offre limitée dans le temps comme étant le choix entre d'une part un appel sur la *Gate*, et d'autre part une activité urgente (si elle ne l'est pas a priori nous insérons l'action `i`) mais qui est retardé par un délai égal à la limite de l'offre. Cette nouvelle construction peut alors s'insérer sans difficulté parmi les branches du choix qui nous posait problème initialement car nous savons traduire un choix comprenant un appel suspensible ainsi qu'un choix comprenant un délai suspensible. C'est-à-dire que nous appliquons la transformation proposée figure 3.30 avec une latence nulle.

vii) Transformation d'une offre limitée non suspensible

Dans le cas d'une offre limitée non suspensible, la limite de l'offre est fixée par rapport au temps universel, non suspensible. Par contre, l'appel sur la *Gate* reste suspensible. Cela nécessite certaines transformations.

L'offre limitée est d'abord transformée en un choix. Ensuite, nous appliquons les transformations portant sur le choix que nous avons déjà évoquées. Cette construction est exposée figure 3.34.

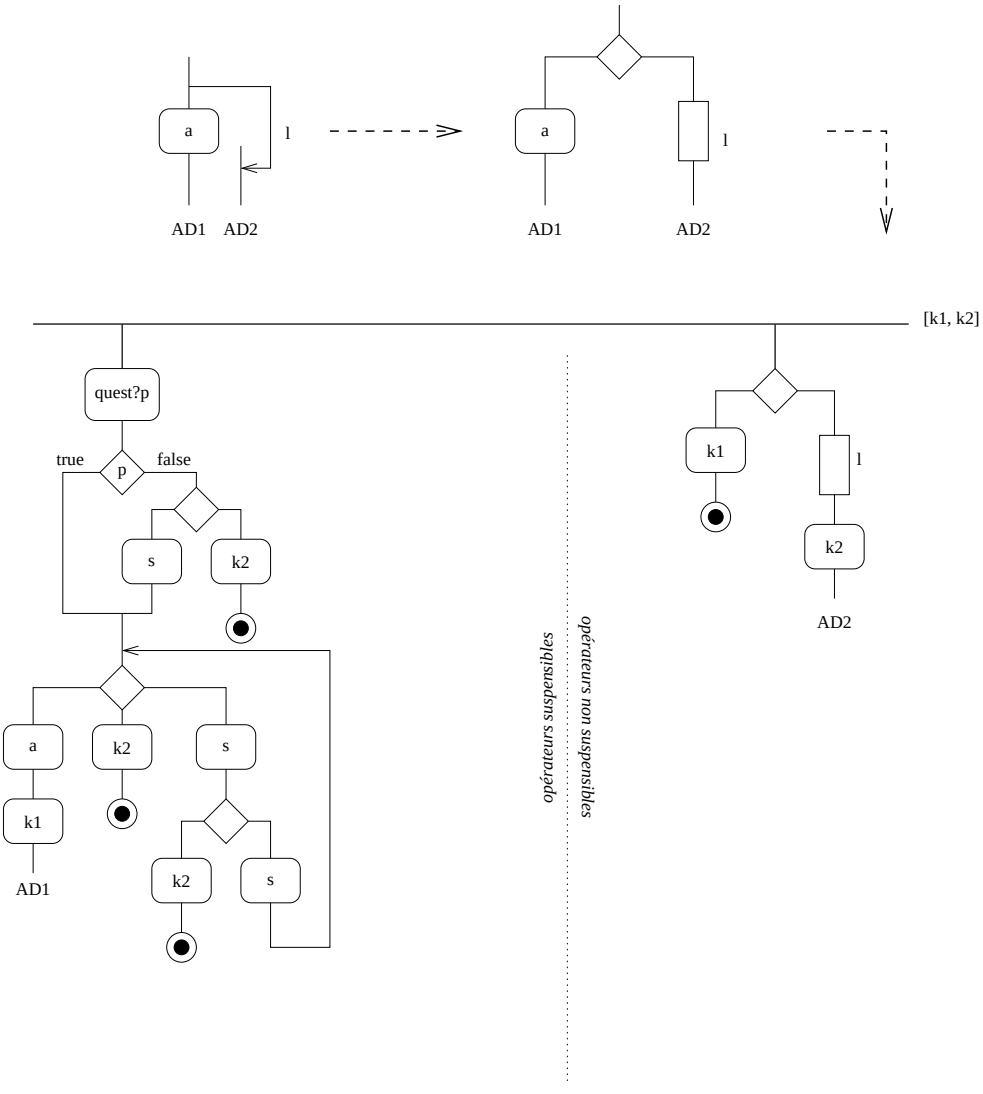


FIG. 3.34 – *Transformation d'une offre limitée non suspensible*

viii) Outillage des compositions parallèles

Chacune des branches d'une composition parallèle (avec ou sans synchronisation) doit se synchroniser sur l'action `s` (qui signale que l'activité est suspendue ou réactivée). Il y a un risque que des branches d'une composition parallèle se bloquent lors d'une tentative de synchronisation sur l'action `s` car d'autres branches ne sont pas prêtes à faire `s` (n'offrent pas cette action). L'idée consiste donc à adjoindre à chaque branche de la composition parallèle un petit processus pour «renommer» l'action `s` en une action $\texttt{s}_n$ locale à la branche.

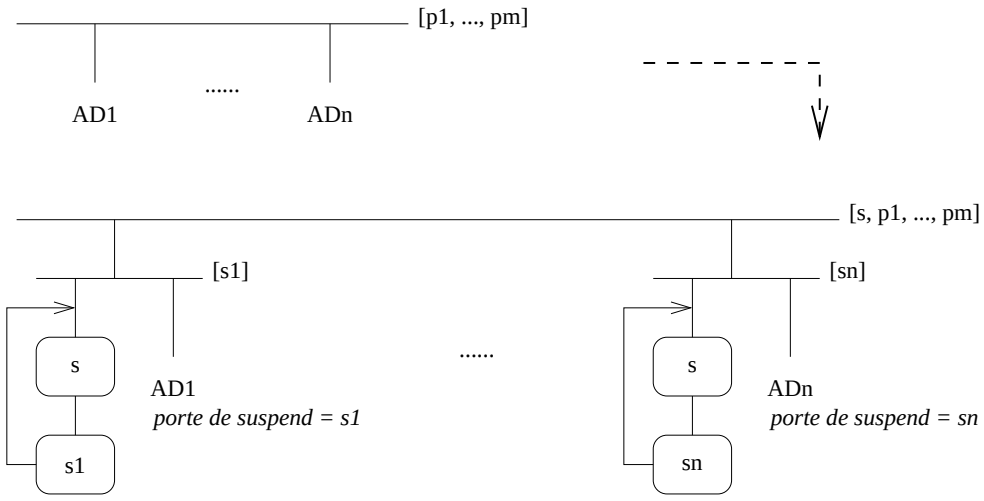


FIG. 3.35 – *Outillage des compositions parallèles suspensibles*

Remarque : il n'est pas nécessaire de prévoir un mécanisme pour détruire ce processus rajouté car lorsque l'$\texttt{AD}_n$, avec lequel il est synchronisé sur la porte $\texttt{s}_n$, se termine (par un `exit` ou un `stop`), il devient inactif (i.e. ne réalise aucune action). Mais pour des raisons d'occupation mémoire de `rtl` nous pouvons envisager de forcer sa destruction (i.e. le remplacer par le processus `exit`).

ix) Exemple de transformation d'activité suspensible

L'exemple présenté dans la figure 3.36 reprend tous les opérateurs suspensibles et les place dans le contexte d'un choix, ce qui correspond au cas le plus délicat.

3.4 Conclusion

Dans ce chapitre nous nous sommes penchés sur les liens possibles entre méthodes formelles et semi-formelles, avec pour objectif de faciliter l'écriture de spécifications formelles, et même, de cacher complètement à l'utilisateur, la mise en œuvre des techniques de validation formelle.

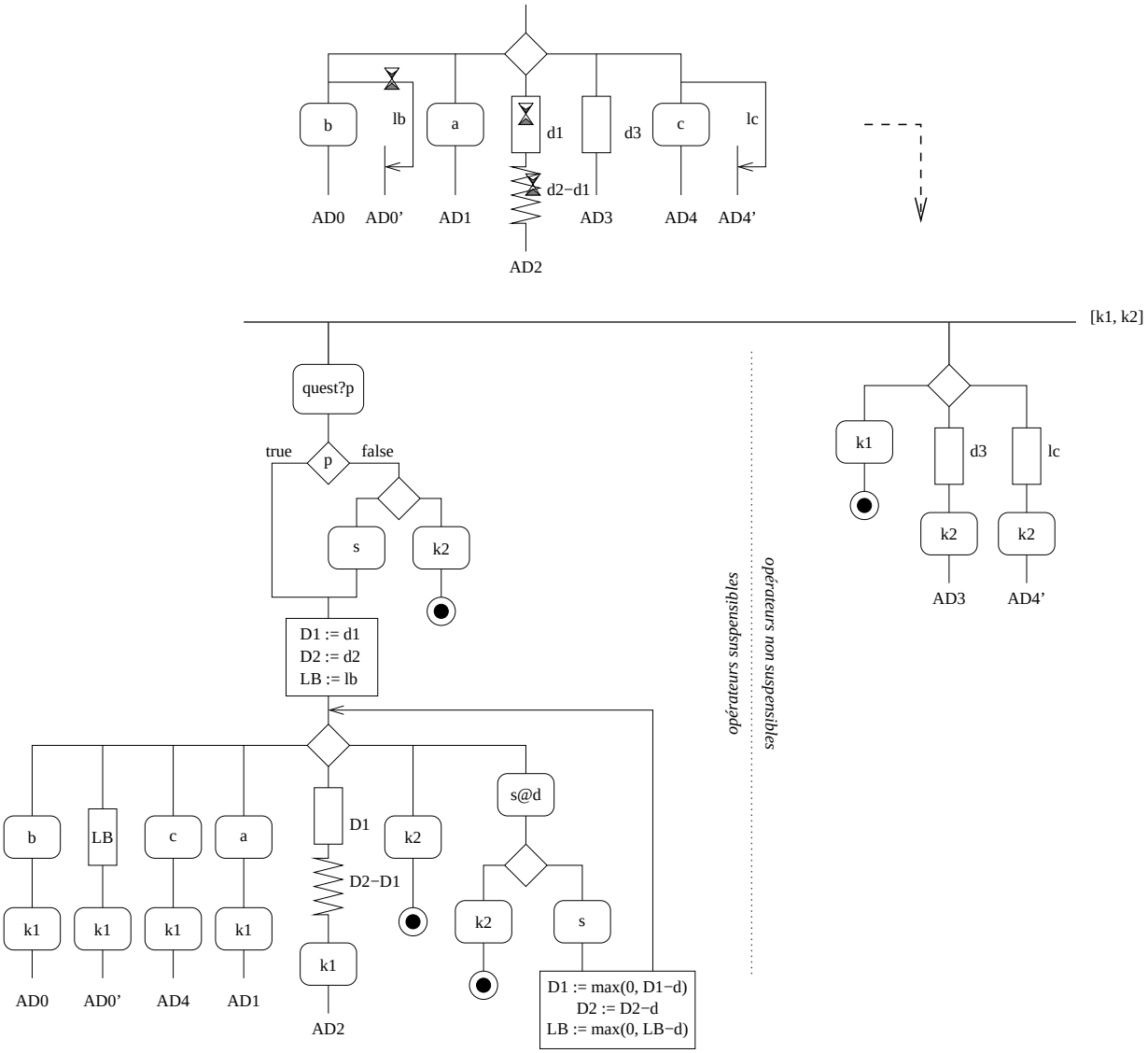


FIG. 3.36 – *Exemple de transformation d'activité suspensible*

Nous avons présenté TURTLE, un profil UML pour la conception et la validation de systèmes temps-réel. Les diagrammes de classes sont étendus avec un stéréotype (*Tclass*) et deux types abstraits (*Gate* et *Composer*). Une sémantique précise est donnée aux associations entre classes (voir les classes *Parallel*, *Synchro*, *Invocation*, *Sequence* et *Preemption*). Le comportement d'une *Tclass* est décrit par un diagramme d'activité étendu par trois opérateurs temporels : un délai déterministe, un délai non déterministe et une offre limitée dans le temps.

