Etude d'un environnement de programmation et de vérification des systèmes réactifs, multi-langages et multi-outils

Muriel Jourdan

To cite this version:

Muriel Jourdan. Etude d'un environnement de programmation et de vérification des systèmes réactifs, multi-langages et multi-outils. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 1994. Français. NNT : . tel-00005099

HAL Id: tel-00005099
https://theses.hal.science/tel-00005099

Submitted on 25 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée par

Muriel JOURDAN

pour obtenir le grade de DOCTEUR
de l'UNIVERSITÉ JOSEPH FOURIER - GRENOBLE I
(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

(Spécialité : Informatique)

Etude d'un environnement de programmation et de vérification des systèmes réactifs, multi-langages et multi-outils

---

Date de soutenance : 29 septembre 1994

Composition du jury :		
	Président	J. Voiron
	Rapporteurs	G. Berry
		M. Sintzoff
	Examinateurs	A. Arnold
		F. Maraninchi
		D. Pilaud

Thèse préparée au sein du Laboratoire de Génie Informatique puis de l'Unité Mixte de Recherche Vérimag

Remerciements

Je tiens à remercier :

Monsieur *Jacques Voiron*Γ Professeur à l'Université Joseph FourierΓ pour m'avoir fait l'honneur de présider le jury de cette thèse ;

Messieurs *Gérard Berry*Γ Maître de Recherches à l'Ecole Nationale Supérieure des Mines de ParisΓet *Michel Sintzoff*ΓProfesseur à l'Université catholique de LouvainΓpour avoir accepté de juger ce travail et pour l'intérêt qu'ils lui ont accordé ;

Monsieur *André Arnold*ΓProfesseur à l'Université de Bordeaux IΓqui a accepté de participer à ce jury ;

Mademoiselle *Florence Maraninchi*ΓMaître de Conférences à l'Université Joseph FourierΓqui a dirigé cette thèse et qui par ses nombreux conseils a permis son aboutissement. J'aimerais plus simplement remercier Florence pour les trois années très riches en enseignementΓpassées à ses cotés.

Cette thèse a été réalisée dans le cadre d'une bourse doctorale de recherche de la région Rhône-Alpes établie sur la base d'un projet commun entre le projet SPECTRE et la société VERILOG. Je remercie Messieurs *Joseph Sifakis*Γdirecteur du projet SPECTREΓet *Daniel Pilaud*ΓResponsable du Centre d'Etudes VERILOG Rhône-AlpesΓqui m'ont accueillie et m'ont permis de mener à bien mon projet.

Je tiens également à remercier l'ensemble des membres du projet SPECTRE qui contribuent à faire de ce projet un cadre de travail privilégié. Plus particulièrementΓje remercie :

Les “équipes” LUSTRE : *N. Halbwachs, F. Lagnier, P. Raymond* et *C. Ratel*ΓKRONOS : *X. Nicollin, A. Olivero, J. Sifakis* et *S. Yovine* et ALDÉBARAN : *J. C. Fernandez, A. Kerbrat* et *L. Mounier*Γque j'ai fort souvent sollicitées ;

*Chantal Costes*Γ*Sabine Maury* et *Christine Servonnet*Γqui anime le secrétariat du projetΓ pour leur bonne humeur et leur disponibilité jamais mises en défaut ;

*Alain Girault*Γ*Alain Kerbrat* et *Claire Loiseaux* pour leur amitié et leur soutien précieux ;

Pour terminerΓj'associe à ses remerciements mes parents et mes deux frères pour la famille que nous formons ;

Et bien sûrΓ*Alain* ... pour tout.

Table des matières

Introduction **11**

I Le modèle automates booléens : vérification et programmation **21**

1 Le modèle automates booléens : vérification **23**

1.1 Les automates booléens modèles de systèmes réactifs 24

1.2 Vérification par simulation comportementale 26

1.2.1 Expression d'une propriété 26

1.2.2 Relation de satisfaction 27

1.2.3 Cas des points d'observation disjoints 28

1.2.4 Cas des ensembles de points d'observation non disjoints 31

1.3 Vérification à l'aide d'une logique temporelle 37

1.3.1 Expression d'une propriété 37

1.3.2 Relation de satisfaction 38

1.3.3 Points d'observation disjoints 40

1.3.4 Cas des ensembles de points d'observation non disjoints 42

1.4 Présentation des techniques de vérification développées autour du langage Esterel 43

1.4.1 La liaison Esterel-EMC 44

1.4.2 La liaison Esterel-Auto 45

1.5 Récapitulation 46

2 Argos : un langage impératif de composition d'automates **47**

2.1 Présentation informelle du langage Argos 47

2.1.1 Les automates . . . 48
2.1.2 La mise en parallèle . . . 49
2.1.3 La déclaration d'événements internes . . . 51
2.1.4 L'opérateur de raffinement . . . 55
2.2 Définition d'Argos . . . 60
2.2.1 Syntaxe abstraite des programmes Argos . . . 60
2.2.2 Eléments de syntaxe concrète . . . 61
2.2.3 Programmes principaux . . . 62
2.2.4 Calcul du modèle d'un programme . . . 63
2.2.5 Contraintes de l'environnement sur les entrées d'un programme . . . 68
2.2.6 Evolution d'Argos depuis sa définition initiale . . . 68
2.3 Un exemple de programmation et de vérification à partir du langage Argos . . . 69
2.3.1 Présentation de l'exemple . . . 69
2.3.2 La description en Argos . . . 70
2.3.3 Des exemples de preuves . . . 72
2.4 Récapitulation . . . 78

3 Deux extensions au langage Argos 79
3.1 Le langage Argos avec des variables de contrôle . . . 80
3.1.1 Présentation informelle . . . 80
3.1.2 Notations et définitions . . . 82
3.1.3 Définition d'Argos avec variables de contrôle . . . 83
3.2 Les états temporisés . . . 90
3.2.1 Présentation informelle . . . 90
3.2.2 Définition d'Argos temporisé . . . 93
3.2.3 Un exemple de programme Argos avec temporisation . . . 96
3.2.4 Les limitations de la macro-notation . . . 96
3.3 Récapitulation . . . 99

4 ArgoLus : un langage mixte impératif et déclaratif 101
4.1 Le langage déclaratif flots de données Lustre . . . 102
4.1.1 Présentation informelle du langage Lustre . . . 102

4.1.2 Exemples de programmes Lustre simples 104
4.1.3 Les horloges dans les programmes Lustre 105
4.1.4 L'appel de nœuds en Lustre 106
4.1.5 Contrôleur de feux de voiture 106
4.1.6 Les erreurs possibles dans un programme Lustre 107
4.1.7 La sémantique flots de données de Lustre 109
4.2 Une sémantique flots de données d'Argos 115
4.2.1 Syntaxe abstraite et sémantique statique 115
4.2.2 Sémantique flots de données d'Argos 115
4.2.3 Liens avec la définition formelle d'Argos par construction du modèle . . . 117
4.3 Le langage mixte ArgoLus 118
4.3.1 Les étapes de la définition d'ArgoLus 118
4.3.2 Lustre avec structures de bloc 120
4.3.3 Argos avec structures de bloc 121
4.3.4 Syntaxe et sémantique statique du langage ArgoLus 122
4.3.5 Sémantique flots de données du langage ArgoLus 124
4.4 Un exemple de programme en ArgoLus 125
4.4.1 Présentation du système à décrire 126
4.4.2 Interface avec l'environnement 127
4.4.3 Programmation en ArgoLus 127
4.4.4 Apports d'ArgoLus par rapport aux langages Lustre et Argos 131
4.5 La mise en œuvre du langage ArgoLus 131
4.5.1 La traduction structurelle d'Argos en Lustre 132
4.5.2 La traduction structurelle d'un programme ArgoLus en Lustre 142
4.5.3 La solution symétrique : traduction d'ArgoLus en Argos 146
4.6 Conclusion 147

II Le modèle graphes temporisés : vérification et programmation 149

5 Le modèle graphes temporisés : vérification 151

5.1 Les graphes temporisés 152

5.2 Les systèmes temporisés 153
5.2.1 Sémantique d'exécution des systèmes temporisés 154
5.2.2 Les systèmes bien temporisés 156
5.3 Sémantique des graphes temporisés en termes de systèmes temporisés 156
5.4 La vérification des systèmes temporisés 157
5.4.1 Principe général 157
5.4.2 La logique temporelle temps réel TCTL 159
5.5 Les graphes temporisés modèles de systèmes réactifs 160

6 Sémantique d'Argos en termes de graphes temporisés 163
6.1 Sémantique d'Argos en termes de graphes temporisés 164
6.1.1 Principe général 164
6.1.2 Exemples de programmes Argos temporisés avec leur modèle 166
6.1.3 Graphes temporisés modèles de programmes Argos 167
6.1.4 Nouvelles définitions du déterminisme et de la réactivité 169
6.1.5 Syntaxe abstraite et sémantique statique 170
6.1.6 Calcul du graphe temporisé modèle d'un programme Argos 171
6.2 Bonne temporisation des graphes temporisés décrits en Argos 174
6.2.1 Théorème 175
6.2.2 Lemmes 175
6.2.3 Les étapes de la démonstration du théorème 6.1 178
6.3 Vérification en temps continu et exécution en temps discret 182
6.3.1 Contexte et présentation de la problématique 182
6.3.2 Discrétisation des séquences d'exécution d'un système temporisé 184
6.3.3 Pas d'exécution concret et résultat de préservation 187
6.4 Un exemple d'application à la robotique 192
6.5 La modélisation en Argos d'une tâche-robot 193
6.6 La vérification de l'absence de blocage temporel 194
6.6.1 Expression de la propriété par une formule TCTL 194
6.6.2 Extensions 197
6.7 Conclusion 198

7 Réalisations et expérimentations 199

7.1 Réalisations 199

7.1.1 Le compilateur Argos 199

7.1.2 Algorithme symbolique de compilation du langage Argos 201

7.1.3 Adaptation de la méthode des potentiels du langage Esterel 205

7.1.4 Mise en œuvre de la sémantique d'Argos temporisé en termes de graphes temporisés 208

7.1.5 Autres extensions 209

7.2 Expérimentations 210

7.2.1 Aldébaran 210

7.2.2 Kronos 212

7.2.3 Lésar 214

Bilan et perspectives 217

7.3 Bilan 217

7.3.1 La définition du langage ArgoLus 217

7.3.2 La sémantique d'Argos en termes de graphes temporisés 218

7.3.3 Les extensions apportées au langage Argos et l'optimisation de son compilateur 220

7.3.4 Interactions entre ces différents travaux 220

7.4 Perspectives 221

7.4.1 Le développement d'un atelier de programmation synchrone et de vérification multi-outils et multi-langages 221

7.4.2 Valorisation de l'approche synchrone dans de nouveaux domaines 222

7.4.3 Argos : jeu de constructeurs de diverses formes d'automates 223

III Annexes 233

A La traduction structurelle d'Argos vers Lustre 235

B Preuve de la divergence des séquences types 239

C Preuve du théorème 6.2 241

Introduction

Dans ce travail nous nous sommes intéressés à une classe particulière de systèmes informatiquesΓ les *systèmes réactifs*. Plus précisémentΓnotre intérêt s'est porté d'une part sur leur programmation dans le cadre de *l'approche synchrone*Γd'autre part sur leur vérification formelle à partir de *modèle* représentant — de façon plus ou moins abstraite — l'ensemble des comportements possibles du système.

Ces deux centres d'intérêts sont reliés par la notion de modèle. En amontΓles *langages synchrones* offrent des formalismes de haut niveau pour décrire les modèles. En avalΓdes techniques de vérification formelle les analysent en tenant compte de la propriété à vérifier.

Différents langages synchrones existent. Ils diffèrent par leur style de programmation. De mêmeΓde nombreux outils de vérification formelle mettant en œuvre des méthodes différentes existent. Ils diffèrent par le type de propriétés qu'ils permettent de vérifierΓpar leurs performancesΓet par la qualité de leur diagnostic lorsqu'une propriété n'est pas satisfaite.

Notre objectif est d'utiliser cet existant pour définir un environnement de programmation et de vérification des systèmes réactifs *multi-langages* et *multi-outils*.

Nature des systèmes réactifs

Il a été proposé dans [HP85] une typologie des systèmes informatiques en trois classes :

- Les *systèmes transformationnels* sont les systèmes les plus classiques en informatique. Ils prennent un ensemble de données structurées au début de leur exécutionΓopèrent sur ces données un ensemble de transformations complexesΓet produisent à la fin de leur exécution un ensemble de résultats. Un compilateur en est un exemple type.

- Les *systèmes réactifs* interagissent continûment avec leur environnementΓsans maîtriser le rythme de ces interactionsΓen réagissant à des entrées par l'émission de sorties appropriées. Ces sorties dépendent des entrées à l'instant courantΓmais aussi potentiellement de toutes les interactions qui précèdent cet instant. L'absence de maîtrise par le système du rythme de ses interactions avec son environnement est une caractéristique fondamentale des systèmes réactifsΓcar elle s'accompagne de l'interdiction de perdre des entrées. La plupart des systèmes industriels dits "temps réel" — systèmes de contrôle-commandeΓ automatismesΓ... — sont des systèmes réactifs.

- Les *systèmes interactifs* sont proches des systèmes réactifs. Ils interagissent eux aussi continûment avec leur environnementΓmais cette fois à une vitesse qui leur est propre. Autrement ditΓils ont la possibilité de mémoriser certaines entrées avant de les prendre en compteΓsans que cela ait d'importantes conséquences sur le déroulement du système. Un système d'exploitation est un exemple de système interactif.

Comme la présentation ci-dessus le montreΓles systèmes réactifs sont soumis à des contraintes temporelles strictesΓconcernant à la fois leur rythme d'acquisition des entrées et leur temps de réponse. Il est par conséquent peu surprenant que leur programmation constitue un domaineΓ où l'efficacité du code produit a une importance considérable.

Les systèmes réactifs sont des systèmes à la fois *déterministes* et *parallèles*. Ils sont déterministes car leurs sorties sont complètement déterminées par les valeurs et les instants d'occurrence de leurs entrées. Il s'ensuit que le comportement d'un système réactif est reproductible. Ils sont parallèles car il est naturel de concevoir un système réactif comme un ensemble de composants s'exécutant en parallèle et coopérant à la réalisation du comportement global souhaité. Cette décomposition en activités parallèles n'implique pas forcément une implantation parallèle du système ; et même dans ce casΓle parallélisme d'implantation ne correspond pas nécessairement à la décomposition initiale. Ces caractéristiques permettent de comprendre pourquoi les langages parallèles de type ADA ne sont pas adaptés à la programmation des systèmes réactifs. En effetΓle parallélisme présent dans ces langages est un parallélisme d'implantation qui introduit du non-déterminisme. Une argumentation détaillée de cette inadéquation peut être trouvée dans [Ber89]. Des langages de haut niveau déterministes et intégrant la notion de parallélisme de description — par opposition au parallélisme d'implantation – doivent être utilisés pour spécifier et programmer les systèmes réactifs. L'approche synchrone permet de définir de tels langagesΓ les langages synchrones.

Vérification formelle des systèmes réactifs basée sur la notion de modèle

Les systèmes réactifs interviennent dans des domaines soumis à des contraintes de sûreté particulièrement sévères : nucléaireΓavioniqueΓ. . Par conséquentΓnon seulement les langages utilisés pour leur programmation doivent posséder une sémantique rigoureuseΓmais il s'avère aussi nécessaire de développer pour ces systèmesΓdes méthodes de vérification formelle.

Les propriétés intéressantes à vérifier sur un système réactif expriment essentiellement des relations d'ordre entre les occurrences d'entrées et de sorties du système. Par exempleΓsi le système réactif considéré est un contrôleur de passage à niveau automatiqueΓil est intéressant de vérifier que la barrière se ferme toujours après avoir reçu l'ordre de se baisser.

Nous nous intéressons aux méthodes de vérification basées sur l'analyse d'un modèle du système à vérifier. Ce modèle constitue une certaine abstraction de l'ensemble des comportements possibles du système. Plus précisément les modèles que nous considérons sont des *systèmes de transitions étiquetées* ou *automates* plus ou moins complexes.

De nombreuses formes d'automates peuvent être utilisées pour modéliser les systèmes réactifsΓde manière plus ou moins abstraite. La richesse et la nature de l'information portée par

les états et/ou les transitionsΓdéterminent l'expressivité du modèle et la possibilité d'analyser automatiquement des propriétés sur ce modèle.

Automates booléens

Par exempleΓdes automates qui ne comportent que des informations booléennes sur les transitions se prêtent bien à toutes sortes de techniques d'analyse. Théoriquement toutes les propriétés que l'on souhaite vérifier sont décidablesΓet des techniques comme les *graphes de décision binaires* permettent de les mettre en œuvre de manière efficace.

En contrepartieΓle pouvoir d'expression d'un tel modèle est assez faible et représente pour des systèmes réels — ne manipulant pas seulement des données booléennes — une assez forte abstraction de la réalité. CependantΓla satisfaction d'une certaine classe de propriétésΓles propriétés de *sûreté*Γsur le modèle booléen est préservée sur le système complet. Les propriétés de sûreté forment une classe de propriétés particulièrement intéressantes [Pnu92ΓHLR92]. InformellementΓelles permettent d'exprimer que le système n'effectue jamais quelque chose de mauvais. Une autre classe de propriétés existeΓles propriétés de *vivacité*. InformellementΓelles permettent d'exprimer que quelque chose de bon arrivera éventuellement ou inévitablement. Une caractérisation formelle de ces deux types de propriétés peut être trouvée dans [Lam77ΓAS86]. L'étude de la modélisation des systèmes réactifs en automates booléens présente donc un intérêt certain. Celui-ci est encore augmenté par le fait qu'une large classe de systèmes réactifs intéressants — par exemple les circuits matériels [Ber91] — peuvent être décrits en ne manipulant que des données booléennes. Dans ce casΓce type de modèle n'est plus une abstraction du système.

Les deux méthodes de vérification formelle basées sur l'analyse de ce type d'automateΓauxquelles nous nous intéressons dans cette thèseΓsont *la simulation comportementale* [Mil80ΓMil89] et *la vérification à l'aide de logique temporelle* [Pnu77ΓLam83]. Nous en donnons ici une présentation informelle.

Prouver une propriété par simulation comportementale consiste d'une part à exprimer la propriété sous forme d'un automateΓd'autre part à "comparer" les deux automates ainsi obtenus. La relation de comparaison — appelée relation de simulation — consiste à vérifier que tous les comportements du modèle sont contenus dans l'ensemble des comportements exprimés par la propriété. Plusieurs outils mettent en œuvre cette méthode de vérification formelleΓparmi lesquels on peut citer Auto [Ver87ΓdSV89]ΓAldébaran [Fer88ΓMou92]Γle Concurrency-WorkBench [CPS89]Γ... Cette méthode de vérification ne permet de vérifier que des propriétés de sûreté.

Prouver une propriété par une méthode à base de logique temporelle consiste d'une part à exprimer la propriété par une formule de cette logiqueΓd'autre part à calculer l'ensemble des états qui satisfont cette formule pour vérifier que l'état initial du modèle en fait partie. Plusieurs outils mettent en œuvre cette méthode de vérification formelleΓparmi lesquels on peut citer EMC [CES83]ΓXésar [RRSV87ΓRod88]. Cette méthode de vérification permet de vérifier des propriétés de sûretéΓmais aussi des propriétés de vivacité — ces dernières ne sont préservées sur le système réel que si le modèle n'est pas une abstraction trop forte de la réalité —.

L'utilisation des outils mettant en œuvre ces deux méthodes de vérification formelle se heurte en pratique à la même limiteΓconnue sous le nom *d'explosion du nombre d'états*. La raison de

cette explosion est liée au fait que l'unique moyen de mémorisation de l'histoire d'un système réactif dans un automate booléenΓest d'associer un état à cette histoire.

Automates interprétés

A l'opposé des automates booléens se trouvent les *automates interprétés*Γdont les transitions portent des conditions et des affectations à des variables de domaines quelconques. Les possibilités d'analyse automatique sont réduites. De nombreuses propriétésΓcomme l'accessibilité d'une configuration — état et valeurs des variables — sont indécidables. En revanche ces modèles sont assez riches pour servir d'intermédiaire entre le programme et le code exécutable. De plusΓil est toujours possible d'en extraire un *squelette de contrôle* booléen pour se ramener à la vérification sur un modèle booléen. Dans ce casΓseule la satisfaction des propriétés de sûreté est préservée sur l'automate interprété.

De nombreux travaux [AD90ΓACD90ΓHal93ΓHL92ΓFer93] portent sur la définition de modèles intermédiaires entre ces deux extrêmes. L'idée est de trouver un compromis entre pouvoir d'expressionΓcompacité de la représentation et décidabilité de propriétés intéressantes. Parmi ces formes intermédiairesΓnous nous sommes intéressés aux *graphes temporisés* [ACD90].

Graphes temporisés

Les graphes temporisés sont des automates contrôlés par des variablesΓétudiés pour modéliser des systèmes fortement contraints par l'évolution du temps. Les variablesΓappelées *horloges*Γsont utilisées pour mémoriser les informations relatives au temps dans le système. Cette "spécialisation" des variables permet de leur associer un jeu d'actions réduitΓqui rend possible l'élaboration de méthodes de vérification formelle de propriétés intéressantesΓdont les propriétés *quantitatives temporelles temps réel*. Dans cette caractérisationΓ*temporelle* fait référence à l'ordre d'exécution des actions du systèmeΓalors que *temps réel* fait référence à l'écoulement du temps entre ces actions. Si l'on reprend le contrôleur de passage à niveau automatiqueΓde telles propriétés permettent par exemple d'exprimer que la barrière est toujours ferméeΓune minute après avoir reçu l'ordre de se baisser.

Le comportement d'un système réactif peut dépendre fortement de l'évolution du temps. Dans un tel casΓil est intéressant d'utiliser les graphes temporisés comme modèles. En effetΓd'une part ce modèle permet de vérifier une forme de propriétés intéressantesΓd'autre part il est moins sensible que les automates booléens au phénomène d'explosion du nombre d'états. En effetΓune des causes de ce phénomène est la présence dans les programmes de structures de contrôle à base de compteursΓpour exprimer des durées. Dans un automate booléenΓles valeurs de ces compteurs sont mémorisées par des étatsΓalors que dans un graphe temporiséΓles variables sont utilisées à cet effet. Il s'ensuit que les limites pratiques d'utilisation des outils de vérification basés sur les automates booléens sont repoussées par ceux associés aux graphes temporisés.

Sur ces graphes temporisésΓdes techniques de vérification à base de logique temporelle temps réel ont été définies [ACD90]. Le système KRONOS [Yov93] est un outil développé à partir d'un algorithme symbolique [HNSY92ΓNic92] qui met en œuvre cette technique de vérification.

Différentes formes de modèles viennent d'être présentées. Elles sont plus ou moins expres-

sivesΓcompactes et intéressantes pour des aspects de vérification. Nous présentons à présent les langages synchrones qui peuvent être utilisés pour décrire ces modèles.

Approche synchrone

Présentation générale

L'approche synchrone est fondée sur l'hypothèse d'exécution en temps nul des systèmes réactifs. Cette hypothèse peut être considérée de deux points de vue :

- **du point de vue externe**Γelle permet d'envisager l'exécution d'un système réactif sous la forme d'une boucle infinie dont le corps est constitué des trois phases suivantes :

  1. Lecture des entrées ;
  2. Calcul des sorties ;
  3. Ecriture des sorties ;

  et dans laquelle la phase de calcul des sorties s'exécute idéalement en temps nul. Cette hypothèse constitue de manière évidente une approximation de la réalité. Cette approximation est valide si l'exécution de cette phase de calcul est suffisamment rapide pour ne pas perdre des entrées. Il est par conséquent nécessaireΓpour valider l'hypothèse de synchronismeΓde savoir prédire le temps d'exécution de la phase de calcul des sortiesΓafin d'assurer qu'il est toujours plus petit que le délai minimal séparant les entrées du système ;

- **du point de vue interne**Γelle permet de décomposer un système réactif en un ensemble de sous-systèmesΓqui s'exécutent en parallèle et communiquent en temps nul. Cette possibilité permet de définir des langages de haut niveauΓdont les sémantiques sont compositionnelles : les *langages synchrones*. IciΓl'hypothèse d'exécution en temps nul n'est pas une approximation de la réalitéΓcar la décomposition parallèle et les communications sont purement statiques et sont calculées lors de la compilation du programme.

Si l'ensemble des comportements possibles d'un système réactifΓc'est-à-dire la fonction qui à une séquence d'entrées associe une séquence de sortiesΓpeut être représentée par un *modèle* finiΓalors la vérification formelle des systèmes réactifs fondée sur l'analyse de ce modèle devient envisageable. Ces preuves effectuées sur le modèle du système ne sont intéressantes que s'il est possible de prouver leur préservation lors de l'exécution du système. Pour celaΓil est important que le formalisme utilisé en tant que modèle soit proche d'une forme exécutable. De plusΓla modélisation de l'ensemble des comportements d'un système réactif par un formalisme proche d'une forme exécutableΓpermet de répondre au problème de prédiction du temps de calcul des sorties.

Les modèles à base d'automates que nous venons de présenter satisfont cette exigence. En effetΓla traduction d'un automate dans un langage impératif à l'aide de variables et de "`Goto`" ne présente aucune difficultéΓpeut être automatisée et prouvée correcte. Par conséquentΓle principe du "What You Prove Is What You Execute" [Ber89] est satisfait.

La sémantique des langages synchrones est définie par la fonction qui associe à tout programmeΓle modèle qu'il décrit. Les compilateurs de ces langages mettent en œuvre cette association. Des outils de vérification formelle sont utilisés sur ces modèles et des traducteurs sont chargés de traduire ces modèles en code exécutable.

Langages synchrones

Idées communes aux langages synchrones

Les systèmes réactifs se décomposent de façon naturelle en sous-systèmes qui évoluent en parallèle et communiquent entre euxΓafin d'obtenir le comportement global souhaité. Les langages synchrones permettent de structurer un programme en respectant cette décomposition naturelle en composantes parallèles. CependantΓil est important de noter que cette décomposition ne reflète pas nécessairement une possible répartition sur une architecture parallèle. Le problème de la répartition de code pour les systèmes réactifs dans le cadre de l'approche synchrone est étudié dans [GMP+90ΓMaf93ΓGir94].

Toutes ces composantes s'exécutent et communiquent entre elles en temps nul. Le mode de communication commun aux langages synchrones est la diffusion synchrone [Ber89]. Lorsqu'un message est émis par une composanteΓtoutes les composantes peuvent le lire et ont au même instant la même vision du message. La validité de l'hypothèse de communication en temps nulΓtient dans le fait que ces communications n'ont aucune réalité physique : elles sont calculées statiquement lors du calcul du modèle associé à un programme.

EnfinΓtous les langages synchrones partagent la possibilité de vérifier des propriétés de sûreté par la *méthode des observateurs* [HLR93]. Le principe de cette méthode consiste à placer en parallèle avec le programme un observateur — une composante — quiΓen fonction des entrées et des sorties du programmeΓdécide si le comportement de ce dernier est correct. Le programme global — constitué de l'observateur et du programme à observer — émet une sortie particulière lorsque le comportement du programme est incorrect. Ce principe de vérification est valide si et seulement si l'observateur ne modifie pas le programme qu'il observe. Le mode de communication par diffusion synchroneΓutilisé dans les langages synchronesΓpermet d'ajouter des composantes qui n'influencent pas le comportement du reste du programmeΓet donc de décrire des observateurs.

Un des avantages de la méthode de vérification par observateurs est qu'il est possible d'utiliser le compilateur du langage comme outil de vérificationΓmême si celui-ci est sûrement moins performant qu'un outil spécifique qui aurait pour seul rôle de mettre en œuvre la méthode. Un tel outil spécifique est disponible dans l'environnement d'un seul des langages synchrones que nous allons présenter. Il se nomme Lésar [Rat92] et permet de vérifier des programmes Lustre par la méthode des observateurs.

Principaux langages synchrones

Le plus ancien des langages synchrones est le langage *Esterel* [BCG87ΓBS91]. Il a été défini par l'équipe de G. Berry à Sophia-AntipolisΓéquipe commune au Centre de Mathématiques Appliquées de l'Ecole des Mines et à l'INRIA. C'est un langage impératif textuel qui contient à la fois des structures de contrôle classiques et des structures spécifiques aux systèmes réactifs.

Les entrées et sorties d'un programme Esterel peuvent être accompagnées d'une valeur de type quelconque. La sémantique des programmes Esterel est définie en termes d'automates interprétés.

Le formalisme des *Statecharts* [Har87] est un formalisme impératif de description des systèmes réactifs défini par D. Harel et A. Pnueli. Il est fondé sur l'utilisation d'automates parallèles et hiérarchisés. Différentes versions de la sémantique des Statecharts existent [HPSS86ΓPS91]. Le principal reproche couramment fait aux Statecharts est la présence de transitions inter-niveaux dues à la sémantique de la décomposition hiérarchique. Ces transitions inter-niveaux sont à l'origine des difficultés pour exprimer la sémantique des Statecharts de manière compositionnelle.

Pour remédier au manque de cohérence des sémantiques des StatechartsΓ et résoudre les problèmes liés à la décomposition hiérarchiqueΓ le langage *Argos* [Mar90ΓMar92] a été défini à Grenoble par F. Maraninchi. Il propose un jeu d'opérateurs sur les automates pour définir structurellement des automates complexes. Ce langage est le plus récent des langages synchrones. Il ne permet de décrire que des systèmes réactifs dont les entrées et sorties sont booléennes. La sémantique d'Argos est définie en termes d'automates booléens.

EnfinΓ deux langages synchrones de style déclaratif ont été définisΓ *Signal* [GBBG85Γ GGaCLM91] et *Lustre* [CHPP87ΓHCRP91bΓHCRP91a]. Le modèle informatique sous-jacent à ces deux langages est le modèle "flots de données" [Kah74]. Dans ces deux langagesΓun programme est un réseau d'opérateurs agissant en parallèle au rythme de leurs entrées. De plusΓ les opérateurs sont des primitives idéales qui réagissent en temps nul. Le langage Signal a été défini à l'IRISAΓdans l'équipe de A. Benveniste et P. Le Guernic. Le langage Lustre a été défini à Grenoble dans l'équipe de P. Caspi et N. Halbwachs. Les entrées et sorties des programmes de ces langages sont de type quelconque. Les sémantiques de ces deux langages sont définies en termes d'automates interprétés.

Chacun de ces langages de programmation possède un environnement de programmation qui lui est spécifique. ToutefoisΓgrâce à l'existence d'un format commun d'expression des automates interprétés — format OC [Pa93] — certains outils sont partagés.

Objet de la thèse

Nous venons de présenter différentes formes de modèlesΓleurs intérêts par rapport à des aspects de vérification formelle et de génération de codeΓ ainsi qu'une classe de langages pouvant être utilisés pour les décrire. Cette présentation constitue en quelque sorte le contexte initial de notre travail. Une analyse de ce contexte permet de mettre en évidence un certain nombre de besoinsΓ que nous présentons ci-dessous. Ceux-ci ont motivé les différents axes de notre travail.

- Homogénéisation des environnements de programmation des langages synchrones pour que ceux-ci interviennent le moins possibleΓ dans le choix du langage pour programmer un système réactif particulier. Cette homogénéisation est un des objectifs du projet Synchrone [Pa93] qui vise à définir un socle commun de formats intermédiaires des environnements des différents langages synchronesΓpour partager le plus d'outils possibles. Une autre solution pour atteindre cette homogénéisation consiste à étudier les traductions structurelles entre les différents langages.

- Définition d'un formalisme permettant de mélanger au niveau source plusieurs langages synchrones. L'existence de langages synchrones déclaratifs et impératifs permet une réponse adaptée à chaque système réactifΓainsi qu'à la culture des programmeurs. CependantΓil n'existe pas de possibilité de faire cohabiter ces styles au sein d'un seul et unique programmeΓalors que ce besoin en langage mixte est reconnu par la communauté des langages synchrones [BB91]. Il n'y a en effet aucune raison pour qu'un même style soit adapté à la description de toutes les composantes d'un système réactif.

- Connexion au plus grand nombre d'outils de vérification formelle. Nous avons vu qu'un nombre important d'outils de vérification formelle existentΓd'une part sur les automates booléensΓmais aussi sur les graphes temporisés. Il est important de pouvoir utiliser à partir des systèmes réactifs la plus large palette possible d'outilsΓcar ils n'offrent pas tous le même type de fonctionnalités. Par exempleΓcertains permettent de prouver des propriétés de vivacité et/ou des propriétés quantitatives temporelles temps-réel. Ils diffèrent aussi par le mode d'expression des propriétésΓpar leurs performances et enfin par la qualité du diagnostic effectué en cas de non satisfaction d'une propriété. Pour atteindre cet objectif :
  - Un premier constat est évident : il n'existe aucun langage synchrone dont la sémantique est exprimée en termes de graphes temporisés. Par conséquentΓles possibilités très intéressantes de vérification formelle offertes par ce modèle ne sont pas exploitées dans l'approche synchrone ;
  - Un deuxième constat apparaît lors d'expériences réelles de vérification formelle de systèmes réactifs modélisés par des automates booléens. La plupart des outils de vérification — soit par simulation comportementaleΓsoit par utilisation d'une logique temporelle — s'avèrent en pratique peu adaptés à la vérification des automates booléens modèles de systèmes réactifs ; leur utilisation nécessite le recours à des transformations "ad hoc" du modèleΓpeu satisfaisantes dans une démarche de vérification formelle.

Contribution

Le travail effectué dans cette thèse constitue des éléments de réponse à ces différents besoinsΓpour la programmation et la vérification des systèmes réactifs ayant des entrées et sorties booléennes :

- Définition des traductions structurelles entre Argos — langage impératif à base d'automates — et Lustre — langage déclaratif flots de données —. Ces traductions permettent en particulier de mettre en évidence les spécificités de ces deux langages. De plusΓla traduction d'Argos en Lustre montre comment des structures de contrôle impérativesΓcomme la composition hiérarchiqueΓse traduisent sous forme déclarative. EnfinΓl'intérêt de cette traduction est qu'elle est suffisamment simple pour qu'il soit possible de retrouver dans le programme LustreΓun nombre important d'informations pertinentes du programme Argos initial. Cette possibilité d'effectuer de "la remontée dans le source" est particulièrement intéressante pour utiliser les outils travaillant directement sur les programmes LustreΓcomme par exemple le simulateur qui permet d'exécuter pas à pas un programme.

- Définition d'un langage mixte impératif/déclaratifΓfondé sur les langages Argos et Lustre. Ce nouveau langage se nomme ArgoLus. Il permet de décrire un système soit comme un

programme Argos contenant des composantes décrites en LustreΓsoit inversement comme un programme Lustre contenant des composantes décrites en Argos. Ce mélange permet en particulier d'introduire une structure de contrôle hiérarchique dans le langage LustreΓ qui facilite l'expression de comportements constitués d'un enchaînement de phases. C'est en recherchant des solutions pour mettre en œuvre le langage ArgoLusΓque les traductions structurelles entre Argos et Lustre ont été définies. En effetΓnous avons choisi de traduire tout programme ArgoLus soit en un programme LustreΓsoit en un programme ArgosΓpour pouvoir profiter des environnements — en particulier des compilateurs — associés à ces langages.

- Connexion au plus grand nombre d'outils de vérification formelle. Cet axe de travail se décompose en deux parties :
  - Extension d'Argos et expression de la sémantique du langage étendu en termes de graphes temporisés. Le fait d'atteindre cette nouvelle forme de modèles à partir d'un langage synchroneΓpermet non seulement de repousser dans certaines situations les limites d'utilisation des outils basés sur les automates booléens ; mais aussiΓde prouver d'autres formes de propriétés intéressantes : les propriétés quantitatives temporelles temps réel ;
  - Définition d'un cadre formel pour l'utilisation d'outils classiques de vérification formelle — de type simulation comportementale ou à base de logique temporelle — pour les systèmes réactifs. Une présentation de ces méthodes de vérification plus générale que leur présentation usuelleΓpermet de mettre en évidence le fait que la plupart des outils implémentent en réalité un cas particulier. Une fois celui-ci identifié nous définissons une méthode systématique de transformation de la propriété à vérifier de manière à s'y ramener. Cette transformation permet donc d'utiliser les outils existantsΓsans les modifier.

Organisation du document

La suite de ce document est organisée en deux partiesΓselon le type de modèle considéré :

- La partie I est dédiée à la vérification et à la programmation des automates booléens modèles de systèmes réactifs ;
- La partie II est dédiée à la vérification et à la programmation des graphes temporisés modèles de systèmes réactifs.

Les deux parties sont construites selon un plan commun :

- Leur premier chapitre présente la définition du modèle et les méthodes de vérification qui l'accompagnent ;
- Les chapitres suivants développent essentiellement des considérations sur les langages de programmation qui permettent de décrire le modèle.

Ces deux parties sont complétées par un chapitre qui présente le travail de réalisation et d'expérimentation effectué au cours de cette thèse.

Nous présentons à présent le contenu des différents chapitres.

- **Partie I : le modèle automates booléens : vérification et programmation**
  - **Le modèle automates booléens : vérification**

    Dans ce chapitreΓsont présentées en détail les méthodes de vérification par simulation comportementale et par utilisation d'une logique temporelle. Le cas particulier mis en œuvre par la majorité des outils existants est présentéΓainsi que la transformation systématique de la propriété pour s'y ramener.
  - **Le langage Argos**

    Le langage Argos est utilisé pour décrire des automates booléens complexes de manière compositionnelle. Ce chapitre contient une présentation informelle du langageΓsa définition formelle et se termine par un exemple de programmation. Sur cet exempleΓ nous présentons une application possible des résultats du chapitre précédent.
  - **Deux extensions au langage Argos**

    Ces extensions sont le résultat des divers exemples de systèmes réactifs programmés en Argos. Ce sont des "macro-notations" puisqu'il est possible de les traduire en terme du langage Argos initial. Par conséquent les modèles des programmes exprimables en Argos "étendu" sont toujours des automates booléens.
  - **La définition du langage ArgoLus**

    Nous présentons dans ce chapitre le langage mixte ArgoLus. C'est dans ce chapitre que sont présentés le langage Lustre et les traductions structurelles entre Argos et Lustre.
- **Partie II : le modèle graphes temporisés : vérification et programmation**
  - **Le modèle graphes temporisés : vérification**

    Ce chapitre présente la deuxième forme de modèle des systèmes réactifs à laquelle nous nous sommes intéressés. Une méthode de vérification à base de logique temporelle temps réel est présentée.
  - **La sémantique d'Argos en termes de graphes temporisés**

    Dans ce chapitreΓle langage Argos est vuΓnon plus comme un moyen de décrire des automates booléensΓmais comme un moyen de décrire des graphes temporisés complexes. Il contient la nouvelle définition d'ArgosΓdeux résultats théoriques importants pour des aspects de vérification et un exemple d'application sur un problème de robotique.
- **Réalisations et expérimentations**

  Ce chapitre présente les réalisations et les expérimentations qui accompagnent les différents travaux effectués au cours de cette thèse.

Partie I

Le modèle automates booléens : vérification et programmation

Chapitre 1

Le modèle automates booléens : vérification

Dans toute cette première partieΓles systèmes réactifs sont modélisés par des automates booléens. Nous précisons dans ce chapitre la forme de ces automates modèles de systèmes réactifs. D'autres choix concernant cette notion — automates booléens modèles de systèmes réactifs — peuvent être effectués. Nous nous intéressons aussi aux possibilités de vérification qui sont associées à ce modèle. Pour satisfaire à notre objectif d'élaboration d'un environnement multi-outilsΓ nous répondons au problème concret suivant : comment utiliser les outils de vérification fondés soit sur la *simulation comportementale* [Mil80ΓMil89]Γsoit sur l'utilisation d'une *logique temporelle* [Pnu77ΓLam83]Γpour la vérification des automates booléens modèles de systèmes réactifs.

L'application de ces méthodes aux automates modèles de systèmes réactifsΓen raison principalement de la forme des étiquettes de leurs transitionsΓn'est pas immédiate. Pour répondre à cette questionΓnous présentons ces méthodes d'une manière originaleΓplus générale que leurs présentations usuelles. C'est cette généralisation qui nous permet d'identifier le cas particulier implémenté par de nombreux outils.

La solution que nous proposons en réponse à ce problème d'inadéquation consiste à se ramener systématiquement à ce cas particulierΓpar une transformation de la propriété à vérifier. Cette transformation n'est pas particulière aux automates booléens modèles de systèmes réactifsΓmais peut s'appliquer à d'autres formes d'automates.

Le plan suivi dans ce chapitre est le suivant :

- la section 1.1 présente notre définition des automates booléens modèles de systèmes réactifs ;
- les sections 1.2 et 1.3 présentent les deux méthodes de vérification sur ces automatesΓ auxquelles nous nous sommes intéressés. Elles se différencient par la forme d'expression des propriétés et par la relation de satisfaction qui détermine si l'automate modèle d'un système satisfait ou non une propriété ;
- la section 1.4 présente un travail proche de celui présenté dans ce chapitre pour la vérification des programmes Esterel.

Bien que les méthodes de vérification soient présentées sans particulariser le type de systèmes

modélisés sous forme d'automatesΓles modèles de systèmes réactifs nous permettent d'illustrer les présentations de ces méthodes. Ceci explique pourquoi nous présentons les automates booléens modèles de systèmes réactifsΓavant de présenter les méthodes de vérification.

1.1 Les automates booléens modèles de systèmes réactifs

Nous commençons par formaliser la notion d'automate sans préjuger de la forme des étiquettes.

Définition 1.1 Les systèmes de transitions étiquetées ou automates
Soit $\mathcal{L}$ un ensemble d'étiquettes, un système de transitions étiquetées S est défini par un triplet (Q, q_{init}, T), où :

- *Q désigne l'ensemble des états de S,*
- *$q_{init} \subseteq Q$ est l'état initial de S,*
- *$T \subseteq Q \times \mathcal{L} \times Q$ est l'ensemble des transitions de S.*

■

Par la suiteΓnous nous intéressons uniquement aux automates ayant un seul état initial. Par conséquentΓnous pouvons remplacer dans la définition ci-dessus $q_{init} \subseteq Q$ par $q_{init} \in Q$.

L'utilisation des automates en tant que modèles des systèmes réactifs dont les entrées et sorties sont purement booléennes est basée sur les principes suivants :

- Les états de l'automate mémorisent l'histoire du système nécessaire pour calculer ses réactions futures.
- Les transitions modélisent une réaction du systèmeΓen précisant :
  - Son état courantΓc'est l'état source de la transition ;
  - Son interaction avec l'environnementΓc'est l'étiquette de la transition. Elle précise la condition sur les entrées qui déclenche la réactionΓet l'ensemble des sorties émises par le système lors de la réaction ;
  - L'état atteint après cette réactionΓc'est l'état but de la transition.

Nous choisissons de représenter les étiquettes des automates modèles de systèmes réactifs par une structure à deux parties :

- Un *monôme d'entrées* exprime la condition booléenne sur le statut des événements dans l'environnement qui constitue la condition de déclenchement d'une transition ;
- Un ensemble de sorties exprime la réaction du système lorsqu'une transition est déclenchée.

Définition 1.2 Monômes sur un ensemble E
L'ensemble des monômes définis sur un ensemble E est noté $\mathcal{M}(E)$. Cet ensemble est isomorphe à $2^E \times 2^E$. Soit m_e un élément de $\mathcal{M}(E)$, m_e est défini par un couple de sous-ensembles de E que l'on note $({m_e}^+, {m_e}^-)$.

- *m_e est* non contradictoire *lorsque ${m_e}^+$ et ${m_e}^-$ sont disjoints. $\mathcal{M}^\star(E)$ est l'ensemble des monômes non contradictoires sur E.*
- *m_e est* complet *par rapport à l'ensemble E lorsque l'union de ${m_e}^+$ et de ${m_e}^-$ est égale à l'ensemble E. L'ensemble des monômes de ce type sur l'ensemble E est noté $\mathcal{M}_c(E)$.*

L'ensemble des monômes complets et non contradictoires sur un ensemble E est noté $\mathcal{M}_c^\star(E)$.

■

Par la suite nous notons un monôme d'entrées $m_e = ({m_e}^+, {m_e}^-)$ en séparant tous ses éléments par des points et en surlignant les éléments qui font partie de ${m_e}^-$. Par exempleΓle monôme $(\{e1, e3\}, \{e2\})$ est noté $e1.e3.\overline{e2}$. La condition de déclenchement égale à ce monôme signifie que la transition est déclenchée si et seulement si les entrées $e1$ et $e3$ sont présentes dans l'environnement et $e2$ y est absent.

Un étiquette est notée m_e/OΓoù O est un ensemble de sorties. L'ensemble des automates ayant cette forme d'étiquettes est noté $\mathcal{A}$. Ces automates ne manipulant que des données booléennesΓnous les appelons automates booléens.

Nous définissons les fonctions qui associent à un élément de $\mathcal{A}$ l'ensemble de ses entréesΓnotée InΓet l'ensemble de ses sortiesΓnotée Out.

Définition 1.3 Ensembles des entrées et des sorties d'un automate
Soit $(Q, q_{init}, T) \in \mathcal{A}$, nous définissons :

- $In((Q, q_{init}, T)) = \{\alpha | \exists (q, m_e/O, q') \in T \wedge \alpha \in {m_e}^+ \cup {m_e}^-\}$;
- $Out((Q, q_{init}, T)) = \{\alpha | \exists (q, m_e/O, q') \in T \wedge \alpha \in O\}$.

■

Par la suiteΓlorsque le contexte le permet — il n'y a pas d'ambiguïté sur l'automate désigné — on utilise In et Out sans paramètre.

Nous définissons à présent différentes propriétés sur les éléments de $\mathcal{A}$.

Définition 1.4 Non recouvrement des ensembles d'entrées et de sorties
Un automate $(Q, q_{init}, T) \in \mathcal{A}$ satisfait la propriété de non recouvrement des entrées et sorties si et seulement si $In \cap Out = \emptyset$. ■

L'ensemble des éléments de $\mathcal{A}$ qui satisfont la propriété de non recouvrement des ensembles d'entrées et de sorties est noté $\mathcal{A}^\emptyset$.

Définition 1.5 Complétude et non contradiction des monômes
Un automate $(Q, q_{init}, T) \in \mathcal{A}$ *satisfait la propriété de complétude — sur l'ensemble des entrées de l'automate — et de non contradiction des monômes qui apparaissent sur ces transitions si et seulement si* $T \subseteq Q \times \mathcal{M}_c^\star(In) \times 2^{Out} \times Q$. ■

L'ensemble des éléments de $\mathcal{A}$ qui satisfont la propriété de complétude et de non contradiction des monômes est noté $\mathcal{A}^c$.

Définition 1.6 Déterminisme
Soit $(Q, q_{init}, T) \in \mathcal{A}$*, cet automate est déterministe si et seulement si*

$$\forall q \in Q, \forall m_e \in \mathcal{M}_c^\star(In), |\{(q, m_e/O, q') \in T\}| \leq 1$$ ■

Définition 1.7 Réactivité
Soit $(Q, q_{init}, T) \in \mathcal{A}$*, cet automate est réactif si et seulement si*

$$\forall q \in Q, \forall m_e \in \mathcal{M}_c^\star(In), |\{(q, m_e/O, q') \in T\}| \geq 1$$ ■

L'ensemble des éléments de $\mathcal{A}$ déterministes (resp. réactifs) est noté $\mathcal{A}_d$ (resp. $\mathcal{A}_r$).

Ces quatre propriétés peuvent être combinées. Par exempleΓl'ensemble $\mathcal{A}_{dr}^{\emptyset,c}$ est l'ensemble des éléments de $\mathcal{A}$ qui les satisfont toutes.

Définition 1.8 Automate booléen modèle de système réactif
Un automate booléen modèle de système réactif *est un élément de* $\mathcal{A}$ *qui vérifie les quatre propriétés ci-dessus. Par conséquent, c'est un élément de* $\mathcal{A}_{dr}^{\emptyset,c}$. ■

1.2 Vérification par simulation comportementale

Nous présentons à présent la méthode de vérification par simulation comportementale. Nous nous plaçons dans un cadre plus général que celui des systèmes réactifs. Par conséquentΓdans toute cette présentation nous utilisons un ensemble d'étiquettes nommé $\mathcal{L}$ quelconque.

Vérifier une propriété par simulation comportementale consiste à traduire la propriété sous forme d'un automate et à comparer l'automate modèle du programme avec celui de la propriété. IntuitivementΓl'automate de la propriété contient tous les "bons" comportements possibles du programme vis à vis de cette propriété ; la relation de comparaison vérifie que chaque réaction du programme a un équivalent possible dans la propriété.

1.2.1 Expression d'une propriété

Toute propriété est construite en faisant référence à *des points d'observation* des réactions de l'automate modèle. Ces points d'observationΓnotés ωΓsont des prédicats sur l'ensemble des

étiquettes de l'automate. Pour un ensemble d'étiquettes $\mathcal{L}$Γ nous notons $\Omega_{\mathcal{L}}$ l'ensemble des points d'observation de $\mathcal{L}$.

Dans toute la suite du chapitreΓnous notons $A_{\mathcal{L}}$ un automate construit sur l'ensemble $\mathcal{L}$ d'étiquettes.

Étant donné l'ensemble $\mathcal{L}$ des étiquettes d'un automateΓune propriété sous forme comportementaleΓest un automate dont les transitions sont étiquetées par des points d'observation de $\mathcal{L}$. Elle est notée $\varphi_{\mathcal{L}}$Γet $\pi(\varphi_{\mathcal{L}})$ dénote l'ensemble des points d'observation utilisés dans $\varphi_{\mathcal{L}}$.

Définition 1.9 Propriété sous forme comportementale
Soit $A_{\mathcal{L}}$ un automate, $\varphi_{\mathcal{L}} = (Q, q_{init}, T)$ exprime une propriété de $A_{\mathcal{L}}$ si et seulement si $T \subseteq Q \times \Omega_{\mathcal{L}} \times Q$. ■

1.2.2 Relation de satisfaction

InformellementΓnous appelons séquence d'exécution d'un automate un cheminΓc'est-à-dire une alternance d'états et d'étiquettesΓissu de l'état initial. Une propriété $\varphi_{\mathcal{L}}$ est satisfaite par un automate $A_{\mathcal{L}}$Γce qui se note $A_{\mathcal{L}} \models_c \varphi_{\mathcal{L}}$Γsi et seulement si à chaque séquence d'exécution s de l'automate (étiquetée par des éléments de $\mathcal{L}$) il est possible d'associer une séquence d'exécution s' de la propriété (étiquetée par des points d'observation) telle qu'à chaque pasΓl'étiquette de s satisfait le point d'observation correspondant dans s'.

Définition 1.10 Relation de satisfaction
Soient $A_{\mathcal{L}} = (Q, q_{init}, T)$ et $\varphi_{\mathcal{L}} = (Q', q'_{init}, T')$, $A_{\mathcal{L}} \models_c \varphi_{\mathcal{L}}$ si et seulement si

$\exists \mathcal{R}el \subseteq Q \times Q'$ telle que

- $(q_{init}, q'_{init}) \in \mathcal{R}el$,
- $\forall (q_1, q'_1) \in \mathcal{R}el$,
  $(q_1, l, q_2) \in T \Rightarrow \exists (q'_1, \omega, q'_2) \in T'$ *telle que* $\omega(l)$ *et* $(q_2, q'_2) \in \mathcal{R}el$.

■

Cette définition montre que la propriété doit contenir toutes les séquences d'exécution qui expriment un “bon comportement” du système. Nous pouvons donc la considérer comme un filtre qui rejette un système dès qu'il peut avoir un “mauvais comportement”.

Illustrons cette définition par un exemple de preuve sur les systèmes réactifs. Nous souhaitons prouver qu'un système réactif n'émet jamais la sortie o. Pour cela nous utilisons un point d'observation qui est satisfait par une étiquette de la forme m_e/O si et seulement si o n'appartient pas à l'ensemble de sorties O. Nous notons ce point d'observation $\omega_{\overline{o}}$. Cette propriété est l'automate de la figure 1.1. La flèche qui pointe sur l'unique état de l'automate indique qu'il est l'état initial. Le système ne vérifie pas cette propriété si et seulement si parmi l'ensemble de tous ses comportementsΓun au moins est tel que la sortie o est émise.

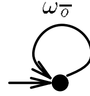


Figure 1.1: automate exprimant l'absence d'émission de la sortie o dans un système réactif

Toute propriété n'est pas exprimable sous forme d'un automate. Par exempleΓil est impossible de prouver par simulation comportementale qu'un système réactif est toujours capable dans ses réactions futures d'émettre la sortie o. Il est nécessaire pour exprimer ce type de propriétés d'utiliser des automates plus complexesΓtels les automates de Büchi. De plusΓla méthode de vérification par simulation comportementale traite globalement l'ensemble des séquences d'exécution d'un programme. Il n'est donc pas possible de vérifier des propriétés concernant un sous-ensemble des séquences d'exécution du programme. Par exempleΓil est impossible par simulation comportementale de vérifier qu'il existe au moins une réaction du système réactif qui émet la sortie o. Les propriétés qui peuvent être vérifiées par simulation comportementale sont les propriétés de *sûreté* [Lam77ΓAS86].

Dans la section qui suit nous présentons un cas particulier de situation de vérification pour lequel la relation de satisfaction se simplifie. Ce cas particulier est identifié par une propriété des points d'observation utilisés : ils sont *disjoints*.

1.2.3 Cas des points d'observation disjoints

La notion de points d'observation disjoints est définie par l'intermédiaire de la fonction $\texttt{Est_Obs_Par}_{\mathcal{L},\theta}$.

Définition 1.11 fonction $\texttt{Est_Obs_Par}_{\mathcal{L},\theta}$
Soient $\mathcal{L}$ un ensemble d'étiquettes et θ un ensemble de points d'observation de $\mathcal{L}$, la fonction $\texttt{Est_Obs_Par}_{\mathcal{L},\theta}$ de profil $\mathcal{L} \rightarrow 2^{\theta}$ est définie par :

$$\texttt{Est_Obs_Par}_{\mathcal{L},\theta}(l) = \{\, \omega \in \theta \mid \omega(l) \,\}$$

Nous appelons $\texttt{Im}_{\mathcal{L},\theta}$ l'ensemble image de cette fonction. ■

Définition 1.12 Ensemble de points d'observation disjoints
L'ensemble θ est un ensemble de points d'observation disjoints si et seulement si :

$$\forall l \in \mathcal{L}, \ |\, \texttt{Est_Obs_Par}_{\mathcal{L},\theta}(l) \,| \ \leq 1$$ ■

Lorsque les points d'observation utilisés par la propriété sont disjointsΓil est possible de renommer chaque étiquette de l'automate :

- Par l'unique point d'observation qu'elle satisfait s'il existe ;

- Par elle-même sinon.

Ce renommage permet de simplifier la relation de satisfactionΓtout en préservant le résultat de la relation initiale.

Définition 1.13 Renommage du programme
Soient une propriété $\varphi_{\mathcal{L}}$ et un automate $A_{\mathcal{L}} = (Q, q_{init}, T)$. Si $\pi(\varphi_{\mathcal{L}})$ est un ensemble de points d'observation de $\mathcal{L}$ disjoints, alors il est possible d'appliquer aux étiquettes des transitions de $A_{\mathcal{L}}$ la fonction de renommage ρ, définie par :

$$\rho(l) = \textit{ si } \texttt{Est_Obs_Par}_{\mathcal{L},\pi(\varphi_{\mathcal{L}})}(l) = \emptyset \textit{ alors } l \textit{ sinon } \omega \textit{ tel que } \{\omega\} = \texttt{Est_Obs_Par}_{\mathcal{L},\pi(\varphi_{\mathcal{L}})}(l)$$

L'automate après renommage est noté $A^{\rho}_{\mathcal{L}}$. ■

La relation de satisfaction simplifiée est notée $\models^s_c$.

IntuitivementΓ$A_{\mathcal{L}} \models^s_c A_{\mathcal{L}'}$ si et seulement si à chaque séquence d'exécution s de l'automate $A_{\mathcal{L}}$Γil est possible d'associer une séquence d'exécution s' de $A_{\mathcal{L}'}$ telle qu'à chaque pasΓl'étiquette de s est syntaxiquement identique à celle correspondante dans s'.

Définition 1.14 Relation de satisfaction simplifiée
Soient $A_{\mathcal{L}} = (Q, q_{init}, T)$ et $A_{\mathcal{L}'} = (Q', q'_{init}, T')$, $A_{\mathcal{L}} \models^s_c A_{\mathcal{L}'}$ si et seulement si

$\exists \mathcal{R}el \subseteq Q \times Q'$ *telle que*

- $(q_{init}, q'_{init}) \in \mathcal{R}el$,
- $\forall (q_1, q'_1) \in \mathcal{R}el$,
  $(q_1, l, q_2) \in T \Rightarrow \exists (q'_1, l', q'_2) \in T'$ *telle que* $l = l'$ *et* $(q_2, q'_2) \in \mathcal{R}el$.

■

Cette relation de satisfaction correspond à la définition de la simulation donnée par Milner dans [Mil80]. C'est cette relation qui est généralement présentée comme la relation de satisfaction de la simulation comportementale.

Remarquons que s'il existe dans $\mathcal{L}$ une étiquette qui n'appartient pas à $\mathcal{L}'$ alors $A_{\mathcal{L}}$ et $A_{\mathcal{L}'}$ ne peuvent pas être mis en relation par $\models^s_c$.

Propriété 1.1 Résultat de préservation
Soient une propriété $\varphi_{\mathcal{L}}$ et un automate $A_{\mathcal{L}} = (Q, q_{init}, T)$ tels que $\pi(\varphi_{\mathcal{L}})$ est un ensemble de points d'observation de $\mathcal{L}$ disjoints, alors

$$A_{\mathcal{L}} \models_c \varphi_{\mathcal{L}} \iff A^{\rho}_{\mathcal{L}} \models^s_c \varphi_{\mathcal{L}}$$ ■

Démonstration 1.1

L'idée de cette preuve est de montrer que la relation $\mathcal{R}el$ entre les états de l'automate et de la propriété est la même, que ce soit pour la relation générale ou la relation simplifiée. Le résultat sur les états initiaux est immédiat, puisque l'état initial de $A^{\rho}_{\mathcal{L}}$ est égal à celui de $A_{\mathcal{L}}$.

Nous prenons pour la suite les notations suivantes :

$A_{\mathcal{L}} = (Q, q_{init}, T)$, $A^{\rho}_{\mathcal{L}} = (Q, q_{init}, T^{\rho})$ *et* $\varphi_{\mathcal{L}} = (Q', q'_{init}, T')$.

Nous commençons par prouver que :

$\forall (q_1, q'_1) \in \mathcal{R}el,$
$(q_1, l, q_2) \in T \Rightarrow \exists (q'_1, \omega, q'_2) \in T'$ *telle que* $\omega(l)$ *et* $(q_2, q'_2) \in \mathcal{R}el$
$\Rightarrow$
$\forall (q_1, q'_1) \in \mathcal{R}el,$
$(q_1, l, q_2) \in T^{\rho} \Rightarrow \exists (q'_1, \omega, q'_2) \in T'$ *telle que* $l = \omega$ *et* $(q2, q2') \in \mathcal{R}el$

Nous pouvons déduire de l'hypothèse de l'implication, le résultat intermédiaire suivant :

$\forall l \in \mathcal{L},\ |\texttt{Est_Obs_Par}_{\mathcal{L},\pi(\varphi_{\mathcal{L}})}(l)| \geq 1 \qquad (i).$

Les points d'observation étant disjoints, nous concluons l'égalité suivante :

$\forall l \in \mathcal{L},\ |\texttt{Est_Obs_Par}_{\mathcal{L},\pi(\varphi_{\mathcal{L}})}(l)| = 1$

Par conséquent d'après la définition 1.13, toutes les étiquettes de T^{ρ} sont des points d'observation et la fonction ρ est une bijection de $\mathcal{L}$ vers $\pi(\varphi_{\mathcal{L}})$. La fonction réciproque ρ^{-1} est donc définie. La partie droite de l'implication à prouver se réécrit en :

$\forall (q_1, q'_1) \in \mathcal{R}el,$
$(q_1, \omega, q_2) \in T^{\rho} \Rightarrow \exists (q'_1, \omega, q'_2) \in T'$ *telle que* $(q_2, q'_2) \in \mathcal{R}el$

Or $(q_1, \omega, q_2) \in T^{\rho} \Rightarrow (q_1, \rho^{-1}(\omega), q_2) \in T$.

L'hypothèse de l'implication à démontrer nous permet de conclure à l'existence de $(q'_1, \omega', q'_2) \in T'$ telle que $\omega'(\rho^{-1}(\omega))$ et $(q_2, q'_2) \in \mathcal{R}el$.

Or par application de (i), nous sommes sûrs que ω est le seul point d'observation qui satisfait $\rho^{-1}(\omega)$, d'où $\omega = \omega'$, ce qui nous permet de conclure la preuve.

La preuve de l'implication dans le sens inverse s'effectue suivant le même schéma, le résultat intermédiaire (i) se déduisant de la même manière des hypothèses.

■

Applications

Pour certains domaines, toutes les situations de vérification intéressantes sont telles que les points d'observation utilisés sont disjoints. C'est le cas par exemple de la vérification des protocoles de communication.

L'ensemble des évolutions possibles d'un protocole peut être modélisé par un automate booléen. Sur cet automateΓles méthodes de vérification formelle par simulation comportementale ou à base de logique temporelle peuvent être utilisées. Pour construire ces automates modèlesΓdes environnements de programmation basés sur des langages de spécification comme Lotos [BB88] ou Estelle [BD88] peuvent être utilisés. Par exempleΓl'environnement Cæsar-Aldébaran [FGM+92] est développé autour du langage de spécification Lotos et de l'outil de vérification formelle Aldébaran [Fer88ΓMou92ΓKer].

Ces langages spécifient un protocole en termes des actions qu'il effectue. Ces actions sont considérées comme atomiquesΓelles peuvent indiquer par exemple l'émission ou la réception d'un message. Ce sont ces noms d'actions qui étiquettent les transitions de l'automate modèle.

Considérons à présent les propriétés exprimées sur ces automates. L'atomicité des étiquettes conduit à des points d'observation qui sont eux aussi très simples. L'expérience montre qu'ils distinguent :

- Soit des actions particulières : ce sont celles qui sont pertinentes pour la propriété ;
- Soit des ensembles d'actions : ce sont celles qui ne sont pas pertinentes pour la propriété.

Par conséquentΓces points d'observation sont toujours disjoints : soit une action est pertinente pour la propriétéΓsoit elle ne l'est pas.

De nombreux outils de vérification comportementale pour ce type d'application on été développés : Auto [Ver87Γ dSV89]Γ Aldébaran [Fer88Γ Mou92Γ Ker]Γ le Concurrency-WorkBench [CPS89]Γ... Mis à part Auto (développé dans l'environnement du langage Esterel)Γ dont nous parlerons dans la section 1.4.2Γils sont tous basés sur la comparaison syntaxique des étiquettes de l'automate modèle du programme avec celles de la propriété. Par conséquentΓils implémentent la relation de satisfaction simplifiée. La comparaison par égalité syntaxique peut être mise en œuvre de façon efficace. La généralisation de ces outils à une relation de comparaison quelconque entre étiquettes soulève de nombreux problèmes pratiques quiΓs'ils sont mal résolusΓpeuvent diminuer considérablement les performances des outils. D'où l'intérêt de définir une traduction systématiqueΓqui permet de se ramener dans tous les casΓà la comparaison des étiquettes par égalité syntaxique.

C'est cette traduction que nous présentons dans la section qui suit.

1.2.4 Cas des ensembles de points d'observation non disjoints

Avant de présenter la transformation de la propriété qui permet de se ramener à des points d'observation disjointsΓnous justifions son intérêt en présentant des exemples de situations de vérification de systèmes réactifs.

La vérification des systèmes réactifs

Supposons que les automates modèles de systèmes réactifs soient de la forme définie en début de ce chapitre. Leurs étiquettes modélisent les réactions du système. Nous avons vu qu'elles sont structurées en deux parties :

- Un monôme complet et non contradictoire sur les entrées du système réactif définit la condition de déclenchement de la transition ;
- Un ensemble définit les sorties émises si la transition est déclenchée.

Les propriétés que l'on souhaite exprimerΓ établissent des relations entre un sous-ensemble des entrées (qui peut être vide) et un sous-ensemble des sorties. On peut vouloir par exemple prouver que deux sorties sont exclusivesΓou encoreΓque chaque occurrence d'une entrée $e1$ qui suit immédiatement une occurrence de l'entrée $e2$Γest accompagnée de la sortie $o1$ mais pas de la sortie $o2$.

De manière généraleΓ il faut observer à chaque réaction du programme le statut d'un sous-ensemble d'entrées et/ou de sorties. Les points d'observation expriment donc une configuration d'événements. Nous supposons dans toute la suiteΓque ces configurations sont données sous forme de monômes d'événements non contradictoires. Nous notons les points d'observation $\texttt{obs(}m_e\texttt{)}$.

IntuitivementΓune étiquette du modèle d'un système réactif satisfait un point d'observationΓ si et seulement si le statut de tous les événements qui apparaissent dans le point d'observation est identique à celui de l'étiquetteΓ en considérant qu'une sortie non émise est absente pour la réaction. Par exempleΓ l'étiquette $e1.\overline{e2}.e3/o1, o2$ satisfait les points d'observation $\texttt{obs}(e1.\overline{e2}.o1), \texttt{obs}(e1.\overline{o3})$ et $\texttt{obs}(o1)$ alors qu'elle ne satisfait ni $\texttt{obs}(\overline{o1})$ ni $\texttt{obs}(e1.o3)$.

FormellementΓune étiquette de la forme m_e/O satisfait un point d'observation $\texttt{obs(}m_e{}'\texttt{)}$Γsi et seulement si :

- Pour toute entrée $\alpha \in m_e{}'^+ \cup m_e{}'^-$Γ$(\alpha \in m_e{}'^+ \Rightarrow \alpha \in m_e{}^+)$ et $(\alpha \in m_e{}'^- \Rightarrow \alpha \in m_e{}^-)$;
- Et pour toute sortie $\alpha \in m_e{}'^+ \cup m_e{}'^-$Γ$(\alpha \in m_e{}'^+ \Rightarrow \alpha \in O)$ et $(\alpha \in m_e{}'^- \Rightarrow \alpha \notin O)$.

Selon la propriété et l'ensemble d'étiquettes de l'automate modèle du systèmeΓles points d'observation peuvent ne pas être disjoints. Considérons un système qui admet pour sorties $o1$ et $o2$. On veut vérifier que ce système n'émet jamais en même temps ces deux sorties. Les bons comportements du système à chaque réaction sont : soit n'émettre ni $o1$ ni $o2$Γsoit émettre $o1$ mais pas $o2$Γsoit émettre $o2$ mais pas $o1$. L'automate qui exprime cette propriété sous forme comportementale est construit à partir des trois points d'observation suivants $\texttt{obs}(\overline{o1}.\overline{o2}), \texttt{obs}(o1.\overline{o2})$ et $\texttt{obs}(\overline{o1}.o2)$. Il est donné par la figure 1.2. L'ensemble de trois étiquettes accolées à l'unique transition de la figureΓpermet de représenter de manière conciseΓtrois transitions qui ont les mêmes états source et but que la transition dessinée. Cette convention de dessin est utilisée dans toute la suite.

Quel que soit l'ensemble des étiquettes de l'automate qui modélise le système réactifΓles points d'observation utilisés dans cet automate sont disjoints. Il est donc possible de renommer le programme et d'appliquer la relation simplifiée.

Supposons à présent queΓpour un système réactif qui possède des entrées $e1$ et $e2$Γet dont l'ensemble des sorties contient $o1$ et $o2$Γla propriété à vérifier soit la suivante :

Chaque occurrence d'une entrée $e1$ *qui suit immédiatement une occurrence de l'entrée* $e2$*, est accompagnée de la sortie* $o1$ *mais pas de la sortie* $o2$*.*

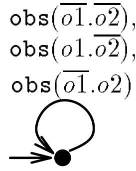


Figure 1.2: automate exprimant l'exclusion mutuelle de $o1$ et $o2$

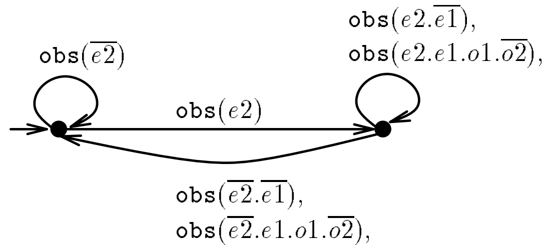


Figure 1.3: automate possible pour exprimer le lien entre $e1$Γ$e2$Γ$o1$ et $o2$

Pour exprimer cette propriété par un automate il faut distinguer deux états selon que l'on se trouve ou pas après une occurrence de $e2$. Lorsqu'on est sûr de ne pas être après une occcurrence de $e2$Γla seule chose intéressante à observer dans le programme est le statut de $e2$Γpuisque si cet événement est présentΓalors la réaction suivante doit être analysée comme une réaction qui suit une occurrence de $e2$. Lorsqu'on est sûr d'être après une occurrence de $e2$Γil faut continuer à observer le statut de cet événement pour savoir analyser la réaction suivanteΓmais il faut aussi observer le statut de $e1$Γ$o1$ et $o2$ pour déterminer si la réaction du programme est correcte. Elle l'est si la présence de $e1$ est accompagnée de la présence de $o1$ et de l'absence de $o2$Γou si $e1$ est absent. Par conséquentΓl'automate de la figure 1.3 est un automate possible pour exprimer la propriété énoncée.

Il suffit que l'ensemble des étiquettes de l'automate modèle du système réactif contienne l'étiquette $e1.e2/o1$ pour que les points d'observation qui apparaissent dans cet automate ne soient pas disjoints. En effetΓcette étiquette satisfait à la fois obs($e2$) et obs($e2.e1.o1.\overline{o2}$).

Considérons à présent l'automate de la figure 1.4. Cet automate peut aussi être utilisé pour vérifier la propriété précédente. Il exprime les bons comportements possibles du système dans chacun des deux états mais de manière beaucoup plus détaillée. Contrairement à l'automate de la figure 1.3Γles points d'observation utilisés dans cet automate sont disjointsΓquel que soit les étiquettes de l'automate modèle du système réactif à vérifier.

Cet exemple nous montre d'une part qu'il peut exister plusieurs automates pour exprimer la même propriété ; et d'autre part qu'il existe toujours un moyen de l'exprimer à partir de points d'observation disjoints quel que soit l'ensemble d'étiquettes de l'automate.

Néanmoins le domaine des systèmes réactifs est un bon exemple pour illustrer le besoin d'une méthode systématique permettant de transformer une propriété pour que les points d'observation

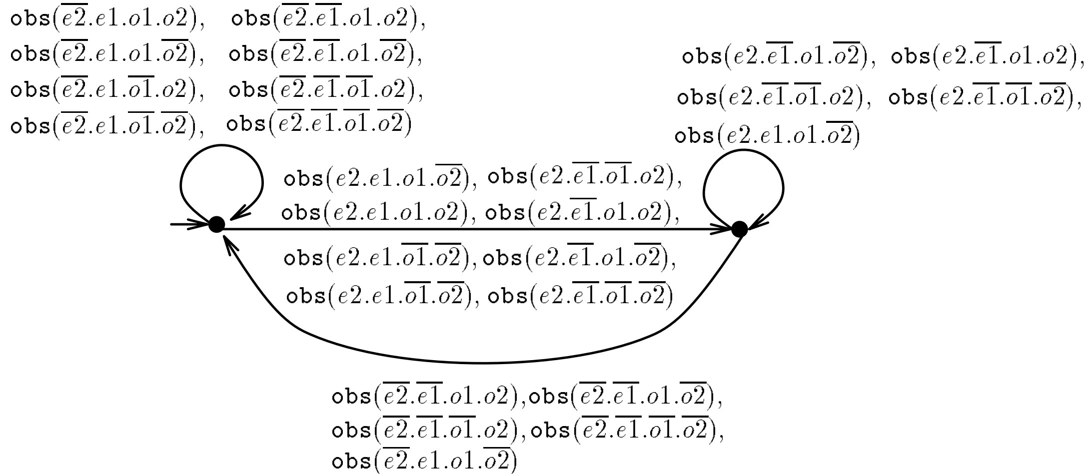


Figure 1.4: automate construit sur des points d'observation disjoints

utilisés soient disjoints. En effet, cet exemple illustre bien le problème posé par la vérification des systèmes réactifs : pour obtenir une propriété utilisant des points d'observation disjoints, il est nécessaire de détailler à un tel point les réactions possibles du programme que l'automate devient complexe et peu intuitif par rapport à son énoncé. Pourtant il existe une forme d'expression de la propriété beaucoup plus intuitive, qui pourrait être utilisée si des outils plus généraux — mettant en œuvre la relation de satisfaction générale définie en 1.10 — existaient.

La solution que nous proposons consiste à automatiser le passage de l'automate de la figure 1.3 à celui de la figure 1.4, de manière à conserver la description intuitive de la propriété tout en permettant l'utilisation d'outils basés sur la comparaison syntaxique des étiquettes de l'automate et de la propriété.

C'est cette transformation, qui nécessite à la fois une modification de l'ensemble des points d'observation et de la structure — ajout de transitions — de la propriété initiale, que nous présentons dans la section qui suit. La présentation que nous effectuons n'est pas particulière au domaine des systèmes réactifs, puisqu'elle est paramétrée par un ensemble d'étiquettes quelconque. Elle peut donc être appliquée à tout autre domaine, où l'expression de la propriété de manière intuitive, conduit à des points d'observation non disjoints.

Transformation de la propriété

La transformation que nous présentons ne s'effectue pas de manière indépendante de l'automate modèle du système à vérifier. En effet, c'est en tenant compte des étiquettes de cet automate que le nouvel ensemble de points d'observation disjoints est construit. En un certain sens, la transformation est minimale, puisque selon l'ensemble des étiquettes de l'automate, la nouvelle propriété est plus ou moins détaillée. Par conséquent, sur l'exemple précédent il n'est pas certain que l'automate de la figure 1.4 soit l'image de l'automate de la figure 1.3. Par exemple, si l'ensemble des étiquettes de l'automate modèle est tel que $o1$ et $o2$ ne sont jamais émis simultanément alors chaque point d'observation peut se contenter de préciser le statut d'un seul des

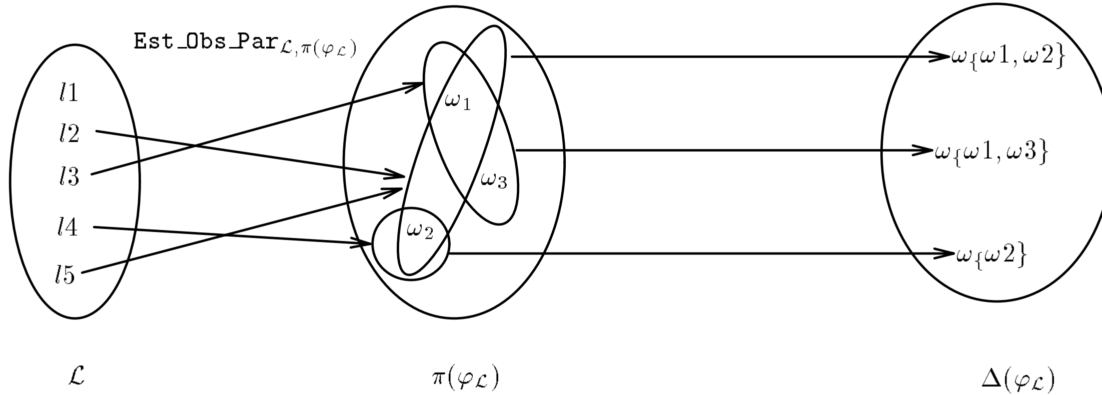


Figure 1.5: exemple de modification d'une propriété

deux événements.

Etant donnée une propriété $\varphi_\mathcal{L}$Γla construction du nouvel ensemble de points d'observation se fait en considérant le domaine image $\texttt{Im}_{\mathcal{L},\pi(\varphi_\mathcal{L})}$ de la fonction $\texttt{Est_Obs_Par}_{\mathcal{L},\pi(\varphi_\mathcal{L})}$. Le nouvel ensemble de points d'observation est noté $\Delta_{\varphi_\mathcal{L}}$. Il est construit en associant à chaque ensemble de points d'observation contenu dans $\texttt{Im}_{\mathcal{L},\pi(\varphi_\mathcal{L})}$Γexcepté l'ensemble videΓun prédicat satisfait par une étiquette si et seulement si cette étiquette satisfait tous les points d'observation de l'ensemble en questionΓet uniquement ceux-ci.

Considérons par exempleΓl'ensemble d'étiquettes $\mathcal{L} = \{l1, l2, l3, l4, l5\}$. Supposons que la propriété soit construite à partir des points d'observation $\omega 1, \omega 2$ et $\omega 3$. La figure 1.5 présente la fonction $\texttt{Est_Obs_Par}_{\mathcal{L},\pi(\varphi_\mathcal{L})}$Γainsi que les nouveaux points d'observation. Dans ce dernier ensembleΓ$\omega_{\{\omega 1,\omega 2\}}$ est satisfait par $l2$ et $l5$Γcar ces étiquettes ne satisfont que $\omega 1$ et $\omega 2$.

Définition 1.15 Nouvel ensemble de points d'observation
Soit une propriété $\varphi_\mathcal{L}$ construite sur $\pi(\varphi_\mathcal{L})$, le nouvel ensemble de points d'observation $\Delta_{\varphi_\mathcal{L}}$ est défini par : $\Delta_{\varphi_\mathcal{L}} = \{\omega_\theta \mid \theta \in \texttt{Im}_{\mathcal{L},\pi(\varphi_\mathcal{L})} \wedge \theta \neq \emptyset\}$ *avec,*

$$\forall l \in \mathcal{L}, \omega_\theta(l) - (\forall \omega \in \theta,\ \omega(l)) \wedge (\forall \omega \in \{\pi(\varphi_\mathcal{L}) \setminus \theta\},\ \neg\omega(l)).$$

■

Ces nouveaux points d'observation de $\mathcal{L}$ sont disjoints par construction.

La fonction de transformation de la propriété est notée $\mathcal{M}$. Elle remplace chaque transition de la propriété initiale (étiquetée par un élement de $\pi(\varphi_\mathcal{L})$) par un ensemble de transitions ayant les mêmes états source et but que la transition remplacéeΓmais qui sont étiquetées par des éléments de $\Delta_{\varphi_\mathcal{L}}$. Plus précisémentΓsi l'étiquette à remplacer est ωΓalors une transition étiquetée par ω_θ fait partie de l'ensemble remplaçant si et seulement si ω appartient à θ. En reprenant l'exemple précédentΓsi la propriété initiale est donnée par la figure 1.6 `(a)`Γalors la propriété modifiée est donnée par la figure 1.6`(b)`.

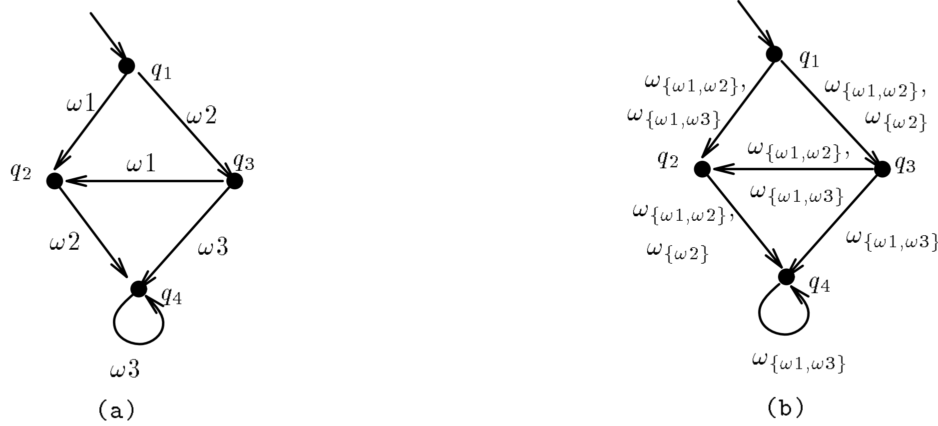


Figure 1.6: transformation d'une propriété comportementale

Définition 1.16 Transformation de la propriété
Soit une propriété $\varphi_{\mathcal{L}} = (Q, q_{init}, T)$ construite sur $\pi(\varphi_{\mathcal{L}})$, la propriété transformée construite sur $\Delta_{\varphi_{\mathcal{L}}}$, image de la fonction $\mathcal{M}$, a le même ensemble d'états et le même état initial que $\varphi_{\mathcal{L}}$. Son ensemble de transitions, noté $T^{\mathcal{M}}$, est défini par : $T^{\mathcal{M}} = \{(q, \omega_\theta, q') | \exists (q, \omega, q') \in T \wedge \omega \in \theta\}$

Nous notons $\varphi_{\mathcal{L}}^{\mathcal{M}}$ la propriété modifiée. ■

Propriété 1.2 Résultat de préservation
Soient une propriété $\varphi_{\mathcal{L}}$ et un automate $A_{\mathcal{L}} = (Q, q_{init}, T)$, alors

$$A_{\mathcal{L}} \models_c \varphi_{\mathcal{L}} \iff A_{\mathcal{L}} \models_c \varphi_{\mathcal{L}}^{\mathcal{M}}$$ ■

Remarque 1.1 *Les points d'observation de $\varphi_{\mathcal{L}}^{\mathcal{M}}$ étant disjoints, le premier résultat de préservation (propriéte 1.1) s'applique, d'où :*

$$A_{\mathcal{L}} \models_c \varphi_{\mathcal{L}} \iff A_{\mathcal{L}}^{\rho} \models_c^s \varphi_{\mathcal{L}}^{\mathcal{M}}$$ ■

Démonstration 1.2
L'idée à la base de cette démonstration est la même que celle utilisée pour démontrer le premier résultat de préservation. Elle consiste à montrer que la même relation entre états peut être utilisée pour prouver les deux relations de satisfaction.

Le résultat concernant les états initiaux est immédiat car la fonction $\mathcal{M}$ ne modifie pas l'état initial de la propriété.

Nous prenons pour la suite les notations suivantes :

$A_{\mathcal{L}} = (Q, q_{init}, T), \varphi = (Q', q'_{init}, T')$ *et* $\varphi_{\mathcal{L}}^{\mathcal{M}} = (Q', q'_{init}, T^{\mathcal{M}})$.

Nous commençons par prouver que :

$\forall (q_1, q_1') \in \mathcal{R}el,$
$(q_1, l, q_2) \in T \Rightarrow \exists (q_1', \omega, q_2') \in T'$ *telle que* $\omega(l)$ *et* $(q_2, q_2') \in \mathcal{R}el$
$\Rightarrow$
$\forall (q_1, q_1') \in \mathcal{R}el,$
$(q_1, l, q_2) \in T \Rightarrow \exists (q_1', \omega_\theta, q_2') \in T^{\mathcal{M}}$ *telle que* $\omega_\theta(l)$ *et* $(q_2, q_2') \in \mathcal{R}el$

Soient $(q_1, l, q_2) \in T$ *et* $\theta = \texttt{Est_Obs_Par}_{\mathcal{L},\pi(\varphi_\mathcal{L})}(l)$ *; par hypothèse,* $\exists (q_1', \omega, q_2') \in T'$ *telle que* $\omega(l)$ *et* $(q_2, q_2') \in \mathcal{R}el$*. Or, par définition de* θ*, nous avons* $\omega \in \theta$*. Par conséquent, d'après la fonction* $\mathcal{M}$*,* $(q_1', \omega_\theta, q_2') \in T^{\mathcal{M}}$*. Or, par construction de* ω_θ*, nous avons* $\omega_\theta(l)$ *(*θ *est exactement l'ensemble des points d'observation qui sont satisfaits par* l*), ce qui nous permet de conclure.*

Pour prouver l'implication inverse, l'existence de $(q_1', \omega_\theta, q_2') \in T^{\mathcal{M}}$ *et la définition de* $\mathcal{M}$ *nous permettent de déduire l'existence de* $(q_1', \omega, q_2') \in T'$ *telle que* $\omega \in \theta$*.*

Or, $\omega_\theta(l) \wedge \omega \in \theta \Rightarrow \omega(l)$*, ce qui nous permet de conclure.*

■

1.3 Vérification à l'aide d'une logique temporelle

La vérification d'un modèle de système réactif à l'aide d'outils basés sur une logique temporelle soulève le même type d'inadéquation que pour la simulation comportementale. En effetΓceux-ci sont aussi basés sur la comparaison syntaxique des étiquettes de l'automate modèle et de la propriétéΓalors que d'autres types de relations de comparaison peuvent s'avérer utiles pour exprimer la propriété de manière plus intuitive. C'est le même type de situation que celle présentée par les figures 1.3 et 1.4.