Nous avons également présenté un algorithme de traduction d'une spécification TURTLE en une spécification RT-LOTOS, technique de description formelle supportée par un outil de validation. Les diagrammes TURTLE peuvent ainsi être simulés et vérifiés sur la base d'une analyse d'accessibilité RT-LOTOS, et ce, en faisant en sorte que le formalisme RT-LOTOS reste complètement caché au concepteur du système.

Nous avons exposé la mise en œuvre de la méthodologie sur un exemple académique. Le profil TURTLE a été évalué sur un logiciel temps-réel embarqué et plus spécifiquement pour la validation de la reconfiguration dynamique de logiciels embarqués à bord de satellites [Apv02].

Des extensions de la méthodologie ont été présentées, et plus particulièrement deux extensions du profil TURTLE, l'invocation et le suspend/resume.

Conclusion

Les travaux exposés dans ce mémoire ont été motivés par la volonté d'étendre le cadre d'utilisation des méthodes de vérification formelle des systèmes contraints par le temps, et plus spécifiquement du langage de description formelle RT-LOTOS.

Contribution de l'étude

Après avoir passé en revue différentes techniques de description formelle de systèmes temps-réel, nous nous sommes concentrés sur le langage RT-LOTOS.

Intuitivement, le non-déterminisme temporel couvre deux notions distinctes : l'*incertitude* et l'*opportunité* [Boy01]. Dans le premier cas, une entité peut attendre une action dans un intervalle temporel sans certitude sur la date précise de son occurrence, et ceci s'exprime en RT-LOTOS par l'opérateur d'offre limité dans le temps. Par ailleurs, une entité peut se réserver l'opportunité de choisir une date au sein d'un intervalle temporel pour réaliser une action, et ceci s'exprime en RT-LOTOS par l'opérateur de latence. Peu de langages de description de systèmes temporisés permettent de spécifier cette distinction et n'offrent qu'une fonctionnalité générique pour spécifier un intervalle temporel. Le langage RT-LOTOS est un langage qui autorise cette distinction, et cela le rend particulièrement attractif et justifie son choix pour les travaux exposés ici.

Dans un premier temps, nous nous sommes intéressés aux techniques de validation par simulation et par vérification formelle. Dans ce cadre nous avons développé un procédé de simulation performant qui réalise la simulation, non plus sur les spécifications RT-LOTOS elles-mêmes, mais sur les automates temporisés dérivés des spécifications RT-LOTOS. Dans un deuxième temps, nous avons cherché à faire le lien avec des techniques de vérification de modèle (model-checking). Pour ce faire, nous avons formalisé et implémenté la conversion des automates temporisés dérivés des spécifications RT-LOTOS en automates temporisés acceptés par des outils existants de model-checking, tels que `Kronos`.

La méthode formelle RT-LOTOS est dotée d'une technique de validation basée sur la construction d'un graphe minimal d'accessibilité. Cette technique, éventuellement combinée avec la technique des observateurs, permet de répondre à des questions telles que : «Est-ce que tel comportement particulier attendu peut effectivement se produire ?» ou encore «Est-il vrai que telle situation d'erreur ne peut en aucun cas se produire ?». Par contre, cette technique peut difficilement répondre à des questions du type «Compte tenu des contraintes du système, quelles sont les dates possibles pour tel événement ?».

Pour répondre à ces dernières questions d'une manière générale, pour toutes les actions du système modélisé, nous avons proposé et implémenté une technique permettant de raffiner le graphe minimal d'accessibilité et d'en extraire les plages temporelles de toutes les actions. Ceci prend la forme d'une nouvelle variante d'automate temporel, appelée *Time Labelled Scheduling Automata* (`TLSA`), que nous avons formalisé. Le `TLSA` est un modèle opérationnel, prévu pour exprimer globalement les contraintes temporelles qu'un système doit satisfaire. Il décrit de manière simple comment ordonnancer les actions au cours du temps.

De plus, le `TLSA` étant synthétisé à partir du graphe minimal d'accessibilité, nous sommes en mesure de filtrer les comportements non souhaitables avant la synthèse. Nous avons élaboré et outillé ces techniques.

En amont du problème de la vérification formelle, se pose celui de la conception des systèmes informatiques. Pour répondre à ce problème précis, un grand nombre de méthodes pragmatiques et d'outils ont vu le jour au cours des dernières décennies. Parmi les plus populaires apparaît la méthode `UML`. Pourtant, même si ces méthodes sont très prisées par le monde industriel, elles souffrent encore du manque de support formel. Face à ce constat, nous avons cherché à faire le lien entre méthodes informelles et méthodes formelles. Pour cela, nous avons d'abord doté `UML` d'extensions permettant d'exprimer des contraintes temporelles, et défini un profil `UML` appelé `TURTLE`. Puis, dans un deuxième temps, nous avons doté `TURTLE` d'une sémantique formelle définie par traduction dans le langage RT-LOTOS.

Cela nous permet, d'une part, d'inscrire la conception de systèmes temps-réel, au sein des méthodes de conception industrielle, et d'autre part, cela nous permet de tirer parti de l'ensemble des techniques de validation développées pour RT-LOTOS, et notamment, de proposer une méthodologie de vérification formelle pour des systèmes temps-réel spécifiés avec `UML` et `TURTLE`.

Un autre avantage notable est que nous pouvons employer les méthodes classiques d'extensions de `UML` pour étendre `TURTLE` et définir de nouveaux profils dédiés à des domaines d'application particuliers. Le profil `TURTLE` est un profil ouvert. Nous avons notamment développé deux extensions du profil `TURTLE` : l'invocation de classes, et un mécanisme de suspension/reprise de tâches.

Perspectives

Le travaux entrepris au cours de cette thèse autorisent de nombreuses suites possibles que nous envisageons ci-après.

De nouvelles techniques d'analyse pour RT-LOTOS

La technique d'analyse d'accessibilité employée sur le langage RT-LOTOS consiste à engendrer un automate temporisé `DTA` à partir des actions sémantiques d'une spécification RT-LOTOS, puis à construire un graphe de régions sur cet automate. Les travaux présentés dans ce mémoire s'appuient sur cette technique, soit au niveau de l'automate temporisé pour mener des simulations rapides et effectuer des vérifications pas *model-checking*, soit au niveau du graphe de régions pour définir la cohérence d'un système et synthétiser un automate d'ordonnancement. Cependant, nous avons pu voir que toutes les constructions RT-LOTOS ne sont pas exprimables en un `DTA`. De plus, le `DTA`, comme le graphe de régions, peut être sujet au risque d'explosion combinatoire inhérent à la sémantique de parallélisme par entrelacement. Il est vrai qu'un utilisateur averti peut contrôler dans une certaine mesure ce risque d'explosion combinatoire, en adoptant un style de spécification RT-LOTOS judicieux et en employant des actions composites par exemple. Malgré cela, il semble souhaitable d'étudier de nouvelles techniques d'analyse pour le langage RT-LOTOS. Il est peu probable que nous trouvions une technique résolvant complètement le problème de l'explosion combinatoire. Il est cependant possible d'éviter certaines étapes coûteuses de la technique classique d'analyse de spécification RT-LOTOS, principalement celle qui consiste à passer par un automate temporisé `DTA`. Ainsi, un travail futur consistera à définir l'expression directe de systèmes RT-LOTOS en terme de *système de transitions temporisées*, c'est-à-dire en un système dans lequel chaque nœud est un état du système RT-LOTOS associé à une date, et dans lequel chaque transition associe une action à un intervalle de tir (éventuellement ouvert, non borné). Il sera dès lors possible d'exploiter les techniques existantes de dépliage de l'espace d'états, soit pour l'analyse d'accessibilité et la vérification de formules LTL [BM83, BD91, Ber01], soit pour l'analyse de propriétés de vivacité et de formules CTL ou HML [YR98, BV02], l'objectif final étant de doter le langage RT-LOTOS d'un outil de vérification similaire à TINA [Tin].

Le modèle `TLSA` pour le test de conformité

Le test de conformité se définit [IT97] comme l'évaluation, au moyen de tests, de la conformité d'une implémentation à sa spécification. Les tests constituent un moyen d'extraire des connaissances sur un système donné en l'expérimentant. Ce concept inclut l'environnement dans lequel les tests sont exécutés, et la modélisation formelle de cet environnement.

Le modèle `TLSA` a été employé avec succès pour réaliser la présentation de documents multimédia interactifs. La méthodologie développée avec le modèle `TLSA` semble promet-

teuse, notamment dans le cadre du test de conformité : des automates TLSA pourraient êtres employés pour confronter des implémentations à leur modélisation.

L'idée consiste à utiliser le TLSA comme moteur principal d'un *testeur*. À supposer que la spécification soit donnée dans le langage RT-LOTOS, en plus des moyens classiques de validation, nous pouvons engendrer un TLSA, c'est-à-dire un automate temporisé qui implémente formellement cette spécification. Il serait alors possible de confronter cet objet formel à l'objet physique qui est l'implémentation matérielle et logicielle du système : le TLSA fournirait à l'implémentation sous test les entrées qu'elle est supposée accepter, et testerait la conformité des sorties qu'elle est supposée engendrer. Tout écart du comportement de l'implémentation sous test par rapport à sa spécification formelle serait alors détecté. La figure 3.37 donne le principe d'un tel testeur.