Pour répondre à ce problèmeΓnous suivons la même approche que dans le cas de la simulation comportementale : présentation de la méthode d'une manière plus générale que les présentations usuellesΓmise en évidence d'un cas particulier plus simple à mettre en œuvreΓpuis transformation de la propriété pour satisfaire les conditions d'application de la relation simplifiée.

1.3.1 Expression d'une propriété

Une propriété est une formule d'une logique temporelle. Différentes logiques temporelles existent. Nous présentons la logique CTL ("Computational Tree Logic" [EC82]).

L'ensemble des formules de CTL est donné par la grammaire suivante :

Définition 1.17 Syntaxe des formules de CTL
$\varphi \ ::= \ p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists X \varphi \mid \varphi_1 \, \exists\mathcal{U} \, \varphi_2 \mid \varphi_1 \, \forall\mathcal{U} \, \varphi_2$

Où p *est un prédicat de base sur les états de l'automate à vérifier. Ce prédicat appartient à un ensemble de prédicats de base* P*, muni d'une fonction d'interprétation* $\mathcal{I}$ *qui, étant donnés un prédicat* p *de* P *et un état* q *de l'automate modèle, détermine si* q *satisfait* p*.* ■

1.3.2 Relation de satisfaction

La sémantique des formules de la logique temporelle s'exprime par une relation entre une formule et un état de l'automate modèle du système à vérifier. Cette relation dépend de la fonction d'interprétation de l'ensemble des prédicats de baseΓnous la notons $\models_{\mathcal{I}}$. Elle exprime le fait qu'un état satisfait une formule donnée.

Définition 1.18 Relation de satisfaction
Soient $A_{\mathcal{L}} = (Q, q_{init}, T)$ et φ une formule de CTL, $A_{\mathcal{L}} \models_l \varphi$ si et seulement si $q_{init} \models_{\mathcal{I}} \varphi$. ■

La relation $\models_{\mathcal{I}}$ est définie de manière inductive sur la structure des formules. Informellement :

- $\exists X \varphi$ est satisfait par un état q si et seulement s'il existe un état successeur de q qui satisfait φ.
- $\varphi_1 \; \exists\mathcal{U} \; \varphi_2$ est satisfait par un état q si et seulement s'il existe une séquence d'exécution issue de q telle que tous les états rencontrés satisfont φ_1 jusqu'à ce qu'un état satisfasse φ_2.
- $\varphi_1 \; \forall\mathcal{U} \; \varphi_2$ est satisfait par un état q si et seulement si toutes les séquences d'exécution issues de q sont telles que tous les états rencontrés satisfont φ_1 jusqu'à ce qu'un état satisfasse φ_2.

Avant de présenter la sémantique formelle des formulesΓnous définissons une notion de séquence d'exécutionΓdans laquelle seuls les états visités sont conservés (on ne conserve pas les étiquettes).

Définition 1.19 Séquence d'exécution
Etant donné un automate $A_{\mathcal{L}} = (Q, q_{init}, T)$, $(q_0, q_1, q_2, \ldots)$ est une séquence d'exécution issue de l'état q si et seulement si $q_0 = q$ et $\forall i, \exists l \in \mathcal{L}$ telle que $(q_i, l, q_{i+1}) \in T$. ■

Définition 1.20 Sémantique des formules

- *$q \models_{\mathcal{I}} p$ ssi $\mathcal{I}(p, q)$.*
- *$q \models_{\mathcal{I}} \neg\varphi$ ssi $\neg(q \models_{\mathcal{I}} \varphi)$.*
- *$q \models_{\mathcal{I}} \varphi_1 \wedge \varphi_2$ ssi $q \models_{\mathcal{I}} \varphi_1 \wedge q \models_{\mathcal{I}} \varphi_2$.*
- *$q \models_{\mathcal{I}} \exists X \varphi$ ssi $\exists (q, l, q') \in T$ telle que $q' \models_{\mathcal{I}} \varphi$.*
- *$q \models_{\mathcal{I}} \varphi_1 \; \exists\mathcal{U} \; \varphi_2$ ssi il existe une séquence d'exécution $(q_0, q_1, q_2, \ldots)$ issue de q pour laquelle il existe $k \geq 0$ tel que $q_k \models_{\mathcal{I}} \varphi_2$ et pour tout $0 \leq j < k, q_j \models_{\mathcal{I}} \varphi_1$.*
- *$q \models_{\mathcal{I}} \varphi_1 \; \forall\mathcal{U} \; \varphi_2$ ssi pour toutes les séquences d'exécution $(q_0, q_1, q_2, \ldots)$ issues de q il existe $k \geq 0$ tel que $q_k \models_{\mathcal{I}} \varphi_2$ et pour tout $0 \leq j < k, q_j \models_{\mathcal{I}} \varphi_1$.*

■

Les abréviations suivantes sont couramment utilisées :

- $\varphi_1 \vee \varphi_2$ désigne $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$Γc'est la disjonction usuelle ;
- $\varphi_1 \Rightarrow \varphi_2$ désigne $\neg\varphi_1 \vee \varphi 2$Γc'est l'implication ;
- $\forall X \varphi$ désigne $\neg\exists X \neg\varphi$Γcette formule est satisfaite pour un état q si et seulement si tous les successeurs de q satisfont φ.
- $\forall\Diamond\varphi$ désigne $true\ \forall\mathcal{U}\ \varphi$Γcette formule est satisfaite pour un état q si et seulement siΓsur toutes les séquences d'exécution issues de qΓil existe un état qui satisfait φ.
- $\exists\Diamond\varphi$ désigne $true\ \exists\mathcal{U}\ \varphi$Γcette formule est satisfaite pour un état q si et seulement s'il existe une séquence d'exécution issue de q sur laquelle il existe un état qui satisfait φ.
- $\forall\Box\varphi$ désigne $\neg\exists\Box\neg\varphi$Γcette formule est satisfaite pour un état q si et seulement si sur toutes les séquences d'exécution issues de q tous les états satisfont φ.
- $\exists\Box\varphi$ désigne $\neg\forall\Diamond\neg\varphi$Γcette formule est satisfaite pour un état q si et seulement s'il existe une séquence d'exécution issue de q sur laquelle tous les états satisfont φ.

L'évaluation de la relation de satisfaction d'une formule par un état est mise en œuvre par des algorithmes dit de "model-checking" [CES86ΓQS81]. Ces algorithmes calculent l'ensemble des états qui satisfont une formule. Plusieurs outils de vérification mettent en œuvre ce type d'algorithmesΓparmi lesquels on peut citer les systèmes EMC [CES86] et Xésar [RRSV87].

Dans la présentation qui vient d'être faiteΓnous avons volontairement omis de préciser la nature des prédicats de base. Pourtant ils déterminent en partie l'évaluation d'une propriété sur un état. Pour mieux présenter ce que sont ces prédicats de baseΓcommençons par considérer un exemple de vérification de système réactif déjà traité par simulation comportementale. Reprenons l'exemple visant à prouver qu'une sortie o n'est jamais émise par un système réactif. Une solution pour exprimer cette propriétéΓconsiste à définir un prédicat $p_{\overline{o}}$ sur les états de l'automate modèle du système qui est vrai dans un état q si et seulement si aucune des transitions issues de q n'émet la sortie o ; puis à vérifier la formule $\forall\Box p_{\overline{o}}$ sur l'état initial du modèle. En faisant ainsiΓon vérifie que tous les états atteignables à partir de l'état initialΓsatisfont $p_{\overline{o}}$. Le prédicat de base $p_{\overline{o}}$ est défini à l'aide du point d'observation $\omega_{\overline{o}}$. Ce point d'observation est satisfait par une étiquette m_e/O si et seulement si o n'appartient pas à l'ensemble de sorties O. La fonction d'interprétation du prédicat $p_{\overline{o}}$ est défini par $\mathcal{I}(p_{\overline{o}}, q) - \forall(q, l, q') \in T, \omega_{\overline{o}}(l)$.

Ce sont donc les prédicats de base qui permettent de transformer l'information présente sur les transitionsΓen propriétés sur les états de l'automate. De plusΓc'est au niveau des prédicats de base qu'interviennent les points d'observation des étiquettes. De la même manière que dans la méthode par simulation comportementaleΓla propriété est construite à l'aide d'un ensemble de points d'observationΓune propriété en logique temporelle est elle aussi construite sur un tel ensemble. Nous pouvons donc paramétrer une propriété φ par l'ensemble des étiquettes $\mathcal{L}$ qu'elle observeΓd'où $\varphi_{\mathcal{L}}$Γet utiliser $\pi(\varphi_{\mathcal{L}})$ comme l'ensemble des points d'observation utilisés par $\varphi_{\mathcal{L}}$.

Les prédicats de base peuvent prendre des formes multiples. Certains sont relatifs aux transitions et permettent de transformer des informations portées par les étiquettes en propriétés d'états. Parmi ceux-ciΓles plus couramment utilisés sont :

- $\texttt{enable}_\omega$ est satisfait par un état si et seulement si une transition issue de cet état satisfait le point d'observation ω. Par conséquent la définition de la fonction d'interprétation pour ce prédicat est :

  $\mathcal{I}(\texttt{enable}_\omega, q) \iff \exists (q, l, q') \in T$ telle que $\omega(l)$

- $\texttt{after}_\omega$ est satisfait par un état si et seulement si toutes les transitions qui aboutissent dans cet état satisfont le point d'observation ω. Par conséquent la définition de la fonction d'interprétation sur ce prédicat est :

  $\mathcal{I}(\texttt{after}_\omega, q) \iff \exists (q', l, q) \in T \wedge \forall (q', l, q) \in T, \omega(l)$

L'interprétation des prédicats de la forme $\texttt{after}_\omega$ soulève un problème particulier. IntuitivementΓle rôle de ce prédicat est d'identifier dans une séquence d'exécution les états qui suivent une étiquette qui satisfait le point d'observation ω. C'est un rôle symétrique à $\texttt{enable}_\omega$Γpuisque ce prédicat identifie les états qui précèdent l'exécution d'une transition étiquetée par une étiquette qui satisfait ω. Or la définition que l'on vient de donner de la fonction d'interprétation sur ce prédicat ne correspond pas à cette idée intuitiveΓet il n'est pas possible de la modifier pour qu'il en soit ainsi.

Pour s'en convaincreΓconsidérons l'automate de la figure 1.7(a)Γet le prédicat $\texttt{after}_{l1}$Γoù $l1$ est le point d'observation satisfait uniquement par l'étiquette $l1$. D'après la définition présentée ci-dessusΓseul l'état q_2 satisfait $\texttt{after}_{l1}$. NéanmoinsΓdans la séquence d'exécution (q_1, q_3, q_4) l'état q_4 suit une étiquette $l1$.

La solution pour remédier à ce problème consiste à transformer l'automate en dupliquant certains étatsΓde telle manière que toutes les transitions qui atteignent un étatΓsatisfassent exactement le même ensemble de point d'observations. Un telle modification qui préserve la propriété à vérifier existe et peut être automatisée. Elle est présentée dans le cas particulier des ensembles de points d'observation disjoints dans [Rod88]. Le résultat de cette transformation de l'automate pour l'évaluation de $\texttt{after}_{l1}$ conduit à l'automate de la figure 1.7(b). Dans toute la suiteΓnous supposons que ce type de transformation est effectuée avant d'évaluer un prédicat de la forme $\texttt{after}_\omega$.

Pour résumer cette sectionΓle calcul de la satisfaction d'une propriété par un automate s'effectue en deux phases :

- Evaluation de la fonction d'interprétation des prédicats de base utilisés dans la propriétéΓsur les états de l'automate modèle du système à vérifier. Cette première phase peut nécessiter la modification de l'automate. De plusΓc'est dans cette première phase que sont pris en compte les points d'observation des étiquettes.

- Evaluation de la sémantique de la formule en tenant compte de la fonction d'interprétation définie dans la première phaseΓde manière à déterminer si l'état initial de l'automate satisfait ou non la formule.

1.3.3 Points d'observation disjoints

Si les points d'observation utilisés dans la propriété sont disjoints alors la première phase se simplifie. En effetΓsi on applique à l'automate la fonction de renommage définie en 1.13Γalors

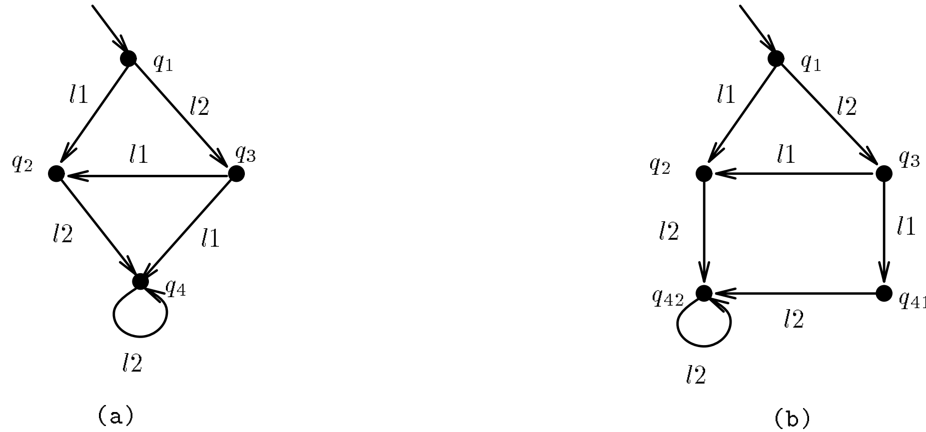


Figure 1.7: évaluation des prédicats de la forme $\texttt{after}_\omega$

l'évaluation d'un prédicat ne nécessite plus la mise en œuvre de relations complexes pour déterminer si une étiquette satisfait ou non un point d'observationΓmais simplement une comparaison syntaxique d'étiquettes. Nous notons $\mathcal{I}^s$ la forme simplifiée de $\mathcal{I}$. Par exempleΓla définition de $\mathcal{I}^s$ sur les prédicats de base $\texttt{after}_\omega$ et $\texttt{enable}_\omega$ est donnée ci-dessous.

Définition 1.21 Définition de $\mathcal{I}^s$ sur $\texttt{after}_\omega$ et $\texttt{enable}_\omega$

- $\mathcal{I}^s(\texttt{enable}_\omega, q) \iff \exists(q, l, q') \in T$ *telle que* $\omega = l$.
- $\mathcal{I}^s(\texttt{after}_\omega, q) \iff \exists(q', l, q) \in T \wedge \forall(q', l, q) \in T,\ \omega = l$. ∎

L'évaluation de la sémantique des formules de la logique — seconde phase de l'analyse d'une propriété sur un automate — s'effectue dans le cas d'un ensemble de points d'observation disjointsΓen utilisant cette relation d'interprétation simplifiée.

Définition 1.22 Relation de satisfaction simplifiée
Soient $A_\mathcal{L} = (Q, q_{init}, T)$ et $\varphi_\mathcal{L}$ une formule de CTL,

$$A_\mathcal{L} \models^s_l \varphi_\mathcal{L} \iff q_{init} \models_{\mathcal{I}^s} \varphi_\mathcal{L}. \quad ∎$$

Une propriété de préservation similaire à celle énoncée dans le cadre de la simulation comportementale peut être exprimée. La fonction de renommage des étiquettes du programme est celle définie en 1.13.

Propriété 1.3 Résultat de préservation
Soient une propriété $\varphi_\mathcal{L}$ et un automate $A_\mathcal{L} = (Q, q_{init}, T)$ tels que $\pi(\varphi_\mathcal{L})$ est un ensemble de points d'observation de $\mathcal{L}$ disjoints, alors

$$A_\mathcal{L} \models_l \varphi_\mathcal{L} \iff A^\rho_\mathcal{L} \models^s_l \varphi_\mathcal{L} \quad ∎$$

Démonstration 1.3
L'idée de la démonstration est de prouver que quels que soient l'automate et le prédicat de base, la fonction d'interprétation $\mathcal{I}$ est équivalente à la fonction simplifiée $\mathcal{I}^s$. Nous effectuons la démonstration pour le prédicat de base $\texttt{enable}_\omega$*. Le même type de démonstration peut être effectué pour* $\texttt{after}_\omega$*.*

Nous utilisons les notations suivantes : $A_\mathcal{L} = (Q, q_{init}, T)$ *et* $A^\rho_\mathcal{L} = (Q, q_{init}, T^\rho)$*.*

Il nous faut prouver l'équivalence suivante :

$\forall q \in Q, \mathcal{I}(\texttt{enable}_\omega, q) \iff \mathcal{I}^s(\texttt{enable}_\omega, q)$

Supposons dans un premier temps que $\mathcal{I}(\texttt{enable}_\omega, q)$*. D'après la définition, cela signifie que* $\exists(q, l, q') \in T$ *telle que* $\omega(l)$*. Par conséquent,* $|\texttt{Est_Obs_Par}_{\mathcal{L},\pi(\varphi_\mathcal{L})}(l)| \geq 1$*. L'hypothèse sur l'ensemble des points d'observation de la propriété (ils sont disjoints) nous permet de déduire l'égalité suivante :* $|\texttt{Est_Obs_Par}_{\mathcal{L},\pi(\varphi_\mathcal{L})}(l)| = 1$*.*

D'après la fonction de renommage, $(q, \omega, q') \in T^\rho$*. Ce qui nous permet de conclure* $\mathcal{I}^s(\texttt{enable}_\omega, q)$*.*

Supposons à présent que $\mathcal{I}^s(\texttt{enable}_\omega, q)$*. D'après la définition, cela signifie que* $\exists(q, \omega, q') \in T^\rho$ *telle que* $\omega = l$*. D'après la fonction de renommage, une telle transition appartient à* T^ρ *si et seulement si* $\exists(q, l, q') \in T$ *telle que* $\omega(l)$*. Ce qui nous permet de conclure* $\mathcal{I}(\texttt{enable}_\omega, q)$*.*

L'équivalence $\forall q \in Q, \mathcal{I}(\texttt{enable}_\omega, q) \iff \mathcal{I}^s(\texttt{enable}_\omega, q)$ *est donc démontrée.* ■

Comme pour la simulation comportementaleΓ c'est pour la vérification des protocoles de communication que des outils mettant en œuvre la relation simplifiée ont été définis. C'est le cas par exemple du système Xésar [Rod88]Γdans lequel des prédicats de la forme `after` et `enable` sont disponibles pour exprimer les propriétés.

1.3.4 Cas des ensembles de points d'observation non disjoints

La méthode systématique pour se ramener à une formule construite à l'aide de points d'observation disjointsΓ est similaire à celle présentée pour la simulation comportementale. Le nouvel ensemble de points d'observation est construit de manière identiqueΓ à partir de l'ensemble des étiquettes de l'automate à vérifier. Les nouveaux points d'observation — ensemble $\Delta(\varphi_\mathcal{L})$ — sont donc de la forme ω_θΓ où θ est l'ensemble image d'une étiquette par la fonction $\texttt{Est_Obs_Par}_{\mathcal{L},\pi(\mathcal{L})}$. La propriété est transformée de la manière suivante : chaque prédicat de base paramétré par ω — par exemple de la forme $\texttt{after}_\omega$ ou $\texttt{enable}_\omega$ — est remplacé par la disjonction d'un ensemble de prédicats de la forme ω_θ. Tous les éléments de $\Delta(\varphi_\mathcal{L})$ tels que θ contient le point d'observation ω font partie de cet ensemble.

Nous reprenons à présent l'exemple illustrant la transformation d'une propriété dans le cadre de la simulation comportementale. Les données de cet exemple sont présentées par la figure 1.5. Supposons que la propriété à vérifier soit exprimée par la formule suivante :

$\forall\Box(\texttt{after}_{\omega 1} \Rightarrow \forall\Diamond(\texttt{enable}_{\omega 2} \wedge \texttt{enable}_{\omega 3}))$Γalors cette propriété est modifiée et devient :

$$\forall\Box((\texttt{after}_{\omega_{\{\omega 1,\omega 2\}}} \vee \texttt{after}_{\omega_{\{\omega 1,\omega 3\}}}) \Rightarrow \forall\Diamond((\texttt{enable}_{\omega_{\{\omega 2\}}} \vee \texttt{enable}_{\omega_{\{\omega 1,\omega 2\}}}) \wedge \texttt{enable}_{\omega_{\{\omega 1,\omega 3\}}}))$$

Définition 1.23 Transformation de la propriété
Soit une propriété $\varphi_{\mathcal{L}}$ construite sur $\pi(\varphi_{\mathcal{L}})$, la propriété transformée construite sur $\Delta_{\varphi_{\mathcal{L}}}$, est définie en remplaçant toutes les occurrences d'un prédicat paramétré par ω ($\mathtt{p}_\omega$) par l'expression $\mathtt{p}_{\omega_{\theta 1}} \vee \mathtt{p}_{\omega_{\theta 2}} \vee \ldots \vee \mathtt{p}_{\omega_{\theta n}}$ où $\forall i,\ \omega_{\theta i} \in \Delta_{\varphi_{\mathcal{L}}} \wedge \omega \in \theta i$.

Nous notons $\varphi_{\mathcal{L}}^{\mathcal{M}}$ la propriété modifiée. ■

Propriété 1.4 Résultat de préservation
Soient une propriété $\varphi_{\mathcal{L}}$ et un automate $A_{\mathcal{L}} = (Q, q_{init}, T)$, alors

$$A_{\mathcal{L}} \models_l \varphi_{\mathcal{L}} \iff A_{\mathcal{L}} \models_l \varphi_{\mathcal{L}}^{\mathcal{M}}$$ ■

Remarque 1.2 *Etant donné que les points d'observation de $\varphi_{\mathcal{L}}^{\mathcal{M}}$ sont disjoints, le premier résultat de préservation (propriété 1.3) s'applique, d'où :*

$$A_{\mathcal{L}} \models_l \varphi_{\mathcal{L}} \iff A_{\mathcal{L}}^{\rho} \models_l^s \varphi_{\mathcal{L}}^{\mathcal{M}}$$ ■

Démonstration 1.4
La propriété est prouvée si l'équivalence suivante est satisfaite :

$$\forall q \in Q,\ \mathcal{I}(\mathtt{p}_\omega, q) \iff (\mathcal{I}(\mathtt{p}_{\omega_{\theta 1}}, q) \vee \mathcal{I}(\mathtt{p}_{\omega_{\theta 2}}, q) \vee \ldots \vee \mathcal{I}(\mathtt{p}_{\omega_{\theta n}}, q)) \text{ où } \forall i,\ \omega_{\theta i} \in \Delta_{\varphi_{\mathcal{L}}} \wedge \omega \in \theta i$$

Nous effectuons la démonstration uniquement pour le prédicat $\mathtt{enable}_\omega$. *Nous commençons par supposer que* $\mathcal{I}(\mathtt{enable}_\omega, q)$. *D'après l'interprétation du prédicat* $\mathtt{enable}_\omega$, *cela signifie que* $\exists (q, l, q') \in T$ *telle que* $\omega(l)$. *Soit θ l'ensemble image de l'étiquette de cette transition par* $\mathtt{Est_Obs_Par}_{\mathcal{L},\pi(\varphi_{\mathcal{L}})}$ ($\theta = \mathtt{Est_Obs_Par}_{\mathcal{L},\pi(\varphi_{\mathcal{L}})}(l)$), *nous sommes sûrs que* $\theta \in \Delta_{\varphi_{\mathcal{L}}} \wedge \omega \in \theta$. *Par conséquent,* $\mathtt{enable}_{\omega_\theta}$ *fait partie de la disjonction susceptible de remplacer* $\mathtt{enable}_\omega$ *dans la formule. De plus, par définition de θ, l'étiquette l satisfait ω_θ. Cela nous permet de conclure,* $\mathcal{I}(\mathtt{enable}_{\omega_\theta}, q)$, *et l'implication*

$$\mathcal{I}(\mathtt{enable}_\omega, q) \Rightarrow (\mathcal{I}(\mathtt{enable}_{\omega_{\theta 1}}, q) \vee \mathcal{I}(\mathtt{enable}_{\omega_{\theta 2}}, q) \vee \ldots \vee \mathcal{I}(\mathtt{enable}_{\omega_{\theta n}}, q))$$

$$\text{où } \forall i,\ \omega_{\theta i} \in \Delta_{\varphi_{\mathcal{L}}} \wedge \omega \in \theta i$$

Nous prouvons à présent l'implication inverse. Nous supposons qu'il existe un point d'observation $\omega_{\theta i} \in \Delta_{\varphi_{\mathcal{L}}}$ *tel que* $\omega \in \theta i \wedge \mathcal{I}(\mathtt{enable}_{\omega_{\theta i}}, q)$. *D'après l'interprétation du prédicat* `enable` *cela signifie* $\exists (q, l, q') \in T$ *telle que* $\omega_{\theta i}(l)$. *Or* $\omega \in \theta i \wedge \omega_{\theta i}(l) \Rightarrow \omega(l)$. *Ce qui permet de conclure* $\mathcal{I}(\mathtt{enable}_\omega, q)$.

■

1.4 Présentation des techniques de vérification développées autour du langage Esterel

Le travail qui sert de base à cette comparaison est principalement celui présenté dans la thèse de V. Lecompte [Lec89], intitulée “Vérification automatique de programmes Esterel”. Deux

approches de vérification formelle y sont développées et comparées. La première est basée sur l'utilisation d'un outil de vérification de formules d'une logique temporelleΓen l'occurrence CTL. Cet outil se nomme EMC [CES83]. La deuxième est basée sur l'utilisation de l'outil d'analyse comportementale des systèmes de transitions étiquetées nommé Auto [dSV89ΓVer87]. Ce travail se différencie du nôtre par le fait que les modèles des programmes Esterel sont des automates étiquetés par des arbres de décision comprenant des tests du statut d'un signalΓde la valeur d'un signal (les entrées et sorties d'un programme Esterel peuvent être valuées)Γde la valeur d'une variable ; et des actions telles l'émission d'un signalΓl'affectation d'une variable. Cette structure de décision provient directement du programme Esterel. De ces modèles généraux avec variables et valeurs d'événementsΓil est possible d'extraire un squelette de contrôle en ne conservant que les informations booléennes. Seule la satisfaction d'une propriété de sûreté prouvée sur ce squelette de contrôle est préservée par le modèle général. Les transitions du squelette de contrôle d'un programme Esterel sont elles aussi étiquetées par des arbres de décision. Nous commençons par présenter la partie qui concerne la connexion à EMCΓpuis nous présentons la méthode basée sur l'utilisation d'Auto.

1.4.1 La liaison Esterel-EMC

La logique CTL est la logique temporelle utilisée par l'outil EMC. Comme nous l'avons vuΓc'est une logique à base de prédicats d'états. Nous avons montré dans la section 1.3.2 qu'un moyen pour définir ces prédicats d'étatsΓà partir d'un automate dans lequel les états sont anonymes et les étiquettes porteuses d'informationΓconsiste à définir des prédicats du type `after` et `enable`. La vérification d'une propriété s'effectue en deux phases : évaluation des prédicats de base utilisés dans la propriété sur l'ensemble des états de l'automateΓpuis évaluation de la formule. Dans la solution que nous avons présentée — c'est la plus courante —Γla première phase s'effectue à l'aide de prédicats d'étiquettes de la forme `enable` et `after`. Ce n'est pas le choix fait pour définir la liaison Esterel-EMC. En effetΓune première étape d'évaluation des prédicats de base sur les étatsΓspécifique à la forme des modèles des programmes Esterel est effectuée.

Cette première étapeΓdont l'objectif est de transformer un automate étiqueté en un automate avec propriété sur les étatsΓest intimement liée à la forme des étiquettes des modèles de programmes Esterel. Comme nous l'avons déjà signaléΓces étiquettes sont des arbres de décision comportant des tests et des actions d'émission. Par exempleΓl'automate de la figure 1.8 `(a)` (seules les transitions issues de l'état initial sont détaillées) correspond en Esterel à un automate de la forme 1.8 `(b)`. Sur ce dernierΓle point d'interrogation indique que l'on teste la présence d'un signal et le point d'exclamation que l'on émet un signal.

La première phase de définition des propriétés d'états crée des états supplémentaires par rapport à ceux qui sont déjà présents dans le modèle. Un état supplémentaire est créé pour chaque test et pour chaque actionΓprésents dans les étiquettes de l'automate initial. Nous avons indiqué sur la figure 1.8 `(b)` ces nouveaux états par des cercles. Les états de l'automate résultat sont tous étiquetés par des propriétés qui expriment la nature de l'état : état présent dans l'automate initialΓétat correspondant à un test de signal ou à un test de variableΓétat correspondant à une émissionΓ.. Lorsqu'un état correspond à un testΓalors des propriétés supplémentaires lui sont attribuées pour préciser le signal testé. Il en est de même pour les états qui correspondent à des actions. L'introduction d'états supplémentaires dans le modèle ne permet plus de faire correspondre à la notion d'étatΓl'idée intuitive de point d'arrêt entre deux réaction du programme. Il

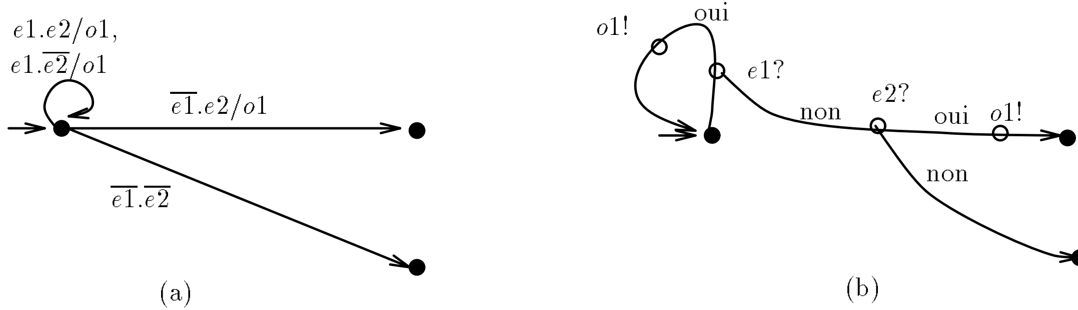


Figure 1.8: forme des modèles de programmes Esterel

s'ensuit que les opérateurs CTLΓtels qu'ils sont classiquement définisΓn'ont plus le sens souhaité. Par exempleΓl'opérateur $\exists X$ ne fait plus référence à la réaction suivanteΓmais à un état successeur qui n'a pas de sens particulier. Pour remédier à ce problèmeΓla sémantique de tous les opérateurs CTL est modifiée de manière à retrouver l'interprétation usuelle de la notion d'état. Cette nouvelle sémantique a été mise en œuvre dans l'outil de vérification EMC.

Les formules qui expriment les propriétés sont basées sur ces nouveaux opérateurs et sur les prédicats de base définis lors de la première phase. L'expression des formules d'une logique temporelle estΓde manière généraleΓun problème complexe. Il nous semble que ce problème est rendu plus délicat encore par l'introduction de ces deux paramètres spécifiques à la vérification des programmes Esterel — définitions spécifiques des prédicats d'états et nouvelle sémantique des opérateurs —.

1.4.2 La liaison Esterel-Auto

Le système Auto permet d'analyser des systèmes de transitions étiquetées. L'idée principale de la méthode de vérification des programmes Esterel consiste à réduire la taille du modèle du système par abstraction de séquences d'actions — séquences d'étiquettes — puis par minimisation de l'automate abstrait. Une fois réduitΓce modèle peut être directement observé ou être comparé automatiquement à une propriété exprimée en termes des séquences d'actions abstraites. Si l'on considère uniquement la possibilité d'abstraction des actionsΓet non pas des séquences d'actionsΓ alors la démarche proposée est similaire à celle que nous venons d'exposerΓmême si les façons de les présenter sont différentes. Par exempleΓAuto ne travaille pas du tout sur la propriétéΓmais sur le modèle du systèmeΓen tenant compte d'un ensemble d'actions abstraites. Ces actions abstraites sont les équivalents des points d'observation dans notre méthode. Ce sont des prédicats sur les étiquettes. On peut par exemple tester la présence d'un signal à l'intérieur d'une étiquette plus complexe. Le modèle du système est modifié en tenant compte de ces directives d'abstraction. Toute étiquette qui satisfait plusieurs actions abstraites — cas des ensembles non disjoints — est dupliquée. Il s'ensuit que la structure du modèle du système à vérifier est modifiée alors que dans la solution que nous avons présentéeΓc'est la structure de la propriété que nous modifions.

Différents critères d'observation sont mis en œuvre dans l'outil Auto. La minimisation d'un automate est intégrée à ces critères d'observation. L'automate réduit peut être visualisé afin de

déterminer si le comportement du programme est correct. A cet effetΓl'outil Autograph [Roy90] a été développé dans l'environnement Esterel. Il propose une interface graphique de placement automatique d'un système de transitions étiquetées. L'automate réduit peut aussi être comparé à un automate exprimant le comportement souhaité du programme en termes d'actions abstraites. Les relations de comparaisonΓpar exemple la simulationΓsont basées sur la comparaison par égalité syntaxique des étiquettes.

Par rapport à la solution que nous avons présentéeΓle système Auto permet de définir des abstractions de séquences d'actionsΓc'est-à-dire de définir des points d'observation d'une séquence d'étiquettes. Cette possibilité permet une abstraction plus importante qu'avec la seule possibilité d'abstraction des actions. Elle n'est pas du tout étudiée dans le cadre formel que nous présentons.

1.5 Récapitulation

Dans ce chapitre nous avons défini un moyen pour utiliser une classe d'outils de vérification importanteΓbasée soit sur la simulation comportementaleΓsoit sur une logique temporelleΓpour la vérification des systèmes réactifs. Nous avons montré que ces outils implémente un cas particulier des relations de satisfaction associées à ces méthodes de vérification. Ces cas particuliers sont plus simples à mettre en œuvre car ils sont basés sur des comparaisons syntaxiques d'étiquettes alors que le cas général nécessite des relations de comparaison plus complexes.

La solution que nous avons proposée est basée sur une transformation systématique de la propriété à vérifier. La propriété ainsi transformée satisfait les conditions d'application des relations de satisfaction simplifiéesΓmises en œuvre par les outils.

L'outil visé n'étant pas modifiéΓnotre méthode permet de choisir celui le mieux adapté à chaque situationΓen fonction de divers critères dont les performances en temps et en mémoireΓ la qualité du diagnostic en cas de non satisfaction de la propriétéΓ...

La transformation de la propriété n'est pas particulière au domaine des systèmes réactifsΓ puisqu'elle est indépendante de l'information portée par les étiquettes. Par conséquentΓelle s'applique à tout autre domaine où les systèmes sont modélisables par des automates et où l'expression intuitive des propriétés conduit à l'utilisation de relations de comparaison autres que l'égalité syntaxique.

Des exemples illustrant l'application de cette méthode sur des systèmes réactifs décrits en Argos sont présentés dans la section 2.3. Les expérimentations effectuées à l'aide des outils Aldébaran — simulation comportementale — et Kronos [Yov93] — évaluateur de la logique TCTL qui est une extension de CTL (cf. chapitre 5) — sont présentées dans le chapitre 7.

Chapitre 2

Argos : un langage impératif de composition d'automates

Après avoir vu comment utiliser une certaine classe d'outils de vérification sur les automates booléens modèles de systèmes réactifsΓnous nous intéressons à présent à un langage de haut niveauΓle langage ArgosΓpermettant de décrire ce type d'automates. De la vérification des systèmes réactifsΓnous passons donc à leur programmationΓsans pour autant nous désintéresser totalement des aspects de vérificationΓpuisque ce chapitre se termine par une illustration des méthodes de preuvesΓprésentées dans le chapitre précédent.

Nous présentons dans ce chapitre le langage Argos tel qu'il est défini dans [Mar92]. C'est le plus récent des langages synchrones. Sa première définition se trouve dans [Mar90]. Nous indiquonsΓà la fin de ce chapitreΓla façon dont cette première définition a évolué pour aboutir à la version que nous présentons. Un exemple de programmation en Argos d'un système réactif non trivial peut être trouvé dans [JM94a].

Ce chapitre est organisé en trois parties :

- La section 2.1 présente de manière informelle le langage Argos ;
- La section 2.2 présente la définition du langage ;
- La section 2.3 contient un exemple de programme Argos et de vérification formelle en utilisant la méthode par observateursΓla simulation comportementale et la logique temporelle CTL.

2.1 Présentation informelle du langage Argos

Le langage Argos est un langage de haut niveau dédié à la description des systèmes réactifs ayant des entrées et des sorties booléennes. Le terme *d'événements* est utilisé pour qualifier les entrées et sorties d'un programme Argos.

La philosophie du langage Argos est la suivante :

- Lorsque le système réactif est suffisamment simple pour être décrit par un automateΓle programme Argos est cet automate ;
- Lorsque le système réactif est complexeΓle programme Argos est construit en identifiant dans ce système des sous-parties suffisamment simples pour être décrites par des automates ; puis en combinant ces sous-programmes à l'aide d'opérateursΓ afin d'obtenir le comportement global souhaité.

Le langage Argos peut donc être vu comme un jeu de constructeurs d'automates.

Deux opérateurs de composition plus un opérateur de communication forment l'ensemble des opérateurs du langage. L'opérateur de *mise en parallèle* permet de décrire l'évolution simultanée de plusieurs sous-programmes. Celui de *raffinement* permet d'introduire dans les programmesΓdes structures de contrôle hiérarchiques. Le dernier opérateur permet quant à lui de faire communiquer des sous-programmes. On le nomme opérateur *de déclaration d'événements internes*Γcar les communications sont établies à l'aide d'événements particuliers qui ne sont ni des entrées ni des sorties du systèmeΓmais des événements locaux définis pour les besoins de la description.

La fonction sémantique définit l'automate décrit par un programme Argos. Si le programme est un programme principal et s'il ne contient pas d'erreursΓalors cet automate est un modèle de système réactif.

La présentation qui suitΓcorrespond à la vision qui fait de l'ensemble des opérateurs d'Argos un jeu de constructeurs d'automates booléens. Nous présentons pour chaque opérateur du langageΓdes programmes simplesΓaccompagnés de leur modèle. De plusΓnous accompagnons cette présentation d'exemples qui prouvent l'adéquation de ces opérateurs à des considérations intuitives sur la construction des systèmes réactifs. En effetΓil est possible de définir un nombre illimité de constructeurs d'automatesΓmais seuls ceux qui correspondent à des utilisations intuitives ont un réel intérêt.

Le langage Argos possède deux syntaxesΓl'une textuelle et l'autre graphique. Seule la syntaxe graphique est ici présentée.

2.1.1 Les automates

Les automates sont les objets de base du langage Argos. Ils servent à décrire les systèmes réactifs simples.

Considérons par exempleΓ la description d'un interrupteur de lampe. L'unique entrée de ce système réactif est l'ordre de commutation de l'interrupteurΓnoté *com*. Ses sorties sont les ordres de mises hors et sous tension de la lampeΓnotés ON et OFF. Une spécification possible du comportement de l'interrupteur est la suivante : la première commutation allume la lampeΓ les suivantes inversent son état physique. Un tel système réactif se décrit très simplement par l'automate de la figure 2.1(a).

L'état L_{off} est l'état initial de l'automate. Par conséquentΓsi l'entrée *com* est présente lors de première réaction du programmeΓla sortie ON est émise et l'état L_{on} devient l'état *actif* de l'automate. Le modèle de ce programme est l'automate de la figure 2.1(b).

Figure 2.1: interrupteur simple

2.1.2 La mise en parallèle

L'opérateur de mise en parallèle correspond à la décomposition d'un système réactif en composantes — ou sous-programmes — qui évoluent simultanément. Par exempleΓun système réactif qui gère deux lampes commandées par deux interrupteurs indépendantsΓpeut être vu comme le résultat de l'évolution simultanée de deux systèmes réactifsΓcontrôlant chacun une de ces lampes. Un tel système se décrit en Argos en utilisant l'opérateur de mise en parallèle.

L'opérateur de mise en parallèle est une forme de produit cartésien d'automates. Deux automates combinés avec cet opérateur sont initialement placés dans leur état initial. A chaque réactionΓils perçoivent les mêmes entréesΓchangent d'état en même temps et unissent leurs sorties.

Considérons l'exemple de la figure 2.2`(a)`. La ligne en pointillé indique que les automates `A` (états $A1$ et $A2$) et `B` (états $B1$ et $B2$) sont mis en parallèle. Le modèle de ce programme est donné par la figure 2.2`(b)`. Chaque transition de ce modèleΓrésulte d'une "union" entre une transition du modèle de `A` et une transition du modèle de `B`.

Par exemple :

- La transition $((A1, B1), e1.e2/s1, s2, (A2, B2))$ s'obtient à partir des transitions :
  - $(A1, e1/s1, A2)$ du modèle de l'automate `A`Γ
  - $(B1, e2/s2, B2)$ du modèle de l'automate `B`.
- La transition $((A1, B1), e1.\overline{e2}/s1, (A2, B1))$ s'obtient à partir des transitions :
  - $(A1, e1/s1, A2)$ du modèle de l'automate `A`Γ
  - $(B1, \overline{e2}, B1)$ du modèle de l'automate `B`.

La description d'un système réactif composé de deux lampes indépendantes s'effectue simplement en composant en parallèle deux automates décrivant chacun une lampe (cf. figure 2.3).

Sur ces deux exemplesΓtoutes les configurations d'états des automates du programme sont accessiblesΓcar les deux composantes sont indépendantes. En particulierΓelles ne partagent aucune entrée. Ce n'est pas toujours le casΓpar exemple le programme de la figure 2.4 `(a)` n'est jamais dans une configuration où $A2$ et $B1$ sont simultanément actifs. Ceci est mis en évidence par le modèle (cf. figure 2.4 `(b)`) du programme.

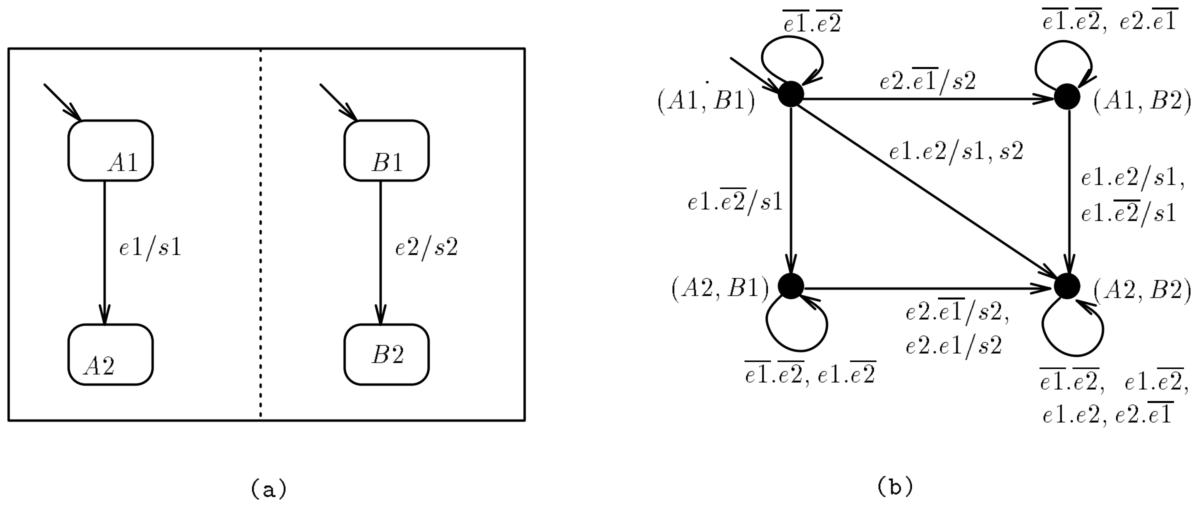


Figure 2.2: mise en parallèle

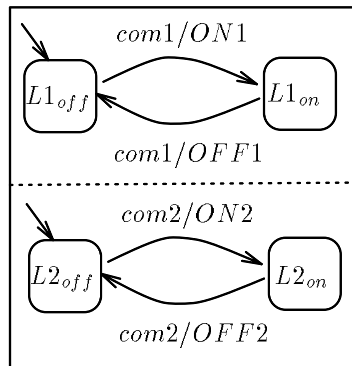


Figure 2.3: double interrupteur

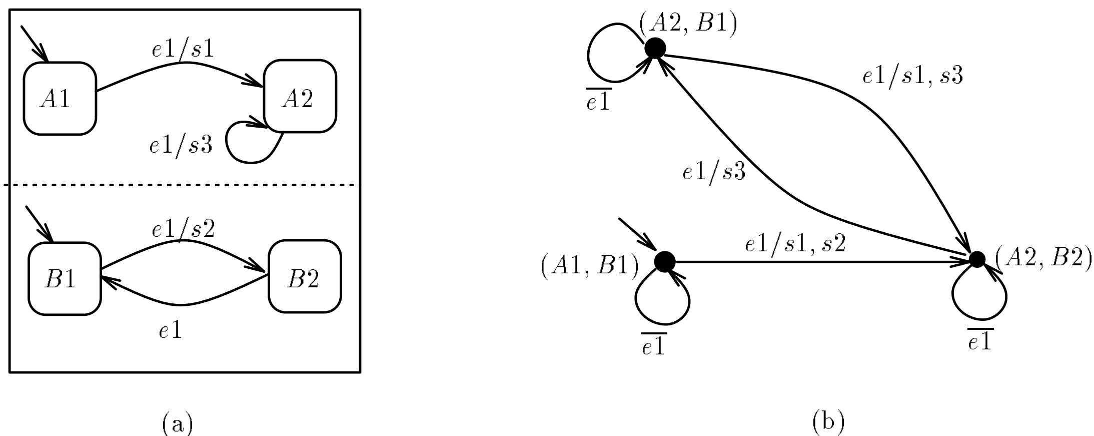


Figure 2.4: deux composantes non indépendantes en parallèle

2.1.3 La déclaration d'événements internes

Définition

L'utilisation d'événements internes permet d'établir des communications entre plusieurs composantes. Dans un même programmeΓun événement interne peut être utilisé à la fois comme une entrée dans une composante et comme sortie dans une autre. C'est de cette manière que la communication entre deux composantes est établie.

Un événement interne n'est pas "visible" à l'extérieur du programmeΓce n'est ni une entréeΓ ni une sortie de ce programme et il n'apparaît pas dans son modèle. Pour connaître le statut d'un événement interne le principe suivant doit être appliqué :

Un événement interne est présent pour une réaction donnée si et seulement s'il est émis lors de cette réaction.

C'est le principe de la communication par diffusion synchrone.

La portée d'un événement interne peut être limitée à une partie du programme. Ceci permet de développer des programmes de manière modulaire.

Pour illustrer l'utilisation de la communication par événements internesΓconsidérons un système réactif composé de deux lampes et d'un seul interrupteur. Les deux premières commutations de l'interrupteur modifient l'état physique de la première lampeΓles deux suivantes celui de la seconde et ainsi de suite. Un programme Argos possible pour décrire ce systèmeΓest donné par la figure 2.5. Les événements $com1$ et $com2$ sont déclarés internes à tout le programmeΓgrâce à un cadre — qui délimite la portée de la déclaration — accompagné d'un cartouche contenant le nom des événements. L'idée consiste à simuler la présence de deux interrupteurs indépendants en ajoutant une troisième composante qui redistribue l'unique ordre de commutation. IntuitivementΓle résultat d'une communication peut être interprété comme une réaction en chaîne. Par exempleΓdans l'état initialΓla présence de com provoque l'émission de $com1$ qui provoque à son tour l'émission de $ON1$.

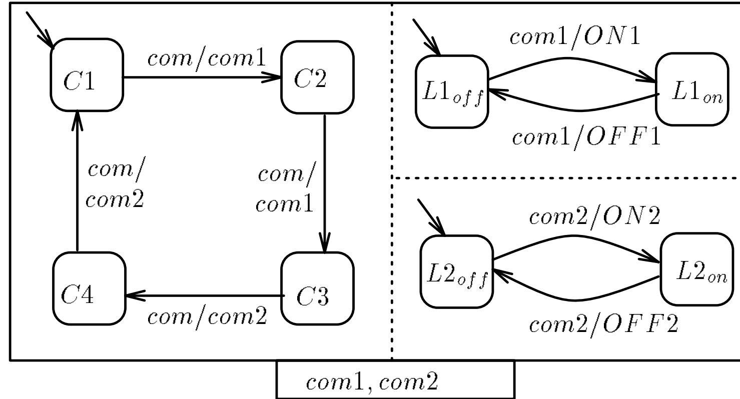


Figure 2.5: interrupteur contrôlant deux lampes

Calcul de la communication par diffusion synchrone

Pour déterminer le statut des événements internes lors de la réaction du programme à une configuration d'entrées donnéesΓdans un état global donnéΓon peut exprimer l'étiquetage des transitions issues de cet état global par un système d'équations booléennesΓdont les variables représentent les événements internes.

Considérons le programme de la figure 2.6(`a`). Les entrées de ce programme sont $e\Gamma f$ et g ; ses sorties sont $s1$ et $s2$. L'événement a est déclaré interne à tout le programme. Supposons que l'on cherche à calculer la réaction de ce programme dans sa configuration initiale. Pour cela le statut de a doit être déterminé. Il apparaît que a est émis dans différentes situations :

- g est présent ;
- e et f sont tous les deux présents ;
- e est absent et f est présent.

Ceci se traduit par l'équation $a = g \ \vee \ (e \wedge f) \ \vee \ (\neg e \wedge f)$. La résolution de cette équation — le statut des entrées $e\Gamma f$ et g donnant les valeurs des variables correspondantes — détermine le statut de a. Connaissant ce statutΓil est ensuite facile de calculer la réaction du programme. On obtient le modèle donné par la figure 2.6(`b`). Notons que dans ce cas très simpleΓle système d'équations a une solution unique. Ce n'est pas toujours le cas.

Remarque 2.1 *Le calcul de la communication par diffusion synchrone à l'aide d'un codage des transitions en système d'équations booléennes, donne en fait la sémantique d'un opérateur unique de mise en parallèle avec synchronisation et encapsulation. Si l'on considère les opérateurs de mise en parallèle et d'encapsulation séparément, il faut donner un sens à la composition parallèle de deux systèmes, lorsqu'un événement est émis par l'un et utilisé en entrée par l'autre, indépendamment de toute encapsulation qui rendrait cet événement interne aux deux systèmes. Nous en donnons l'idée ici. Considérons le programme de la figure 2.7(*`a`*)*. *Le modèle du programme*

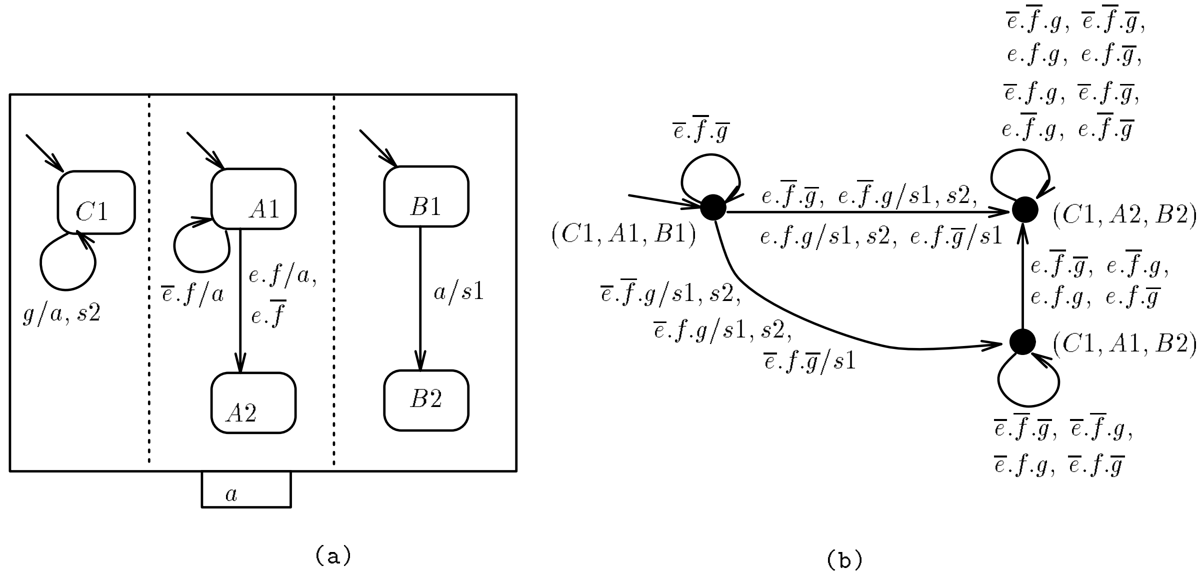


Figure 2.6: exemple de communication

opérande de la déclaration d'événements internes est donné par la figure 2.7(b)*. L'application de la sémantique de l'opérateur de déclaration d'événements internes sur cet automate consiste à rejeter les transitions qui sont en contradiction avec le principe de la diffusion synchrone ; c'est le cas lorsqu'un événement interne est supposé présent dans la configuration d'entrées et qu'il n'est pas émis, ou inversement, lorsqu'un événement interne est supposé absent alors qu'il est émis. Sur les transitions restantes, l'information concernant les événements internes est éliminée. On obtient le modèle donné par la figure 2.7*(c)*. Une généralisation de cette approche est proposée dans [JM94b].* ■

Cas où le système d'équations n'a pas une solution unique, les problèmes de causalité

Lorsque pour un état du programme et une configuration d'entrées donnésΓle système d'équations booléennes n'admet pas de solutionΓon ne sait pas déterminer les statuts des événements internes. On considère que c'est une mauvaise utilisation du mode de communication et on rejette le programme.

Considérons le programme Argos décrit par la figure 2.8(b)Γet calculons ses réactions à partir de la configuration initiale. Le statut de l'événement interne a est défini par l'équation $a = b$Γ celui de b par $b = e \wedge \neg a$. La résolution de ce système en supposant la présence de e n'admet pas de solution.

Le système peut aussi avoir plusieurs solutionsΓauquel cas la réaction du système à la configuration d'entrées donnée peut être considérée non-déterministe. C'est le cas du programme de la figure 2.8(a). On rejette également ce type de programme.

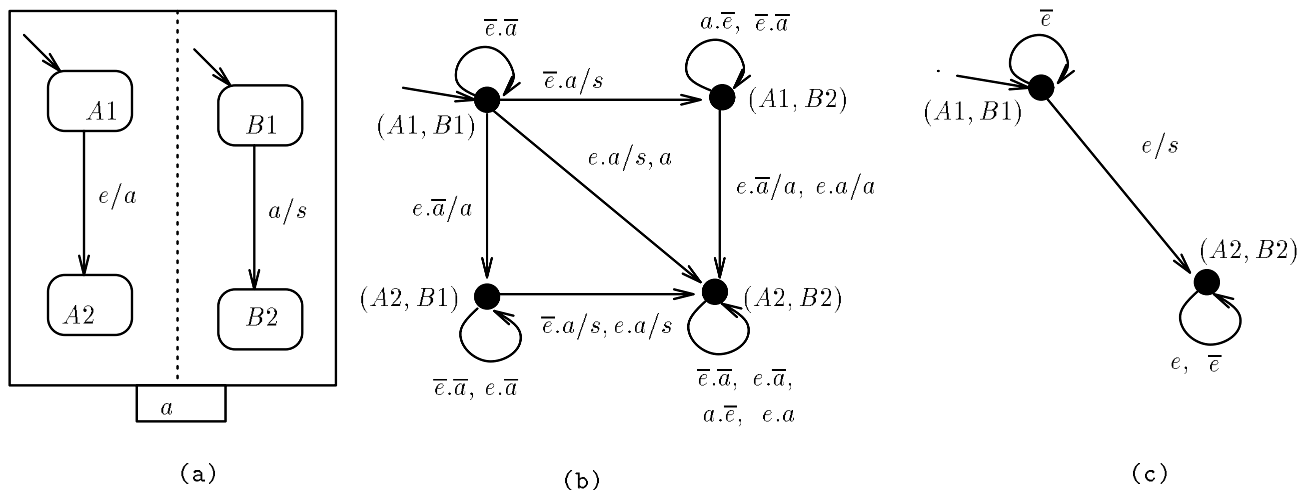


Figure 2.7: sémantique de l'opérateur de déclaration d'événements internes

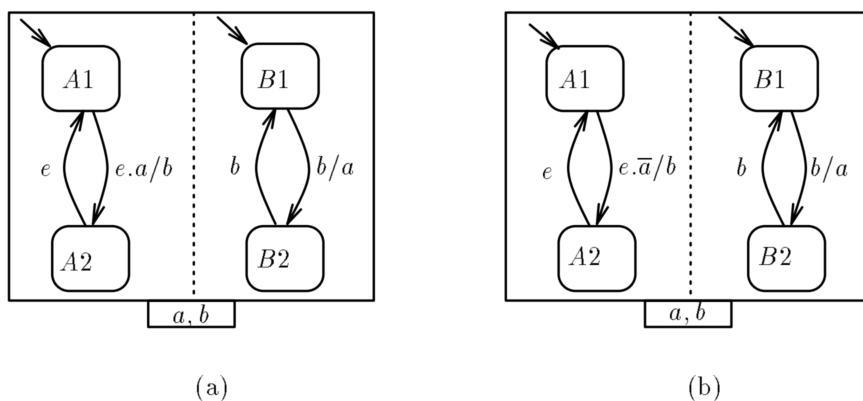


Figure 2.8: problèmes de causalité

2.1.4 L'opérateur de raffinement

Principe général

Lorsqu'on utilise l'opérateur de mise en parallèleΓtoutes les composantes participent à toutes les réactions du programme. L'opérateur de raffinement permetΓluiΓde limiter la participation de certaines composantes à certaines réactions du programme. De faitΓil introduit une structuration hiérarchique des programmes ArgosΓpour traduire par exemple la présence de modes ou de phases dans le système réactif décrit.

L'idée intuitive du fonctionnement de cet opérateur est la suivante : un automateΓappelé *automate de contrôle*Γlance et interrompt des sous-programmes en fonction de ses propres réactions. Plus précisémentΓun sous-programme à contrôler est associé à un des états de l'automate de contrôle. Cette association signifie que l'activité du sous-programmeΓc'est-à-dire sa participation à la réaction du programmeΓest liée à l'activité de cet état : à chaque fois que l'on entre dans l'étatΓle sous-programme est lancéΓà chaque fois que l'on en sortΓle sous-programme est interrompu. Un sous-programme qui est lancé dans la réaction couranteΓparticipe à la réaction suivante. Tout se passe alors comme s'il était placé en parallèle avec l'automate de contrôle. A l'inverseΓun sous-programme interrompu dans la réaction couranteΓne participe pas à la réaction suivanteΓtout se passe alors comme s'il n'existait pas.

Nous présentons un premier exemple d'utilisation de l'opérateur de raffinement dans lequel un sous-programme est interrompu par la présence d'une entrée du programme. C'est une "vraie" interruptionΓpar opposition aux cas :

- De terminaison normale : détection de la fin d'une phase ;
- De terminaison sur erreur : détection d'une erreur dans le déroulement d'une phase ;

dans lesquels la cause de l'interruption est interne au sous-programme.

La prise en compte des interruptions

Considérons le programme de la figure `2.9(a)`. L'automate de contrôle est l'automate `C`. Le fait que l'automate `A` soit placé à l'intérieur de l'état $C1$ indique qu'il est associé à cet état. On dit que l'automate `A` *raffine* l'état $C1$. Aucun sous-programme n'est associé à l'état $C2$. Dans la configuration initiale du programmeΓl'état $C1$ est actif puisqu'il est l'état initial de l'automate de contrôle. Par conséquent l'automate `A` participe à la première réaction du programme et il est par définition placé dans son état initial. Tout se passe alors comme si les deux automates étaient placés en parallèle. Dès que e est présentΓla participation de l'automate `A` est interrompue. Elle reprendra à la prochaine occurrence de e. Celle-ci place de nouveau l'automate `A` dans son état initial.

L'automate modèle de ce programme est donné par la figure `2.9(b)`. On peut distinguer parmi ces réactions deux classes de situations :

- Les situations où l'automate de contrôle et l'automate `A` évoluent simultanément. C'est le cas de la transition $((C1, A1), e.e1/s1, (C2))$. Remarquons que les sorties émises par `A` sont prises en compteΓmême si celui-ci est interrompu par la réaction ;

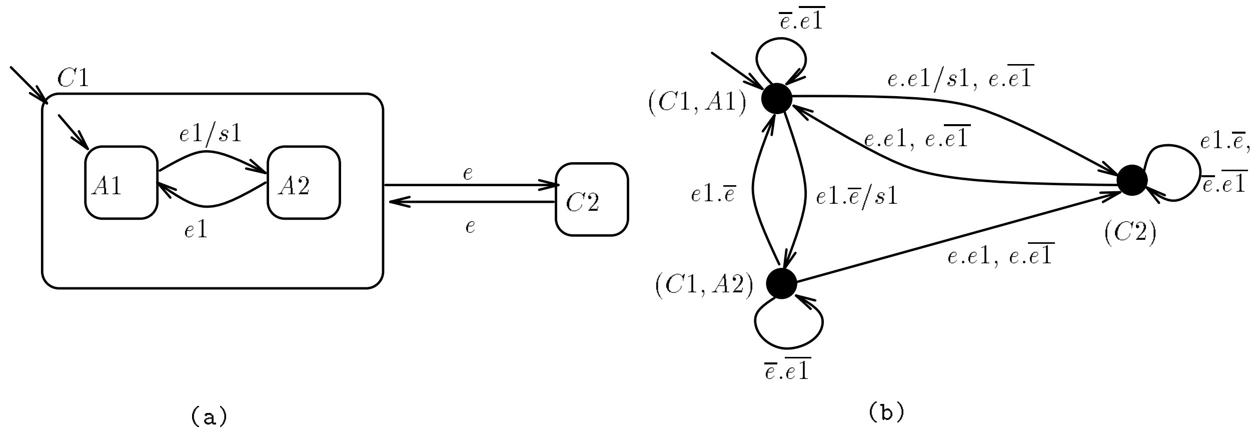


Figure 2.9: terminaison par interruption

- Les situations où seul l'automate `A` évolue. C'est le casΓ de la transition $((C1, A1), e1.\overline{e}/s1, (C1, A2))$.

Terminaison normale ou sur erreur

Que ce soit pour décrire une terminaison normale ou sur erreurΓil est nécessaire de faire communiquer la composante qui raffine un état et l'automate de contrôle. Cette communication peut être vue comme une demande d'interruption de la composante à l'automate de contrôle. Dans le programme de la figure 2.10 `(a)`Γc'est l'événement interne fin qui exprime cet ordreΓlors de la réception de la deuxième occurrence de $e1$. Le modèle de ce programme est donné par la figure 2.10 `(b)`.

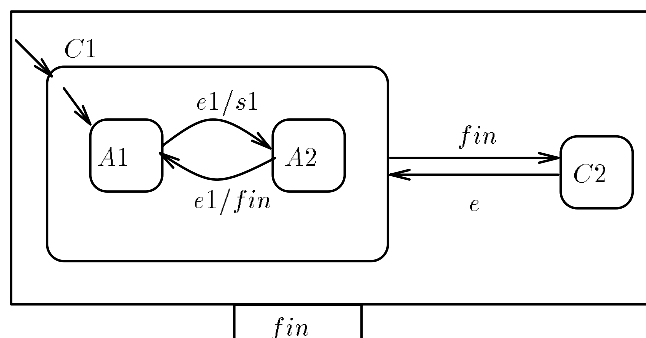
Remarquons que le type de communication établi entre la composante qui raffine un état et l'automate de contrôle n'est possible que si les sorties émises par la composante sont prises en compteΓmême au moment où la composante est interrompue.

La réinitialisation d'une composante

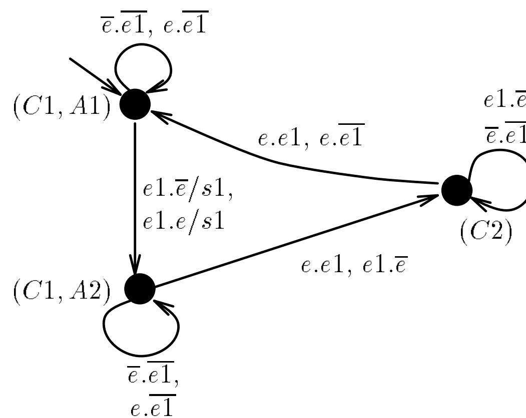
Il est possible à l'aide de l'opérateur de raffinementΓde réinitialiser un sous-programmeΓce qui signifie le relancer dans son état initial. Cette réinitialisation s'exprime dans l'automate de contrôleΓen explicitant une transition qui boucle sur l'état raffiné.

Supposons que l'automate `A` des exemples qui précèdentΓest lancé sur la première occurrence de l'entrée e (l'état $C2$ devient initial)Γpuis réinitialisé à chaque occurrence de e. Cette réinitialisation se décrit en Argos comme indiqué par la figure 2.11`(a)`. Le modèle de ce programme est donné en 2.11`(b)`.

Pour les états non raffinésΓune boucleΓsi elle n'est pas accompagnée de l'émission de sortiesΓ peut être omise sans que cela modifie le programme. Par contreΓune boucle sur un état raffinéΓ

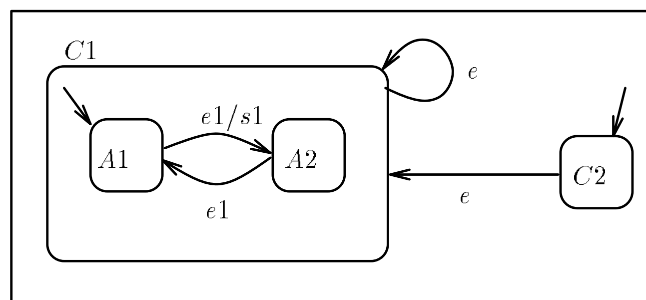


(a)

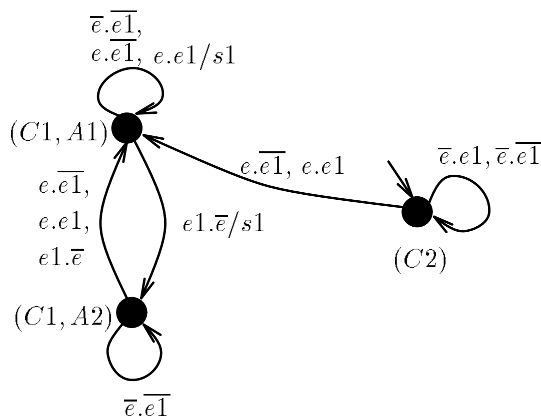


(b)

Figure 2.10: terminaison normale



(a)



(b)

Figure 2.11: réinitialisation d'un sous-programme

même si elle n'est pas accompagnée de l'émission de sortieΓest significative pour le comportement du programmeΓpuisqu'elle indique une réinitialisation.

Une utilisation atypique du raffinement : le dialogue instantané

Le terme de dialogue instantané est utilisé lorsqu'une composanteΓpour déterminer sa réaction couranteΓdemande des informations à une autre composante du programme. C'est un dialogue car il y a interrogation de la part d'une composanteΓet réponse de l'autre ; il est instantané car tout se passe dans la même réaction du programme.

Considérons le système formé d'une lampeΓd'un interrupteurΓet d'un bouton à deux positions A et B. Sa spécification est la suivante : une commutation de l'interrupteur n'a d'effet sur la lampe que si le bouton est déjà en position B lorsque l'interrupteur est manipulé. Nous appelons ce système un interrupteur conditionné. Ses entrées sont les ordres de commutation reliés à l'interrupteur (com_L) et au bouton (com_B). Ses sorties sont les ordres d'extinction (OFF) et d'allumage (ON) de la lampe. InitialementΓle bouton est en position A et la lampe est éteinte.

Le programme Argos de la figure 2.12 décrit ce système. Il contient deux composantes mises en parallèleΓl'une gère la lampe (c'est celle du dessus)Γl'autre le bouton. Lorsque la composante qui gère la lampe reçoit un ordre de commutationΓelle ne peut décider de son effet sans connaître la position du bouton. Or cette information est connue par l'autre composante ; il faut donc établir un dialogue entre ces deux composantes. Ce dialogue s'effectue de la manière suivante : lorsque la composante qui gère la lampe reçoit l'entrée com_LΓelle interroge la composante associée au boutonΓpour connaître la position de ce dernier. L'événement interne $B?$ traduit cette interrogation. Si le bouton est en position BΓl'état B est actif et l'événement interne $B!$ est émis — même si le bouton change de position au même instant —. La présence de $B!$ provoque la réaction de la composante qui gère la lampe : émission d'un ordre d'allumage ou d'extinction et changement d'état.

Le dialogue entre les deux composantes est instantané puisque dans la même réaction com_L est reçuΓ$B?$ émis et reçuΓ$B!$ émis et reçuΓet enfin ON émis. Parmi cette succession d'actionsΓ seules la réception de com_L et l'émission de ON sont visibles à l'extérieur du programme puisque les autres actions concernent des événements internes.

Il existe d'autres solutions pour exprimer une communication de type dialogue instantané entre deux composantes. Par exempleΓl'utilisation du raffinement n'est pas essentielle (cf. figure 2.13). Son seul intérêt est de rendre le comportement du programme plus intuitif.

Ce système peut aussi être décrit en Argos sans utiliser le mécanisme du dialogue instantané (cf. figure 2.14). L'idée est la suivante : la composante qui gère le bouton émet à chaque réaction où la position B est activeΓun événement interne particulier ; la composante qui gère la lampe utilise cette information lorsqu'elle en a besoin. Ce n'est plus une situation de dialogueΓmais plutôt un “monologue”.

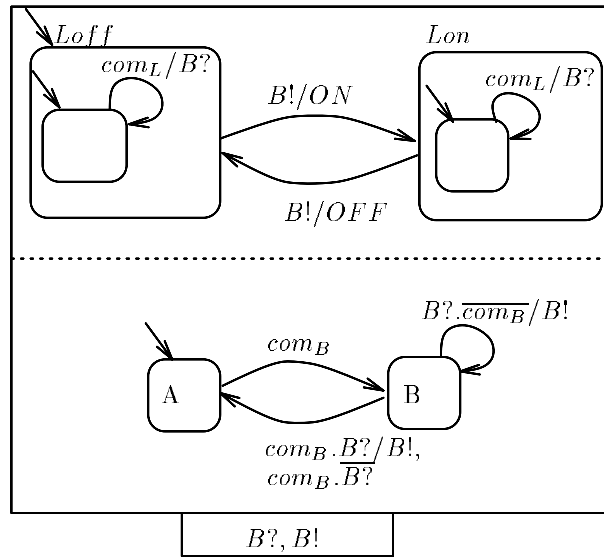


Figure 2.12: exemple de dialogue instantané

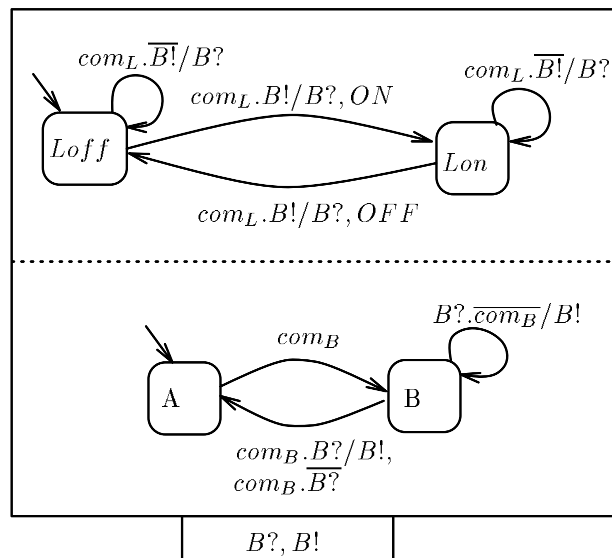


Figure 2.13: exemple de dialogue instantané sans raffinement

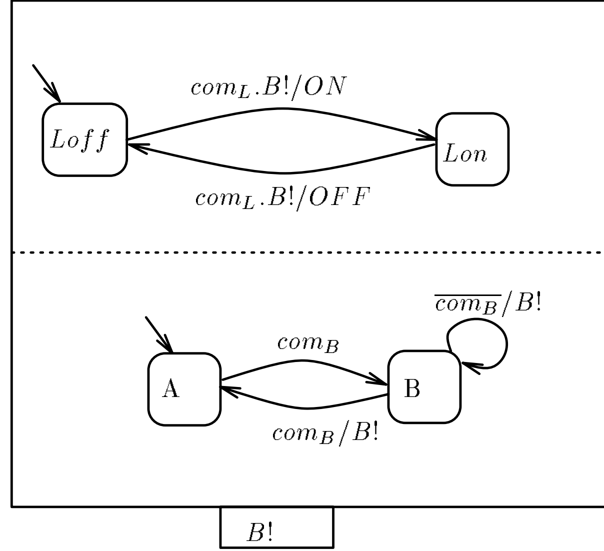


Figure 2.14: remplacement du dialogue par un "monologue"

2.2 Définition d'Argos

Le langage est formellement défini par la fonction sémantique qui associe à tout programme ArgosΓ son modèle. Cette fonction est présentée de manière opérationnelle (sous forme algorithmique).

2.2.1 Syntaxe abstraite des programmes Argos

La forme syntaxique des programmes Argos est définie par la grammaire suivante :

Définition 2.1 Syntaxe abstraite des programmes Argos

P	$::=$	$P \| P$		*mise en parallèle*
	$\mid$	$\overline{P^Y}$		*déclaration d'événements internes*
	$\mid$	$\mathbf{R}_A(R_1, \ldots, R_n)$	*avec* $A \in \mathcal{A}_d^{\emptyset,c}$	*raffinement paramétré par* A
R	$::=$	NIL $\mid P$		

■

L'expression $\overline{P^Y}$ signifie que les événements contenus dans Y sont déclarés internes au programme P. $\mathbf{R}_A(R_1, \ldots, R_n)$ indique que l'état d'indice i de l'automate de contrôle A est raffiné par le sous-programme R_i. Cet opérateur admet autant d'opérandes que l'automate de contrôle admet d'états. NIL dénote le "programme nul" qui ne possède ni entrée ni sortie. Il permet d'indiquer qu'un état n'est pas raffiné. Plus exactementΓdans cette syntaxeΓun état non raffiné est un état raffiné par NIL. Par conséquentΓles automates apparaissant uniquement à travers

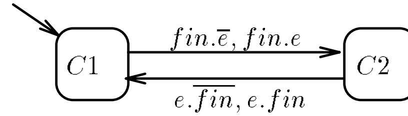


Figure 2.15: automate C avec des monômes complets

l'opérateur de raffinementΓun automate dont tous les états sont non raffinésΓest le paramètre d'un opérateur de raffinement dont tous les opérandes sont égaux à NIL. Les automates paramètres des opérateurs de raffinement utilisent des monômes complets et non contradictoires sur l'ensemble de leurs entrées. Leurs ensembles d'entrées et de sorties sont disjoints. De plusΓce sont des automates déterministes.

L'ensemble des programmes Argos corrects du point de vue syntaxique est noté $\mathcal{P}$. Avant de définir le sous-ensemble de $\mathcal{P}$ des programmes principauxΓnous précisons quelques éléments de syntaxe concrète.

2.2.2 Eléments de syntaxe concrète

Simplification de l'écriture des automates

La simplification présentée dans ce paragraphe est le résultat des expérimentations effectuées avec le langage Argos. D'autres peuvent être définies.

Dans la syntaxe abstraite des programmes ArgosΓles monômes d'événements qui expriment les conditions de franchissement des transitions sontΓpar définitionΓcomplets et non contradictoires sur l'ensemble de toutes les entrées de l'automate. De plus les automates utilisés dans le programme sont déterministes.

Le fait que dans cette syntaxe abstraiteΓles monômes des étiquettes soient completsΓsimplifie l'écriture de la sémantique du langage. CependantΓcela alourdit considérablement l'écriture des automatesΓcar dans de nombreux casΓtous les événements ne sont pas pertinents dans tous les états. Une simplification possible consiste à autoriser des monômes incomplets. L'absence d'un événement dans un monômeΓindique que la transition est déclenchéeΓqu'il soit présent ou absent.

Cette simplification a été utilisée dans la présentation informelle du langage ArgosΓnotamment dans la figure 2.10. En effetΓl'ensemble des entrées de l'automate C est $\{e, fin\}$. Il s'ensuit que les monômes apparaissant dans cet automate devraient être complets sur cet ensembleΓ comme l'indique la figure 2.15. Il ne le sont pas car e n'est pas pertinent pour l'état $C1$Γet inversement l'événement fin ne l'est pas pour $C2$.

La phase qui consiste à compléter les monômes (et donc à dupliquer des transitions) est appelée *phase de saturation*. La vérification du déterminisme des automates du programmeΓ s'effectue une fois la phase de saturation effectuée.

Dans la suiteΓtous les programmes Argos sont donnés en utilisant cette simplification de l'écriture des monômes.

La réutilisation de composantes (les procédures)

Il est possible en Argos de réutiliser des composantes grâce à un mécanisme d'appel de procédure. Une procédure Argos est composée d'un nomΓd'une liste d'événements d'entréesΓd'une liste d'événements de sorties (listes des paramètres formels de la procédure) et d'un corps qui est un programme Argos. Un appel de procédure est constitué d'un nomΓd'une liste d'événements d'entrées et d'une liste d'événements de sorties (liste des paramètres effectifs). Une phase d'expansion des procédures recopie chaque appel par le corps de la procédure appelée en renommant les paramètres formels par les paramètres effectifs.

2.2.3 Programmes principaux

Un programme Argos est un programme principal si et seulement si tous les événements utilisés à la fois comme des entrées par certaines composantes et comme des sorties par d'autresΓsont déclarés internes au programme. Seuls les programmes principaux décrivent des modèles de systèmes réactifsΓpuisque les modèles des autres programmes ne vérifient pas la propriété de non recouvrement des entrées et des sorties (exemple de la figure 2.7).

Définition 2.2 Ensemble des programmes principaux
Un programme correct syntaxiquement est un programme principal s'il n'existe aucun événement utilisé à la fois en entrée et en sortie, c'est-à-dire :

$$\forall P \in \mathcal{P},\ P \in \mathcal{P}_{princ} - \mathcal{I}(P) \cap \mathcal{O}(P) = \emptyset \quad \blacksquare$$

Nous notons $\mathcal{P}_{princ}$ le sous-ensemble des programmes principaux.

Les fonctions $\mathcal{I}$ et $\mathcal{O}$ calculent respectivement l'ensemble des entrées et l'ensemble des sorties d'un programme (définition ci-dessous). Les événements internes du programme ne font pas partie de ces ensembles.

Définition 2.3 Ensemble des entrées d'un programme

$$\begin{array}{lcl}
\mathcal{I}(P_1 \| P_2) & = & \mathcal{I}(P_1) \cup \mathcal{I}(P_2) \\
\mathcal{I}(\mathbf{R}_{(Q,q_{init},T)}(R_1,\ldots,R_n)) & = & In \cup \bigcup_{i=1}^{i=n} \mathcal{I}(R_i) \\
\mathcal{I}(\overline{P^Y}) & = & \mathcal{I}(P) - Y \\
\mathcal{I}(\textsc{nil}) & = & \emptyset
\end{array}$$

$\blacksquare$

Définition 2.4 Ensemble des sorties d'un programme

$$\begin{array}{lcl}
\mathcal{O}(P_1 \| P_2) & = & \mathcal{O}(P_1) \cup \mathcal{O}(P_2) \\
\mathcal{O}(\mathbf{R}_{(Q,q_{init},T)}(R_1,\ldots,R_n)) & = & Out \cup \bigcup_{i=1}^{i=n} \mathcal{O}(R_i) \\
\mathcal{O}(\overline{P^Y}) & = & \mathcal{O}(P) - Y \\
\mathcal{O}(\textsc{nil}) & = & \emptyset
\end{array}$$

$\blacksquare$

2.2.4 Calcul du modèle d'un programme

Présentation de la fonction sémantique

Le rôle de la fonction sémantique, notée $\mathcal{S}$, est d'associer à tout programme Argos syntaxiquement correct, l'automate qu'il détermine : son modèle. Le profil de cette fonction est le suivant :

$$\mathcal{S} : \mathcal{P} \;\rightarrow\; \mathcal{A}^{c}_{dr} \cup \{\text{ERROR}\}$$

La valeur d'erreur signifie que le programme contient un problème de causalité. Si le programme est un programme principal qui ne contient pas de problème de causalité, alors son modèle appartient à $\mathcal{A}^{\emptyset,c}_{dr}$, c'est un modèle de système réactif.

La fonction sémantique est dirigée par la syntaxe :

$$\mathcal{S}(P \;\mathsf{op}\; P') = \mathcal{F}_{\mathsf{op}}(\mathcal{S}(P), \mathcal{S}(P'))$$

où op est un opérateur binaire (le même type d'égalité existe pour un opérateur unaire). La fonction $\mathcal{F}_{\mathsf{op}}$ exprime la sémantique de chaque opérateur.

Comme il est montré dans [Mar92], cette caractéristique permet d'établir un certain nombre de bonnes propriétés de la sémantique, dont l'existence d'une relation d'équivalence entre programmes qui est une congruence pour les opérateurs du langage.

Mise en œuvre de la fonction sémantique

Etant donné un programme P, la fonction sémantique $\mathcal{S}$ doit définir : l'ensemble des états, l'état initial et l'ensemble des transitions du modèle de P.

Les configurations d'un programme Argos sont des configurations d'états des automates actifs du programme. La forme syntaxique d'un état du modèle doit donc permettre de retrouver deux informations : ensemble des automates actifs et ensemble des états courants des automates actifs. Pour atteindre ces deux objectifs, nous avons choisi de désigner un état du modèle d'un programme P par la structure de P, dans laquelle une information supplémentaire a été ajoutée aux triplets qui définissent les automates du programme. Cette information est une valeur de l'ensemble des états de l'automate ; elle indique l'état courant de l'automate. A partir de cette forme syntaxique, il est possible de retrouver l'ensemble des automates actifs du programme, par un parcours de la structure à partir de la racine, ainsi que l'ensemble des états courants de ces automates.

Définition 2.5 Forme syntaxique des états du modèle d'un programme

$$\begin{array}{lcll} p & ::= & p \| p & \\ & | & \overline{p^{Y}} & \\ & | & \mathbf{R}_{(Q,q_{init},q_c,T)}(r_1,\ldots,r_n) & \textit{où } q_c \in Q \textit{ est l'état courant de l'automate} \\ r & ::= & nil | p & \end{array}$$

■

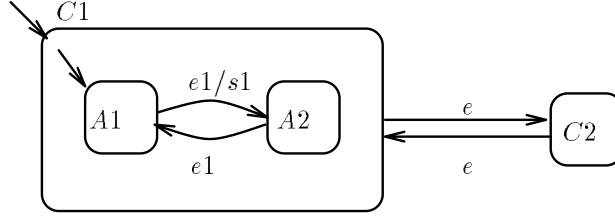


Figure 2.16: états du modèle d'un programme

Cette définition présente l'inconvénient d'associer plusieurs états du modèle à une configuration du programme. En effetΓdans celle-ciΓseuls les états courants des automates actifs sont pertinents. Par conséquent tous les états du modèle qui définissent le même ensemble d'automates actifs et donnent à ces automates des états courants identiquesΓreprésentent la même configuration du programme. Par exempleΓdans la figure 2.16 les expressions (les ensembles d'états et de transitions ne sont pas détaillés) :

- $\mathbf{R}_{(Q_C,C1,C2,T_C)}(\mathbf{R}_{(Q_A,A1,A1,T_A)}(nil, nil), nil)$
- $\mathbf{R}_{(Q_C,C1,C2,T_C)}(\mathbf{R}_{(Q_A,A1,A2,T_A)}(nil, nil), nil)$

qui diffèrent par l'état courant de l'automate `A` — $A1$ dans un casΓ$A2$ dans l'autre — désignent toutes les deux la configuration du programme où l'état $C2$ est actif. Nous définissons la forme normale de ces états comme étant celle dont l'état courant de chaque automate inactif est son état initial. Sur l'exemple précédentΓla première des deux expressions est la forme normale.

L'ensemble des états potentiels du modèle d'un programme PΓest donné par l'ensemble des états qui ont la même structure que P. Parmi ces états potentielsΓle modèle ne retient que les états accessibles à partir de l'état initial.

L'état initial du modèle de P est calculé par la fonction *init*. Il s'obtient en plaçant chaque automate dans son état initial .

Définition 2.6 Calcul de l'état initial du modèle

$$\begin{array}{lcl}
init(\textsc{nil}) & = & nil \\
init(P_1 \| P_2) & = & init(P_1) \| init(P_2) \\
init(\overline{P^Y}) & = & \overline{init(P)}^Y \\
init(\mathbf{R}_{(Q,q_{init},T)}(R_1, \ldots, R_n)) & = & \mathbf{R}_{(Q,q_{init},q_{init},T)}(init(R_1), \ldots, init(R_2))
\end{array}$$

■

L'algorithme général de calcul des transitions du modèle de P est donné ci-dessousΓil permet aussi de définir l'ensemble des états du modèle.