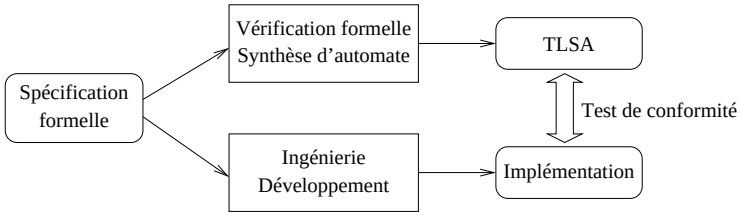


FIG. 3.37 – *Conformité entre implémentation et TLSA*

Les auteurs de [ENDE97] évoquent la difficulté de tester des systèmes temporels. Ils proposent pour cela l'utilisation d'automates temporels conjointement avec une autre entité chargée du test de la partie contrôle du système (automate non temporisé). Dans [LC97], les auteurs apportent une définition formelle du test de systèmes temporels avec la notion de *testeur canonique*, et préconisent les *machines à état étendues à entrées/sorties* (ETIOSM), dérivées du testeur canonique, pour spécifier le système à tester. Les tests de conformité à réaliser sont dérivés de la spécification en ETIOSM. Une autre possibilité proposée dans [CKKP99, EDKE98, CL00] est la génération, à partir de la spécification, de séquences de test dans une syntaxe normalisée dédié au test, TTCN [ISO98], qui permet notamment d'armer des *timers*.

Une approche utilisant le TLSA apparaît comme une réponse possible aux difficultés du test des systèmes temporels. Tout d'abord, le TLSA est un automate permettant d'exprimer autant l'aspect contrôle du système que l'aspect temporel. Par ailleurs, il permet d'exprimer des contraintes temporelles sur des intervalles avec un certain indéterminisme. Il offre donc un pouvoir d'expression supérieur à ce que l'on peut exprimer en armant des timers. Pour finir, bien que les syntaxes ETIOSM et TTCN soient normalisées, le TLSA offre en plus une spécification formelle du testeur et il devrait être possible de dériver des séquences de test TTCN à partir d'un TLSA.

Implémentation et extensions de TURTLE

Le profil TURTLE est doté d'une sémantique caractérisée par la traduction de spécification TURTLE en spécification RT-LOTOS. Cette traduction a été définie de manière claire et non ambiguë. Il serait souhaitable maintenant de l'implémenter.

Un point notablement délicat dans les méthodologies liant conception et analyse est la phase de remontée des résultats d'analyse au niveau de la conception. Pour ce faire, nous proposons de confronter le graphe d'accessibilité issu d'une spécification TURTLE aux diagrammes de séquences initialement envisagés par le concepteur. Une spécification est conforme à un diagramme de séquences s'il existe dans son graphe d'accessibilité un chemin retraçant les séquences décrites par le diagramme. Par ce biais nous pouvons proposer une exploitation des résultats de l'analyse d'accessibilité qui est plus proche des méthodologies de développement des utilisateurs d'UML. Cette approche doit maintenant être étayée par une méthodologie précise. Dans le cadre de l'exploitation de l'analyse d'accessibilité, nous envisageons également un autre axe d'étude consistant à proposer une extension au profile TURTLE qui offrirait au concepteur un certain nombre d'observateurs prédéfinis pour contrôler des propriétés récurrentes, comme par exemple, la durée effective d'une activité.

Par ailleurs, nous avons la possibilité de spécialiser TURTLE par domaine d'application en construisant de nouveaux profils d'extension. Nous avons proposé quelques extensions au profile TURTLE. Il nous semble intéressant d'en proposer d'autres. Notamment nous envisageons d'étendre le profil temps-réel système pour proposer un mécanisme d'interruption/reprise d'activité pour le doter, par exemple, d'un mécanisme de priorités entre les tâches. Le profil temps-réel système que nous avons développé s'est avéré suffisant dans le cas d'étude sur lequel nous l'avons employé [AdSSSL02, Apv02]. Cependant, les dispositifs de séquencement et d'ordonnancement de tâches emploient fréquemment des mécanismes d'interruption/reprise avec priorité. Par conséquent, pour étudier les systèmes ayant recours à de tels dispositifs, il nous faut étendre notre profil temps-réel système dans ce sens.

Il nous semble également intéressant de bâtir un profil pour le domaine du multimédia avec des opérateurs permettant de spécifier la composition logique et temporelle de médias au sein d'un document, tout en permettant d'analyser la cohérence temporelle du document spécifié, et de proposer une vérification de la cohérence. Des travaux sur ce thème ont montré l'intérêt d'avoir recours à une méthode formelle pour analyser les documents multimédia qui sont décrits avec un langage de haut niveau tel que SMIL 2.0 (voir [SC00, SLC01]). Ainsi, nous pourrions proposer également une méthodologie semblable, et gageons-le, à peu de frais.

Publications de l'auteur

[AdSSL+01] L. Apvrille, P. de Saqui-Sannes, C. Lohr, P. Sénac, et J.P. Courtiat. « A New UML Profile for Real-time System Formal Design and Validation ». Dans 4^{th} *International Conference on the Unified Modeling Language*, pages 287–301, Toronto, Canada, octobre 2001. Springer Verlag.

[AdSSSL02] L. Apvrille, P. de Saqui-Sannes, P. Sénac, et C. Lohr. « Reconfiguration dynamique de protocoles embarqués à bord de satellites ». Dans *Colloque Francophone de l'Ingenierie des Protocoles*, pages 441–454, Montréal, Canada, mai 2002. Hermes Science.

[BL00] M. Boyer et C. Lohr. « Modélisation de systèmes temporisés ». Dans *Atelier CNES. Modélisation et Simulation des Systèmes Communicants*, Toulouse, France, décembre 2000.

[CL99] J.P. Courtiat et C. Lohr. « RT-Lotos et rtl. Un retour sur expérience ». Dans *Atelier CNES. Vérification des logiciels réactifs : techniques de description formelle et de modélisation*, Toulouse, France, octobre 1999.

[CSLO00] J.P. Courtiat, C.A.S. Santos, C. Lohr, et B. Outtaj. « Experience with RT-Lotos, a temporal extension of the Lotos formal description technique ». *Computer Communications*, 23:1104–1123, 2000.

[dSSAL+01] P. de Saqui-Sannes, L. Apvrille, C. Lohr, P. Sénac, et J.P. Courtiat. « UML et RT-Lotos : Vers une intégration informel/formel au service de la validation de systèmes temps réel ». Dans *Modélisation des Systèmes Réactifs*, pages 513–528, Toulouse, France, octobre 2001.

[dSSAL+02] P. de Saqui-Sannes, L. Apvrille, C. Lohr, P. Sénac, et J.P. Courtiat. « UML and RT-Lotos An Integration for Real-Time System Validation ». *Journal Européen des Systèmes Automatisés*, 36(7):1029–1042, 2002.

[LC02a] C. Lohr et J.P. Courtiat. « De la spécification à l'ordonnancement de systèmes contraints par le temps ». Dans *Colloque Francophone de l'Ingenierie des Protocoles*, pages 209–222, Montréal, Canada, mai 2002. Hermes Science.

[LC02b] C. Lohr et J.P. Courtiat. « From the specification to the scheduling of time-dependent systems ». Dans 7^th *International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, numéro 2469 dans Lecture Notes in Computer Science, pages 129–145, Oldenburg, Germany, septembre 2002. Springer Verlag.

[SLC01] P.N.M. Sampaio, C. Lohr, et J.P. Courtiat. « An integrated environment for the presentation of consistent SMIL 2.0 documents ». Dans *ACM Symposium on Document Engineering*, pages 115–124, Atlanta, Georgia, USA, novembre 2001.

Références

[ACD90] R. Alur, C. Courcoubetis, et D.L. Dill. « Model-checking for real-time systems ». Dans 5^{th} *IEEE Symposium on Logic in Computer Science*, 1990.

[ACdOP97] L. Andriantsiferana, J.P. Courtiat, R.C de Oliveira, et L. Picci. « An experiment in using RT-Lotos for the formal specification and verification of a distributed scheduling algorithm in a nuclear power plant monitoring system ». Dans 10^{th} *International Conference on Formal Description Techniques*, Osaka, Japan, novembre 1997. Chapman&Hall.

[ACH92] R. Alur, C. Courcoubetis, et N. Halbwachs. « Minimization of timed transition systems ». Dans *CONCUR'92*, volume 630 de *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[ACHWT92] R. Alur, C. Courcoubetis, N. Halbwachs, et H. Wong-Toi. « An Implementation of Three Algorithms for Timing Verification Based on Automata Emptiness ». Dans 13^{th} *Real-time Systems Symposium*, pages 157–166, Phoenix, Arizona, 1992. IEEE Computer Society Press.

[AD91] R. Alur et D.L. Dill. « The theory of timed automata ». Dans *REX Workshop "Real-Time: Theory in Practice"*, volume 600 de *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[AD94] R. Alur et D.L. Dill. « A theory of timed automata ». *Theoretical Computer Science*, 126(2):183–235, 1994.

[AFH94] R. Alur, L. Fix, et T.A. Heizinger. « Event-Clock Automata: A Determinizable Class of Timed Automata ». Dans 6^{th} *Annual Conference on Computeraided Verification*, Lecture Notes in Computer Science 818, pages 1–13. Springer Verlag, 1994.

[AG92] A. Arnold et I. Guessarian. *Mathématiques pour l'informatique*. Masson, 1992.

[AGS00] K. Altisen, G. Gossler, et J. Sifakis. « A Methodology for the Construction of Scheduled Systems ». Dans 6^th *International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 1926 de *Lecture Notes in Computer Science*, pages 106–120, Pune, India, septembre 2000. Springer Verlag.

[AMP94] E. Asarin, O. Maler, et A. Pnueli. « Symbolic Controller Synthesis for Discrete and Timed Systems ». Dans *Hybrid Systems*, pages 1–20, 1994.

[AMPS98] E. Asarin, O. Maler, A. Pnueli, et J. Sifakis. « Controller Synthesis for Timed Automata ». Dans *System Structure and Control*. Elsevier Science, 1998.

[APFR01] C. André, M.A. Peraldi-Frati, et J.P. Rigault. « Scenario and Property Checking of Real-Time Systems Using a Synchronous Approach ». Dans

4^{th} *IEEE Int. Symp on Object-Oriented Real-Time Distributing Computing*, pages 438–444, Magdeburg, Allemagne, mai 2001.

[Apv02] L. Apvrille. « *Contribution à la reconfiguration dynamique de logiciels embarqués temps-réel : application à un environnement de télécommunication par satellite* ». PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, France, juin 2002.

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Masson, 1992.

[BB87] T. Bolognesi et E. Brinksma. « Introduction to the ISO Specification Language Lotos ». *Computer Networks and ISDN Systems*, 14(1), 1987.

[BD91] B. Berthomieu et M. Diaz. « Modeling and verification of time-dependent systems using time Petri nets ». *IEEE Transactions on Software Engineering*, 17(3), 1991.

[BD97] H. Bowman et J. Derrick. « Extending Lotos with Time: True Concurrency Perspective ». Dans *ARTS'97, AMAST Workshop on Real-Time Systems, Concurrent and Distributed Software*, volume 1231 de *Lecture Notes in Computer Science*, pages 382–399. Springer Verlag, mai 1997.