Algorithme 2.1 Le calcul des transitions du modèle d'un programme P

Trans $\leftarrow \emptyset$ {ensemble des transitions}
$I \leftarrow \mathcal{I}(P)$ {ensemble des entrées}
Access $\leftarrow \{init(P)\}$ {ensemble des états accessibles}
A_Explorer $\leftarrow \{init(P)\}$ {ensemble des états accessibles encore à explorer }
Tant que (A_Explorer $\neq \emptyset$) faire
    $p \leftarrow$ Choisir_etat(A_Explorer) {état source}
    A_Explorer $\leftarrow$ A_Explorer $\setminus p$
    Pour chaque $m_e \in \mathcal{M}_c^\star(I)$ faire {monôme déclencheur}
        Si (CalculTrans(p, m_e).resultat = OK) faire
            $O \leftarrow$ CalculTrans(p,m_e).sorties {ensemble des sorties}
            $p_b \leftarrow$ CalculTrans(p,m_e).but {état but}
            Trans $\leftarrow$ Trans $\cup$ $\{(\mathrm{p}, m_e, O, p_b)\}$
            A_Explorer $\leftarrow$ A_Explorer $\cup$ $\{p_b\}$
            Access $\leftarrow$ Access $\cup$ $\{p_b\}$
        Sinon
            Exit(erreur)

■

Le cœur de cet algorithme se situe dans la fonction `CalculTrans` quiΓà partir d'un état p du modèle et d'un monôme d'événements m_e complet et non contradictoire sur l'ensemble des entrées du programmeΓcalcule la transition issue de pΓdéclenchée par m_e. Ce calcul peut échouer. La seule cause de cet échec est la détection d'un problème de causalité dans le programmeΓce qui signifie que pour une réaction donnéeΓil n'est pas possible de définir le statut d'un événement interne de manière unique. Cette situation suffit à stopper le calcul des transitions du modèle. Le programme P dans ce cas est incorrect du point de vue sémantique. Le résultat de $\mathcal{S}$ sur de tels programmes est la valeur spéciale ERROR.

Lorsque le calcul d'une transition réussitΓl'ensemble des événements générés par la transitionΓ ainsi que son état butΓsont calculés par la fonction `CalculTrans`. Nous pouvons alors passer au calcul d'une autre transitionΓsoit en modifiant le monôme d'événementsΓsoit en changeant d'état source. Il existe nécessairement un nombre fini de transitionsΓcar d'une part le nombre d'états potentiels du modèle d'un programme est fini et d'autre part le nombre de transitions issues de chaque état est finiΓpuisqu'égal au nombre de monômes complets et non contradictoires sur l'ensemble des entrées du programme.

Nous présentons à présent en détail la fonction `CalculTrans` qui calcule l'ensemble des transitions du modèle.

La fonction `CalculTrans` détermine si la réaction du programme à un monôme m_e dans un état p est correctement définie — il n'y a pas de problème de causalité —. Si c'est le casΓelle définit cette réaction : l'ensemble des événements qu'elle génère et la nouvelle configuration du programme. La fonction est définie par un ensemble de règles opérationnelles de la forme :

$$\frac{Condition}{p \xrightarrow{m_e/O} p'} \qquad \text{[Exemple]}$$

Une telle règle doit s'interpréter comme suit : si la *Condition* est vérifiée alors la transition issue de pΓdéclenchée par m_e existeΓest uniqueΓadmet O comme ensemble d'événements générés et p' comme état but. Pour vérifier simplement les problèmes d'unicité dans le calcul d'une transitionΓl'ensemble des règles est telΓque les conditions d'application d'une règle sont deux à deux exclusives. Par conséquentΓla fonction `CalculTrans` échoue si et seulement si aucune des règles n'est applicable.

Ces règles utilisent une fonction quiΓétant donné l'état p d'un modèleΓpermet de définir le programme P tel que p fait partie des états du modèle de P. Cette fonction est notée *prog*(p). Elle consiste simplement à "oublier" l'information donnée par les états courants pour ne conserver que la structure du programme.

Définition 2.7 Calcul du programme dont p est un état du modèle

$$
\begin{array}{lcl}
prog(p_1 \| p_2) & = & prog(p_1) \| prog(p_2) \\
prog(\mathbf{R}_{(Q,q_{init},q_c,T)}(r_1,\ldots,r_n)) & = & \mathbf{R}_{(Q,q_{init},T)}(prog(r_1),\ldots,prog(r_n)) \\
prog(\overline{p^Y}) & = & \overline{prog(p)^Y} \\
prog(nil) & = & \text{NIL}
\end{array}
$$

■

Chaque règle de calcul des transitions tient compte de la structure de l'état source (type de l'opérateur principal). Elles sont présentées en fonction de ce type. Les règles sont inductives : des conditions portent sur l'ensemble de transitions des modèles des opérandes. Une condition de la forme $p \xrightarrow{m_e/O} p'$ est vérifiée s'il existe dans le modèle de *prog*(p) une unique transition issue de p déclenchée par m_e qui génère O et atteint p'. Les monômes des transitions de ce modèle sont construits sur l'ensemble des entrées du sous-programme *prog*(p). Cet ensemble pouvant être inclus dans l'ensemble des entrées du programmeΓnous utilisons sur les monômes une fonction de *restriction*Γdéfinie ci-dessous. Cette fonction permet de restreindre le monôme à un ensemble d'événements donné.

Définition 2.8 Restriction d'un monôme à un sous-ensemble de E
Soit $m_e \in \mathcal{M}(E)$ et F un sous-ensemble de E, $m_e[F] = (m_e{}^+ \cap F, m_e{}^- \cap F)$ est le monôme obtenu par restriction de m_e à F. ■

La propriété suivante se vérifie facilement : $m_e \in \mathcal{M}_c^\star(E) \implies m_e[F] \in \mathcal{M}_c^\star(F)$.

La mise en parallèle

Soient $I_i = \mathcal{I}(prog(p_i))$ pour $i \in \{1,2\}$;

$$
\frac{p_1 \xrightarrow{m_e[I_1]/O_1} p_1' \qquad p_2 \xrightarrow{m_e[I_2]/O_2} p_2'}{p_1 \| p_2 \xrightarrow{m_e/O_1 \cup O_2} p_1' \| p_2'} \qquad \text{[Par]}
$$

Une transition issue d'une mise en parallèle de deux composantes est la combinaison de deux transitionsΓchacune issue des modèles des composantes. Cette combinaison s'exprime par l'union des ensembles de sorties et la mise en parallèle des états buts.

Le raffinement

Soient $R_i = prog(r_i) \ \forall i \in [1..n]$ et $I_c = \mathcal{I}(R_c)$;

$$\frac{\exists (q_c, m_e[In]/O, q_d) \in T \qquad r_c \xrightarrow{m_e[I_c]/O_c} r'_c}{\mathbf{R}_{(Q,q_{init},q_c,T)}(r_1,\ldots,r_n) \xrightarrow{m_e/O \cup O_c} \mathbf{R}_{(Q,q_{init},q_d,T)}(r_1,\ldots,init(R_c),\ldots,r_n)} \quad \text{[Raff1]}$$

$$\frac{\forall O, q_d,\ \nexists (q_c, m_e[In]/O, q_d) \in T \qquad r_c \xrightarrow{m_e[I_c]/O_c} r'_c}{\mathbf{R}_{(Q,q_{init},q_c,T)}(r_1,\ldots,r_n) \xrightarrow{m_e/O_c} \mathbf{R}_{(Q,q_{init},q_c,T)}(r_1,\ldots,r'_c,\ldots,r_n)} \quad \text{[Raff2]}$$

Pour que ces règles soient complètement définiesΓil faut prévoir le cas où le sous-programme raffinant un état est égal à NIL.

$$\frac{}{nil \xrightarrow{\emptyset/\emptyset} nil} \quad \text{[Nil]}$$

La règle [Raff1] s'applique lorsque le monôme d'événements engendre une réaction dans l'automate de contrôle. Dans le cas contraireΓc'est [Raff2] qui s'applique.

Lorsqu'il y a évolution simultanée de l'automate de contrôle et du sous-programme raffinant l'état courant de l'automate (règle [Raff1])Γla réaction de ce dernier influence uniquement la construction de l'ensemble des sorties de la transition globale. Dans l'état but de cette transitionΓ le sous-programme qui raffine l'état quitté de l'automate de contrôle — état q_c — est réinitialisé. En effetΓtous les automates qu'il contient deviennent inactifs. Le sous-programme qui raffine l'état atteint de l'automate de contrôle est par construction déjà dans son état initialΓil n'a donc pas besoin d'être modifié. Si ces deux états sont identiques (la transition de l'automate de contrôle est une boucle) alors la réinitialisation du sous-programme R_c exprime bien la réaction souhaitée.

Lorsque seul le sous-programme raffinant l'état courant de l'automate réagit (règle [Raff2])Γ la réaction globale est égale à celle du sous-programme.

La déclaration d'événements internes

Soient $P = prog(p)$ et $I = \mathcal{I}(P)$;

$$\frac{\begin{array}{l}\forall m_Y \in \mathcal{M}_c^\star(Y \cap I), p \xrightarrow{m_e \wedge m_Y/O_i} p_i \\ \exists!\ m_Y \in \mathcal{M}_c^\star(Y \cap I)\ telle\ que\ \ p \xrightarrow{m_e \wedge m_Y/O_i} p_i \quad {m_Y}^+ \subseteq O_i \quad {m_Y}^- \cap O_i = \emptyset\end{array}}{\overline{p^Y} \xrightarrow{m_e/O_i - Y} \overline{{p_i}^Y}} \quad \text{[Int]}$$

L'idée intuitive de cette règle est la suivante. Si on complète le monôme m_e par une configuration quelconque des statuts des événements internes (représentée par m_Y)Γalors la transition issue de p déclenchée par le monôme résultat doit nécessairement exister (première partie de la

condition). Autrement ditΓcette première condition permet de vérifier qu'il n'y a pas de problème de causalité à l'intérieur du programme opérande de la déclaration d'événements internes. De plusΓune seule des hypothèses faites sur le statut des événements internes doit être cohérente (deuxième partie de la condition) : tous les événements internes supposés présents doivent être générés (testé par $m_Y{}^+ \subseteq O_i$)Γtous ceux supposés absents ne doivent pas l'être (testé par $m_Y{}^- \cap O_i = \emptyset$).

2.2.5 Contraintes de l'environnement sur les entrées d'un programme

Il apparaît souvent dans les spécifications d'un programme des informations qui expriment des relations entre les entrées du programme. Par exempleΓelles peuvent être :

- des relations d'exclusion mutuelle : les entrées ne sont jamais simultanément présentes ;
- des relations d'implication : la présence d'une entrée implique la présence d'une autre entrée.

Ces relations entre les entrées du programme doivent être prise en compte lors de la génération du modèle. En effetΓsi la présence dans l'automate de transitions qui ne seront jamais déclenchées ne pose pas de problème en ce qui concerne l'exécution du programmeΓelle peut par contre invalider le processus de vérification formelle — présence de transitions dans le modèle qui ne sont pas des réactions possibles du programme —.

Le langage Argos permet la prise en compte des contraintes exprimant des relations d'exclusion mutuelle entre des entrées du programme — En EsterelΓil est possible d'exprimer en plus des relations d'exclusion mutuelleΓdes relations d'implication —.

Pour prendre en compte ces relations dans l'algorithme de génération du modèle d'un programmeΓil suffit d'éliminer de l'ensemble des monômes considérés (égal à $\mathcal{M}_c^\star(I)$) tous les monômes qui sont en contradiction avec une des relations d'exclusion mutuelle.

2.2.6 Evolution d'Argos depuis sa définition initiale

Le principal changement dans la définition d'Argos depuis [Mar90] concerne le mode de communication. Dans la définition initiale du langageΓcoexistaient deux modes de communication : le *rendez-vous* et la communication par *événements générés*. Le mode de communication par rendez-vous permettait de bloquer un automate en attente d'un événement émis par une autre composante. La communication par événements générés traduisait un mécanisme de type réaction en chaîne. La présence de ces deux modes de communications était liée à l'impossibilité de nier la présence d'un événement dans la condition de déclenchement des transitions des automates. En effet ces conditions permettaient uniquement de tester la présence d'un seul événement. Ces deux modes de communications ont été unifiés par l'introduction de négation des événements dans les conditions de déclenchement des transitionsΓen un mode de communication plus puissant : la diffusion synchrone.

Il est important de noter que la définition formelle du langage Argos a considérablement été modifiée par cette modification du mode de communication.

2.3 Un exemple de programmation et de vérification à partir du langage Argos

L'objet de cette section est la description et la vérification d'un contrôleur de feux de voiture. Dans une première partieΓnous décrivons la spécification de ce contrôleur. Nous donnons ensuite sa description en Argos. La seconde partie est consacrée à la vérification de propriétés sur le modèle du programmeΓen utilisant différentes méthodes formelles à la disposition du programmeur : méthode des observateursΓsimulation comportementale et vérification à l'aide de la logique CTL. La méthode de vérification à l'aide d'observateurs et celle par simulation comportementale sont basées sur les mêmes fondements. CependantΓles modes d'expression de la propriété dans les deux cas sont différentsΓd'où l'intérêt de présenter les deux méthodes.

2.3.1 Présentation de l'exemple

Le système considéré contrôle les cinq types de feux d'une voiture : veilleusesΓcodesΓpharesΓanti-brouillard et longue-portée. Ces feux sont respectivement contrôlés par les couples de sorties suivants :

- $(onV, offV)$ ordre d'allumage et d'extinction des veilleuses ;
- $(onC, offC)$ ordre d'allumage et d'extinction des codes ;
- $(onP, offP)$ ordre d'allumage et d'extinction des phares ;
- $(onAB, offAB)$ ordre d'allumage et d'extinction des anti-brouillard ;
- $(onLP, offLP)$ ordre d'allumage et d'extinction des longue-portée.

Pour gouverner ce systèmeΓl'utilisateur dispose d'une part d'une manette dont nous détaillons le fonctionnement par la suite ; d'autre partΓde deux boutons poussoirs associés respectivement aux anti-brouillard (bouton `AB`) et aux longue-portée (bouton `LP`). Une action sur l'un des deux boutons – pression ou relâchement – n'entraîne pas nécessairement la modification de l'état physique de la lampe associée. En effetΓle contrôleur est tel que les anti-brouillard ne peuvent être allumés qu'en présence des codes. Par conséquentΓles anti-brouillard ne sont allumés que si les deux conditions suivantes sont réunies : les codes sont allumés et le bouton `AB` est en position basse. Le même type de comportement concernant les longue-portée et les phares est spécifié. Les actions sur les boutons sont modélisées par les entrées cAB et cLP qui désignent aussi bien un relâchement qu'une pression sur le bouton en question.

Il est possible de faire tourner la manette dans le sens direct ou indirect. Selon la position courante de la manetteΓun ou les deux sens sont autorisés. Par ces actions de rotationΓil est possible d'atteindre trois positions que nous appelons `éteint, veilleuses, codes-phares`. Une rotation dans le sens direct (resp. indirect) est modélisée par l'entrée TD (resp. TI). La figure 2.17 décrit précisément le schéma de parcours de ces trois positions.

En position `éteint`Γaucune des lampes n'est allumée. En position `veilleuses`Γseules les veilleuses sont allumées. Lorsque l'on passe en position `codes-phares` les codes sont allumés. Il est alors possible en tirant la manette vers soi d'allumer les phares. La manette dans ce cas

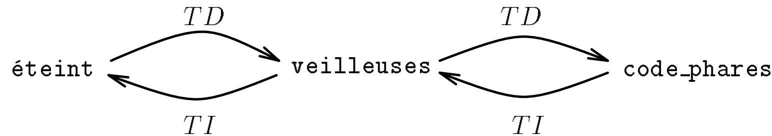


Figure 2.17: positions de la manette

revient tout de suite dans sa position initiale. Pour repasser en codes, il faut de nouveau tirer la manette vers soi. Cette action est modélisée par l'entrée cp.

En résumé, les entrées du système sont TD, TI, cp, cAB et cLP. Nous considérons dans la suite que toutes ces entrées sont deux à deux exclusives. Ce qui se note en Argos par $TD\#TI\#cp\#cAB\#cLP$.

Initialement, le contrôleur se trouve dans la configuration suivante : les deux boutons sont en position haute, la manette est en position `éteint`.

2.3.2 La description en Argos

Nous commençons par identifier dans ce système trois composantes qui évoluent en parallèle :

- Une composante de gestion du bouton des anti-brouillard (`C_AB`)
- Une composante de gestion du bouton des longue-portée (`C_LP`)
- Une composante de gestion de la manette (`C_M`)

Les composantes `C_AB` et `C_LP` sont de simples automates à deux états, qui mémorisent la position des boutons poussoirs. Elles sont identiques, au nom des événements près. Nous détaillons uniquement la composante `C_AB`. Pour prendre en compte la contrainte sur le fonctionnement des anti-brouillard — l'état du bouton `AB` n'est pertinent que lorsque les codes sont allumés — les composantes `C_AB` et `C_M` doivent communiquer. Nous définissons pour cela les événements internes suivants :

- $ABhaut$ et $ABbas$ sont des sorties de `C_AB` et des entrées de `C_M`. $ABbas$ (resp. $ABhaut$) indique le passage de l'état haut (resp. bas) à bas (resp. haut) du bouton `AB`. Lors de la description de la composante `C_M`, le statut de cet événement doit être pris en compte pour allumer (resp. éteindre) les anti-brouillard si les codes sont eux aussi allumés ;
- $pecAB$ ($pecAB$ pour prise en compte du statut des anti-brouillard) est une sortie de la composante de `C_M` et une entrée de `C_AB`. Il indique l'allumage des codes. Par conséquent, si le bouton des anti-brouillard est en position basse, les anti-brouillard doivent être allumés (émission de la sortie $onAB$).

Le même type de communication doit être établi entre les composantes `C_M` et `C_LP` pour gérer le fonctionnement des longue-portée. La figure 2.18 permet de préciser les entrées et sorties

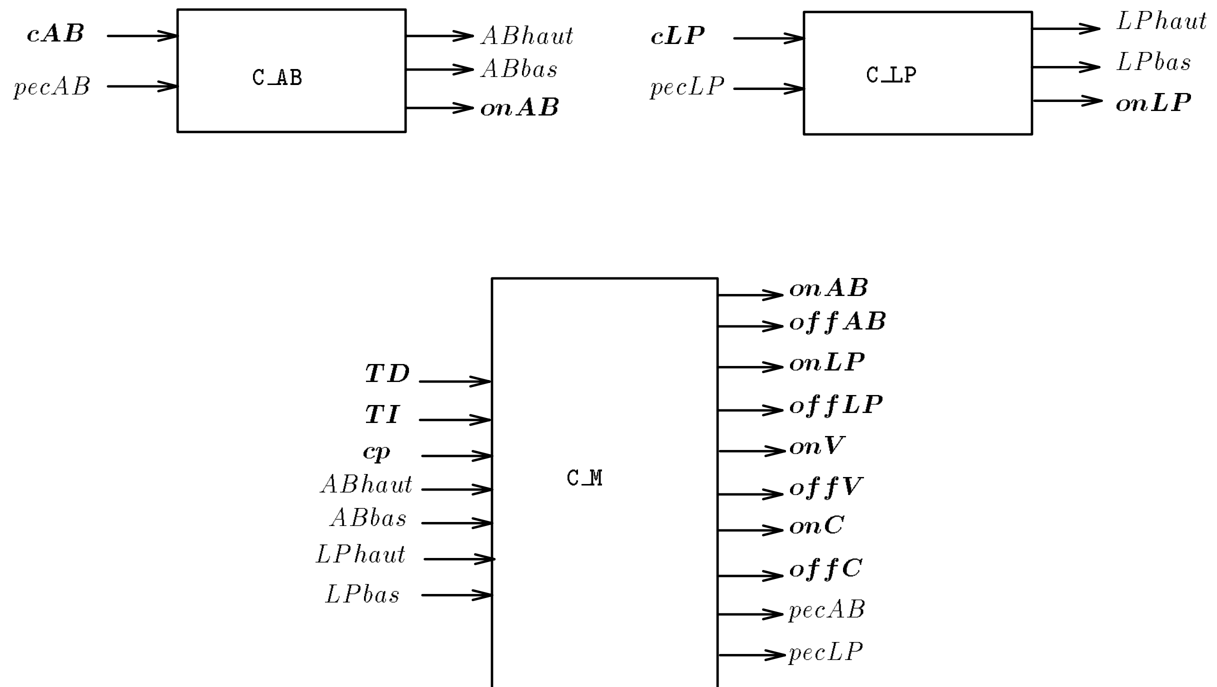


Figure 2.18: entrées et sorties des trois composantes du programme

de chaque composante. Sur cette figureΓles entrées et sorties du programme apparaissent en caractères gras pour les distinguer des événements internes.

Puisque les deux composantes `C_AB` et `C_LP` sont identiques aux noms des événements prèsΓ nous les définissons comme des appels à une procédure générique que nous appelons `bouton_cond` (bouton conditionné). La définition de cette procédure est donnée par la figure 2.19. Les noms d'événements ne font plus référence à un bouton particulier. Les relations d'exclusion mutuelle entre les entrées du programme assurent que l'ordre de commutation d'un bouton et l'ordre de prise en compte du statut d'un bouton (événement interne *pec*) ne sont jamais présents en même temps.

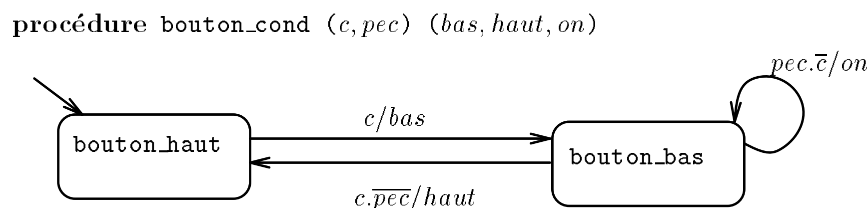


Figure 2.19: procédure `bouton_cond`

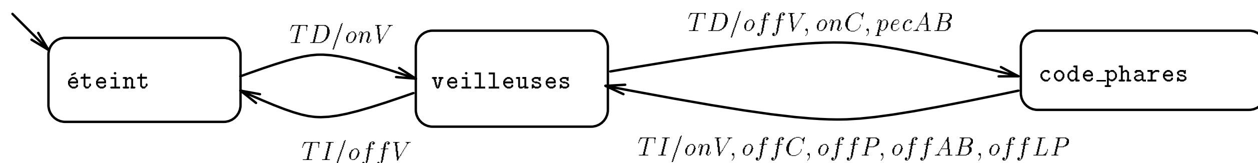


Figure 2.20: automate de contrôle de la composante C_M

La composante C_M se décompose hiérarchiquement en suivant les trois positions principales de la manette. L'automate principal est donné par la figure 2.20. L'état codes_phares de cet automate est raffiné par une composante quiΓà partir de l'entrée *cp*Γgère les codes et les phares. Lorsque cette composante est lancéeΓles codes sont allumés. Il faut donc prendre en compte la position du bouton ABΓce qui se traduit par l'émission de $pecAB$. Lorsque cette composante est interrompueΓles codes ou les phares peuvent être allumés. Par prudenceΓles deux commandes d'extinction $offC$ et $offP$ sont émises. Une d'entre elles est inutile. Cette solution suppose qu'il n'est pas grave de demander l'extinction d'une lampe déjà éteinte. Dans le cas contraireΓil suffirait d'ajouter des composantes en parallèle à tout le programmeΓpour mémoriser l'état de chaque lampe. Ces composantes auraient pour rôle de filtrer les commandes “absurdes” venant du programme. La même hypothèse est faite pour les anti-brouillard et les longue-portée.

L'état code_phares est raffiné par une composante qui gère l'entrée *cp*Γon la nomme C_CP. Comme le montre la figure 2.21Γc'est un automate à deux états. L'état initial (codes) (resp. l'état phares) mémorise le fait que les codes (resp. phares) sont allumés. Lorsque l'état codes (resp. phares) est actifΓune action sur le bouton AB (resp. LP) doit être suivie d'un effet immédiat sur les anti-brouillard (resp. les longue-portée). Ce comportement est traduit par un raffinement des deux états par une composante chargée de rendre les actions sur les boutons effectives.

2.3.3 Des exemples de preuves

Nous souhaitons dans cette partieΓdonner des exemples précis de preuvesΓen utilisant différentes méthodes à la disposition du programmeur.

Nous allons dans un premier temps prouver que le fonctionnement des anti-brouillard tel qu'il est défini par le programme Argos est conforme à la spécification. Cette propriété nous permet d'illustrer la méthode de vérification par observateurs et celle par simulation comportementale. Pour mettre en évidence l'application de la méthode à base de la logique CTLΓnous définissons une seconde propriété plus adéquate à une expression en termes de formules d'une logique temporelle.

Exemple de vérification par la méthode des observateurs

La technique de vérification par observateurs consiste à placer en parallèle au programme à vérifierΓune composante qui observe les entrées et sorties du programme et détermine en fonction de celles-ciΓsi la propriété est satisfaite. Pour celaΓelle génère une sortie d'erreur dès que le

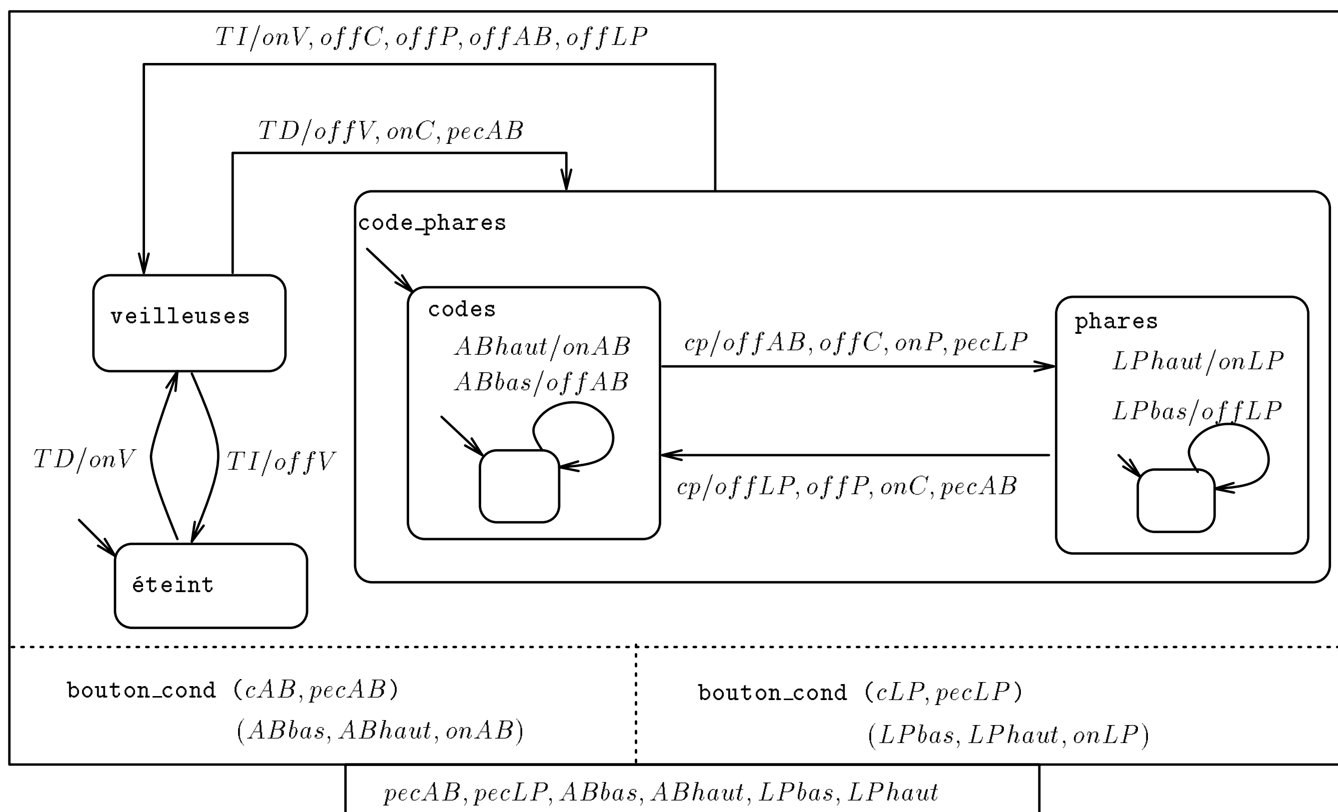


Figure 2.21: programme Argos du contrôleur de feux de voiture

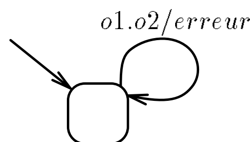


Figure 2.22: observateur de l'exclusion mutuelle de $o1$ et $o2$

programme a un comportement incorrect.

Illustrons ce principe sur un cas très simple. Supposons qu'un programme ait deux sorties $o1$ et $o2$ et que la propriété à vérifier soit l'exclusion mutuelle de ces deux sorties. Pour cela il suffit de placer le programme en parallèle avec l'observateur donné par la figure 2.22Γde rendre internes les événements $o1$ et $o2$Γde calculer le modèle de ce nouveau programme et enfinΓde regarder sur ce modèle si la sortie $erreur$ est émise. Si c'est le casΓalors le programme peut émettre simultanément $o1$ et $o2$. Par conséquentΓil ne vérifie pas la propriété.

La propriété que nous souhaitons vérifier par le programme Argos du contrôleur de feux de voiture s'énonce comme suit :

Si les codes sont allumés, alors une action sur le bouton `AB` *est suivie d'un effet immédiat sur les anti-brouillard. Si les codes sont éteints, cette action n'est suivie d'aucun effet. De plus, si les codes sont éteints (resp. allumés) alors que le bouton* `AB` *est en position basse, les anti-brouillard doivent eux aussi être éteints (resp. allumés).*

L'observateur qui correspond à cette propriété est donné par la figure 2.23. D'après l'énoncé de la propriétéΓil faut connaître la position du bouton `AB`Γet le statut des codes pour déterminer si les ordres d'extinction ou d'allumage des anti-brouillard sont corrects. Les quatre états de l'observateur nous permettent de mémoriser ces informations. Les événements cAB, onC et $offC$ permettent de naviguer entre ces états. Le comportement du programme est incorrect dans quatre situations :

- Alors que les codes sont éteints (resp. allumés) et que le bouton `AB` est en position basseΓ l'allumage (resp. l'extinction) des codes n'est pas accompagné de celui (resp. celle) des anti-brouillard ;
- Alors que les codes sont allumés et que le bouton `AB` est en position basse (resp. haute)Γun relâchement du (resp. une pression sur le) bouton `AB` n'est pas accompagné de l'extinction (resp. l'allumage) des anti-brouillard.

Ces quatre comportements d'erreur se traduisent par quatre transitions qui génèrent la sortie $erreur$ dans l'observateur. L'automate "observateur" obtenu est placé en parallèle avec le programme Argos du contrôleur de feux de voiture. Les sorties de ce programme qui sont utilisées en entrées par l'observateurΓdoivent être déclarées internes au programme résultant de la mise en parallèle. Le modèle de ce nouveau programme est calculé. Aucune de ces transitions n'émet la sortie d'erreur. Par conséquentΓle programme satisfait la propriété.

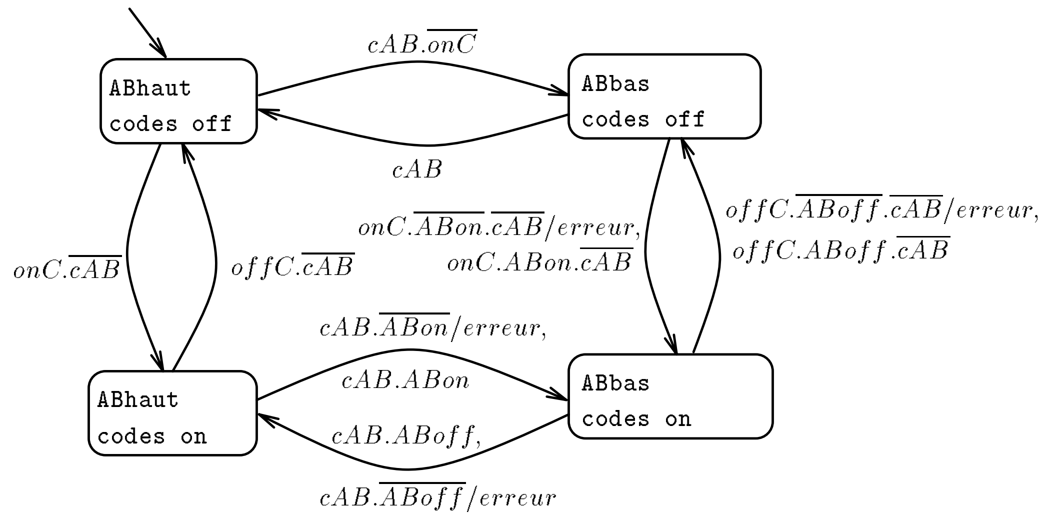


Figure 2.23: observateur qui détecte les erreurs de fonctionnement des anti-brouillard

Un exemple de vérification par simulation comportementale

Nous souhaitons dans cette section prouver la propriété précédenteΓmais en utilisant cette fois la simulation comportementale. Il nous faut définir l'automate qui par simulation avec le programmeΓdétermine si celui-ci satisfait ou non la propriété. A la différence de l'observateur qui exprime les cas de dysfonctionnement du programmeΓl'automate à construire pour la vérification comportementaleΓexprime tous les comportements qui sont en accord avec la propriété.

Par exempleΓà partir de l'état initial du systèmeΓla séquence

- allumage des codesΓ
- commutation du bouton `AB` accompagnée de l'allumage des anti-brouillardΓ
- extinction des codes accompagnée de celle des anti-brouillardΓ

est une séquence qui est cohérente par rapport à la propriété que l'on souhaite vérifier. Par contreΓla séquence composée des deux premières actionsΓpuis de l'extinction des codes non accompagnée de celle des anti-brouillard est une séquence incohérente par rapport à la propriété. L'automate qui traduit la propriété doit contenir toutes les séquences cohérentes que peut faire le systèmeΓsans contenir une seule séquence incohérente.

Cet automate contient quatre états qui permettent de connaître exactement l'état du bouton `AB` et l'état des codes. Nous reprenons les noms d'états donnés pour l'observateur. Dans l'état initial (`AB haut codes off`)Γla réaction du programme doit être observée de manière à déterminer :

- Si le bouton `AB` est commuté : nous définissons à cet effet le point d'observation `obs(`cAB`)` ;
- Si les codes sont allumés : nous définissons à cet effet le point d'observation `obs(`onC`)` ;

- si aucune des deux actions précédentes n'est effectuée : nous définissons à cet effet le point d'observation `obs(`$\overline{onC}.\overline{cAB}$`)`.

Remarquons que les deux premiers points d'observation sont disjoints pour notre programme. En effetΓd'après la relation d'exclusion mutuelle associée au programmeΓune étiquette du modèle du programme ne peut pas contenir à la fois cAB et onC dans l'ensemble des événements présents.

Le même type d'analyseΓétat par étatΓconduit à l'automate de la figure 2.24. L'automate est construit à partir de neuf points d'observation. Ces neuf points d'observation ne sont pas disjoints. Par exempleΓ le modèle du programme contient des transitions étiquetées par $TD.\overline{TI}.\overline{cp}.\overline{cLP}.\overline{cAB}/onC, offV$. Or cette étiquette satisfait les points d'observation `obs(`$\overline{cAB}.\overline{offC}$`)` et `obs(`$onC$`)`. Pour pouvoir utiliser les outils de vérification par simulation comportementale — basés sur la comparaison syntaxique des étiquettes du programme à celles de la propriété — il faut transformer la propriété de manière à se ramener à un ensemble de points d'observation disjoints. Nous appliquons pour cela la fonction de transformation définie en 1.16. Nous obtenons une propriété construite à partir de sept points d'observation et qui contient vingt-cinq transitions. Il est important de remarquer que la relative petite taille de l'automate résultat est due au fait que la transformation s'effectue en tenant compte des étiquettes du modèle du programme. Il s'ensuit qu'un tel automate ne peut être donné directement par l'utilisateur qui lui n'a pas une connaissance précise de ces étiquettes. Une construction directe d'un automate traduisant la propriété à l'aide de points d'observation disjoints nécessiterait — pour être sûr d'avoir un ensemble de points d'observation disjoints — la prise en compte dans chaque état du statut de tous les événements qui interviennent dans la propriété. Puisque pour cette propriétéΓnous observons le statut de cinq événements différentsΓle nombre de transitions à expliciter dans chaque état est particulièrement important (de l'ordre de 32).

Il est important d'éviter cette explosion du nombre de transitions carΓsans préjuger du mode de fonctionnement des outilsΓil paraît intuitif de penser que plus le nombre de transitions de la propriété est élevéΓplus le temps de recherche pour une étiquette de l'automate d'une étiquette de la propriété syntaxiquement équivalenteΓest grand. Cette remarque intuitive s'avère justifiée dans le cas de l'outil Aldébaran que nous avons utilisé pour vérifier notre programme. Le résultat obtenu est positif.

Exemple de vérification à l'aide de la logique CTL

La méthode de vérification par simulation comportementale permet uniquement de vérifier des propriétés de sûreté. Nous présentons à présent la preuve d'une propriété de vivacité à l'aide d'une logique temporelle.

Cette propriété s'énonce de la manière suivante :

Lorsque les anti-brouillard sont éteints, il est toujours possible de les allumer.

Autrement ditΓnous voulons vérifier que notre système ne peut se placer dans une configuration telleΓque les anti-brouillard ne puissent plusΓquelle que soit l'action du conducteur sur la manette et les boutonsΓêtre allumés.

Ce type de propriétés de vivacité — accessibilité d'un état à partir duquel une action est possible — présente un intérêt dans de nombreux systèmes réactifs. En effetΓelle permet de prouver

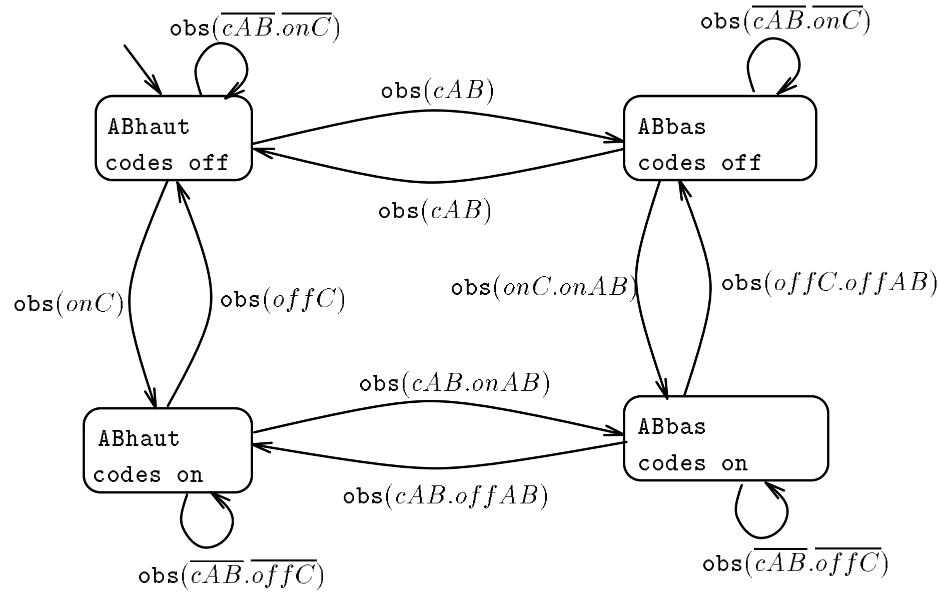


Figure 2.24: propriété comportementale exprimant le bon fonctionnement des anti-brouillard

qu'une situation particulière – lampe éteinteΓalarme enclenchée – n'est jamais irrémédiable et que l'environnement peut toujours se comporter de manière à inverser cette situation.

Les points d'observation nécessaires pour exprimer cette propriété sont :

- la sortie $ABon$ est émiseΓnous le notons obs($ABon$)Γ
- la sortie $ABoff$ est émiseΓnous le notons obs($ABoff$).

Il s'avère que pour l'ensemble des étiquettes de notre programmeΓ ces deux points d'observation sont disjoints. Cela est dû aux relations d'exclusion mutuelle sur les entrées. Nous nous trouvons donc dans une situationΓoù il suffit de renommer les étiquettes du modèle qui satisfont un des deux points d'observation par ce point d'observation.

Il faut à présent traduire la propriété en une formule CTL. Pour celaΓil faut exprimer le fait queΓpour toutes les exécutions du programme — issues de l'état initial — après toute émission de $ABoff$ il doit exister une séquence d'exécution qui émet $ABon$. D'où la formuleΓ

$$\forall\Box(\texttt{after}_{\texttt{obs}(ABoff)} \implies \exists\Diamond\texttt{after}_{\texttt{obs}(ABon)}).$$

Nous rappelons les significations intuitives des opérateurs $\forall\Box$ et $\exists\Diamond$.

- $\forall\Box\varphi$ est satisfaite pour un état q si et seulement si sur toutes les séquences d'exécution issues de qΓtous les états satisfont φ.
- $\exists\Diamond\varphi$ est satisfaite pour un état q si et seulement s'il existe une séquence d'exécution issue de q sur laquelle un état satisfait φ.

Cette formule CTL est satisfaite par notre programme.

2.4 Récapitulation

Nous avons présenté dans ce chapitre le langage Argos tel qu'il a été défini dans [Mar92]. C'est un langage impératif à base d'automates parallèles et hiérarchisés. Sa sémantique est définie en termes d'automates booléens. Nous avons aussi présenté dans ce chapitre un exemple de système réactif décrit en Argos. Cet exemple met en évidence le style de programmation de ce langage. De plusΓil nous a permis d'illustrer les différentes techniques de vérification formelle auxquelles nous nous sommes intéressés dans notre travail : la méthode des observateursΓla simulation comportementale et l'utilisation d'une logique temporelle.

Chapitre 3

Deux extensions au langage Argos

L'objet de ce chapitre est de définir deux extensions au langage Argos que nous venons de présenter. Ces extensions répondent toutes les deux à des besoins apparus dès la conception du langage. Ce sont deux macro-notationsΓpuisqu'il est possible de les traduire à l'aide des constructions de base d'Argos. Par conséquentΓles modèles des programmes écrits en utilisant ces extensions sont toujours des automates booléens.

La première extension concerne l'introduction de variables comme alternative à la mémorisation de l'histoire du système réactif sous forme d'états d'automate. Ces variables sont appelées *variables de contrôle*. Leur introduction dans le langage Argos est présentée dans la section 3.1.