[BDS95] J. Bryans, J. Davies, et S. Schneider. « Towards a denotational semantics for ET-Lotos ». Dans *CONCUR'95*, Lecture Notes in Computer Science, pages 269–283, Philadelphia, USA, août 1995. Springer Verlag.

[Ber01] B. Berthomieu. « La Méthode des Classes d'États pour l'Analyse des Réseaux Temporels - Mise en Œuvre, Extensio à la multi-sensibilisation ». Dans *Modélisation des Systèmes Réactifs*, pages 275–290, Toulouse, France, octobre 2001. Hermes Science.

[BF98] J.M. Bruel et R.B. France. « Transforming UML Models to Formal Specifications ». Dans *International Conference on the Unified Modelling Language (UML): Beyond the Notation*, volume 1618. Springer Verlag, 1998.

[BJLY98] J. Bengtsson, B. Jonsson, J. Lilius, et W. Yi. « Partial Order Reductions for Timed Systems ». Dans *International Conference on Concurrency Theory*, volume 697 de *Lecture Notes in Computer Science*, pages 485–500, Nice, France, septembre 1998. Springer Verlag.

[Bjo00] M. Bjorkander. « Real-Time Systems in UML (and SDL) ». *Embedded System Engineering*, octobre/novembre 2000.

[BK85] J.A. Bergstra et J.W. Klop. « Algebra of communicating process with abstraction ». *Theorectical Computer Science*, 37(1):77–121, 1985.

[BK97] H. Bowman et J. Katoen. « A true concurrency semantics for ET-Lotos ». Rapport Technique, University Erlangen-Nurnberg, décembre 1997.

[BL91a] T. Bolognesi et F. Lucidi. « Lotos-like process algebras with urgent or timed interactions ». Dans 4^{th} *International Conference on Formal Description Techniques*, volume C-2 de *IFIP Transactions*, Sydney, Australia, novembre 1991. Elsevier Science.

[BL91b] T. Bolognesi et F. Lucidi. « Timed Process Algebras with Urgent Interactions and a Unique Powerful Binary Operator ». Dans *REX (Research and Education in Concurrent Systems) - Workshop*, Lecture Notes in Computer Science, pages 124–148, Mook, Hollande, juin 1991. Springer Verlag.

[BM83] B. Berthomieu et M. Menasche. « An Enumerative Approach for Analyzing Time Petri Nets ». Dans *IFIP Congress*, pages 41–46, Paris, 1983.

[BNT94] T. Bolognesi, E. Najm, et P.A.J. Tilanus. « G-Lotos: A Graphical Language for Concurrent Systems ». *Computer Networks and ISDN Systems*, 26(9):1101–1127, 1994.

[Bou99] H. Boucheneb. « *Conception et analyse d'un modèle temporisé unificateur à base de réseaux de* Petri *de Haut Niveau* ». PhD thesis, Université des Sciences et de la technologie Houari Boumedienne, Alger, février 1999.

[Boy01] M. Boyer. « *Contribution à la modélisation des systèmes à temps contraint et application au multimédia* ». PhD thesis, Université Toulouse III, Toulouse, France, juillet 2001.

[Bru99] J.M. Bruel. « Integrating Formal and Informal Specification Techniques. Why? How? ». Dans *IEEE Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, USA, 1999.

[BST97] S. Bornot, J. Sifakis, et S. Tripakis. « Modeling Urgency in Timed Systems ». Dans *Compositionality: The Significant Difference (COMPOS'97)*, numéro 1536 dans Lecture Notes in Computer Science, pages 103–129, Bad Malente, Germany, 1997. Springer Verlag.

[Bur98] A. Burgueño Arjona. « *Formalisation et vérification des systèmes temps réels hybrides (in both french and english)* ». PhD thesis, ENSAE, Toulouse, France, juin 1998.

[BV02] B. Berthomieu et F. Vernadat. « State class constructions for branching analysis of Time Petri nets ». (preliminary) 02130, LAAS, mars 2002.

[CAD] « CADP ». `http://www.inrialpes.fr/vasy/cadp.html`.

[Cas99] E. Castel. « Simulation distribuée d'applications temps-réel ». Rapport Technique, ENSICA - LAAS/CNRS, juin 1999.

[CCS93] J.P. Courtiat, M.S. Camargo, et D.E. Saïdouni. « RT-Lotos : Lotos Temporisé pour la Spécification de Systèmes Temps Réeel ». Dans *Colloque Francophone de l'Ingenierie des Protocoles*, pages 427–441, Montréal, Canada, septembre 1993.

[CdC93] R.J. Coelho da Costa. « *Systèmes de transitions étiquetés causaux : une nouvelle approche pour la description du comportement événementiel de systèmes concurrents* ». PhD thesis, Université Paul Sabatier, Toulouse, France, mai 1993.

[CdCCdO96] J.P. Courtiat, L.F.R. da Costa Carmo, et R.C. de Oliveira. « A general-purpose multimedia synchronization mechanism based on causal relations ». Dans *IEEE Journal on Selected Areas in Communications*, volume 14. $n°1$, janvier 1996.

[CdO94] J.P. Courtiat et R.C. de Oliveira. « About time nondeterminism and exception handling in a temporal extension of Lotos ». Dans 14^{th} *International Conference on Protocol Specification, Testing and Verification*, Vancouver, Canada, juin 1994. Chapman&Hall.

[CdO95] J.P. Courtiat et R.C. de Oliveira. « On RT-Lotos and its application to the formal design of multimedia protocols ». Dans *Annals of Telecommunications*, volume 50. $n°$11–12, 1995.

[CdO96] J.P. Courtiat et R.C. de Oliveira. « Proving temporal consistency in a new multimedia synchronization model ». Dans *ACM Multimedia'96*, Boston, USA, novembre 1996.

[CES86] E.M. Clarke, E.A. Emerson, et A.P. Sistla. « Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications ». *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, avril 1986.

[CKKP99] B.M Chin, S.U. Kim, S.W. Kang, et C.H. Park. « Modeling and Interoperability Test Generation of a Real-Time QoS Monitoring Protocol ». *ETRI*, 21(4):52–64, 1999.

[CL00] R. Castanet et P. Laurençot. « Testing Real-Time Systems ». Dans 15^{th} *World Conference on Nondestructive Testing*, Roma, Italy, octobre 2000.

[CM00] R.G. Clarck et A.M.D. Moreira. « Use of E-Lotos in Adding Formality to UML ». *Journal of Universal Computer Science*, 6(11):1071–1087, 2000.

[CMS99] A. Cerone et A. Maggiolo-Schettini. « Time-base expressivity of time Petri nets for system specification ». *Theoretical Computer Science*, 216(1–2):1–53, mars 1999.

[CRdO95] J.P. Courtiat et R.C. R. de Oliveira. « A reachability analysis of RT-Lotos specifications ». Dans 8^{th} *International Conference on Formal Description Techniques*, Montreal, Canada, octobre 1995. Chapman&Hall.

[DBS95] J. Davies, J. Bryans, et S. Schneider. « Real-time Lotos and Timed Observations ». Dans 8^{th} *International Conference on Formal Description Techniques*, pages 383–397, Montreal, Canada, octobre 1995. Chapman&Hall.

[DdB01] S. Dupuy et L. du Bousquet. « A Multi-formalism Approach for the Validation of UML Models ». *Formal Aspects of Computing*, 12(4):228–230, 2001.

[Die97] H. Dierks. « Synthesising Controllers from Real-Time Specifications ». Dans *Tenth International Symposium on System Synthesis*, pages 126–133. IEEE Computer Society Press, septembre 1997.

[Die01] H. Dierks. « PLC-Automata: a new class of implementable real-time automata ». *Theoretical Computer Science*, 253(1):61–93, février 2001.

[Dij75] E.W. Dijkstra. « Guarded commands, nondeterminacy, and formal derivation of programs ». *Communications of the ACM*, 18(8):453–457, août 1975.

[DMP91] R. Dechter, I. Meiri, et J. Pearl. « Temporal constraint networks ». *Artificial Intelligence*, 49(1-3):61–95, 1991.

[Dou99] B.P. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison-Wesley Longman, 1999.

[DP98] J. Delatour et M . Paludetto. « UML/PNO, a way to merge UML and Petri net objects for the analysis of real-time systems ». Dans *Object-Oriented Technology and Real Time Systems*, volume 1543, Bruxelles, Belgique, 1998.

[DS93] M. Diaz et P. Sénac. « Time stream Petri nets, a model for multimedia streams synchronisation ». Dans *Proceedings of MultiMedia Modeling'93*, Singapore, novembre 1993.

[DY96] C. Daws et S. Yovine. « Reducing the number of clock variables of timed automata ». Dans 17^{th} *IEEE Real-Time Systems Symposium*, pages 73–81, Washington, USA, décembre 1996. IEEE Computer Society Press.

[EC82] E.A. Emerson et E.M. Clarke. « Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons ». *Science of Computer Programming*, 2(3):241–266, décembre 1982.

[ECM+99] A.S. Evans, S. Cook, S. Mellor, J. Warmer, et A. Wills. « Advanced Methods and Tools for a Precise UML ». Dans 2^{nd} *International Conference on the Unified Modeling Language UML'99*, Lecture Notes in Computer Science, Colorado, USA, 1999. Springer Verlag.

[EDKE98] A. En-Nouaary, R. Dssouli, F. Khendek, et A. Elqortobi. « Timed Test Cases Generation Based on State Characterization Technique ». Dans 19^{th} *IEEE Real-Time Systems Symposium*, Madrid, Spain, décembre 1998.

[EJP97] E.A. Emerson, S. Jha, et D. Peled. « Combining Partial Order and Symmetry Reductions ». Dans *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 de *Lecture Notes in Computer Science*, pages 19–34, Enschede, The Netherlands, avril 1997. Springer Verlag.

[ELo01] « E-Lotos: Enhancements to Lotos », septembre 2001. ISO/IEC 15437.

[EM85] H. Ehrig et B. Mahr. « Fundamentals of Algebraic Specification 1, Equations and Initial Semantics ». *EATCS - Monographs on Theoretical Computer Science*, 6, 1985.

[ENDE97] A En-Nouaary, R. Dssouli, et A. Elqortobi. « Géneration de Tests Temporisés ». Dans 6^{th} *Colloque Francophone de l'Ingenierie des Protocoles*, Liege, Belgique, octobre 1997.

[EST] « Esterel Studio ». `http://www.esterel-technologies.com/v2/index.html`.

[Feh99] A. Fehnker. « Scheduling a steel plant with timed automata ». Rapport Technique CSIR9910, Computing Science Institute Nijmegen, 1999.

[GA98] L.Z. GRANVILLE et M.J.B. ALMEIDA. « E-Dart: A Specification Environment for the E-LOTOS Formal Technique ». Dans *European Concurrent Engineering Conference*, Erlangen, Allemagne, avril 1998.

[Gen89] H.J. GENRICH. Equivalence Transformation of PrT-Nets. Dans *Advances in Petri nets*, numéro 424 dans LNCS, page 179. Springer Verlag, 1989.

[GLo89] « G-LOTOS: a Graphical Syntax for LOTOS », 1989. ISO/IEC JTC1/SC21 N3253.

[God96] P. GODEFROID. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 de *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[God99] P. GODEFROID. « Exploiting Symmetry when Model-Checking Software ». Dans *Formal Methods for Protocol Engineering and Distributed Systems*, volume 156 de *IFIP*, pages 257–275, Beijing, China, octobre 1999. Kluwer Academic Publisher.

[Gro93] J.F. GROOTE. « Transition system specifications with negative premises ». *Theoretical Computer Science*, 118(2):263–299, septembre 1993.

[Gue81] I. GUESSARIAN. *Algebraic Semantics*. Numéro 99 dans Lecture Notes in Computer Science. Springer Verlag, 1981.

[GV90] J.F. GROOTE et F. VAANDRAGER. « An efficient algorithm for branching bisimulation and stuttering equivalence ». Dans 17^{th} *International Colloquium on Automata, Languages and Programming*, volume 443 de *Lecture Notes in Computer Science*, pages 626–638. Springer Verlag, juin 1990.

[GW93] P. GODEFROID et P. WOLPER. « Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties ». *Formal Methods in System Design*, 2(2):149–164, avril 1993.

[Hal90] J.A. HALL. « Seven Myths of Formal Methods ». *IEEE Software*, 7(5):11–19, septembre 1990.

[HD96] C. HEITMEYER et Mandrioli D., éditeurs. *Formal Methods for Real-Time Computing*, volume 4 de *Trends in Software*. John Wiley & Sons Ltd, Chichester, UK, 1996.

[Her97] C. HERNALSTEEN. « A timed automaton model for ET-LOTOS verification ». Dans *Formal Description Techniques X/Protocol Specification, Testing and Verification XVII*, pages 193–204, London, UK, novembre 1997. Chapman&Hall.

[HNSY94] T.A. HENZINGER, X. NICOLLIN, J. SIFAKIS, et S. YOVINE. « Symbolic model checking for real-time systems ». *Information and Computation*, 2(111):193–244, 1994.

[Hoa69] C.A.R. HOARE. « An axiomatic basis for computer programming ». *Communications of the ACM*, 12(10):576–580, 583, octobre 1969.

[Hoa78] C.A.R. HOARE. « Communicating Sequential Processes ». *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa89] C.A.R. HOARE. *Communicating Sequential Processes.* Prentice-Hall, 1989.

[Hol91] G.J. HOLZMANN. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[HR98] J. HILLSTON et J. RIBAUDO. Stochastic process algebra: a new approach to performance modeling. Dans *Modeling and Simulation of Advanced Computer Systems.* Gordon Breach, 1998.

[ISO98] ISO/IEC. « Open systems intercinnection - Conformance testing methodology and framework ». Part 3: Tree and Tabular Combined Notation ISO/IEC 9646-3, Information Technology, 1998.

[IT97] ITU-T. « Framework on formal methods in conformance testing ». Rapport Technique Recommendations Z.500, International Telecommunication Union, mars 1997.

[JJP98] C. JARD, J.M. JÉZÉQUEL, et F. PENNANEAC'H. « Vers l'utilisation d'outils de validation de protocoles dans UML ». *Technique et Science Informatiques*, 17(9):1083–1098, septembre 1998.

[Kha97] W. KHANSA. « *Réseaux de Petri p-temporels : contribution à l'étude des systèmes à événements discrets* ». PhD thesis, Université de Savoie, Annecy, France, mars 1997.

[KL96] I. KANG et I. LEE. « An Efficient State Space Generation for Analysis of Real-Time Systems ». Dans *International Symposium on Software Testing and Analysis*, pages 4–13, 1996.

[KLK00] I. KANG, I. LEE, et Y.S. KIM. « An Efficient State Space Generation for Analysis of Real-Time Systems ». *Transactions of Software Engineering*, 26(5):453–477, 2000.

[KMH00] L. KHATIB, N. MUSCETTOLA, et K. HAVELUND. « Verification of Plan Models Using UPPAAL ». Dans 1^{st} *International Workshop on Formal Approaches to Agent-Based Systems*, volume 1871 de *Lecture Notes in Computer Science*, pages 114–122. Springer Verlag, 2000.

[KMH01] L. KHATIB, N. MUSCETTOLA, et K. HAVELUND. « Mapping Temporal Planning Constraints into Timed Automata ». Dans 8^{th} *International Workshop on Temporal Representation and Reasoning*, Cividale del Friuli, Italy, juin 2001. IEEE Computer Society Press.

[Kro] KRONOS. `http://www-verimag.imag.fr/TEMPORISE/kronos`.

[Lam77] L. LAMPORT. « Proving the correctness of multiprocess programs ». *IEEE Transactions on Software Engineering*, 3(2):125–143, mars 1977.

[LC97] P. LAURENÇOT et R. CASTANET. « Integration of Time in Canonical Testers for Real-Time Systems ». Dans 3^{rd} *Workshop on Object-Oriented Real-Time Dependable Systems*, Newport Beach, USA, février 1997. IEEE Computer Society Press.

[Led91] G. LEDUC. « An Upward Compatible Timed Extension to LOTOS ». Dans 4^{th} *International Conference on Formal Description Techniques*, volume C-

2 de *IFIP Transactions*, pages 217–232, Sydney, Australia, novembre 1991. Elsevier Science.

[Led94] G. LEDUC. A Method for Applying LOTOS at an Early Design Stage and its Application to the ISO Transport Protocol. Dans *The OSI95 Transport Service with Nultimedia Support*, volume 1, pages 151–180. Springer Verlag, University of Liège, Belgium, juillet 1994.

[LFHH92] L. LOGRIPPO, M. FACI, et M. HAJ-HUSSEIN. « An Introduction to LOTOS: Learning by Examples ». *Computer Networks and ISDN Systems*, 23(5):325–342, 1992. Errata in 25(1) (1992) 99-100.

[LG00] A. LE GUENNEC. « Méthodes formelles avec UML : Modélisation, validation et génération de tests ». Dans *Colloque Francophone de l'Ingenierie des Protocoles*, pages 151–166, Toulouse, France, octobre 2000.

[Liv78] C. LIVERCY. *Théorie des programmes, schémas, preuves, sémantique.* Dunod, 1978.

[LL92] G. LEDUC et L. LÉONARD. « A timed LOTOS supporting a dense time domain and including new timed operators ». Dans 5^{th} *International Conference on Formal Description Techniques*, volume C-10 de *IFIP Transactions*, pages 87–102, Lannion, France, octobre 1992. Elsevier Science.

[LL93] L. LÉONARD et G. LEDUC. « An Enhanced Version of Timed LOTOS and its Application to a Case Study ». Dans 6^{th} *International Conference on Formal Description Techniques*, volume C-22 de *IFIP Transactions*, pages 483–498, Boston, USA, octobre 1993. Elsevier Science.

[LL97] L. LÉONARD et G. LEDUC. « An introduction to ET-LOTOS for the description of time-sensitive systems ». Dans *Computer Networks and ISDN Systems*, volume 29, pages 271–292, 1997.

[LL98a] F. LAROUSSINIE et K.G. LARSEN. « CMC: a Tool for Compositionale Model-Checking of Real-Time Systems ». Dans 11^{th} *International Conference on Formal Description Techniques*, pages 439–456, Paris, France, novembre 1998. IFIP, Kluwer Academic Publishers.

[LL98b] L. LÉONARD et G. LEDUC. « A Formal definition of time in LOTOS ». *Formal Aspecs of Computing*, 10(3):248–266, 1998.

[LLdF+95] G. LEDUC, L. LÉONARD, D. de FRUTO, L. LLANA, L. MIGUEL, J. QUEMADA, et G. RABAY. « Time Extended LOTOS ». Dans *Annex C of ISO/IEC JTC1/SC21/WG7 N1053: Revised Working Draft on Enhancements to* LOTOS *(V2)*, Liege, Belgique, septembre 1995.

[Lot88] « LOTOS, a formal description technique based on temporal ordering of observational behavior », 1988. ISO Standard 8807.

[MBC+94] M.A. MARSAN, A. BIANCO, L. CIMINIERA, R. SISTO, et A. VALENZANO. « A LOTOS Extension for the Performance Analysis of Distributed Systems ». *IEEE/ACM Transaction on Networking*, 2(2), 1994.

[Mer74] P. MERLIN. « *A study of the recoverability of computer system* ». PhD thesis, Dep. Comput. Sci., Univ. California, Irvine, 1974.

[MF76] P. MERLIN et D.J. FABER. « Recoverability of communication protocols ». *IEEE Transactions on Communications*, COM-24(9), septembre 1976.

[MFV92] C. MIGUEL, A. FERNÁNDEZ, et L. VIDALLER. « Extending LOTOS towards performance evaluation ». Dans 5^{th} *International Conference on Formal Description Techniques*, volume C-10 de *IFIP Transactions*, pages 103–118, Lannion, France, octobre 1992. Elsevier Science.

[Mil89] R. MILNER. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, 1989.

[MOSS99] M. MÜLLER-OLM, D.A. SCHMIDT, et B. STEFFEN. « Model-Checking: A Tutorial Introduction ». Dans *Static Analysis,* 6^{th} *International Symposium, (SAS'99)*, Lecture Notes in Computer Science, pages 330–354, Venice, Italy, septembre 1999. Springer Verlag.

[MT90] F. MOLLER et C. TOFTS. « A temporal calculus of communicating systems ». Dans *Theories of Concurrency: Unification and Extension*, volume 458 de *Lecture Notes in Computer Science*, Amsterdam, Netherlands, août 1990. Springer Verlag.

[Nak97] A. NAKATA. « *Symbolic Bisimulation Checking and Decomposition of Real-Time Service Specifications* ». PhD thesis, Osaka University, Japon, janvier 1997.

[NHT93] A. NAKATA, T. HIGASHINO, et K. TANIGUCHI. « LOTOS enhancement to specify time constraint among non-adjacent actions using first order logic ». Dans 6^{th} *International Conference on Formal Description Techniques*, volume C-22 de *IFIP Transactions*, pages 451–466, Boston, USA, octobre 1993. Elsevier Science.

[NS91] X. NICOLLIN et J. SIFAKIS. « An overview and synthesis on timed process algebras ». Dans *REX Workshop "Real-Time: Theory in Practice"*, volume 600 de *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[NS94] X. NICOLLIN et J. SIFAKIS. « The Algebra of Timed Processes ATP: Theory and Application ». *Information and Computation*, 114(1):131–178, 1994.

[NSY93] X. NICOLLIN, J. SIFAKIS, et S. YOVINE. « From ATP to Timed Graphs and Hybrid Systems ». *Acta Informatica*, 30(2):181–202, 1993.