La seconde extension est une structure qui permet de limiter le temps de stationnement dans un état. Cette nouvelle forme d'états est appelée *états temporisés*. Cette notion d'état temporisé était déjà présente dans une des versions des Statecharts. L'introduction des états temporisés dans le langage Argos est présentée dans la section 3.2.

Pour définir la sémantique de ces deux extensionsΓdeux solutions équivalentes sont possibles :

- Définir la fonction d'expansion qui transforme tout programme Argos étendu en un programme Argos utilisant les opérateurs de base. Le modèle du programme étendu est alors obtenu en appliquant la fonction sémantique $\mathcal{S}$ définie dans le chapitre précédent ;

- Définir une nouvelle fonction sémantique qui détermine le modèle d'un programme étendu quelconque.

La première solution peut sembler plus simple à mettre en œuvre. C'est le cas lors de l'introduction des états temporisés. Par contre pour les variables de contrôleΓnous présentons la deuxième solution car il se trouve que définir la fonction d'expansion est d'un niveau de complexité équivalent à la définition d'une nouvelle fonction sémantique.

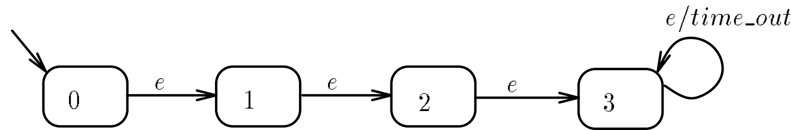


Figure 3.1: programme Argos d'un chien de garde de borne quatre

3.1 Le langage Argos avec des variables de contrôle

3.1.1 Présentation informelle

Dans le langage ArgosΓle seul moyen jusqu'à présent de mémoriser de l'information sur l'histoire des entrées d'un système réactifΓest de créer un état d'un automate et de lier l'activité de ce dernier à cette information.

Les variables de contrôle sont une alternative à la mémorisation sous forme d'étatsΓ d'informations sur l'histoire des entrées d'un système réactif. Ce sont alors les valeurs de ces variables qui sont reliées à une signification précise sur cette histoire. L'accès à cette information s'effectue à l'aide de conditions sur les valeurs des variables de contrôle.

Pour certains systèmesΓl'utilisation de variables engendre des descriptions plus concises et/ou plus proches des spécifications. C'est le cas par exempleΓdes programmes qui contiennent des composantes de type compteurΓcomme des *chiens de garde*. De telles composantes ont pour rôle de compter les occurrences d'un événement et d'avertir leur environnement lorsqu'une borne est dépassée.

Nous prenons pour exemple un programme Argos simpleΓconstitué d'un chien de garde de borne 4 qui compte les occurrences d'une entrée e. Pour décrire ce programme en ArgosΓil suffit de définir un automate à quatre étatsΓchaque état mémorisant le nombre d'occurrences de e déjà reçues (cf. figure 3.1). Avec ce principeΓun chien de garde dont la borne est 100Γpossède 100 états. La définition en extension de cet automate à structure régulière est peu satisfaisante. Une définition plus conciseΓet surtout indépendante de la borne est obtenue en utilisant une variable de contrôle. Celle-ci mémorise le nombre des occurrences de e déjà reçues ; elle peut donc prendre pour notre exemple quatre valeurs. Le programme Argos avec variables de contrôle de la figure 3.2Γdécrit le même chien de garde que le celui de la figure 3.1. La portée d'une variable de contrôle est indiquée de la même manière que pour les événements internes. La déclaration de la variable contient son domaine de valeurs (sur l'exemple : [0..3])Γainsi que sa valeur initiale (sur l'exemple : 0).

L'expansion de ce programme conduit au programme de la figure 3.1. Elle associe un état à chaque valeur possible de la variable. On évalue statiquement les conditions et les affectations qui portent sur les variables. Le domaine de valeurs possibles pour toute variable doit être fini.

La portée des variables de contrôle n'est pas limitée aux automatesΓet peut s'étendre à tout un programme Argos. Il n'y a en effet aucune raison d'interdire l'utilisation de variables de contrôle partagées par des automates mis en parallèleΓsi c'est de cette manière que le programme se conçoit le plus naturellement. Supposons qu'un automate complexe utilisant des variables se

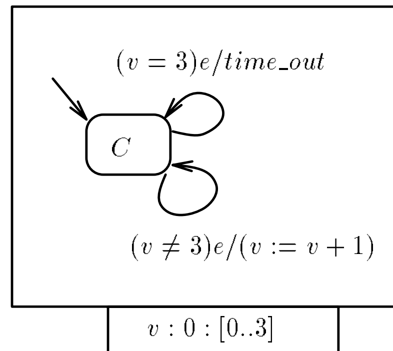


Figure 3.2: programme Argos avec variables de contrôle d'un chien de garde de borne quatre

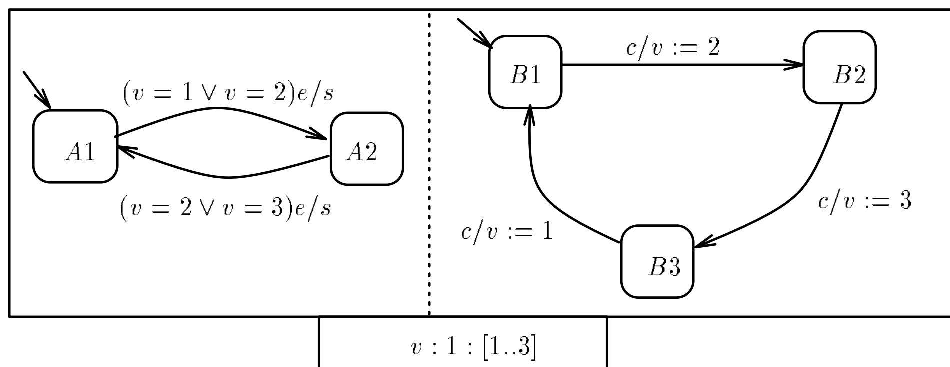


Figure 3.3: communication par variables de contrôle

décrive de manière plus intuitive à l'aide d'une mise en parallèle de deux automates plus simplesΓ partageant des variables. Puisque l'opérateur de mise en parallèle traduit du parallélisme de description — par opposition à du parallélisme d'implantation — un tel partage ne pose pas de véritables difficultés et peut être autorisé dans le langage.

Par ailleursΓles variables de contrôle peuvent être utilisées pour exprimer certains types de communication entre deux composantes (cf. figure 3.3) : l'une d'entre elles est chargée de mettre à jour la variable qui est lue par l'autre composante. Pour pouvoir communiquer à l'aide de variablesΓl'information échangée ne doit pas porter sur la réaction courante du systèmeΓmais sur les réactions passées. En effetΓcomme les états des automatesΓles variables ne renseignent que sur le passé d'un système.

En contre-partieΓcette absence de limitation dans l'utilisation des variables est à l'origine de problèmes d'écriture simultanée à une même variable qu'il faut être capable de détecter.

3.1.2 Notations et définitions

Notations

- VAR dénote un ensemble de noms de variablesΓv est un représentant de cet ensemble.
- $\text{TYPE}^{\not\infty}$ dénote un ensemble de noms de types finisΓ $\tau^{\not\infty}$ est un représentant de cet ensemble.
- VAL dénote l'ensemble de valeurs obtenus en effectuant l'union des domaines des types appartenant à $\text{TYPE}^{\not\infty}$Γet en ajoutant une valeur spéciale notée $\perp$. $\mathtt{v}$ dénote un élément de cet ensemble.

La notion de contexte

Un contexte W est une fonction de l'ensemble des variables dans l'ensemble des valeurs. Nous notons $\mathcal{W}$ l'ensemble de tous les contextesΓet $W_\perp$ le contexte défini par $\forall v \in$ VAR $W_\perp(v) = \perp$. Un contexte est dit défini sur un ensemble de variables VΓlorsque V contient toutes les variables auxquelles W associe des valeurs différentes de $\perp$.

Deux opérations sont définies sur les contextes : la modification d'un contexte en un pointΓ et la modification d'un contexte en tenant compte d'un autre contexte.

Définition 3.1 Modification d'un contexte en un point
L'application de cette fonction à un contexte W est notée $W\{v \leftarrow \mathtt{v}\}$, où $\mathtt{v}$ est la nouvelle valeur à donner à la variable v.

$$W\{v_1 \leftarrow \mathtt{v}\}(v_2) = \textit{ si } (v_1 = v_2) \textit{ alors } \mathtt{v} \textit{ sinon } W(v_2) \quad \blacksquare$$

Définition 3.2 Modification d'un contexte d'après un autre contexte
L'application de cette fonction à un contexte W_1 est notée $W_1\{W_2\}$, où W_2 est le contexte dont il faut tenir compte dans la modification. Seules les valeurs des variables définies dans W_2 sont modifiées dans W_1.

$$W_1\{W_2\}(v) = W_2(v) \textit{ si } (W_2(v) \neq \perp) \textit{ alors } W_2(v) \textit{ sinon } W_1(v) \quad \blacksquare$$

Expressions, conditions, affectations

L'ensemble des expressions est noté EXPΓil est défini de manière usuelle. Une expression est notée e.

L'ensemble des conditions noté COND est le sous-ensemble des expressions booléennes de EXP. Une condition est notée γΓun ensemble de conditions est noté C.

Nous supposons définie une fonction d'évaluation d'une expression (ou d'une condition) dans un contexte. Elle est notée $\mathcal{E}$.

Une affectation u est un couple associant une variable à une expression. Une liste d'affectations est notée U. L'ensemble des listes d'affectations est noté L_U. Sur ces listesΓ les opérations usuelles d'ajout d'un élément en tête de liste et de concaténation de deux listes sont respectivement notées $\bullet$ et @. La liste vide est notée $\emptyset_L$. Nous supposons défini un prédicat sur l'ensemble des listes d'affectationsΓnoté `doublon`Γsatisfait si et seulement s'il existe dans la liste deux affectations à la même variable.

Nous définissons une fonction d'évaluation d'une liste d'affectations dans un contexteΓnotée $\mathcal{E}_U$. Cette fonction a pour résultat un contexte qui associe à toute variableΓ soit la valeur $\perp$ si cette variable n'apparaît pas dans la liste d'affectationsΓsoit le résultat de l'évaluation de la première expression qui lui est associée dans la liste.

Définition 3.3 Evaluation d'une liste d'affectations dans un contexte
L'évaluation d'une liste d'affectations U dans un contexte W est notée $\mathcal{E}_U(U, W)$. Elle a pour résultat un autre contexte. Toutes les expressions de la liste sont évaluées dans le même contexte.

$$\begin{array}{lcl} \mathcal{E}_U(\emptyset_L, W) & = & W_\perp \\ \mathcal{E}_U((v,e)\bullet U, W) & = & \mathcal{E}_U(U, W)\{v \leftarrow \mathcal{E}(e, W)\} \end{array}$$

■

3.1.3 Définition d'Argos avec variables de contrôle

Syntaxe

Un programme Argos avec variables de contrôle est noté P^c.

Définition 3.4 Syntaxe abstraite des programmes Argos avec variables de contrôle

$$\begin{array}{lcl} P^c & ::= & P^c \| P^c \\ & | & \overline{P^c{}^Y} \\ & | & \mathbf{R}_{(Q, q_{init}, T)}(R_1, \ldots, R_n) \\ & | & P^c\Box(v : e : \ \tau^{\not\propto}) \ \textit{avec} \ v \in \ VAR, \ e \in \ EXP, \ \ \tau^{\not\propto} \in \ TYPE^{\not\propto} \\ R & ::= & \textsc{nil} | P^c \end{array}$$

■

Dans cette grammaireΓ $P^c\Box(v : e : \ \tau^{\not\propto})$ dénote un programme obtenu en déclarant dans le sous-programme P^c la variable de contrôle v de type $\tau^{\not\propto}$. La valeur initiale de cette variable de contrôle est définie par l'expression e.

L'ensemble des transitions T de l'automate paramètre de l'opérateur de raffinementΓest tel que $T \subseteq Q \times \mathcal{M}^\star_c(In) \times$ COND $\times\, 2^{Out} \times L_U \times Q$. Pour une transition $t = (q_s, m_e, \gamma, O, U, q_b)$Γ la condition de franchissement est donnée par le monôme m_e qui exprime la condition sur le statut des entréesΓet par la condition γ qui exprime la condition sur les valeurs des variables du programme. Le franchissement de la transition provoque l'émission des événements contenus dans O et la mise à jour des variables selon la liste d'affectations U.

Il n'est pas possible d'étendre statiquement la notion de déterminisme d'un automate lorsque ses transitions portent des conditions sur les valeurs des variables. Le déterminisme des automates est vérifié lors de la génération du modèle du programme.

Dans la syntaxe concrète des programmes Argos avec variables de contrôleΓ les monômes d'événements des transitions ne sont pas nécessairement complets sur l'ensemble des entrées de l'automate. La phase de saturation s'applique en ne tenant pas compte des conditions sur les variables.

Nous notons $\mathcal{P}^c$ l'ensemble des programmes Argos avec variables de contrôle syntaxiquement corrects. $\mathcal{P}^c_{rinc}$ est le sous-ensemble des programmes principaux.

Définition 3.5 Ensemble des programmes principaux
Soit $P^c \in \mathcal{P}^c; P^c \in \mathcal{P}^c_{rinc} - (\mathcal{I}(P^c) \cap \mathcal{O}(P^c) = \emptyset) \wedge (\mathcal{V}ar\mathcal{L}ib(P^c) = \emptyset)$ ■

Les fonctions $\mathcal{I}$ et $\mathcal{O}$ (définitions 2.3 et 2.4) s'étendent sans difficulté aux programmes avec variables de contrôle. La fonction $\mathcal{V}ar\mathcal{L}ib$ (cf. définition 3.6) calcule l'ensemble des variables libres d'un programme.

Définition 3.6 Ensemble des variables libres d'un programme

$$\begin{array}{lcl}
\mathcal{V}ar\mathcal{L}ib(\textsc{nil}) & = & \emptyset \\
\mathcal{V}ar\mathcal{L}ib({P^c}_1 \| {P^c}_2) & = & \mathcal{V}ar\mathcal{L}ib({P^c}_1) \cup \mathcal{V}ar\mathcal{L}ib({P^c}_2) \\
\mathcal{V}ar\mathcal{L}ib(\mathbf{R}_{(Q,q_{init},T)}(R_1, \ldots, R_n)) & = & V((Q, q_{init}, T)) \cup \bigcup_{i=1}^{i=n} \mathcal{V}ar\mathcal{L}ib(R_i) \\
\mathcal{V}ar\mathcal{L}ib(\overline{P^{cY}}) & = & \mathcal{V}ar\mathcal{L}ib(P^c) \\
\mathcal{V}ar\mathcal{L}ib(P^c \square (v : e :\ \tau^{\not\propto})) & = & \mathcal{V}ar\mathcal{L}ib(P^c) \setminus \{v\}
\end{array}$$

Où V est une fonction qui associe à un automate l'ensemble des variables qui sont utilisées dans les étiquettes de ses transitions. ■

Nous supposons dans toute la suite du chapitreΓ qu'il y a unicité des noms de variables dans tout programme Argos.

Sémantique statique

Les règles de sémantique concernent les problèmes de vérification de type et de cohérence d'utilisation des variables par rapport à leur portée. Pour les aspects de vérification de typeΓ différentes stratégies plus ou moins contraignantes pour l'utilisation des variables peuvent être définies. Etant donné que la définition de la sémantique des variables de contrôle ne dépend pas de la stratégie choisieΓ nous laissons au niveau de la définition qui suit tous les choix possibles.

Calcul du modèle d'un programme Argos avec variables de contrôle

Présentation de la fonction sémantique

Le rôle de la fonction sémantiqueΓ notée S^cΓ est d'associer à tout programme principalΓ l'automate qu'il détermine : son modèle. Le profil de cette fonction est le suivant :

$S^c \ : \ \mathcal{P}^c_{rinc} \rightarrow \mathcal{A}^{\emptyset,c}_{dr} \cup \{\text{ERROR}\}$

La valeur d'erreur signifie que le programme contient soit une erreur de causalité, soit une erreur due à la mauvaise utilisation des variables. Ce type d'erreur est détaillé par la suite.

La fonction S^c est définie à l'aide de la fonction S_W. Cette fonction a pour argument un programme correct syntaxiquement — il n'est pas nécessairement principal — et un contexte défini sur un ensemble qui contient toutes les variables libres du programme. Elle leur associe le modèle du programme calculé en donnant à ses variables libres les valeurs précisées dans le contexte.

Le lien entre les deux fonctions S^c et S_W est le suivant :

$$\forall \ P^c \ \in \ \mathcal{P}^c_{rinc}, \quad S^c(P^c) = S_W(P^c, W_\perp).$$

Nous présentons par la suite la définition de la fonction S_W.

Étant donnés un programme P^c et un contexte W, la fonction S_W doit tout d'abord détecter la présence d'éventuelles erreurs dans le programme. Dans le cas où le programme n'en contient pas, elle doit définir l'ensemble d'états, l'état initial et l'ensemble des transitions du modèle de P^c sous le contexte W.

Les états du modèle sont de la forme (p, W) où p reflète la structure de P^c et mémorise, en plus de l'état courant de chaque automate, la valeur courante de chaque variable définie dans P^c. Le contexte W mémorise les valeurs des variables libres du programme. La grammaire suivante définit la forme syntaxique de p.

Définition 3.7 Syntaxe de la composante p d'un état (p, W) du modèle

$$\begin{array}{lcll} p & ::= & p \| p & \\ & | & \overline{p^Y} & \\ & | & \mathbf{R}_{(Q,q_{init},q_c,T)}(r_1, \ldots, r_n) & (q_c : \textit{état courant de l'automate}) \\ & | & p\Box(v : e : \mathtt{v} : \ \tau^{\not\propto}) & (\mathtt{v} \ \ \textit{valeur courante de la variable}) \\ r & ::= & nil | p & \end{array}$$

■

Selon la configuration des automates du programme, une variable est active ou pas. Seules les variables actives sont pertinentes pour calculer la réaction du programme. Elles déterminent, avec l'ensemble des états courants des automates actifs, la configuration du programme. Afin de manipuler une forme normale, la valeur courante d'une variable inactive est par convention la valeur spéciale $\perp$.

L'état initial du modèle de P^c est égal à $(init_W(P^c, W), W)$. La fonction $init_W$ a pour rôle, d'une part de placer chaque automate du programme dans son état initial, d'autre part de calculer la valeur initiale de chaque variable active dans l'état initial. Ce second calcul peut dépendre de la valeur des variables libres du programme, d'où la présence du contexte W.

Définition 3.8 Etat initial du modèle

$$\begin{array}{lcl}
init_W(\textsc{nil}, W) & = & nil \\
init_W({P^c}_1 \| {P^c}_2, W) & = & init_W({P^c}_1, W) \| init_W({P^c}_2, W) \\
init_W(\overline{P^c}^Y, W) & = & \overline{init_W(P^c, W)}^Y \\
init_W(P^c \square (v : e : \ \tau^{\not\propto}), W) & = & init_W(P^c, W') \square (v : e : \mathcal{E}(e, W) : \ \tau^{\not\propto}) \\
 & & avec\ W' = W\{v \leftarrow \mathcal{E}(e, W)\} \\
init_W(\mathbf{R}_{(Q, q_{init}, T)}(R_1, \ldots, R_n), W) & = & \mathbf{R}_{(Q, q_{init}, q_{init}, T)}(r'_1, \ldots, r'_n) \\
\qquad avec\ r'_i = & & si\ (i = init)\ alors\ init_W(R_{init}, W)\ sinon\ exit(R_i)
\end{array}$$

■

La fonction *exit* calcule la forme normale d'un programme dont toutes les variables sont inactives. L'état résultat est tel que chaque automate a pour état courant son état initialΓet chaque variable a pour valeur courante la valeur spéciale $\perp$.

Définition 3.9 Fonction *exit*

$$\begin{array}{lcl}
exit(\textsc{nil}) & = & nil \\
exit({P^c}_1 \| {P^c}_2) & = & exit({P^c}_1) \| exit({P^c}_2) \\
exit(\overline{P^c}^Y) & = & \overline{exit(P^c)}^Y \\
exit(P^c \square (v : e : \ \tau^{\not\propto})) & = & exit(P^c) \square (v : e : \perp : \ \tau^{\not\propto}) \\
exit(\mathbf{R}_{(Q, q_{init}, T)}(R_1, \ldots R_n)) & = & \mathbf{R}_{(Q, q_{init}, q_{init}, T)}(exit(R_1), \ldots, exit(R_n))
\end{array}$$

■

La détection d'éventuelles erreurs dans un programme s'effectue pendant le calcul de l'ensemble des transitions de son modèle. En plus des erreurs de causalitéΓtrois situations dues à une mauvaise utilisation de variables de contrôle conduisent à des programmes incorrects :

- La valeur courante d'une variable est hors de son domaine de définition. Cette erreur est détectée à l'aide d'un prédicat *DomVar* sur les composantes p des états de l'automate. Ce prédicat est satisfait si et seulement si les valeurs courantes des variables actives de p appartiennent à leur domaine de définition (cf. définition 3.10) ;
- La valeur d'une variable est modifiée plus d'une fois dans la même réaction du programme — problème d'écriture simultanée —. Ce type d'erreur est détecté en gardant lors du calcul d'une réaction la traceΓde toutes les affectations effectuées sous forme de listeΓpuis en utilisant le prédicat `doublon` sur cette liste.
- La réaction d'un automate composante de base du programme n'est pas déterministeΓce qui signifie qu'étant données la configuration des entrées et la valeur des variablesΓplusieurs transitions peuvent être déclenchées.

Définition 3.10 Prédicat $DomVar$

$$
\begin{array}{lcl}
DomVar(\textsc{nil}) & = & true \\
DomVar(p_1 \| p_2) & = & DomVar(p_1) \wedge DomVar(p_2) \\
DomVar(\overline{p^Y}) & = & DomVar(p) \\
DomVar(p\Box(v : e : \mathtt{v} :\ \tau^{\not\propto})) & = & DomVar(p) \wedge (\mathtt{v} \in\ \tau^{\not\propto}) \\
DomVar(\mathbf{R}_{(Q,q_{init},q_{init},T)}(r_1, \ldots, r_n)) & = & DomVar(r_1) \wedge \ldots \wedge DomVar(r_n)
\end{array}
$$

■

Nous présentons à présent l'algorithme général de calcul des transitions du modèle d'un programme Argos avec variablesΓsous un contexte qui précise les valeurs des variables libres de ce programme. De la même manière que pour la mise en œuvre de la sémantique du langage Argos — sans variable de contrôle — cet algorithme permet aussi de détecter les programmes incorrectsΓet de calculer l'ensemble des états du modèle du programmeΓsi celui-ci est correct.

L'algorithme général de calcul des transitions du modèle de P^c dans le contexte W est le suivant :

Algorithme 3.1 Calcul des transitions du modèle de P^c **sous un contexte** W

```
Trans ← ∅
I ← 𝓘(P^c)
(p_init, W_init) ← (init_W(P^c, W), W)
Si DomVar(p_init) faire
   Access ← {(p_init, W_init)}
   A_Explorer ← {(p_init, W_init)}
   Tant que (A_Explorer ≠ ∅) faire
        (p, W) ← Choisir_etat(A_Explorer)
        A_Explorer ← A_Explorer \(p, W)
        Pour chaque m_e ∈ 𝓜_c^*(I) faire
             Si ( CalculTrans(p, W, m_e).resultat = OK ) faire
                O ← CalculTrans(p, W, m_e).sorties
                (p_b, W_b) ← CalculTrans(p, W, m_e).but
                U ← CalculTrans(p, W, m_e).aff
                Si (¬doublon(U) ∧ DomVar(p_b)) faire
                   Trans ← Trans ∪ {((p, W), m_e, O, (p_b, W{W_b}))}
                   A_Explorer ← A_Explorer ∪ {(p_b, W{W_b})}
                   Access ← Access ∪ {(p_b, W{W_b})}
                Sinon
                   Exit(erreur)
             Sinon
                Exit(erreur)
Sinon
   Exit(erreur)
```

■

Le calcul d'une transitionΓrôle de la fonction `CalculTrans`Γpeut conduire à un échec. C'est le cas si l'un des automates du programme est non déterministeΓou si le programme contient une erreur de causalité. Dans le cas où le calcul d'une transition réussitΓles informations calculées sont :

- L'ensemble des sorties émises par la transition ;
- La composante p de l'état but de la transition ;
- Le contexte défini sur les variables dont les valeurs sont modifiées par la réaction ; il contient leurs nouvelles valeurs ; ce contexte permet de calculer la composante W de l'état but de la transition ;
- La liste des affectations effectuées lors de la réaction.

Nous présentons à présent la fonction `CalculTrans`

Nous notons $(p, W) \xrightarrow{m_e/O,U} (p_b, W_b)$ l'existence de la transition $((p, W), m_e, O, (p_b, W_b))$ dans le modèle de $prog(p)$ sous le contexte W; la liste d'affectations U contient toutes les affectations effectuées lors de cette transition.

La mise en parallèle

Soient $I_i = \mathcal{I}(prog(p_i))$ pour $i \in \{1, 2\}$;

$$\frac{(p_1, W) \xrightarrow{m_e[I_1]/O_1,U_1} (p_1', W_1) \qquad (p_2, W) \xrightarrow{m_e[I_2]/O_2,U_2} (p_2', W_2)}{(p_1 \| p_2, W) \xrightarrow{m_e/O_1 \cup O_2, U_1 @ U_2} (p_1' \| p_2', W_1\{W_2\})} \quad [\mathrm{Par}^c]$$

Remarque 3.1 *$W_1\{W_2\} \neq W_2\{W_1\}$ si et seulement si* `doublon(U1@U2)`*. Par conséquent, tant que le programme ne contient pas de problème d'écriture simultanée à une même variable, l'ordre dans lequel nous appliquons la fonction de modification d'un contexte en fonction d'un autre contexte est indifférent. Lorsqu'au contraire il contient un problème de ce type, alors nous ne tenons pas compte de cet ordre car de toute façon le programme est rejeté.* ■

Le raffinement

Soient $R_i = prog(r_i)$ *et* $I_c = \mathcal{I}(prog(r_c))$;

$$\frac{\exists!(q_c, m_e[In], \gamma/O, U, q_d) \in T \text{ telle que } \mathcal{E}(\gamma, W) \qquad (r_c, W) \xrightarrow{m_e[I_c]/O_c,U_c} (r_c'', W'')}{(\mathbf{R}_{(Q, q_{init}, q_c, T)}(r_1, \ldots, r_n), W) \xrightarrow{m_e/O \cup O_c, U @ U_c} (\mathbf{R}_{(Q, q_{init}, q_d, T)}(r_1', \ldots, r_n'), W')} \quad [\mathrm{Raff}^c 1]$$

avec

$r_i' = r_i$ si $i \neq c \wedge i \neq d$
$r_i' = exit(R_i)$ si $i = c \wedge c \neq d$
$r_i' = init_W(R_d, W)$ si $(i = c \wedge c = d) \vee (i = d)$
$W' = W''\{\mathcal{E}_U(U, W)\}$

$$\frac{\forall \gamma, O, U, q_d, \ \nexists(q_c, m_e[In], \gamma/O, U, q_d) \in T \text{ telle que } \mathcal{E}(\gamma, W) \quad (r_c, W) \xrightarrow{m_e[I_c]/O_c,U_c} (r_c'', W'')}{(\mathbf{R}_{(Q, q_{init}, q_c, T)}(r_1, \ldots, r_n), W) \xrightarrow{m_e/O \cup O_c, U_c} (\mathbf{R}_{(Q, q_{init}, q_c, T)}(r_1, \ldots, r_c', \ldots, r_n), W'')} \quad [\mathrm{Raff}^c 2]$$

Pour que ces règles soient complètement définies, il faut prévoir le cas où le sous-programme raffinant un état est égal à NIL. D'où la règle suivante :

$$\frac{}{(nil, W) \xrightarrow{\emptyset/\emptyset,\emptyset_L} (nil, W)} \quad [\text{Nil}^c]$$

La règle [Raff^{c}1] traduit le cas où il y a évolution conjointe de l'automate de contrôle et du sous-programme raffinant, alors que la règle [Raff^{c}2] correspond à celui où seul le sous-programme raffinant évolue. Ces deux règles assurent le déterminisme des automates du programme : la règle [Raff^{c}1] ne peut être appliquée que si une seule transition dans l'automate de contrôle peut être déclenchée ; la règle [Raff^{c}2] ne s'applique que si aucune transition de l'automate de contrôle ne peut être déclenchée. Par conséquent, aucune règle ne s'applique si l'automate est non déterministe, et la fonction `CalculTrans` échoue.

La règle [Raff^{c}1] est telle que les variables actives dans l'état but sont mises à leurs valeurs initiales en tenant compte du contexte de départ de la transition. Les modifications engendrées par les affectations liées à la transition déclenchée dans l'automate de contrôle, ne sont donc pas prises en compte dans le calcul de ces valeurs initiales.

Pour la règle [Raff^{c}1] comme pour la règle [Par^c], l'ordre d'application de la fonction de modification d'un contexte par rapport à un autre contexte est indifférent.

La déclaration de variable de contrôle

Soit $I = \mathcal{I}(prog(p))$;

$$\frac{(p, W\{v \leftarrow \mathtt{v}\}) \xrightarrow{m_e[I]/O,U} (p', W')}{(p \Box (v : e : \mathtt{v} : \ \tau^{\not\infty}), W) \xrightarrow{m_e/O,U} (p' \Box (v : e : W\{W'\}(v) : \ \tau^{\not\infty}), W')} \quad [\text{Var}^c]$$

La nouvelle valeur courante de la variable v est soit contenue dans W', ce qui signifie qu'elle est modifiée par la transition, soit contenue dans W si elle ne l'a pas été.

La déclaration d'événements internes

Soient $P = prog(p)$ et $I = \mathcal{I}(prog(p))$;

$$\frac{\begin{array}{l} \forall m_Y \in \mathcal{M}_c^\star(Y \cap I), (p, W) \xrightarrow{m_e \wedge m_Y/O_i,U_i} (p_i, W_i) \\ \exists! m_Y \in \mathcal{M}_c^\star(Y \cap I) \text{ tel que } (p, W) \xrightarrow{m_e \wedge m_Y/O_i,U_i} (p_i, W_i) \qquad {m_Y}^+ \subseteq O_i \ \wedge \ {m_Y}^- \cap O_i = \emptyset \end{array}}{(\overline{p^Y}, W) \xrightarrow{m_e/O_i - Y,U_i} (\overline{{p_i}^Y}, W_i)} \quad [\text{Int}^c]$$

La déclaration d'événements internes n'est pas affectée par la présence des variables de contrôle.

3.2 Les états temporisés

3.2.1 Présentation informelle

Lors de la programmation d'un système réactifΓil est parfois utile de décrire des situations du type :

Si avant N réceptions de l'entrée e, telle réaction ne s'est pas produite, alors à la Nième réception de e déclencher une réaction particulière.

Considérons le cas d'une montre à affichage numérique ; ce type de spécification est présent à divers endroitsΓnotamment dans la description de la fonction réveil de la montre. Nous appelons *ms* l'entrée qui indique au programme qu'une unité de temps vient de s'écouler. Supposons que l'utilisateur de la montre ait réglé l'alarme à une certaine heure `H`. Lorsque l'heure courante est égale à `H`Γla sonnerie doit être déclenchéeΓet si avant N réceptions de l'entrée *ms* l'utilisateur n'a pas demandé son arrêtΓla sonnerie doit être interrompue pendant un certain temps puis de nouveau relancée. Ce type de fonctionnement doit être répété M fois.

C'est pour exprimer facilement ces situations en ArgosΓque la notion d'état temporisé a été introduite — cette notion était utilisée dans une version des Statecharts —. Sans celle-ciΓelles sont décrites en utilisant des composantes de type compteur dont le rôle consiste à compter les occurrences de l'événement utilisé pour exprimer le délaiΓafin d'indiquer le dépassement d'une borne. Ces compteurs doivent être lancésΓinterrompus et réinitialisés en fonction des réactions des autres composantes du programme.

L'intérêt des états temporisés est de rendre implicite la gestion de ces compteurs. En effetΓ bien que les situations à décrire soient simples à énoncerΓles programmes obtenus en utilisant les composantes de type compteur sont difficiles à manipuler : un compteur est associé à chaque délaiΓle nombre d'états du compteur peut être très important puisqu'égal à la valeur de la borneΓ les communications entre les compteurs et le reste du programme sont nombreuses.

Un état temporisé est un état d'un automate dans lequel on ne peut pas rester plus d'une certaine durée exprimée en nombre d'occurrences d'un événement. C'est une macro-notation : tout programme qui contient des automates dans lesquels apparaissent des états temporisésΓpeut être traduit automatiquement en un programme Argos dans lequel la temporisation se traduit par l'utilisation des opérateurs de base du langage (utilisation explicite de composantes de type compteur).

Afin de comprendre la signification intuitive d'un état temporiséΓil convient d'imaginer qu'à chaque état temporisé par le couple (d, e)Γest associé un compteur des occurrences de e. Ce compteur peut être remis à zéro. Il expire lorsque le nombre d'occurrences de e reçues est égal à d.

Examinons l'automate de la figure 3.4. GraphiquementΓun état temporisé est distingué par une étiquette de la forme [n e]Γoù n est un entier strictement positif et e est le nom d'un événement. Sur cet exempleΓseul l'état C est temporisé par un délai de quatre occurrences de l'événement b. Le carré noir sur l'une des transitions issues de l'état C indique que cette transition est celle déclenchée lorsque le délai expireΓnous l'appelons la transition *d'expiration*.

Cet automate se comporte de la manière suivante. Initialement dans son état AΓl'automate

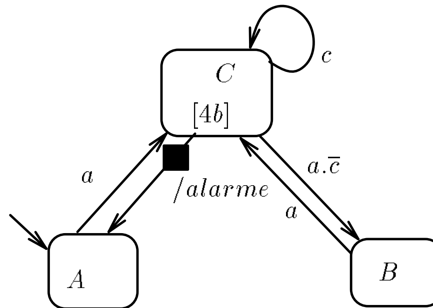


Figure 3.4: exemple d'automate contenant des états temporisés

sur réception de a passe dans l'état C. L'activation de l'état CΓ qui est un état temporiséΓ provoque implicitement le déclenchement d'un compteur des occurrences de b. C'est le cas de toutes les transitions ayant C pour état but. Toutes les transitions issues de C provoquent l'arrêt et la remise à zéro du compteur. La transition qui boucle sur l'état C provoque uniquement la remise à zéro de ce compteur (il est possible de considérer qu'il y a arrêt puis déclenchement). SiΓ une fois le compteur lancéΓ quatre occurrences de b sont reçues sans qu'auparavant aucune des transitions issues de C n'ait été déclenchéesΓ alors lors de la réception du quatrième bΓ la transition d'expiration est déclenchée : la sortie *alarme* est émise et l'état A devient actif. Il est important de noter que le déclenchement de la transition qui boucle sur l'état C suffit à retarder l'expiration du délai. Par conséquentΓ on peut être dans l'état C et "voir passer" plus de quatre occurrences de bΓ sans pour autant quitter C : il suffit pour cela qu'entre ces quatre occurrences de bΓ une occurrence de c soit présente. De plusΓ si au moment où le délai expireΓ une transition qui sort de l'état C peut être déclenchée — ce qui provoquerait la remise à zéro du compteur — la priorité est donnée à la transition d'expiration.

Le programme Argos qui définit le fonctionnement du réveil est donné par la figure 3.5. L'entrée H indique que l'heure courante est égale à l'heure programmée pour le déclenchement du réveil. L'entrée *stop* indique une action de l'utilisateur pour stopper la sonnerie. Les sorties ON et OFF sont les ordres qui commandent le lancement et l'arrêt de la sonnerie. EnfinΓ l'événement interne *arret* émis à chaque fois que la sonnerie s'arrête d'elle-mêmeΓ sert de référence pour exprimer le délai au bout duquel la sonnerie est définitivement arrêtée. Une montre à affichage numérique a été programmée en ArgosΓ la description de ce programme peut être trouvée en [JM92].

L'objectif de la fonction d'expansion est de transformer un programme Argos temporisé en un programme construit avec les opérateurs de base exprimant le comportement souhaité. Pour cela l'idée est la suivante :

- Associer un compteur à chaque état temporisé du programme. Si le délai associé à cet état est d occurrences de eΓ alors le compteur compte les occurrences de e et génère un événement spécial sur réception de la $d$ième occurrence de e. Cet événement est noté $tout_j$ où j est l'indice de l'état temporisé ;

- Transformer chaque automate temporisé en un automate qui ne l'est pas. Cette transformation modifie l'ensemble de transitionsΓ mais ne modifie ni l'ensemble des étatsΓ ni l'état

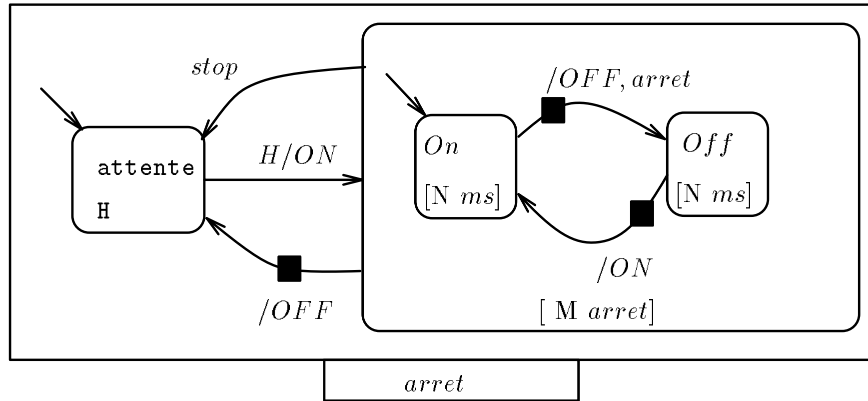


Figure 3.5: fonction réveil d'une montre

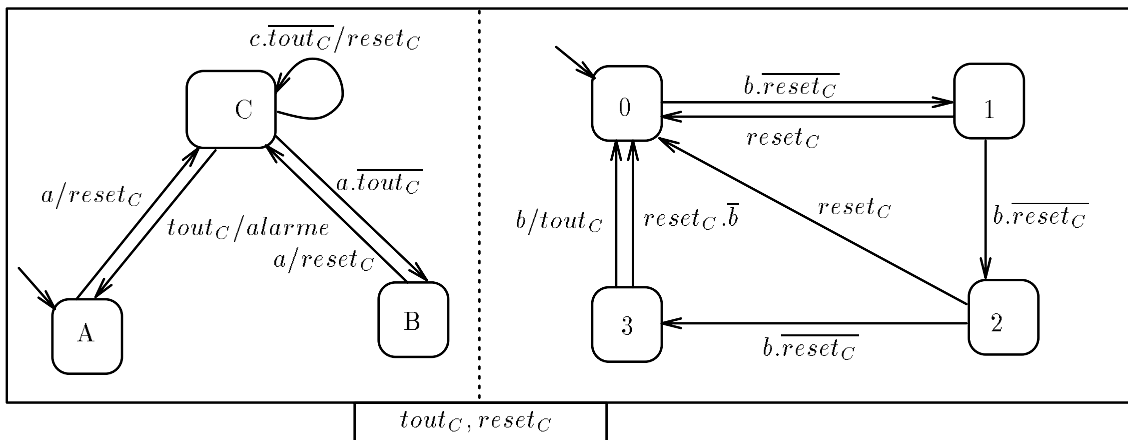


Figure 3.6: exemple d'expansion

initial. L'automate résultat doit contrôler les compteurs associés aux états temporisés contenus dans l'automate initial. Pour cela il génére des ordres de réinitialisation de la forme $reset_j$ à chaque fois que l'on entre dans un état temporisé d'indice j. De plusΓil réagit aux événements de la forme $tout_j$ qui indiquent l'expiration du délai de l'état d'indice j ;

- Mettre en parallèle tous les compteurs définis avec le programme obtenu en modifiant les automates temporisés du programme initialΓet rendre internes tous les événements de la forme $tout_j$ ou $reset_j$ utilisés.

L'application de cette fonction d'expansion au programme temporisé constitué par le seul automate de la figure 3.4Γconduit au programme de la figure 3.6.

3.2.2 Définition d'Argos temporisé

Syntaxe

Un programme Argos temporisé est noté P^t.

Par rapport à la syntaxe abstraite des programmes Argos donnée en 4.4 seuls les automates paramètres des opérateurs de raffinement sont modifiés : ce sont des automates temporisés.

Définition 3.11 Syntaxe abstraite des programmes Argos temporisés

$$
\begin{array}{lcl}
P^t & ::= & P^t \| P^t \\
 & | & \overline{P^{t}}^{Y} \\
 & | & \mathbf{R}_{(Q,q_{init},T,X,\mathcal{T})}(R_1, \ldots, R_n) \\
R & ::= & \text{NIL} | P^t
\end{array}
$$

■

Un automate temporisé est de la forme $(Q, q_{init}, T, X, \mathcal{T})$Γoù ;

- X est un sous-ensemble de QΓqui contient tous les états temporisés de l'automate ;
- $\mathcal{T}$est une fonction appelée *fonction de temporisation*. Elle est définie sur XΓet elle associe à chaque état temporisé un couple (d, e) où :
  - d est un entier donnant la borne de la temporisation ;
  - e est l'événement dont on comptabilise les occurrences.
- $T \subseteq Q \times \mathcal{M}_c^\star(In) \cup \{\square\} \times 2^{Out} \times Q$ est l'ensemble des transitions de l'automate. Dans cet ensemble les transitions d'expiration sont distinguées des autres transitions par la forme de leur monôme. Nous utilisons à cet effet un monôme spécial noté $\square$.

La simplification d'écriture des monômes utilisée dans la syntaxe concrète du langage Argos peut s'appliquer aux programmes Argos temporisés. La phase de saturation s'effectue alors en tenant pas compte des transitions d'expiration.

Sémantique statique des programmes Argos temporisés

Les règles de sémantique statique qui suivent complètent la définition de la syntaxe abstraite.

Définition 3.12 Sémantique statique des programmes Argos temporisés
Les ensembles de transitions des automates temporisés présents dans le programme doivent satisfaire les propriétés suivantes :

- *Tout état temporisé possède une et une seule transition d'expiration ;*
- *Un état non temporisé ne possède pas de transition d'expiration.*

■

Fonction d'expansion des programmes Argos temporisés

Dans toute la suiteΓnous supposons que les états des automates sont indicés globalement.

Le programme Argos résultat de l'application de la fonction d'expansion sur un programme temporisé P^t a la forme $\overline{P \|_{j \in \mathcal{X}(P^t)} C_j}^{Y}$Γoù :

- P est un programme Argos non temporisé de structure identique à celle de P^t. Seuls les automates de P^t ont été modifiés ;
- La fonction $\mathcal{X}$ détermine l'ensemble des indices des états temporisés d'un programme P^t(voir définition 3.13) ;
- C_j est le compteur associé à l'état temporisé d'indice j ;
- Y contient tous les événements internes utilisés pour établir les communications entre les compteurs et le reste du programme : $Y = \{reset_j, tout_j\} \quad \forall j \in \mathcal{X}(P^t)$.

Définition 3.13 Ensemble des états temporisés d'un programme

$$\begin{array}{lcl}
\mathcal{X}({P^t}_1 \| {P^t}_2) & = & \mathcal{X}({P^t}_1) \cup \mathcal{X}({P^t}_2) \\
\mathcal{X}(\overline{P^t}^{Y}) & = & \mathcal{X}(P^t) \\
\mathcal{X}(\mathbf{R}_{(Q, q_{init}, T, X, \mathcal{T})}(R_1, \ldots, R_n)) & = & X \cup \mathcal{X}(R_1) \cup \ldots \mathcal{X}(R_n) \\
\mathcal{X}(\text{NIL}) & = & \emptyset
\end{array}$$

■

Nous commençons par présenter la définition formelle des compteurs. Puis nous définissons la fonction de transformation d'un automate temporisé.

Le compteur associé à un état j temporisé par l'événement e avec la borne dΓcontient d états. Il admet deux entrées : e et $reset_j$ et une sortie : $tout_j$. Chaque état du compteur mémorise le nombre d'occurrences de e déjà comptabilisées. L'ensemble des états du compteur C_j est donc l'intervalle d'entiers $[0..d-1]$. Son état initial est 0. La réception de $reset_j$ provoque la remise à zéro du compteur (retour dans l'état 0)Γla réception du $d$ième e provoque l'émission de $tout_j$.

Définition 3.14 Définition des composantes compteurs
La composante compteur C_j associée à l'état temporisé j si $\mathcal{T}(j) = (d, e)$ est l'automate défini par $([0..d-1], 0, T)$ où T est l'union de deux ensembles $T1$ et $T2$. L'ensemble T_1 contient toutes les transitions issues des états d'indices compris entre 0 et $d-2$. L'ensemble T_2 contient les transitions issues de l'état $d-1$.

- $T_1 = \{(i, e.\overline{reset_j}, \emptyset, i+1), (i, e.reset_j, \emptyset, 0), (i, reset_j.\overline{e}, \emptyset, 0) \; \forall i < d-1\}$
- $T_2 = \{(d-1, e.reset_j, \{tout_j\}, 0), (d-1, e.\overline{reset_j}, \{tout_j\}, 0), (d-1, reset_j.\overline{e}, \emptyset, 0)\}$

■

Remarquons que ces composantes compteurs pourraient être définies en utilisant des variables de contrôle.

L'idée pour transformer un automate temporisé en un automate qui ne l'est pas mais qui se synchronise avec les compteurs associés à ses ex-états temporisés est la suivante :

- Les ensembles d'états et l'état initial de l'automate ne sont pas modifiés ;
- Une transition qui entre dans un état temporisé j doit réinitialiser le compteur (émission de $reset_j$) ;
- Une transition qui sort d'un état temporisé j et qui n'est pas la transition d'expirationΓdoit être conditionnée par l'absence de $tout_j$ pour assurer la priorité à la transition d'expiration ;
- La transition d'expiration qui sort de l'état temporisé j doit être remplacée par un ensemble de transitions. Ces transitions sont toutes conditionnées par la présence de $tout_j$. Le reste de la condition de déclenchement exprime une condition quelconque sur le statut des autres entréesΓce qui traduit la priorité de la transition d'expiration sur les autres transitions issues d'un état temporisé.

Définition 3.15 Modification des automates temporisés
Soit $(Q, q_{init}, T, X, \mathcal{T})$ un automate temporisé, l'automate non temporisé qui lui est associé est défini par le tuple (Q, q_{init}, T').

De plus, si $In((Q, q_{init}, T, X, \mathcal{T})) = In$ et $Out((Q, q_{init}, T, X, \mathcal{T})) = Out$ alors $In((Q, q_{init}, T')) = In \cup \{tout_j / j \in X\}$ et $Out((Q, q_{init}, T')) = Out \cup \{reset_j / j \in X\}$.

L'ensemble T' est défini à partir de T. Chaque transition t de T donne lieu à une transition t' dans T' qui dépend des caractéristiques des états source et but de t (états temporisés ou non) et du monôme de t (transition d'expiration ou non). Dans cet ensemble, les monômes des transitions doivent être complets sur l'ensemble des entrées de l'automate résultat. Pour simplifier la définition, nous présentons une solution où ces monômes sont complets sur l'ensemble des entrées pertinentes pour l'état source de la transition, et non pas sur l'ensemble de toutes les entrées de l'automate. Pour obtenir le véritable ensemble T', il faut encore compléter tous les monômes pour les rendre complets sur l'ensemble de toutes les entrées. C'est l'algorithme de construction de T' à partir de T qui sert de définition :

$T' \leftarrow \emptyset$
Pour tout $(i, m, O, j) \in T$ faire selon i, j et m
    $i \in X \wedge j \in X \wedge m \neq \square$:
        Ajouter $(i, m \bullet \overline{tout_i}, O \cup \{reset_j\}, j)$ à T'
    $i \in X \wedge j \in X \wedge m = \square$:
        Pour tout $m \in \mathcal{M}_c^\star(I)$ faire
            Ajouter $(i, tout_i \bullet m, O \cup \{reset_j\}, j)$ à T'
    $i \in X \wedge j \notin X \wedge m \neq \square$:
        Ajouter $(i, m \bullet \overline{tout_i}, O, j)$ à T'
    $i \in X \wedge j \notin X \wedge m = \square$:
        Pour tout $m \in \mathcal{M}_c^\star(I)$ faire
            Ajouter $(i, tout_i \bullet, O, j)$ à T'
    $i \notin X \wedge j \in X$:
        Ajouter $(i, m, O \cup \{reset_j\}, j)$ à T'
    $i \notin X \wedge j \notin X$:
        Ajouter (i, m, O, j) à T'

■

3.2.3 Un exemple de programme Argos avec temporisation

Nous reprenons dans cette section le contrôleur de feux de voiture décrit dans la section 2.3. Nous ajoutons aux fonctionnalités de ce contrôleur la possibilité d'effectuer des appels de phares. Cette nouvelle fonctionnalité est spécifiée de la manière suivante :

- Un appel de phare dure une seconde et demie ;
- Il est déclenché lorsque la manette est tirée vers le bas de la voiture. Cette nouvelle position de la manette n'est pas stable ; elle revient à sa position initiale si on ne l'y maintient pas. Cette nouvelle action est modélisée par l'entrée *appel*.
- Si les phares sont allumés lorsque la manette est tirée vers le basΓ alors ce sont les longue-portée qui subissent l'appel de phares. Si elles sont elles-mêmes déjà alluméesΓ alors l'état des différentes lampes est inchangé.

En plus de la nouvelle entrée *appel*Γune entrée qui indique le passage du temps est fournie par l'environnement pour que le système sache mesurer une durée. Nous supposons que le contrôleur dispose d'une entrée *ms* qui indique l'écoulement d'une milli-seconde. L'ensemble des sorties du contrôleur n'est pas modifié.

Nous avons choisi de modifier le programme donné par la figure 2.21 de la manière suivante : une composante de gestion des appels de phareΓnotée `C_Appel`Γest mise en parallèle aux trois composantes initiales. Elle ne reçoit pas directement l'entrée *appel* mais a pour entrée deux événements internes générés par la composante qui gère la manette (`C_M`). L'événement interne *appelP* (resp. *appelLP*) indique que les phares (resp. longue-portée) doivent subir un appel. Le programme du contrôleur de feux de voiture avec appel de phares est donné par la figure 3.7.

Ce programme n'est correct que si nous supposons que l'état des phares (ou des longue-portée si ce sont eux qui subissent l'appel) ne peut pas être modifié — les entrées *cp* et *cLP* ne peuvent pas être présentes — durant l'appel. Etant donnée la durée de cet appel (une seconde et demie) cette hypothèse nous semble justifiée.

3.2.4 Les limitations de la macro-notation

La macro-notation des états temporisés peut laisser penser par sa forme qu'elle permet de mesurer des durées. C'est en effet le casΓ mais ces durées ne sont pas mesurées de manière exacte. Considérons l'exemple du programme de la figure 3.8. Dans ce programmeΓl'entrée m indique l'écoulement d'une minute. Ce programme peut laisser penser que la sortie s est émise 3 minutes après la première occurrence de a. En réalitéΓla seule propriété assurée par ce programme est que la durée qui sépare l'émission de s et la réception de aΓest strictement supérieure à 2 et inférieure ou égale à 3.

Le chronogramme de la figure 3.8 permet de comprendre les raisons de cette approximation. D'après la sémantique des états temporisésΓl'émission de s est simultanée avec la 3ième occurrence de m qui suit la réception de a. Les occurrences de a et de m ne sont pas nécessairement simultanées. Par conséquentΓla première occurrence de m qui suit a indique l'écoulement

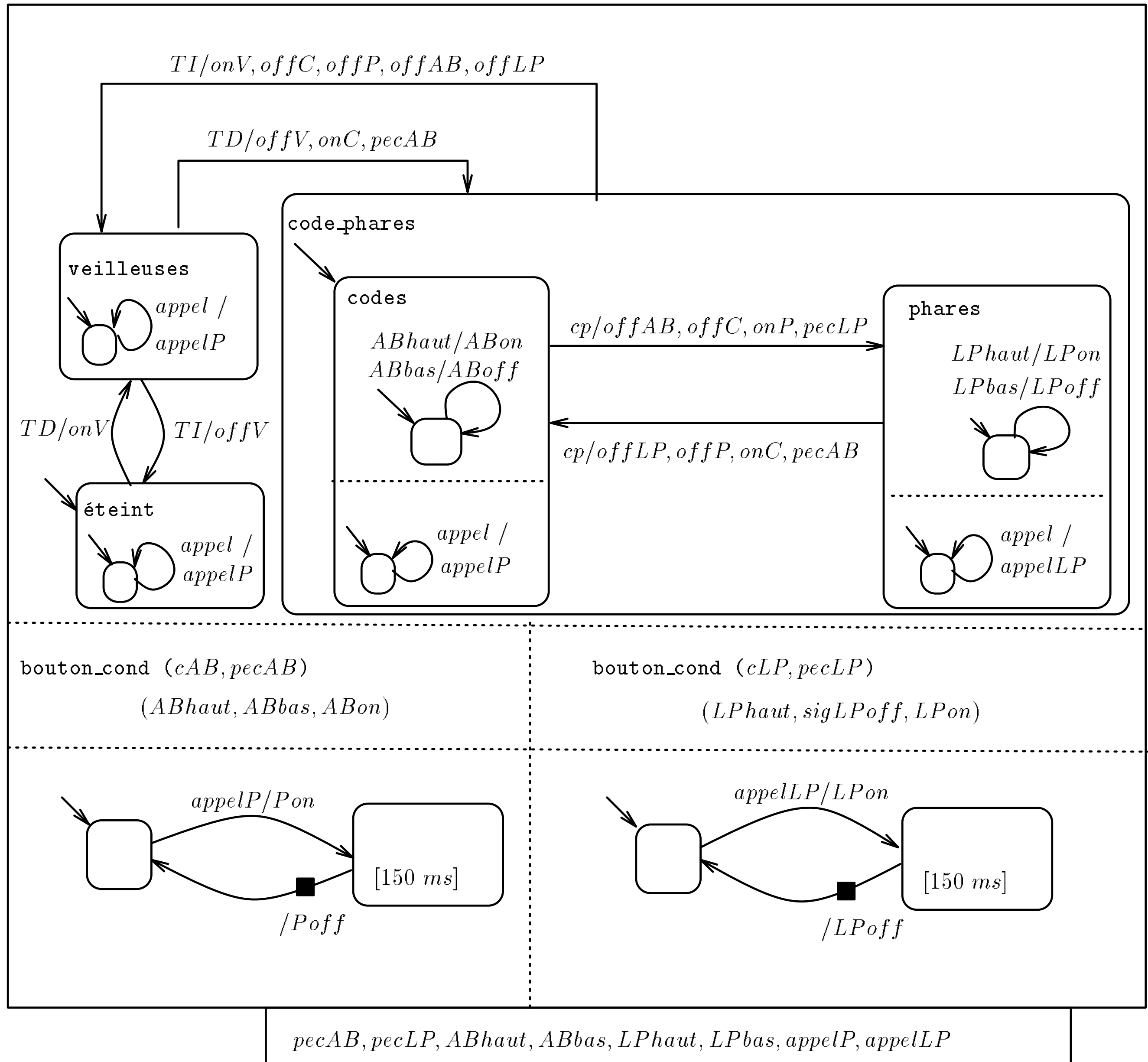


Figure 3.7: contrôleur de feux de voiture avec appel de phares

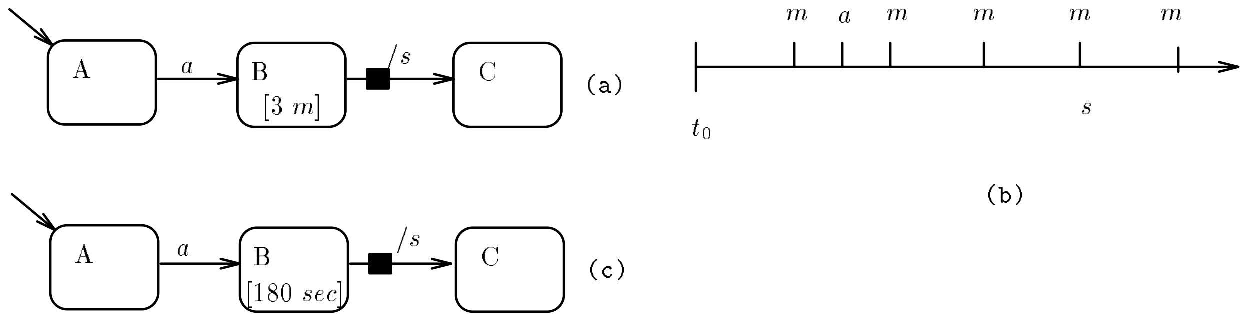


Figure 3.8: expression de durées à l'aide d'états temporisés

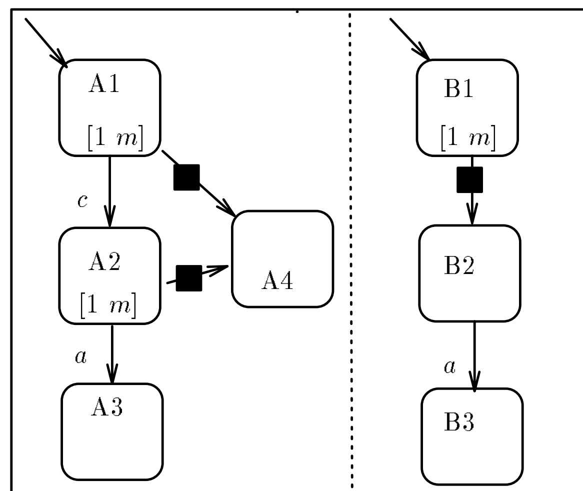


Figure 3.9: exemple de contre-sens sur la signification des états temporisés

d'une minute depuis la dernière occurrence de m et non pas l'écoulement d'une minute depuis la réception de aΓd'où l'approximation.

Remarquons que pour le même exemple l'utilisation d'une entrée qui mesure l'écoulement des secondes (cf. figure 3.8) permet d'obtenir des résultats plus proches de la durée souhaitée. En effet avec ce nouveau programmeΓla durée qui sépare s et a est comprise entre 179 secondes et 180 secondes.

De manière généraleΓsi l'on souhaite exprimer des durées à l'aide d'états temporisésΓil est conseillé d'utiliser des entrées sous-multiples de la base servant à définir ces durées.

Le caractère approché de l'expression des durées à l'aide d'états temporisés est parfois à l'origine de contre-sens important. Considérons l'exemple de la figure 3.9 et répondons à la question suivante : l'état (A3,B3) est-il accessible ?

Une mauvaise compréhension des états temporisés conduit à une réponse affirmative. Le raisonnement suivi est le suivant : avant la première minuteΓc est reçuΓl'automate A passe de l'état A1 à l'état A2 ; une minute depuis l'instant initial s'écoule et l'automate B passe de l'état B1 à l'état B2 ; avant l'écoulement de la première minute depuis la réception de cΓa est reçu et (A3,B3) est atteint. L'erreur dans ce raisonnement est la suivante : la première occurrence de m depuis l'instant initial fait non seulement passer l'automate B de l'état B1 à l'état B2Γmais elle est aussi la première occurrence de m depuis la réception de cΓpar conséquent elle fait passer l'automate A de A2 dans A4 et l'état (A3,B3) est inaccessible.

3.3 Récapitulation

Nous avons défini dans ce chapitre deux macro-notations : les variables de contrôle et les états temporisés. La première donne au programmeur un autre moyen de mémoriser de l'information sur l'histoire du système. Dans certaines situationsΓl'utilisation des variables de contrôle conduit à des programmes plus concis et plus proches des spécifications du système. La seconde macro-notation permet de limiter le temps de stationnement dans un étatΓce qui permet en utilisant l'opérateur de raffinementΓd'exprimer des structures de type “chien de garde temporisé”. Etant donné qu'elles peuvent être toutes les deux traduites à l'aide des opérateurs de base du langage ArgosΓelles ne modifient pas la forme des modèles des programmes.

Chapitre 4

ArgoLus : un langage mixte impératif et déclaratif

L'objectif de ce chapitre est de définir un langage mélangeant au niveau source les langages Argos et Lustre [JLMR94b]. Ce langage se nomme ArgoLusΓil constitue un nouveau moyen de description des automates booléensΓen combinant les avantages :

- De la programmation déclarative flots de données : un programme est un ensemble d'équations qui établissent des relations entre les séquences infinies des valeurs des entrées et des sorties (les flots). Ces équations doivent être vérifiées à chaque réaction du programme ;
- De la programmation impérative à base d'automates parallèles et hiérarchisés.

L'idée à l'origine du langage mixte ArgoLus consiste à identifier dans les deux langages des unités syntaxiques (ou blocs) équivalentes du point de vue sémantique pour pouvoir les interchanger : un bloc Lustre pouvant être utilisé comme un bloc Argos et inversement.

Pour définir formellement le langage ArgoLus nous suivons la démarche suivante :

- Définir les sémantiques des deux langages dans un style commun ;
- Inclure dans ces sémantiques la notion de blocs ;
- Modifier chaque sémantique pour prendre en compte la possibilité d'importer un bloc de l'autre langage.

IntuitivementΓla sémantique d'ArgoLus est obtenue par une union des sémantiques d'Argos et de LustreΓce qui permet d'assurer la préservation dans le langage mixte des sémantiques des langages qui le composent.

Le plan suivi dans ce chapitre est le suivant :

- La section 4.1 présente le langage Lustre. La sémantique de Lustre est présentée sous une forme différente de celle d'Argos qui convient mieux au style flots de données du langage.

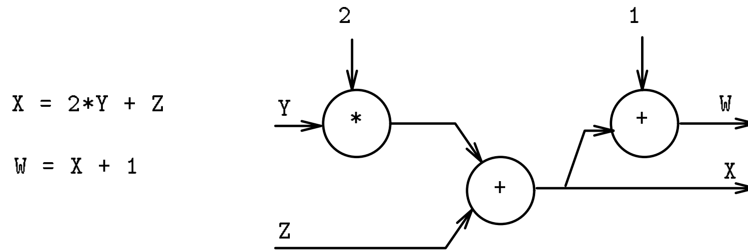


Figure 4.1: flots de données : forme équationnelle et réseau

- La section 4.2 présente une sémantique flots de données du langage Argos équivalente à la forme plus opérationnelle présentée dans le chapitre 2.2 ;
- La section 4.3 présente le langage ArgoLus en donnant : des considérations intuitives sur le langage mixte et les différentes étapes qui conduisent à la définition de sa sémantique comme l'union des sémantiques flots de données d'Argos et de Lustre ;
- La section 4.4 présente un exemple de système réactif décrit en utilisant le langage mixte ArgoLus et discute des apports du langage mixte par rapport aux deux langages qui le constituent ;
- La section 4.5 présente une solution possible pour mettre en œuvre la définition du langage ArgoLus. Cette solution consiste à traduire structurellement tout programme ArgoLus en un programme Lustre ;
- La section 4.6 conclut ce chapitre et présente des travaux proches de ceux présentés dans ce chapitre.

4.1 Le langage déclaratif flots de données Lustre

Des présentations détaillées de Lustre — plus adaptées à une initiation à ce langage que la présentation qui suit — peuvent être trouvées dans [CHPP87ΓHCRP91bΓHCRP91a].

4.1.1 Présentation informelle du langage Lustre

Le style de programmation flots de données

Pour programmer des systèmes réactifsΓle langage Lustre propose une approche *flots de données* [Kah74]. Dans cette approcheΓun système est constitué d'un réseau d'opérateurs agissant en parallèle au rythme de leurs entrées. Un programme Lustre peut être vu comme la transcription textuelle d'un tel réseau : des identificateurs sont associés à certains "fils" du réseau et les relations entre ces identificateurs sont représentées par des équations (cf. figure 4.1).

Pour décrire des systèmes réactifs à partir du modèle flot de donnéesΓ Lustre adopte l'hypothèse de synchronisme : les opérateurs du réseau sont des primitives idéales dont la "tra-

versée" par le flot des données prend un temps nul. Cette approche permet d'introduire de façon très simple une dimension temporelle. Les entrées du réseau sont interprétées comme des fonctions d'un même "temps" de référence ; puisque la traversée des opérateurs prend par hypothèse un temps nulΓla valeur des sorties est instantanément disponible. Les sorties sont donc des fonctions du même temps de référence que les entrées. Dans l'exemple de la figure 4.1Γcela revient à interpréter chaque variable comme une fonction du temps :

$$\begin{array}{lrcl} \forall t & \mathtt{X}(t) & = & 2\mathtt{Y}(t) + \mathtt{Z}(t) \\ & \mathtt{W}(t) & = & \mathtt{X}(t) + 1 \end{array}$$

L'approche flot de données n'est intéressante que s'il est possible de faire référence au passéΓ c'est-à-dire introduire des "retards" dans le réseau (l'équivalent des bascules dans un circuit logique). Ceci est possible en Lustre grâce à l'opérateur de mémorisation `pre`. En interprétant son argument comme une fonction du tempsΓ"`Y = pre X`" peut être vu comme la fonction :

$$\forall t \neq 0 \quad \mathtt{Y}(t) \quad = \quad \mathtt{X}(t-1)$$

L'introduction de retards pose le problème de la valeur d'un flot à l'instant initial. Pour initialiser correctement les flotsΓle programmeur dispose de l'opérateur binaire `->` ; par exempleΓpour "`Z = X -> Y`"Γnous avons :

$$\begin{array}{lrcl} & \mathtt{Z}(0) & = & \mathtt{X}(0) \\ \forall t \neq 0 & \mathtt{Z}(t) & = & \mathtt{Y}(t) \end{array}$$

Ces deux opérateurs permettent d'écrire des définitions récurrentes. Par exempleΓl'équation `N = 0 -> pre N + 1` décrit la suite des entiers naturels.

Nous précisons à présent les notions de flotsΓet les opérateurs Lustre sur ces flots.

Flots et opérateurs sur les flots

En LustreΓune expression de type τ dénote une suite infinie de valeurs du type τΓappelée flot. C'est la suite des valeurs de l'expression à chaque réaction du programme — on emploie plutôt en Lustre le terme d'instant —.

Les types booléenΓentier et réel sont prédéfinis. Les constantes de ces trois types s'écrivent de manière habituelle. Si c est une constanteΓ`c` dénote la suite infinie de c ; par exempleΓla constante entière `1` dénote la suite $(1, 1, 1, \cdots)$.

Les opérateurs arithmétiques et logiques habituels sont prédéfinis. Un opérateur `OP` de $\tau_1 \times \tau_2 \times \cdots \times \tau_n$ dans τ opère point à point sur les suites infinies de ces types. Par exemple :

$$\begin{array}{rrcl} \text{Si} & X & = & (x_1, x_2, \cdots, x_i, \cdots) \\ \text{et} & Y & = & (y_1, y_2, \cdots, y_i, \cdots) \\ \text{alors} & X + Y & = & (x_1 + y_1, x_2 + y_2, \cdots, x_i + y_i, \cdots) \end{array}$$

Le langage possède deux opérateurs qui permettent de manipuler les flots globalement :

- L'opérateur `pre` permet de décaler une suite dans le temps :

  si $X = (x_1, x_2, \cdots, x_i, \cdots)$ alors `pre` $X = (\bot, x_1, x_2, \cdots, x_i, \cdots)$.

  $\bot$ dénote une valeur indéfinie. Elle exprime le fait que la valeur d'une suite avant l'instant initial n'a pas de sens. Nous supposons que cette valeur est absorbante pour tous les opérateurs sur les valeurs.

- L'opérateur `->` permet d'initialiser un flot :

  si $X = (x_1, x_2, \cdots, x_i, \cdots)$ et $Y = (y_1, y_2, \cdots, y_i, \cdots)$ sont deux flots de même typeΓalors

  X `->` $Y = (x_1, y_2, \cdots, y_i, \cdots)$

  Cet opérateur permet d'initialiser correctement les suites de valeurs définies par un opérateur `pre`.

Equations

Un programme Lustre permet de définir des flots de sorties à partir de flots d'entrées etΓéventuellementΓde flots intermédiaires (variables locales). Toute variable *id* définissant un flot de sortie ou localΓest définie par une équation de la forme $id = exp$. Les expressions sont construites avec des constantesΓdes opérateurs sur les valeurs et sur les flotsΓmais aussi des variables locales et des entrées. La sémantique intuitive d'une équation est donnée par les principes suivants :

Principe de substitution : il y a synonymie complète entre l'expression et la variable. En particulierΓdans toute expression il est possible de remplacer la variable par l'expression qui la définitΓet réciproquement. Il en découle naturellement que l'ordre des équations n'a aucune incidence sur la sémantique d'un programme.

Principe de définition : le comportement de la variable est complètement déterminé par l'expression associée et le comportement des variables qui apparaissent dans cette expression. En particulierΓaucune information ne doit être déduite de la manière dont est utilisée la variable dans les autres définitions.

4.1.2 Exemples de programmes Lustre simples

Interrupteur de lampe

Cet exemple est traité en Argos dans la section 2.1.

Ce programme prend en entrée un flot booléen `comL` qui indique à chaque instant si l'interrupteur de la lampe est ou non commuté. Il produit un flot booléen `etat` qui indique à chaque instant si la lampe est éteinte (`etat = false`) ou allumée (`etat = true`). A l'instant initialΓla lampe est supposée éteinte.

Nous mettons en évidenceΓavec cet exempleΓune des différences avec Argos. En ArgosΓle même exemple admet deux sorties qui sont les ordres d'extinction et d'allumage de la lampe.

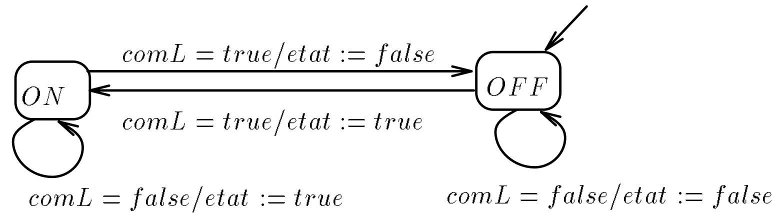


Figure 4.2: équivalent sous forme d'automate du nœud `interrupteur`

Une solution similaire en Lustre avec deux flots booléens pour sortiesΓcorrespondant aux deux ordres d'émission est possibleΓmais ce n'est pas la solution "naturelle" qui correspond au style de programmation Lustre.

```
node interrupteur(comL : bool) returns (etat:bool);
let
     etat = if comL then not (false -> pre(etat)) else (false -> pre(etat));
tel
```

L'opérateur conditionnel `if_then_else` opère point à point sur les flots.

Ce nœud Lustre est à rapprocher d'un automate à deux états (cf. figure 4.2) qui :

- Change d'états lorsque la valeur du flot `comL` est égale à `true` ;
- Met à jour la valeur de la sortie `etat` à chaque réaction du programme ;
- A pour état initial l'état OFF.

Compteur d'occurrences

Cet exemple met en évidence l'utilisation de flots non booléens. Le système décrit est un compteur généralisé d'occurrences d'un signal `incr`Γparamétré par une valeur initiale `init` et une valeur d'incrément `step`. Ces deux paramètres constituent deux des flots d'entrées du programme. Ce sont deux flots entiers. Nous utilisons aussi un flot booléen `reset` qui indique si le compteur doit être réinitialisé dans l'instant courant. L'unique flot de sortie est un flot d'entier `n` qui donne la valeur du compteur à chaque instant.

```
node compteur(init,step:int; reset,incr:bool) returns (n:int);
let
     n = init -> if reset then init
                  else if incr then pre(n) + step else pre(n);
tel
```

4.1.3 Les horloges dans les programmes Lustre

Dans les deux exemples de programmes Lustre présentésΓles flots d'entrées et de sorties portent une valeur à chaque instant. Il n'y a pas comme en ArgosΓde notion d'absence d'un événement.

Le mécanisme des horloges permet en Lustre de définir des flots qui ne sont définis qu'à des instants particuliers. Ce mécanisme est surtout important lorsqu'on manipule des flots non booléens. En effetΓles valeurs portées par un flot booléen s'interprètent naturellement comme une information sur son statut : *true* indique la présenceΓalors que *false* indique l'absence. Puisque par la suiteΓnous nous intéressons qu'aux systèmes ayant des entrées et sorties booléennes — pour montrer la faisabilité d'un mélange au niveau source d'Argos et Lustre — nous ne prenons pas en compte le mécanisme des horloges. Par conséquentΓnous ne détaillons pas dans cette présentation de LustreΓce mécanisme. Une telle présentation peut être trouvée dans [Ray91].

4.1.4 L'appel de nœuds en Lustre

Le langage Lustre est un langage modulaire. Un programmeΓappelé *nœud*Γest constitué d'une spécification d'interface — noms et types des paramètres formels d'entrée et de sortie — et d'un système d'équations définissant les sortiesΓet d'éventuelles variables localesΓen fonction des entrées. Tout nœud Lustre peut être réutilisé dans la définition d'un autre nœudΓsauf dans sa propre définition. Une phase d'expansion permet d'éliminer — en recopiant le corps des nœuds — tous les appels de nœuds qui se trouvent à l'intérieur de la définition d'un autre nœud. Il s'ensuit que la fonction sémantique est définie sur un nœud qui ne contient pas d'appel à d'autres nœuds.

4.1.5 Contrôleur de feux de voiture

Cet exemple est traité en Argos dans la section 2.3. Le programme Lustre admet les mêmes entrées que le programme Argos : `TD, TI, cP, cAB,` et `cLP`. Il est possible en Lustre de définir l'exclusion mutuelle de ces cinq entrées par l'expression `assert #(TD,TI,cP,cAB,cLP)`. Les sorties du programme Lustre sont les états des cinq feux de la voitureΓet non les ordres d'allumage et d'extinction de ces feux. Le programme est construit à l'aide du nœud `interrupteur` défini précédemment et d'un autre nœud nommé `jafter`. Ces deux nœuds sont très souvent utilisés lors de la programmation en Lustre.

Le nœud `jafter` ("just after") prend en entrée un flot booléen et produit un flot de sortie qui indique si l'instant courant suit directement un instant où l'entrée était vraie. Il en découle naturellement que la sortie est initialement fausse.

```
node jafter(x : bool) returns (s : bool);
let
     s = false -> pre x;
tel
```

Nous donnons à présent le nœud principal.

```
node FEUX(TD,TI,cP,cAB,cLP : bool)
returns (veilleuses, codes, phares, longue_portee, anti_brouillard : bool);
var
     codes_phares : bool;
     ab_on : bool;
     lp_on : bool;
```

```
let
     assert #(TD,TI,cP,cAB,cLP);
     ab_on = interrupteur(false, cAB);
     lp_on = interrupteur(false, cLP);
     veilleuses = if jafter(veilleuses) then (not TI and not TD)
                  else if jafter(codes_phares) then TI
                       else TD;
     codes_phares = if jafter(codes_phares) then not TI
                   else jafter(veilleuses) and TD;
     codes = codes_phares and not phares;
     phares = if jafter(phares) then (not TI and not cP)
             else jafter(codes_phares) and cP ;
     longue_portee = phares and lp_on;
     anti_brouillard = codes and ab_on;
tel
```

Le nœud `interrupteur` est utilisé pour mémoriser l'état des boutons associés aux anti-brouillard et aux longue-portée. Ces deux appels sont à rapprocher des appels à la procédure `bouton_cond` dans le programme Argos. Une variable locale `codes_phares` mémorise la position de la manette dans laquelle soit les codesΓsoit les phares sont allumés. Cette variable est à rapprocher de l'état du même nom dans le programme Argos. Les contraintes de fonctionnement des anti-brouillard et des longue-portée s'expriment plus simplement qu'en Argos car l'état des boutons est directement accessible grâce aux variables `ab_on` et `lp_on`.

4.1.6 Les erreurs possibles dans un programme Lustre

Les problèmes de causalité

Nous avons vu qu'en Argos un problème de causalité survient lorsque le statut d'un événement interne ne peut pas être défini de manière unique. La communication en Lustre se traduit par l'utilisation à l'intérieur d'un même programme de variables qui sont à la fois définies par une équationΓcomme des sortiesΓet utilisées en partie droite d'une équationΓcomme des entrées. Les valeurs de ces variables locales ne sont pas définies par l'environnementΓmais par le programme lui-même. Ce sont elles qui établissent les communications entre différentes parties du programme. C'est aussi une diffusion synchroneΓcar à chaque instant la valeur d'une variable est unique et peut être lue par toutes les autres composantes du programme.

Une mauvaise utilisation du mode de communication se traduit par l'impossibilité à un instant donné de définir la valeur d'une variable de manière uniqueΓsoit parce que l'équation qui la définit n'admet pas de solutionΓsoit parce qu'elle en admet plusieurs.

Considérons par exemple les deux programmes Lustre suivantsΓqui sont deux équivalents possibles aux programmes donnés pour illustrer les problèmes de causalité en Argos (cf. figure 2.8).

```
node plusieurs_sol(e : bool) returns (s : bool);
var a,b : bool;
let
     s = e and a ;
     a = e and b;
     b = e and a;
tel
```

Ce système d'équations booléennes admet deux solutions quand `e` est vraiΓles valeurs des variables locales `a` et `b` ne sont donc pas définies de manière unique. En revancheΓdans l'exemple suivantΓle système d'équations quand `e` est vraiΓn'admet pas de solution.

```
node zero_sol(e : bool) returns (s : bool);
var a,b : bool;
let
     s = e and a ;
     a = e and not b;
     b = e and a;
tel
```

On peut montrer qu'un système d'équations qui n'admet pas une solution unique est nécessairement bouckléΓc'est-à-dire que la valeur d'une variable dépend instantanément d'elle-même. Pour simplifier sa sémantiqueΓle langage Lustre utilise un critère syntaxique qui permet de rejeter tous les programmes qui contiennent des systèmes d'équations bouclés.

Ce critère syntaxique est trop fortΓce qui signifie qu'une fausse dépendance — dépendance syntaxique — comme celle présente dans le programme suivantΓconduit au rejet d'un programme correct.

```
node faux_probleme(e : bool) returns (s : bool);
var a,b : bool;
let
     s = e and a ;
     a = e and b and not b;
     b = e and a;
tel
```

L'utilisation d'un critère approché de détection des problèmes de causalité peut sembler surprenantΓcependant :

- Lustre manipule des valeurs de type quelconque et la résolution d'un système d'équations non booléennes estΓdans le cas généralΓun problème indécidable. Le problème est le même pour Argos lors de l'introduction dans le langage de données non-booléennes ;
- L'expérience de la programmation en Lustre montre que ce critère syntaxique est adapté au style de programmation du langage. Autrement ditΓil y a peu de chances pour qu'un programmeur Lustre écrive un programme quiΓbien que correctΓsoit rejeté. Nous verrons plus loin toutefoisΓque cette détection syntaxique doit faire l'objet d'une attention particulière lorsque les programmes Lustre sont générés automatiquement — par exempleΓpar traduction structurelle de programmes Argos —.

Les erreurs d'initialisation des opérateurs `pre`

Il existe en Lustre un type d'erreur qui n'a pas d'équivalent en Argos. Ces erreurs proviennent d'une mauvaise utilisation des opérateurs sur les flots. Considérons le programme Lustre suivant :

```
node pb_initialisation(e : bool) returns (s : bool);
let
    s = e and pre e;
tel
```

Lors de la première réaction du programmeΓla sortie `s` est indéfinie. Par conséquentΓce programme est incorrect. Pour le corrigerΓil faut initialiser correctement le flot défini par `pre e`. Une solution est donnée ci-dessous :

```
node no_pb(e : bool) returns (s : bool);
let
    s = e and (true -> pre e);
tel
```

Il est possible de définir des critères syntaxiques de détection de ce type d'erreurs. Ces critères sont tous trop forts : ils rejettent des programmes corrects. Pour cette raisonΓle compilateur Lustre-V3 [Ray91] implémente un critère de détection dynamique exact. Pour simplifier la sémantique de LustreΓnous présentons ici une détection syntaxique. Le critère utilisé est le suivant : tout programme qui utilise dans une équation un opérateur `pre` non précédé immédiatement d'un opérateur `->` est rejeté.

4.1.7 La sémantique flots de données de Lustre

La sémantique flots de données associe à un flot d'entrées un flot de sorties. Les flots d'entrées et de sorties d'un programme Lustre sont donnés sous forme de séquences infinies de mémoires.

La notion de mémoire

Une mémoire est une fonction partielle de l'ensemble des identificateurs vers l'ensemble des constantes. Dans la suite du chapitreΓnous dénotons une constante par k. La fonction `Def` appliquée à une mémoire permet d'identifier l'ensemble sur lequel elle est définie. Nous notons $Mem(Idf)$Γl'ensemble de toutes les mémoires définies sur l'ensemble d'identificateurs Idf (ce sont des fonctions totales sur Idf).

Un flot d'entrées (resp. de sorties) d'un programme Lustre est une séquence infinie de mémoires définies sur l'ensemble de ses entrées (resp. sorties)Γoù la $i$ème mémoire mémorise la valeur des entrées (resp. sorties) lors la $i$ème activation du programme.

Nous définissons les opérations sur les mémoires suivantes :

- **Affectation** : $\sigma[k/idf]$Γoù $\sigma \in Mem(Idf)$Γdésigne une mémoire $\sigma' \in Mem(Idf \cup \{idf\})$ définie par :

$\sigma'(idf) = k$
$\sigma'(idf') = \sigma(idf') \;\; \forall idf' \neq idf$

- **Concaténation** : $\sigma_1.\sigma_2$Γoù $\sigma_1 \in Mem(Idf1)$Γ$\sigma_2 \in Mem(Idf2)$ désigne une mémoire $\sigma' \in Mem(Idf1 \cup Idf2)$ définie par :

  $\sigma'(idf) = \sigma_1(idf)$ ssi $idf \in Idf1 \setminus Idf2$
  $\sigma'(idf) = \sigma_2(idf)$ ssi $idf \in Idf2$

  La mémoire σ_2 est "prioritaire" par rapport à σ_1.

- **Filtrage** : $\sigma_{|Idf2}$ où $\sigma \in Mem(Idf1)$Γdésigne une mémoire $\sigma' \in Mem(Idf1 \cap Idf2)$ définie par :

  $\sigma'(idf) = \sigma(idf)$ ssi $idf \in Idf1 \cap Idf2$

La mémoire qui n'est définie en aucun point est notée $\sigma_\emptyset$. Une séquence infinie de mémoires est notée : $\sigma_1\sigma_2 \ldots \; \sigma_n \ldots$

Syntaxe abstraite et sémantique statique

Nous ne nous intéressons pas ici au problème de vérification statique des types d'un programme. Les types des variables manipulés par un programme Lustre n'apparaissent donc pas dans la syntaxe abstraite.

Un programme Lustre est défini par trois ensembles de variables et une liste d'équations. Les ensembles de variables définissent respectivement : l'ensemble des entrées du programmeΓ l'ensemble de ses sorties et l'ensemble de ses variables locales.

Définition 4.1 Syntaxe abstraite des programmes Lustre

Pl	::=	(In, Out, Loc, Eqs)	*Programme principal*
Eqs	::=	$Eq \; Eqs \mid Eq$	*Liste d'équations*
Eq	::=	$idf \;=\; Exp$	*Equation*
Exp	::=	$k \mid idf \mid Exp$ `->` Exp	
		$\mid$ `pre`$(Exp) \mid$ `Dataop`$(Exp, \ldots, Exp)$	*Expression*

■

Pour définir les contraintes de sémantique statique qui s'appliquent à un programme LustreΓ nous supposons définies les fonctions suivantes sur les listes d'équations. Ces fonctions sont toutes illustrées sur la liste d'équations L donnée ci-dessous :

```
      | e1 = e2 -> pre(e3 or e4);
L =   | e3 =  e4 or pre(e1);
      | e4 = e3 -> pre(e2)
```

- `IdfD` et `IdfU` définissent respectivement l'ensemble des identificateurs définis par une liste d'équationsΓc'est-à-dire qui apparaissent en partie gauche d'une équationΓet l'ensemble des identificateurs utilisés par les expressions qui apparaissent dans ces équations.

  `IdfD`(L) `= {e1, e3, e4}` `IdfU`(L) `= {e1,e2,e3,e4}`.

- `NoCycle` est un prédicat satisfait si et seulement si la liste d'équations n'est pas bouclée. Elle l'est si et seulement si le graphe de la relation de dépendance binaire $\mathcal{R}el$ entre identificateurs contient un cycle. Le couple $(idf1, idf2)$ appartient à $\mathcal{R}el$ si et seulement s'il existe une équation de la forme $idf1 = exp$ dans la listeΓet si $idf2$ est utilisé dans expΓexcepté le cas où toutes les occurrences de $idf2$ sont à l'intérieur d'une expression argument d'un opérateur `pre`.

  `NoCycle(`L`) = false` car $\mathcal{R}el$= {`(e1,e2), (e3,e4), (e4,e3)`}

- `UDef` est un prédicat satisfait si et seulement si les identificateurs ne sont définis qu'une seule fois.