[Pel98] D. PELED. « Ten Years of Partial Order Reduction ». Dans 10^{th} *International Conference on Computer-Aided Verification*, volume 1427 de *Lecture Notes in Computer Science*, pages 17–28, Vancouver, Canada, juin 1998. Springer Verlag.

[Pet99] P. PETTERSSON. « *Modelling and verification of real-time systems using timed automata: theory and practice* ». PhD thesis, Uppsala University, Uppsala, Sweden, février 1999.

[Pnu77] A. PNUELI. « The temporal logic of programs ». Dans *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, octobre 1977. IEEE Computer Society Press.

[Pnu86] A. PNUELI. « Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends ». Dans *Current Trends in Concurrency: Overviews and Tutorials*, volume 224 de *Lecture Notes in Computer Science*, pages 510–584. Springer Verlag, 1986.

[PPL] « Parama Polyhedra Library ». `http://www.cs.unipr.it/ppl`.

[QAdF89] J. QUEMADA, A. AZCORRA, et D. de FRUTOS. « TIC: A Timed Calculus For LOTOS ». Dans 2^{nd} *International Conference on Formal Description Techniques*, IFIP Transactions, pages 195–209, Vancouver, Canada, décembre 1989. Elsevier Science.

[QdFA93] J. QUEMADA, D. de FRUTOS, et A. AZCORRA. « TIC: A Timed Calculus ». *Formal Aspects of Computing*, 5(3):224–252, 1993.

[QF87] J. QUEMADA et A. FERNÁNDEZ. « Introduction of Quantitative Relative Time into LOTOS ». Dans *International Conference on Protocol Specification, Testing and VerificationVII*, pages 105–121, Zurique, Suisse, mai 1987. Elsevier Science.

[Ram74] C. RAMCHANDANI. « *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets* ». PhD thesis, MIT, Cambridge, février 1974.

[RP84] R.R. RAZOUK et C.V. PHELPS. « Performance Analysis Using Timed PETRI Nets ». Dans *Int. Conf. on Parallel Processing*, pages 126–128, New York, 1984. IEEE Service Cent.

[RTS99] « Real-Time Studio ». Artisan Software Tools, 1999. `http://www.artisansw.com`.

[Saï96] D.E. SAÏDOUNI. « *Sémantique de maximalité : Application au raffinement d'actions en* LOTOS ». PhD thesis, LAAS-CNRS, Université Toulouse III, mai 1996.

[SC00] P.N.M. SAMPAIO et J.P. COURTIAT. « A Formal Approach for the Presentation of Interactive Multimedia Documents ». Dans *ACM Multimedia'2000*, pages 435–438, Los Angeles, USA, octobre 2000.

[SC01] P.N.M. SAMPAIO et J.P. COURTIAT. « Scheduling and Presenting Interactive Multimedia Documents ». Dans *International Conference on Multimedia and Exposition'2001*, pages 1227–1227, Tokyo, Japan, août 2001.

[SDLdSS96] P. SÉNAC, M. DIAZ, A. LÉGER, et P. de SAQUI-SANNES. « Modelling logical and temporal synchronization in hypermedia systems ». *IEEE Journal on Selected Areas in Communications*, 14(1):84–103, janvier 1996.

[Sif77] J. SIFAKIS. « Use of Petri Nets for Performance Evaluation ». Dans E. BEILNER et E. GELENBE, éditeurs, *Measuring Modeling and Evaluating Computer Systems*, pages 75–93, Elsevier Science, 1977.

[Sén96] P. SÉNAC. « *Contribution à la modélisation des systèmes multimédias et hypermédias* ». PhD thesis, Université Paul Sabatier, juin 1996.

[SR98] B. SELIC et J. RUMBAUGH. « Using UML for Modeling Complex Real-Time Systems ». Rapport Technique, Rational Software, mars 1998.

[SSC99] C.A.S. SANTOS, P.N.M. SAMPAIO, et J.P. COURTIAT. « Revisiting the Concept of Hypermedia Document Consistency ». Dans *ACM Multimedia'99*, Orlando, USA, novembre 1999.

[Sto96] N. STOREY. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[TG01] F. TERRIER et S. GÉRARD. « Real Time System Modeling with UML: Current Status and Some Prospects ». Dans 15^{th} *European Conference on Object-Oriented Programming*, Budapest, Hungary, juin 2001.

[TH97] K. TINDELL et H. HANSSON. *Real Time Systems by Fixed Priority Scheduling (Course notes)*. Department of Computer Systems, Uppsala University, octobre 1997.

[Tin] TINA. « TIme PETRI Nets Analyzer ». `http://www.laas.fr/tina`.

[Tra00] I. TRAORÉ. « An Outline of PVS Semantics for UML Statecharts ». *Journal of Universal Computer Science*, 6(11):1088–1108, 2000.

[UML01] Object Managment Group. « *Unified Modeling Language Specification, Version 1.4* », septembre 2001. `http://www.omg.org/cgi-bin/doc?formal/01-09-6`.

[UPP] « UPPAAL ». `http://www.uppaal.com`.

[Val93] A. VALMARI. « On-The-Fly Verification With Stubborn Sets ». Dans 5^{th} *International Conference on Computer-Aided Verification*, volume 697 de *Lecture Notes in Computer Science*, pages 397–408, Elounda, Greece, juin 1993. Springer Verlag.

[VAM96] F. VERNADAT, P. AZÉMA, et F. MICHEL. « Covering Step Graph ». Dans 17^{th} *Int. Conf. on Application and Theory of* PETRI *Nets*, volume 1091, pages 516–535, Osaka, Japan, juin 1996. Springer Verlag.

[Ver97] J.J. VEREIJKEN. « *Discrete-Time Process Algebra* ». PhD thesis, Eindhoven University of Technology, décembre 1997.

[VHJ96] D. VAN HUNG et W. JI. « On the Design of Hybrid Control Systems Using Automata Models ». Dans *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 de *Lecture Notes in Computer Science*, pages 156–167. Springer Verlag, 1996.

[vHTZ89] W.H.P. van HULZEN, P.A.J. TILANUS, et H. ZUIDWEG. « LOTOS Extended With Clocks ». Dans 2^{nd} *International Conference on Formal Description Techniques*, pages 179–193, Vancouver, Canada, décembre 1989. Elsevier Science.

[VM97] F. VERNADAT et F. MICHEL. « Covering Step Graph Preserving Failure Semantics ». Dans 18^{th} *Int. Conf on Application and Theory of* PETRI *Nets*, volume 1248, pages 253–270, Toulouse, France, juin 1997. Springer Verlag.

[VSvSB91] C.A VISSERS, G. SCOLLO, M. van SINDEREN, et E. BRINKSMA. « Specification styles in distributed systems design and verification ». *Theoretical Computer Science*, 89:179–206, 1991.

[Wal83] B. WALTER. « Timed Net for Modeling and Analysing protocols with time ». Dans *Proceedings of the IFIP Conference on Protocol Specification Testing and Verification*, Elsevier Science, 1983.

[Win90] J.M. WING. « A Specifier's Introduction to Formal Methods ». *IEEE Computer*, 23(9):8–24, septembre 1990.

[YL93] M. YANNAKAKIS et D. LEE. « An efficient algorithm for minimizing real-time transition systems ». Dans *CAV'93*, volume 697 de *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[Yov93] S. YOVINE. « *Méthodes et Outils pour la Vérification Symbolique de Systèmes Temporisés* ». PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, mai 1993.

[YR98] T. YONEDA et H. RYUBA. « CTL Model Checking of Time PETRI Nets using Geometric Regions ». *IEICE Transactions on Information and Systems*, E99-D(3):297–395, mars 1998.

Annexes

A

Justification de la traduction des domaines d'urgence en invariants

La sous-section 1.3.4 propose une traduction des automates temporisés de type `DTA` *(voir la sous-section 1.3.2) en automates de type* `Kronos` *([Kro]), notés ici TA. Lors de la construction de l'automate temporisé TA associé au* `DTA`*, certains étapes sont évidentes et d'autres non. Elles sont justifiées ici.*

Preuve de (3)

Le nombre d'horloges manipulées par un automate temporisé est constant tout au long des différents états de l'automate, ce qui n'est pas le cas pour le `DTA` où le nombre d'horloges est variable suivant les états de l'automate afin de minimiser le nombre d'horloges. L'ensemble X sera par conséquent l'ensemble d'horloges associé à un état où le nombre d'horloges est maximal. C'est-à-dire $X = \{c_i \mid 1 \leqslant i \leqslant N\}$ où $N = \max\{Nclock(s) \mid s \in S\}$.

Preuve de (4.e)

Dans le `DTA`, on procède à un renommage des horloge afin de minimiser leur nombre. La construction de la valuation ρ doit :

- satisfaire la sémantique du TA à construire, à savoir $\nu' = \nu[\rho]$
- permettre un renommage des horloges afin de préserver l'association entre les gardes d'activation K et ψ quand la transition sera tirée.
- évaluer les horloges restantes (c'est-à-dire, celles qui n'appartiennent pas à l'ensemble $\{c_i \mid i \in \{1...Nclock(s)\}\}$).

Soit un état (s,s',K,U,a,C,θ) de E, nous avons construit l'état correspondant $(s^{TA},L',\psi,\rho,s'^{TA})$. Soit ν_dta la valuation des horloges du `DTA`, et ν_ta celle du TA associé.

Supposons que, pour deux horloges y et z appartenant à $\{c_i \mid i \in \{1...Nclock(s)\}$, on ait : $\theta(y) = z$ (ou encore $y = \theta^{-1}(z)$).

La valuation ρ étant construite de la façon suivante :

$$\rho : X \to X^*$$
$$\rho(x) = \begin{cases} \theta^{-1}(x) & si\ x \in Im(\theta) \\ 0 & sinon\ (c'est - á - dire\ si\ x \in X - Im(\theta)) \end{cases}$$

La preuve de (4.e) consiste à montrer que $\nu'\_ta(z) = \nu\_ta(y)$.

Calculons $\nu'\_ta(z)$:

$\nu'\_ta(z)=\nu\_ta(\rho(z))$ (D'après la définition de l'automate temporisé et du fait que $z \in X$)

$=\nu\_ta(\theta^{-1}(z))$ (Définition de ρ et $z \in Im(\theta)$)

$=\nu\_ta(y)$ (Car $y = \theta^{-1}(z)$)

d'où l'assertion (4.e).

Preuve de (5)

D'après la définition de l'automate temporisé, l'invariant associé à un état est un prédicat qui caractérise la progression du temps dans cet état.

Dans le `DTA`, on peut le formaliser comme étant le domaine temporel où les actions non urgentes peuvent se produire. Aussi, d'après la définition des transitions implicites du `DTA`, le temps ne peut pas progresser dans le domaine U si l'action a est urgente. Autrement dit, le temps peut progresser jusqu'aux frontières gauches de U incluses. L'ajout des frontières gauches du domaine temporel U est justifié par la possibilité de tir de l'action urgente même à la frontière gauche du domaine temporel où elle peut être tirée.

En résumé :

- Si l'action est urgente et si U est non vide, l'invariant est donc le domaine temporel qui précède le domaine U et que nous avons noté par $Pred(U)$.
- Dans le cas où l'ensemble U est composé de plusieurs contraintes, $Pred(U)$ l'est aussi et est interprété comme une disjonction de ces contraintes car le blocage du temps par l'une des contraintes peut ne pas l'être pour les autres contraintes.
- Dans le cas où U est vide, cela veut tout simplement dire que l'action n'est jamais urgente. Le temps peut donc progresser indéfiniment. La même situation se présente également quand l'action est observable.

Dans le cas où plusieurs transitions a partent de l'état s, l'invariant qui caractérise l'état s sera la conjonction de tous les invariants associées à toutes ces transitions. Ceci

est justifié par le fait que le temps doit progresser de la même façon pour les différentes transitions. Si le temps ne progresse pas pour une transition, il ne va pas le faire pour les autres transitions. C'est la première transition urgente qui bloquera le temps qui sera alors tirée. Le domaine de progression du temps pour cette transition correspond donc à la conjonction des différentes progressions du temps. D'où l'assertion (5).

B

Pré-traitement du graphe minimal d'accessibilité

Le graphe minimal d'accessibilité tel qu'il est produit[1] *par* `rtl` *n'est pas directement exploitable par l'algorithme de synthèse du* `TLSA`*. Nous détaillons ici les diverses manipulations à réaliser ainsi que les outils disponibles à cette fin.*

B.1 Reporter les fonctions C et *theta*

Pour traiter les transitions LOTOS, l'algorithme a besoin des fonctions `C` et `theta` du `DTA`. Le programme `dtargmerge` permet cela : il prend en entrée le `DTA` et le graphe minimal d'accessibilité et ajoute aux transitions LOTOS du graphe minimal d'accessibilité les fonctions `C` et `theta` du `DTA` prises sur les transitions entre les états de contrôle que décrivent les régions. (figure B.1).

[1] la commande est : `rtl -TG3 -p1 -d2 spec.lot > spec.rg`

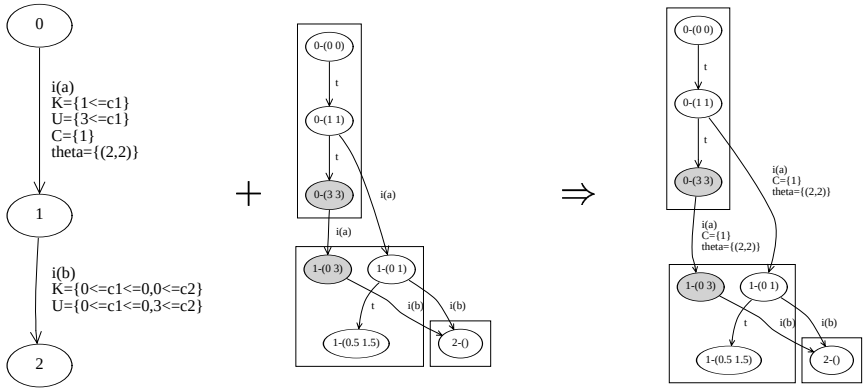


FIG. B.1: Reporter les fonctions `C` et `theta`

B.2 Rerouter les régions-point

L'algorithme actuellement implanté dans `rtl` pour la génération du graphe minimal d'accessibilité (voir la sous-section 1.3.3) trouve parfois des transitions qui aboutissent à des points appartenant à une région déjà créée. Par exemple la transition `( 0-(2 2), i(b), 2-(2) => N26-L3 )` sur la figure B.2.b. Il faut alors regrouper cette région-point avec la région à laquelle elle appartient (il faut trouver celle dont les équations sont vérifiées par les horloges de la région-point).

Dans l'exemple décrit par la figure B.2, l'algorithme de minimisation a trouvé la région `2-(0) 0<=c1<3` comme successeur de la région `0-(0 0)`. Puis il trouve le point `2-(2)` comme successeur de la région `0-(2 2)`. Ce point `2-(2)` appartient bien à la région `2-(0)`, mais l'algorithme se trouve dans une autre branche (marquée par une progression temporelle de 2 unités de temps) de l'arbre d'exploration des configurations. Par conséquent, il ne fait pas pointer cette branche de l'arbre vers le nœud correspondant à la région `2-(0)` mais construit un artéfact de région portant le nom `2-(2)` suivi de la position de la région `2-(0)` dans l'arbre d'exploration. C'est précisément ce type d'artéfact de région que nous avons appelé ici *région-point*.

Nous allons donc réintégrer la région-point `2-(2)` dans la région `2-(0)`. Nous ne perdrons pas l'information comme quoi, en venant de la région `0-(2 2)`, le système arrive dans une autre configuration de la région `2-(0)` différente de la configuration atteinte en venant de `0-(0 0)`. Si ces points d'arrivé sont effectivement différents, l'algorithme de détermination des configurations accessibles au plus tôt le détectera et, au besoin, répliquera la région `2-(0)` (voir paragraphe ii) page 86).

B.3 Projection et minimisation

Pour des questions de facilité d'écriture nous pouvons être amenés à ajouter dans la spécification RT-LOTOS des actions qui n'ont pas d'intérêt dans l'ordonnancement final

a) Spécification RT-LOTOS

```
specification strap: exit

type natural is sorts nat
opns + : nat,nat->nat
     - : nat,nat->nat
endtype

behaviour
  hide a, b in
    ( latency(3) a; stop )
    |||
    ( latency(2) b; stop )
endspec
```

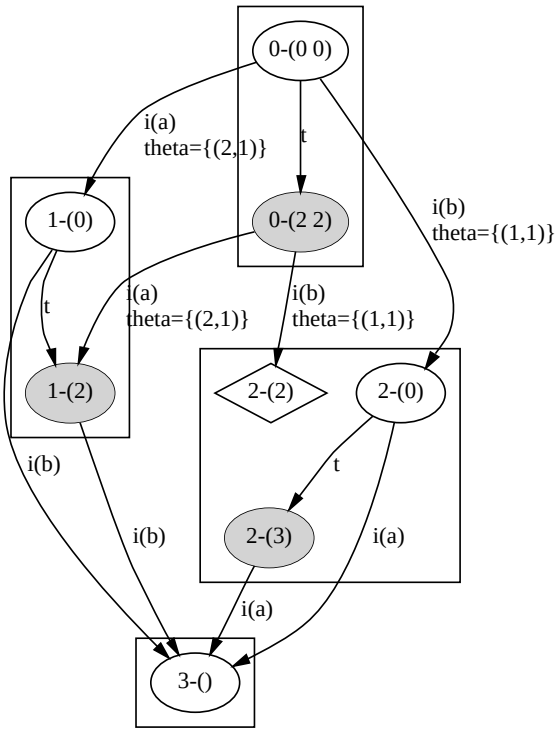
b) Graphe minimal d'accessibilité

```
0-(2 2) URG 0<=c1<3 c2>=2 c2-c1>-1
0-(0 0) 0<=c1<3 0<=c2<2 -1<c2-c1<2
1-(2) URG c1>=2
1-(0) 0<=c1<2
2-(3) URG c1>=3
2-(0) 0<=c1<3
3-()
( 0-(2 2), i(a), 1-(2) )
( 0-(2 2), i(b), 2-(2) => N26-L3 )
( 0-(0 0), i(a), 1-(0) )
( 0-(0 0), i(b), 2-(0) )
( 0-(0 0), t, 0-(2 2) )
( 1-(2), i(b), 3-() )
( 1-(0), i(b), 3-() )
( 1-(0), t, 1-(2) )
( 2-(3), i(a), 3-() )
( 2-(0), i(a), 3-() )
( 2-(0), t, 2-(3) )
```

c) Graphe minimal d'accessibilité rerouté

```
0-(2 2) URG 0<=c1<3 c2>=2 c2-c1>-1
0-(0 0) 0<=c1<3 0<=c2<2 -1<c2-c1<2
1-(2) URG c1>=2
1-(0) 0<=c1<2
2-(3) URG c1>=3
2-(0) 0<=c1<3
3-()
( 0-(2 2), i(a), 1-(2) )
( 0-(2 2), i(b), 2-(0) )
( 0-(0 0), i(a), 1-(0) )
( 0-(0 0), i(b), 2-(0) )
( 0-(0 0), t, 0-(2 2) )
( 1-(2), i(b), 3-() )
( 1-(0), i(b), 3-() )
( 1-(0), t, 1-(2) )
( 2-(3), i(a), 3-() )
( 2-(0), i(a), 3-() )
( 2-(0), t, 2-(3) )
```

d) Graphe minimal d'accessibilité



e) Graphe minimal d'accessibilité rerouté

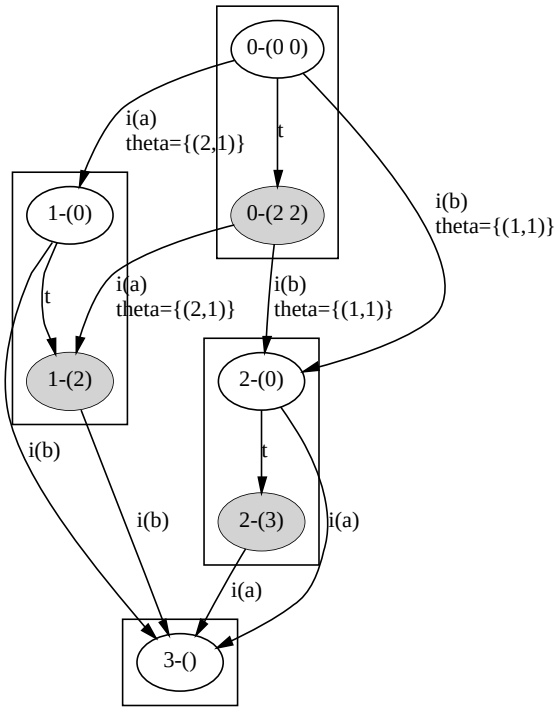


FIG. B.2: Rerouter le graphe minimal d'accessibilité

du système. Il peut alors être souhaitable de ne pas les faire apparaître dans le TLSA. Nous pouvons faire cela simplement en appliquant sur le graphe minimal d'accessibilité des algorithmes classiques de projection ou de minimisation de systèmes temporisés étiquetés, et ce, avant la synthèse du TLSA.

Les outils CADP [CAD], proposent de nombreuses fonctionnalités dans ce sens. Deux outils rg2bcg et bcg2rg ont été développés pour traduire les graphes minimaux d'accessibilité tels qu'ils sont produits par rtl dans le format utilisés par les outils CADP, et réciproquement.

B.3.1 Choix de l'outil CADP

L'outil aldebaran permet notamment la projection observationelle. Les transitions à cacher doivent tout d'abord être renommées en l'action i (l'outil bcg_label permet cela), puis un graphe équivalent au sens de l'équivalence observationnelle est engendré. Cela correspond bien à ce que nous cherchons à faire. aldebaran a des options -ocla, -bcla, -pcla, etc. qui affichent les classes d'équivalences des états après le raffinement de partition. Toutefois, pour certaines équivalences, il y a des phases de saturation préliminaires qui créent ou éliminent des transitions, ce qui fait que les classes risquent de ne pas être affichées correctement.

Une autre manière de faire est de minimiser par équivalence de branchement en utilisant l'outil bcg_min qui utilise l'algorithme définit dans [GV90]. Cet outil dispose d'une option -class qui imprime très fidèlement les classes d'équivalence. En cela il se distingue de l'outil aldebaran qui peut être induit en erreur lorsqu'il imprime les classes d'équivalence qu'il construit, du fait de l'élimination de τ-transitions (action i). C'est donc bcg_min qui a retenu notre attention.

B.3.2 Insertion de la minimisation dans la production du TLSA

Les étapes habituelles de la génération d'un TLSA sont les suivantes :

1. Génération du RG (graphe minimal d'accessibilité associé) et du TLSA à partir de la spécification RT-LOTOS
2. Ajout des fonctions C={...} et theta={...} du DTA sur le RG
3. Élagage des transitions du RG qui conduisent à des états puits ou non désirables.
4. Génération du TLSA.

La minimisation va s'insérer entre les phases 1 et 2, à l'aide des deux outils rg2bcg et bcg2rg.

B.3.3 Traitement des informations temporelles

Les outils CADP ne sont pas prévus pour traiter des LTS qui comportent des informations temporelles. Si nous appliquons la minimisation par équivalence de branchement sur le RG sans tenir compte des régions (uniquement sur les transitions et la structure du

graphe), nous risquons de regrouper des nœuds qui correspondent à des régions complètement différentes.

Pour éviter cela, il faut joindre au graphe une partition des nœuds équivalents du point de vue de leurs régions. Une manière simple de réaliser ceci est d'adjoindre à chaque nœud du graphe une transition spéciale qui reboucle et qui est étiqueté par une marque spéciale caractéristique de la région de la classe d'équivalence (du point de vue des régions) à laquelle elle appartient. L'algorithme de minimisation ne pourra regrouper que les états qui appartiennent à des chemins équivalents (et qui donc passent par cette transition), et ainsi donc, ne regroupera que les nœuds qui correspondent à des régions équivalentes. La marque spéciale choisie est tout simplement la chaîne de caractères représentant les équations de la région sous forme canonique telle qu'elle est produite l'outil `rtl`.

B.3.4 Reconstruction des fonctions C et *theta*

L'outil `dtargmerge` présenté dans la section B.1 puise dans le `DTA` les fonctions $C = \{\ldots\}$ et $theta = \{\ldots\}$ et les ajoute aux transitions du `RG` en vue de la génération du `TLSA`. Le problème est que la phase de minimisation ajoute au `RG` des transitions qui n'existaient pas dans le système original, et donc dans le `DTA`. Il faut donc reconstruire ces fonctions.

Considérons les transitions $1 \xrightarrow{a} \boxed{2 \xrightarrow{i} 3} \xrightarrow{b} 4$. La phase de minimisation va regrouper les états 2 et 3 en un seul état. Supposons que nous décidions d'appeler ce nouvel état 2 (cette opération est réalisée par l'outil `bcg2rg`). Le graphe devient : $1 \xrightarrow{a} 2 \xrightarrow{b} 4$. L'outil `dtargmerge` ne peux trouver une transition correspondant à $2 \xrightarrow{b} 4$ dans le `DTA`. Il signale l'erreur, et va tenter de reconstruire une fonction C : si l'état père n'a pas d'horloge, alors la fonction C remet à zéro toutes les horloges de l'état fils, et si l'état fils a des horloges, alors une fonction *theta* peut être nécessaire mais nous ne pouvons pas la deviner, `dtargmerge` signale alors l'erreur.

Une amélioration à l'outil `bcg2rg` a été apportée : il inclut dorénavant les fonctionnalités de `dtargmerge`. L'avantage est que, du fait que `bcg2rg` connaît les classe d'équivalence, il peut choisir plus finement les fonctions C et *theta* dans le `DTA`. Dans l'exemple précédent, il portera sur la transition $2 \xrightarrow{b} 4$ les fonctions trouvées dans le `DTA` sur la transition $3 \xrightarrow{b} 4$, car il sait que 2 et 3 sont équivalents.

C

Spécification RT-Lotos engendrée depuis TURTLE

Cette annexe présente la spécification RT-LOTOS *de la machine à café n° 1 (sa spécification* TURTLE *est donnée dans la figure 3.11), engendrée par les algorithmes de traduction qui ont étés introduits dans la sous-section 3.1.3 et détaillés dans la section 3.2.*

```
specification VendingMachine : noexit

type natural is boolean
  sorts nat
  opns
    +  : nat,nat->nat
    >  : nat,nat->bool
    == : nat,nat->bool
endtype


behaviour

  hide g_1, g_2, g_3, g_4, g_5 in

  p_Wallet[g_1,g_1] (10)
  |[g_1,g_2]|
  (
    p_Button[g_4]
    |[g_4]|
    (
      p_CoffeeMachine[g_1,g_3,g_4,g_2] (150,200)
      |[g_3]|
      p_Button[g_3]
```

```
    )
  )

where

  process Wallet[putCoin,coinBack](amount:nat) : noexit :=
    [amount > 0] ->
      (
        ( putCoin!1; Wallet[putCoin,coinBack](amount-1) )
        []
        ( coinBack?x:nat; Wallet[putCoin,coinBack](amount+x) )
      )
    []
    [amount == 0] ->
      ( coinBack?x:nat; Wallet[putCoin,coinBack](amount+x) )
  endproc

  process p_Wallet[putCoin,coinBack](amount:nat) : noexit :=
    Wallet[putCoin,coinBack](amount)
  endproc


  process CoffeeMachine[coin_in,coffee,tea,ejectCoin]
                       (coinDelay:nat,buttonDelay:nat) : noexit :=
    coin_in!1;
    (
      coin_in{coinDelay}!1;
      (
        ( coffee{buttonDelay}; delay(100,175)
          CoffeeMachine[coin_in,coffee,tea,ejectCoin](coinDelay,buttonDelay) )
        []
        ( tea{buttonDelay}; delay(120,200)
          CoffeeMachine[coin_in,coffee,tea,ejectCoin](coinDelay,buttonDelay) )
        []
        ( delay(buttonDelay) ejectCoin!2;
          CoffeeMachine[coin_in,coffee,tea,ejectCoin](coinDelay,buttonDelay) )
      )
    )
    []
    ( delay(coinDelay) ejectCoin!1;
      CoffeeMachine[coin_in,coffee,tea,ejectCoin](coinDelay,buttonDelay) )
  endproc

  process p_CoffeeMachine[coin_in,coffee,tea,ejectCoin]
                         (coinDelay:nat,buttonDelay:nat) : noexit :=
    CoffeeMachine[coin_in,coffee,tea,ejectCoin](coinDelay,buttonDelay)
  endproc


  process Button[push] : noexit :=
    latency(230) push; delay(100) Button[push]
```

```
  endproc

  process p_Button[push] : noexit :=
    Button[push]
  endproc

endspec
```

Auteur : Christophe LOHR

Titre :

Contribution à la conception de systèmes temps-réel s'appuyant sur la technique de description formelle RT-LOTOS

Résumé :

Ce mémoire de thèse s'intéresse à la conception de systèmes temps-réel en s'appuyant sur la méthode formelle RT-LOTOS, extension temporelle à l'algèbre de processus LOTOS. Il aborde plusieurs points relatifs à la spécification, la validation et l'ordonnancement de systèmes concurrents sujets à des contraintes logiques et temporelles.

La première partie propose un éventail de méthodes formelles pour la spécification et la validation de systèmes temps-réel. Elle présente également le langage RT-LOTOS et la technique de vérification formelle associée basée sur une analyse d'accessibilité. Elle détaille finalement un ensemble de travaux concernant l'automate temporisé (appelé un `DTA`) dérivé d'une spécification RT-LOTOS, avec comme objectifs d'exécuter des simulations rapides, et de s'interfacer avec des outils de vérification de type *model-checker*.

La deuxième partie présente une étude sur la notion de cohérence temporelle et propose une technique ainsi qu'un modèle formel pour exploiter sous un angle nouveau des informations issues de la vérification formelle par analyse d'accessibilité. Cette approche propose de raffiner le graphe des régions, d'en élaguer certaines branches jugées non souhaitables, d'extraire les dates de tir possible des actions, et de présenter ces informations sous la forme d'un nouveau type d'automate temporisé (appelé un `TLSA`) ayant pour vocation l'ordonnancement dans le temps des actions d'un système.

Enfin, la troisième partie se penche sur les liens possibles entre méthodes formelles et semi-formelles. Dans ce cadre, nous proposons une sémantique formelle pour les diagrammes `UML` s'appuyant sur RT-LOTOS, après avoir défini une extension temps-réel à `UML` (appelée `TURTLE`). Ainsi, nous définissons une méthodologie qui s'inscrit dans les techniques de développement industriel classiques et qui permet une vérification formelle de systèmes temps-réel.

Discipline : Programmation et Systèmes

Mots-clés : systèmes temps-réel, méthodes formelles, RT-LOTOS, automates temporisés, cohérence temporelle, graphe d'accessibilité, `UML`, validation.

Author: Christophe LOHR

Title:

Contribution to the design of real-time systems based on the RT-LOTOS formal description technique

Abstract:

This thesis deals with the design of real-time systems based on the RT-LOTOS formal method, a timed extension to the LOTOS process algebra. It addresses several issues related to the specification, validation and scheduling of concurrent systems subject to logical and temporal constraints.

The first part of the work proposes a review of formal methods for the specification and validation of real-time systems. It also presents the RT-LOTOS language and the associated formal verification technique based on reachability analysis. Finally, it details some works based on the timed automaton (called a `DTA`) derived from an RT-LOTOS specification in order to carry out fast simulations, and to interface with model-checking tools.

The second part presents a study on the concept of temporal consistency and proposes both a technique and a formal model to exploit in a new way the information resulting from the formal reachability analysis. This approach proposes to refine the regions graph, to remove paths considered to be nondesirable, to extract the firing instants of the actions, and to present this information in a new model of timed automaton (called a `TLSA`) which is able to schedule the execution of the actions of a system.

Finally, the third part considers the possible relations between formal and nonformal methods. Within this framework, we propose a formal semantics for `UML` diagrams, after having defined a real-time extension to `UML` (called `TURTLE`). Thus, we define a methodology, which takes place inside traditional industrial development techniques and which allows a formal analysis of real-time systems.

Keywords: Real-Time Systems, Formal Methods, RT-LOTOS, Timed Automata, Temporal Consistency, Reachability Graph, `UML`, Validation

---

Laboratoire d'Analyse et d'Architecture des Systèmes, UPR 8001 du CNRS
7 avenue du Colonel Roche, 31077 Toulouse cedex