  `UDef(`L`) = true`

- `PreOK` est un prédicat satisfait si et seulement si toutes les occurrences de l'opérateur `pre` sont précédées immédiatement de l'opérateur `->`.

  `PreOK(`L`) = false`

Les trois ensembles de variables qui définissent un programme Lustre ne sont pas indépendants de la liste d'équations qui constitue le corps du programme. La contrainte statique (i) — donnée ci-dessous — établit les liens entre ces différentes parties du programme. Les identificateurs définis par la liste d'équations sont partagés entre variables locales et sorties. Les entrées doivent faire partie des identificateurs utilisés par les équations mais ne peuvent pas être des variables locales. EnfinΓtout identificateur qui apparaît dans la liste d'équations doit faire partie d'au moins un des trois ensembles de variables.

Un programme Lustre correct du point de vue de la sémantique statique a des ensembles d'entrées et de sorties disjoints (cf. contrainte (ii)).

La liste d'équations doit de plus satisfaire l'absence de cycleΓl'unicité des définitions et la correcte initialisation des opérateurs `pre` (cf. contraintes (iii)Γ(iv) et (v)).

Définition 4.2 Sémantique statique des programmes Lustre
Un programme Lustre (In, Out, Loc, Eqs) est correct du point de vue de la sémantique statique si et seulement si :

(i) $Loc \subseteq \texttt{IdfD}(Eqs)$, $Out = \texttt{IdfD}(Eqs) \setminus Loc$, $In \subseteq \texttt{IdfU}(Eqs) \setminus Loc$,
$In \cup Out \cup Loc \supseteq \texttt{IdfD}(Eqs) \cup \texttt{IdfU}(Eqs)$

(ii) $In \cap Out = \emptyset$

(iii) $\texttt{NoCycle}(Eqs)$

(iv) $\texttt{UDef}(Eqs)$

(v) $\texttt{PreOK}(Eqs)$

■

Sémantique flots de données du langage Lustre

Le but de la sémantique flots de données est de décrire un programme Lustre Pl comme une “machine” qui construit une séquence infinie de mémoires de sorties à partir d'une séquence infinie de mémoires d'entrées. Par conséquentΓune exécution du programme peut être caractérisée par le prédicat $\sigma_i^1\ \sigma_i^2\ \ldots\ \sigma_i^n\ \ldots \vdash Pl : \sigma_o^1\ \sigma_o^2\ \ldots\ \sigma_o^n\ \ldots$ où les σ_i sont des mémoires sur les entrées du programme et les σ_o sont des mémoires sur ses sorties.

La construction de la séquence infinie des mémoires de sorties est décrite de manière cyclique. Chaque cycle de calcul définit une réaction du programmeΓla $i$ème mémoire de sorties étant associée à la $i$ème mémoire d'entrées de la séquence. Une telle mémoire de sorties ne peut être définie en considérant uniquement le programme et les valeurs courante des entrées à cet instant. Elle dépend aussi d'informations héritées du passé. Le seul élément de mémorisation d'un programme Lustre est l'opérateur `pre`. Par conséquentΓde la même façon qu'en Argos la mémorisation de l'histoire du programme se fait en conservant l'état courant des automatesΓla mémorisation de l'histoire d'un programme Lustre s'effectue en conservant les dernières valeurs prises par les expressions opérandes des opérateurs `pre` du programme.

Nous ajoutons un paramètre à l'opérateur `pre` pour conserver la valeur précédente de son opérande. Les différents états d'un programme ne diffèrent donc que par les constantes associées aux opérateurs `pre`.

Définition 4.3 Forme syntaxique des états du modèle d'un programme Lustre

$$
\begin{array}{lcl}
pl & ::= & (In, Out, Loc, eqs) \\
eqs & ::= & eq\ eqs \mid eq \\
eq & ::= & idf\ =\ exp \\
exp & ::= & k \mid idf \mid exp\texttt{->}exp \\
 & & \mid \texttt{pre}(exp, k) \mid \texttt{Dataop}(exp, \ldots, exp)
\end{array}
$$

■

Par extension de la notion d'état d'un programme LustreΓnous pouvons parler d'état d'une liste d'équationsΓnoté eqsΓd'état d'une équationΓnoté eqΓou encore d'état d'une expressionΓ noté exp.

A l'instant initialΓles expressions calculées par les opérateurs `pre` du programme n'ont pas encore pris de valeur. Grâce à la correcte initialisation des opérateurs `pre` — imposée par la sémantique statique — nous verrons que la valeur donnée à ces expressions à l'instant initial est indifférente. Nous choisissons de les initialiser avec la valeur spéciale $\perp$.

Nous définissons l'état initial d'un programme Lustre $Pl = (In, Out, Loc, Eqs)$ comme étant égal à $(In, Out, Loc, Init(Eqs))$Γoù la fonction $Init$ transforme les occurrences des opérateurs `pre` de la forme $\texttt{pre}(Exp)$ dans la liste d'équations Eqs en $\texttt{pre}(Init(Exp), \perp)$.

Le fait qu'à un instant iΓdans un état plΓle programme Pl associe à une mémoire d'entrées σ_iΓla mémoire de sorties σ_o et atteint le nouvel état pl'Γest noté par le prédicat $\sigma_i \vdash pl \xrightarrow{\sigma_o} pl'$

Ce prédicat est étendu :

- Aux listes d'équations $\sigma_i \vdash eqs \xrightarrow{\sigma_o} eqs'$;
- Aux équations $\sigma_i \vdash eq \xrightarrow{\sigma_o} eq'$.

Ces prédicats signifientΓen prenant pour exemple celui sur les listes d'équationsΓque dans la mémoire σ_iΓla liste d'équations eqs est évaluableΓc'est-à-dire que tous les identificateurs dont on utilise la valeurΓsont définis dans σ_i ; la mémoire σ_o est le résultat de cette évaluationΓet eqs' définit le nouvel état de la liste d'équations après cette évaluation.

Nous utilisons aussi le prédicat $\sigma_i \vdash exp \xrightarrow{k} exp'$ qui signifie que l'expression exp est évaluable dans σ_iΓk est le résultat de cette évaluation et exp' est le nouvel état de l'expression après cette évaluation.

Le programme principal

$$\frac{\forall n \geq 1,\ \sigma_i^n \vdash eqs^n \xrightarrow{\sigma_o^n} eqs^{n+1} \qquad eqs^1 = Init(Eqs)}{\sigma_i^1\ \sigma_i^2\ \ldots\ \sigma_i^n\ \ldots \vdash (In, Out, Loc, Eqs) : \sigma^1_{o|Out}\ \sigma^2_{o|Out}\ \cdots\ \sigma^n_{o|Out}\ \cdots} \quad \text{[Pl]}$$

Le flot des sorties est construit cycle après cycle. Pour le premier cycle d'activationΓl'état courant est constitué par l'état initial du programme. Pour les cycles suivantsΓl'état courant est donné par l'état du programme à la fin du cycle précédent.

La liste d'équations

$$\frac{\sigma_i \vdash eq \xrightarrow{\sigma_o} eq' \qquad \sigma_i.\sigma_o \vdash eqs \xrightarrow{\sigma'_o} eqs'}{\sigma_i \vdash eq\ eqs \xrightarrow{\sigma_o.\sigma'_o} eq'\ eqs'} \quad \text{[Eqs1]}$$

$$\frac{\sigma_i \vdash eqs \xrightarrow{\sigma_o} eqs' \qquad \sigma_i.\sigma_o \vdash eq \xrightarrow{\sigma'_o} eq'}{\sigma_i \vdash eq\ eqs \xrightarrow{\sigma_o.\sigma'_o} eq'\ eqs'} \quad \text{[Eqs2]}$$

Grâce aux contraintes de sémantique statiqueΓles mémoires σ_i, σ_o et σ'_o sont toutes définies sur des ensembles disjoints. Par conséquentΓl'ordre d'application de l'opérateur de concaténation est indifférent.

Deux règles sont nécessaires pour permettre l'évaluation d'une liste d'équations dans n'importe quel ordre. En effetΓil faut éventuellement retarder l'évaluation d'une équation pour que toutes les variables dont elle dépendΓsoient disponibles dans la mémoire courante.

Une équation

$$\frac{\sigma_i \vdash exp \xrightarrow{k} exp'}{\sigma_i \vdash idf = exp \xrightarrow{\sigma_\emptyset[k/idf]} idf = exp'} \quad \text{[Eq]}$$

Une expression

$$\frac{}{\sigma_i \vdash k \xrightarrow{k} k} \quad \text{[E1]}$$

$$\frac{idf \in \mathtt{Def}(\sigma_i)}{\sigma_i \vdash idf \xrightarrow{\sigma_i(idf)} idf} \qquad \text{[E2]}$$

Cette règle indique qu'il est possible d'évaluer une expression réduite à un identificateur si et seulement si la valeur de cet identificateur est définie dans la mémoire courante.

$$\frac{\sigma_i \vdash exp \xrightarrow{k'} exp'}{\sigma_i \vdash \mathtt{pre}(exp, k) \xrightarrow{k} \mathtt{pre}(exp', k')} \qquad \text{[E3]}$$

$$\frac{(\sigma_i \vdash exp_1 \xrightarrow{k} exp'_1) \qquad (\sigma_i \vdash exp_2 \xrightarrow{k'} exp'_2)}{\sigma_i \vdash exp_1 \mathtt{->} exp2 \xrightarrow{k} exp2'} \qquad \text{[E4]}$$

$$\frac{\forall j \leq n,\ (\sigma_i \vdash exp_j \xrightarrow{k^j} exp'_j) \qquad k = \mathtt{Dataop}(k^1, \ldots, k^n)}{\sigma_i \vdash \mathtt{Dataop}(exp_1, \ldots, exp_n) \xrightarrow{k} \mathtt{Dataop}(exp'_1, \ldots, exp'_n)} \qquad \text{[E5]}$$

D'après la règle [E3] et la forme de l'état initial d'un programmeΓl'évaluation d'un opérateur `pre`Γlors de la première activation d'un programme LustreΓfournit toujours pour résultat la valeur $\perp$. Grâce à la règle [E4]Γcette valeur n'est pas propagée si l'opérateur `pre` est “bien gardé” par un opérateur d'initialisation (ce qui est imposé par la sémantique statique).

Restriction aux entrées et sorties booléennes

La sémantique ci-dessus est valable quels que soient les types des flots utilisés dans le programme Lustre. Nous nous limitons par la suite aux flots booléens. Avec cette restrictionΓune mémoire est une fonction partielle de l'ensemble des identificateurs dans l'ensemble $\{\mathtt{true},\mathtt{false},\perp\}$. Puisque la valeur $\perp$ n'est jamais propagéeΓnous pouvons la remplacer indifféremment par `true` ou `false`. Nous choisissons de la remplacer par `true`.

Grâce à cette limitation aux flots booléensΓl'ensemble des états possibles d'un programme Lustre est fini. Il est par conséquent possible de définir l'automate booléen modèle d'un programme LustreΓen donnant par exemple un algorithme de calcul du modèle d'un programmeΓ similaire à celui utilisé pour définir le langage Argos.

4.2 Une sémantique flots de données d'Argos

Pour pouvoir définir la sémantique du langage mixte ArgoLus comme une union des sémantiques des deux langages qui le composentΓnous donnons à la sémantique d'Argos une forme semblable à celle de LustreΓd'où le terme de sémantique flots de données d'Argos. Nous donnons à la fin de cette présentationΓle lien entre la définition d'Argos par cette sémantique flots de donnéesΓ et celle donnée dans la section 2.2.

4.2.1 Syntaxe abstraite et sémantique statique

Pour définir la sémantique flots de données d'ArgosΓnous faisons apparaître dans la syntaxe abstraite du langage la notion de programme principal et une notion de profil des programmes.

Définition 4.4 Syntaxe abstraite des programmes Argos

$$
\begin{array}{lcllll}
Pa & ::= & (In, Out, Ps) & & \textit{Programme principal} \\
Ps & ::= & Ps \| Ps & & \textit{Mise en parallèle} \\
 & | & \overline{Ps^{Y}} & & \textit{Déclaration d'événements internes} \\
 & | & \mathbf{R}_A(R_1, \ldots, R_n) & \textit{avec } A \in \mathcal{A}_d^{\emptyset,c} & \textit{Raffinement paramétré par } A \\
R & ::= & \textsc{nil} \mid Ps & &
\end{array}
$$

■

Définition 4.5 Contrainte statique sur les programmes principaux
Un programme principal (In, Out, Ps) *est correct statiquement si et seulement si*

$$(In = \mathcal{I}(Ps)) \wedge (Out = \mathcal{O}(Ps)) \wedge (In \cap Out = \emptyset)$$ ■

Les fonctions $\mathcal{I}$ et $\mathcal{O}$ sont respectivement définies en 2.3 et 2.4. Elles calculent l'ensemble des événements utilisés soit comme entréesΓsoit comme sorties à l'intérieur du programmeΓsans être déclarés internes à celui-ci.

4.2.2 Sémantique flots de données d'Argos

Comme pour lustreΓles valeurs des entrées et sorties du programmeΓc'est-à-dire leurs statutsΓsont déterminés par des mémoires booléennes définies partiellement sur un ensemble d'identificateurs.

Nous utilisons pour définir la sémantique flots de données d'Argos les prédicats $\sigma \vdash m_e$ et $\sigma \not\vdash m_e$ définis comme suit :

Définition 4.6

- $\sigma \vdash m_e$ *est satisfait si et seulement si* σ *est définie pour tous les événements utilisés dans* m_e*, et la condition booléenne représentée par* m_e *est satisfaite par les valeurs de* σ*.*

  D'où, $\sigma \vdash m_e \iff (\forall \alpha \in {m_e}^+ \ \sigma(\alpha) = \texttt{true}) \wedge (\forall \alpha \in {m_e}^- \ \sigma(\alpha) = \texttt{false})$

- $\sigma \not\vdash m_e$ *est satisfait si et seulement si* σ *est définie pour tous les événements utilisés dans* m_e*, et la condition booléenne représentée par* m_e *n'est pas satisfaite par les valeurs de* σ*.*

  D'où, $\sigma \not\vdash m_e \iff (\forall \alpha \in {m_e}^+ \cup {m_e}^- \ (\sigma(\alpha) = \texttt{true} \vee \sigma(\alpha) = \texttt{false}))$
  $\wedge(\exists \alpha \in {m_e}^+ \ \sigma(\alpha) = \texttt{false}) \vee (\exists \alpha \in {m_e}^- \ \sigma(\alpha) = \texttt{true})$

■

Comme pour un programme LustreΓune exécution d'un programme Argos Pa est caractérisée par le prédicat $\sigma_i^1 \ \sigma_i^2 \ \ldots \ \sigma_i^n \ \ldots \vdash Pa : \ \sigma_o^1 \ \sigma_o^2 \ \ldots \ \sigma_o^n \ \ldots$ où les σ_i et σ_o sont respectivement des mémoires booléennes sur les entrées et sorties du programme. Nous verrons plus loin queΓpar définitionΓce prédicat n'est satisfait que si le programme ne contient pas d'erreur de causalité. Lorsque pour une séquence infinie de mémoires d'entrées $\sigma_i^1 \ \sigma_i^2 \ \ldots \ \sigma_i^n \ \ldots$ une erreur de causalité est détectéeΓalors nous utilisons la notation $\sigma_i^1 \ \sigma_i^2 \ \ldots \ \sigma_i^n \ \ldots \not\vdash Pa$. Ce qui signifie qu'il n'est pas possible de définir la séquence infinie de mémoires de sorties correspondantes.

Le comportement d'un programme principal face à une histoire d'entrées particulière est défini cycle après cycle. Le prédicat $\sigma \ \vdash p \xrightarrow{\sigma'} p'$ définit la réaction d'un programme Pa ou d'une structure de programme Ps (un programme non principal) dans l'état p lors d'un cycle où les valeurs des entrées sont données par la mémoire σ. La mémoire σ' contient les valeurs des sorties du programme ; l'état p' est son nouvel étatΓutile pour calculer sa réaction au cycle suivant.

Nous définissons trois opérations supplémentaires sur les mémoires :

- **Elargissement** : $\sigma \xrightarrow[k]{} Idf2$Γ où $\sigma \in Mem(Idf1)$Γ désigne une mémoire $\sigma' \in Mem(Idf1 \cup Idf2)$ définie par :

  $\sigma'(idf) = \sigma(idf)$ si $idf \in Idf1$
  $\sigma'(idf) = k$ si $idf \in Idf2 \setminus Idf1$

- **Elimination** : $\sigma \setminus Idf2$Γoù $\sigma \in Mem(Idf1)$Γdésigne une mémoire $\sigma' \in Mem(Idf1 \setminus Idf2)$ définie par :

  $\sigma'(idf) = \sigma(idf)$ si $idf \in Idf1 \setminus Idf2$

- **Concaténation avec priorité à `true`** : $\sigma_1 \odot \sigma_2$Γoù $\sigma_1 \in Mem(Idf1)$Γ$\sigma_2 \in Mem(Idf2)$ désigne une mémoire $\sigma' \in Mem(Idf1 \cup Idf2)$ définie par :

  $\sigma'(idf) = \sigma_1(idf)$ si $(idf \in Idf1) \wedge (idf \notin Idf2)$
  $\sigma'(idf) = \sigma_2(idf)$ si $(idf \in Idf2) \wedge (idf \notin Idf1)$
  $\sigma'(idf) = \sigma_2(idf) \vee \sigma_1(idf)$ si $(idf \in Idf2) \wedge (idf \in Idf1)$

Le programme principal

$$\frac{\forall n \geq 1,\ \sigma_i^n \ \vdash p^n \ \xrightarrow{\sigma_o^n} p^{n+1} \qquad p^1 = init(Ps)}{\sigma_i^1\ \sigma_i^2\ \ldots\ \sigma_i^n\ \ldots \vdash (In, Out, Ps) : \sigma_o^1\ \sigma_o^2\ \ldots\ \sigma_o^n\ \ldots} \quad \text{[Pa]}$$

Le calcul de la séquence infinie de mémoires de sorties s'effectue cycle après cycle. Pour le premierΓl'état courant est constitué par l'état initial du programme (la fonction *init* est définie en 2.6).

La mise en parallèle

$$\frac{\sigma_i \ \vdash p1 \ \xrightarrow{\sigma_1} p1' \qquad \sigma_i \ \vdash p2 \ \xrightarrow{\sigma_2} p2'}{\sigma_i \ \vdash p1 \| p2 \ \xrightarrow{\sigma_1 \odot \sigma_2} p1' \| p2'} \quad \text{[P]}$$

Une sortie est émise par le programme si et seulement si l'une des deux composantes l'émet.

Le raffinement

$$\frac{\begin{array}{l} \exists (q_c, m_e/O, q_d) \in T \ telle\ que\ \sigma_i \vdash m_e \\ \qquad \sigma_i \ \vdash r_c \ \xrightarrow{\sigma_c} r_c' \\ \qquad \sigma_O = \ \sigma_\emptyset \xrightarrow[\mathtt{true}]{} O \\ \qquad \sigma_{Out} = \ \sigma_O \xrightarrow[\mathtt{false}]{} Out \end{array}}{\sigma_i \vdash \mathbf{R}_{(Q, q_c, q_{init}, T)}(r_1, \ldots, r_n) \ \xrightarrow{\sigma_{Out} \odot \sigma_c} \mathbf{R}_{(Q, q_d, q_{init}, T)}(r_1, \ldots, init(prog(r_c)), \ldots, r_n)} \quad \text{[R1]}$$

$$\frac{\forall (q_c, m_e/O, q_d) \in T\ \sigma_i \not\vdash m_e \quad \sigma_i \ \vdash r_c \ \xrightarrow{\sigma_c} r_c' \quad \sigma_{Out} = \ \sigma_c \xrightarrow[\mathtt{false}]{} Out}{\sigma_i \vdash \mathbf{R}_{(Q, q_c, q_{init}, T)}(r_1, \ldots, r_n) \ \xrightarrow{\sigma_{Out}} \mathbf{R}_{(Q, q_c, q_{init}, T)}(r_1, \ldots, r_c', \ldots, r_n)} \quad \text{[R2]}$$

Pour que ces règles soient complètement définiesΓil faut prévoir le cas où le sous-programme raffinant un état est NIL.

$$\frac{}{\sigma_i \vdash nil \ \xrightarrow{\sigma_\emptyset} nil} \quad \text{[Nil]}$$

La déclaration d'événements internes

$$\frac{\begin{array}{l} \forall \sigma_Y \in Mem(Y),\ \sigma_i.\sigma_Y \vdash p \ \xrightarrow{\sigma} p' \\ \exists ! \sigma_Y \in Mem(Y)\ telle\ que\ \sigma_i.\sigma_Y \vdash p \ \xrightarrow{\sigma} p' \\ \qquad \forall \alpha \in Y,\ \sigma_Y(\alpha) = \mathtt{true} \Rightarrow \sigma(\alpha) = \mathtt{true} \\ \qquad \quad \sigma_Y(\alpha) = \mathtt{false} \Rightarrow \sigma(\alpha) = \mathtt{false} \end{array}}{\vdash \overline{p^Y} \ \xrightarrow{\sigma \backslash Y} \overline{p'^Y}} \quad \text{[I]}$$

La détection des problèmes de causalité s'effectue de la même manière que dans la première définition du langage Argos : pour toutes les hypothèses possibles sur le statut des événements internesΓla réaction du programme est calculée ; la cohérence de chaque hypothèse est vérifiée en examinant la mémoire de sortie ; un problème de causalité est détecté s'il n'y a pas une unique hypothèse qui est cohérente.

4.2.3 Liens avec la définition formelle d'Argos par construction du modèle

La sémantique flots de données d'Argos définit le flot de sorties qui constitue la réponse du programme à un flot d'entrées donnée. Lors de la définition formelle d'Argos (cf. section 2.2.4)Γ

nous avons présenté la fonction sémantique $\mathcal{S}$ qui associe à tout programmeΓson modèle. Celui-ci contient toutes les réponses du programme aux différents flots d'entrées possibles. Il faut à présent vérifier que ces deux façons de définir la sémantique d'un programme Argos sont cohérentes : étant donnés un programme et un flot quelconque de ses entréesΓle flot des sorties qui constitue la réponse du programme au flot d'entrées ne doit pas dépendre de la méthode utilisée pour le définir. En particulierΓsi ce flot d'entrées est tel que l'on détecte une erreur de causalité dans le programmeΓalors cette erreur doit être détectée par les deux méthodes.

La cohérence des deux définitions formelles d'Argos est assurée par la propriété suivante :

Propriété 4.1
$\forall Pa \in \mathcal{P}_{princ},$

$$\sigma_i^1\ \sigma_i^2\ \ldots\ \sigma_i^n\ \ldots \vdash (In, Out, Ps)\ :\ \sigma_o^1\ \sigma_o^2\ \ldots\ \sigma_o^n\ \ldots\ -$$

$$\forall n \geq 1,\ p^n \xrightarrow{\mathcal{L}(\sigma_i^n, \sigma_o^n)} p^{n+1} \in T\ \ avec\ \mathcal{S}(Ps) = (Q, q_{init}, T)\ et\ p^1 = init(Ps)$$

$$\exists \sigma_i^1\ \sigma_i^2\ \ldots\ \sigma_i^n\ \ldots\ telle\ que\ \sigma_i^1\ \sigma_i^2\ \ldots\ \sigma_i^n\ \ldots \not\vdash (In, Out, Ps) -\ \mathcal{S}(Ps) = \text{ERROR}$$

■

Dans cette propriétéΓ$\mathcal{L}$ est une fonction qui transforme un couple de mémoires booléennes contenant une mémoire d'entrées et une mémoire de sortiesΓen une étiquette structurée — monôme d'événements et ensemble de sorties —.

L'idée de la démonstration de cette propriété est de prouver le résultat intermédiaire suivant :

$$\forall p \in Q, \sigma_i \vdash p \xrightarrow{\sigma_o} p' - p \xrightarrow{\mathcal{L}(\sigma_i, \sigma_o)} p' \in T$$

Cette propriété intermédiaire traduit la satisfaction du prédicat sur les états utilisé dans la sémantique flots de donnéesΓen terme d'existence d'une transition dans le modèle du programme. Cette mise en correspondance est quasi-syntaxique.

4.3 Le langage mixte ArgoLus

4.3.1 Les étapes de la définition d'ArgoLus

L'idée à l'origine du mélange est la suivante : un nœud Lustre dont les entrées et sorties sont purement booléennes peut être compilé sous forme d'un automate booléen qui peut être utilisé comme opérande d'une mise en parallèle ou d'un raffinement dans un programme Argos. Par exempleΓle programme de la figure 4.3 `(a)` dans lequel un nœud Lustre raffine l'état d'un automate Argos modélise un système réactif parfaitement défini. Il suffit pour connaître ce système réactifΓde remplacer le nœud par son modèle (cf. 4.3`(b)`). On obtient un programme Argos dont le modèle est donné par la figure 4.3 `(c)`.

Ce premier type de mélange qui consiste à intégrer des nœuds Lustre dans des programmes Argos nous a donné l'idée du mélange inverse : intégrer des programmes Argos à l'intérieur de nœuds Lustre. La symétrie n'est pas complète puisqu'un programme Argos ne se compile pas en un nœud Lustre. CependantΓl'automate obtenu par compilation du programme Argos se traduit simplement en un ensemble d'équations qui forment un nœud Lustre.

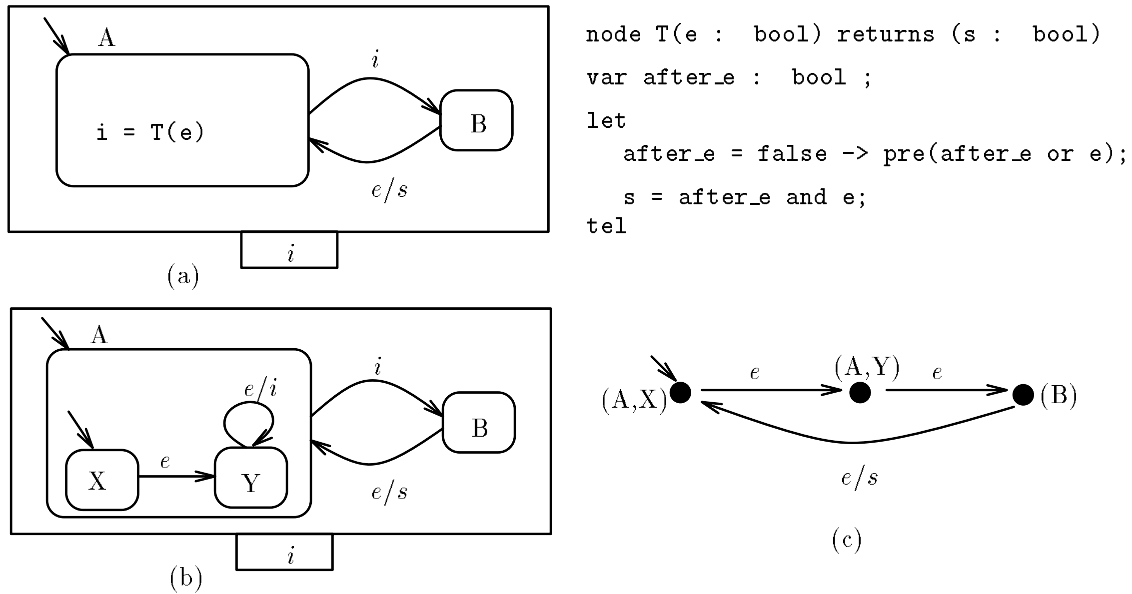


Figure 4.3: intégration de nœuds Lustre à l'intérieur de programmes Argos

Il s'ensuit que les programmes ArgoLus peuvent avoir deux formes :

- Soit ce sont des programmes Argos qui contiennent des nœuds Lustre ;
- Soit ce sont des programmes Lustre qui contiennent des composantes Argos ;

Pour formaliser ces idées intuitivesΓnous commençons par définir dans chaque langage une notion de bloc. Ce sont les unités syntaxiques de base qui pourront être interchangées : un bloc Lustre peut être utilisé à la place d'un bloc Argos et inversement. Ces blocs concrétisent la notion de sous-programme dans chaque langage. Ce sont :

- Des paquets d'équations en LustreΓaccompagnés d'un profil ;
- Des structures d'opérateurs en Argos (une certaine composition d'automates)Γaccompagnées d'un profil.

Ces blocs proviennent syntaxiquement :

- Des appels de nœuds en Lustre ;
- Des appels de procédures en Argos.

Nous commençons par présenter l'introduction de cette notion de bloc dans chaque langageΓ aussi bien au niveau syntaxique que sémantique. Nous présentons ensuite la définition formelle du langage ArgoLus.

4.3.2 Lustre avec structures de bloc

Nous définissons un bloc Lustre comme une liste d'équations à laquelle on attache un ensemble d'entréesΓun ensemble de sorties et un ensemble de variables locales. Un tel bloc peut remplacer une équation dans n'importe quel contexte. Au niveau de la syntaxe concrèteΓun bloc Lustre est défini par un appel de nœud.

Définition 4.7 Syntaxe abstraite des programmes Lustre avec structures de bloc

$$
\begin{array}{lll l}
Pl & ::= & (In, Out, Loc, Eqs) & \textit{Programme principal} \\
Eqs & ::= & Eq\ Eqs \mid Eq & \textit{Liste d'équations} \\
Eq & ::= & idf\ =\ Exp \mid Bl & \textit{Equation} \\
Bl & ::= & (In, Out, Loc, Eqs) & \textit{Bloc Lustre} \\
Exp & ::= & k \mid idf \mid Exp\ \texttt{->}\ Exp & \\
 & & \mid \texttt{pre}(Exp) \mid \texttt{Dataop}(Exp, \ldots, Exp) & \textit{Expression}
\end{array}
$$

■

Pour définir les contraintes de sémantique statique sur un programme Lustre avec structures de blocΓnous commençons par étendre les fonctions définies sur les listes d'équationsΓau cas où une équation est un bloc.

- `IdfD` (ensemble des identificateurs définis par une liste d'équations) et `IdfU` (ensemble des identificateurs utilisés par une liste d'équations) s'étendent de la manière suivante :

  $\texttt{IdfD}((In, Out, Loc, Eqs)) = Out$

  $\texttt{IdfU}((In, Out, Loc, Eqs)) = In$

  Les variables locales au nœud ne sont pas prises en compte par ces fonctions.

- `NoCycle` s'étend en prenant en compte les équations qui constituent le corps de chaque bloc qui apparaît dans le programme pour la définition de la relation $\mathcal{R}el$.

- `UDef` et `PreOK` s'étendent en prenant en compte les équations qui constituent le corps de chaque bloc qui apparaît dans le programme.

Pour ces trois dernières fonctionsΓtout se passe comme si le programme Lustre était totalement expansé.

Nous donnons les contraintes de sémantique statique en faisant référence à celles données pour Lustre dans la définition 4.2.

Définition 4.8 Sémantique statique de Lustre avec structures de bloc

- *La contrainte (i) doit être satisfaite par chaque bloc Lustre qui apparaît dans le programme.*

- *Les contraintes $(i), (ii), (iii), (iv)$ et (v) doivent être satisfaites par le programme principal.*

■

Notons que la contrainte (ii) qui impose aux ensembles d'entrées et de sorties d'être disjointsΓ ne s'applique pas à tous les blocsΓmais seulement au programme principal. Cela traduit seulement le fait que les listes de paramètres effectifs d'entrées et de sorties d'un appel de nœud ne sont pas nécessairement disjointes.

Pour définir la sémantique d'un programme Lustre avec structures de blocΓil suffit de rajouter à l'ensemble déjà définiΓune règle qui prend en considération le cas où une équation est un bloc. Un bloc se comportant comme les équations qui le composentΓcette règle est la suivante :

$$\frac{\sigma_i \vdash eqs \xrightarrow{\sigma_o} eqs'}{\sigma_i \vdash (In, Out, Loc, eqs) \xrightarrow{\sigma_o | Out} (In, Out, Loc, eqs')} \quad [\mathrm{Bl}]$$

4.3.3 Argos avec structures de bloc

Nous définissons un bloc Argos comme une structure d'opérateurs à laquelle on a attaché des ensembles d'entrées et de sorties. Contrairement aux blocs LustreΓl'ensemble des événements internes au bloc n'apparaît pas dans sa définition. En effetΓil est possible en Argos de retrouver cette informationΓgrâce à la présence d'un opérateur de déclaration d'événements internes dans la syntaxe abstraite.

Un bloc Argos peut remplacer un opérande d'opérateur dans n'importe quel contexte. Au niveau de la syntaxe concrèteΓun tel bloc est défini par un appel de procédure.

Définition 4.9 Syntaxe abstraite des programmes Argos avec structures de bloc

Pa	$::=$	(In, Out, Ps)		*Programme principal*
Ps	$::=$	$Ps \| Ps$		*Mise en parallèle*
	$\mid$	$\overline{Ps^{Y}}$		*Déclaration d'événements internes*
	$\mid$	$\mathbf{R}_A(R_1, \ldots, R_n)$	*avec* $A \in \mathcal{A}_d^{\emptyset,c}$	*Raffinement paramétré par A*
	$\mid$	Ba		*Bloc Argos*
R	$::=$	$\textsc{nil} \mid Ps$		
Ba	$::=$	(In, Out, Ps)		

■

Pour définir les contraintes de sémantique statique d'un programme Argos avec structures de blocΓnous commençons par étendre les fonctions $\mathcal{I}$ (ensemble des entrées d'un programme) et $\mathcal{O}$ (ensemble des sorties d'un programme) au cas où un opérande est un bloc Argos. Nous avons :

$\mathcal{I}((In, Out, Ps)) = In$

$\mathcal{O}((In, Out, Ps)) = Out$

Définition 4.10 La sémantique statique d'Argos avec structures de bloc

- *Tous les blocs Argos* (In, Out, Ps) *qui apparaissent dans le programme doivent être tels que* $In = \mathcal{I}(Ps)$ *et* $Out = \mathcal{O}(Ps)$.
- *Le programme principal doit en plus vérifier la contrainte* $In \cap Out = \emptyset$.

■

De la même manière qu'en LustreΓles paramètres effectifs d'entrées et de sorties d'un appel de procédure ne sont pas nécessairement disjoints.

La sémantique d'un programme Argos avec blocs est définie en ajoutant à l'ensemble des règles données précédemmentΓla règle suivante qui définit le comportement d'un bloc.

$$\frac{\sigma_i \vdash ps \xrightarrow{\sigma_o} ps'}{\sigma_i \vdash (In, Out, ps) \xrightarrow{\sigma_o | Out} (In, Out, ps')} \quad \text{[Ba]}$$

4.3.4 Syntaxe et sémantique statique du langage ArgoLus

Au niveau de la syntaxe abstraiteΓun programme ArgoLus a soit la forme d'un programme ArgosΓsoit la forme d'un programme Lustre. Une liste d'équations peut contenir un bloc Argos ou un bloc Lustre. De même un opérande d'un opérateur Argos peut être un bloc Argos ou un bloc Lustre.

Au niveau de la syntaxe concrèteΓcette possibilité d'interchanger des blocs se traduit par la possibilité d'appeler une procédure Argos à l'intérieur d'une liste d'équationsΓet inversementΓun nœud Lustre comme opérande d'un opérateur Argos.

Définition 4.11 Syntaxe abstraite des programmes ArgoLus

Pal	$::=$	$(In, Out, Ps) \mid (In, Out, Locs, Eqs)$	*Programme principal*
Ps	$::=$	$Ps \| Ps$	*Mise en parallèle*
	$\mid$	$\overline{Ps}^{Y}$	*Déclaration d'événements internes*
	$\mid$	$\mathbf{R}_A(R_1, \ldots, R_n)$ *avec* $A \in \mathcal{A}_d^{\emptyset,c}$	*Raffinement paramétré par A*
	$\mid$	$Ba \mid Bl$	*Bloc Argos ou Lustre*
R	$::=$	$\textsc{nil} \mid Ps$	
Ba	$::=$	(In, Out, Ps)	*Bloc Argos*
Eqs	$::=$	$Eq\ Eqs \mid Eq$	*Liste d'équations*
Eq	$::=$	$idf = Exp \mid Bl \mid Ba$	*Equation*
Bl	$::=$	(In, Out, Loc, Eqs)	*Bloc Lustre*
Exp	$::=$	$k \mid idf \mid Exp$ `->` Exp	
		$\mid$ `pre`$(Exp) \mid$ `Dataop`$(Exp, \ldots, Exp)$	*Expression*

■

Afin de définir la sémantique statique du langage ArgoLusΓnous commençons par étendre les fonctions qui servent à exprimer les contraintes statiques de chaque langage aux blocs de l'autre langageΓen commençant par celles du langage Lustre.

- `IdfD` et `IdfU` s'étendent aux blocs Argos de la manière suivante :

  `IdfD`$((In, Out, Ps)) = Out$

  `IdfU`$((In, Out, Ps)) = In$

- `NoCycle` s'étend en ne prenant pas en compte les blocs Argos présents dans la liste d'équations pour la définition de la relation $\mathcal{R}el$.

- `UDef` s'étend en considérant que toutes les sorties d'un bloc Argos sont définies une et une seule fois dans ce bloc.

La sémantique statique du langage Argos s'exprime à l'aide des fonctions $\mathcal{I}$ et $\mathcal{O}$. Ces deux fonctions sont étendues aux blocs Lustre. Nous avons :

$\mathcal{I}((In, Out, Loc, Eqs)) = In$

$\mathcal{O}((In, Out, Loc, Eqs)) = Out$

Les contraintes de sémantique statique qui s'appliquent à un programme ArgoLus dépendent de la structure de celui-ci. S'il a la forme d'un programme Lustre alors il doit satisfaire celles d'un programme Lustre avec structures de bloc. Les extensions apportées aux fonctions qui apparaissent dans ces contraintes doivent être prises en compte. InversementΓsi le programme ArgoLus a la forme d'un programme ArgosΓalors il doit satisfaire les contraintes de sémantique statique d'un programme Argos avec structures de bloc. Il faut ajouter à ces premiers ensembles de contraintes héritées de chaque langageΓdes contraintes spécifiques au langage ArgoLus. Elles s'expriment par les deux énoncés suivants :

- Tout bloc Argos à l'intérieur d'une liste d'équations doit être ferméΓc'est-à-dire que ses ensembles d'entrées et de sorties doivent être disjoints.

- Tout bloc Lustre à l'intérieur d'une structure d'opérateurs Argos doit être ferméΓc'est-à-dire que ses ensembles d'entrées et de sorties doivent être disjoints.

Ces deux contraintes ne sont pas indispensables à la définition d'un langage mixte fondé sur Argos et Lustre. NéanmoinsΓelles permettent de simplifier considérablement la définition d'ArgoLus et ne paraissent pas trop limiter ses possibilités d'utilisation. Seule la première de ces contraintes est importante ; par symétrie nous imposons aussi la vérification de la deuxième. Cette première contrainte assure qu'un événement qui exprime des communications entre opérandes d'un opérateur ArgosΓest déclaré interne par l'opérateur propre au langage ArgosΓet non par une déclaration de variables locales héritée du langage Lustre. Le programme donné par la figure 4.4 est un exemple de programme que nous ne voulons pas considérer. En effetΓbien que i serve à faire communiquer deux automates ArgosΓil n'est pas déclaré interne par un opérateur Argos. Nous verronsΓlors de la présentation de la sémantique du langage ArgoLusΓles problèmes que pose l'existence de programmes de ce type.

```
node incorrect ( e1 :  bool) (s :  bool);
var i :  bool
let
    (i,s) = T(e1,i);
tel
```

```
procédure T(e1,e2)(s1,s2)
```

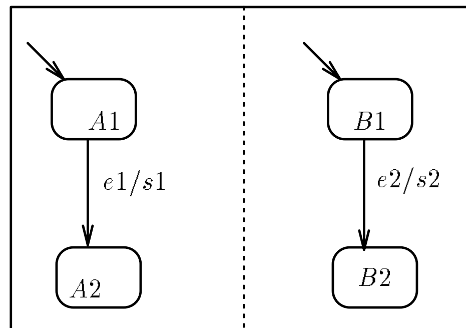


Figure 4.4: exemple de programme ArgoLus incorrect

4.3.5 Sémantique flots de données du langage ArgoLus

Nous définissons la sémantique flots de données en effectuant l'union des règles qui définissent la sémantique des deux langages avec structures de bloc. Ces règles n'ont pas besoin d'être modifiées. Nous présentons cependant par la suite les différents points qui auraient pu être source de modificationsΓet les raisons pour lesquelles ils n'en engendrent pas.

- **Le retardement de l'évaluation des équations en Lustre.**

  Une convention prise lors de la définition de la sémantique flots de données du langage Argos est que la valeur d'un événement est toujours définie lorsque celui-ci est utilisé. Cette propriété est assurée par constructionΓgrâce en particulier à la façon dont sont traités les événements internes. Ce n'est pas le cas en LustreΓoù une équation peut dépendre d'une variable qui n'est pas encore définie dans la mémoire couranteΓd'où la nécessité de prévoir le retardement de l'évaluation d'une équation dans la sémantique flots de données de Lustre.

  En ArgoLusΓ une liste d'équations peut contenir un bloc Argos. Ce bloc peut utiliser une variable définie par une équation qui le suit dans la liste. Il faut donc prévoir le retardement de l'évaluation d'un bloc Argos dans la sémantique d'ArgoLus. Il se trouve que ce retardement est déjà prévu dans la sémantique d'ArgosΓgrâce à la définition des prédicats $\sigma \vdash m_e$ et $\sigma \not\vdash m_e$ (cf. définition 4.6). En effetΓces deux prédicats ne sont pas satisfaits si un des événements utilisés dans le monôme n'est pas défini dans la mémoire courante.

- **L'enrichissement de la mémoire courante lors de l'évaluation d'une liste d'équations en Lustre**

  En LustreΓlorsqu'une liste d'équations est évaluée (cf. [Eqs1] et [Eqs2])Γla mémoire courante est enrichie après chaque évaluation d'une équation. C'est de cette manière que les valeurs des variables locales apparaissent dans la mémoire courante. En ArgosΓcette notion d'enrichissement de la mémoire courante n'existe pas. Les événements internes sont traités d'une toute autre manière. On suppose une certaine valeur aux événements internesΓ

on calcule la réaction du programme sous cette hypothèseΓpuis on vérifie a posteriori la cohérence de l'hypothèse.

La contrainte de sémantique statique du langage ArgoLus qui impose que tout événement qui établit une communication entre deux automates Argos est déclaré interne par l'opérateur ArgosΓnous permet de conserver cette distinction entre les deux langages. En effetΓsi le programme de la figure 4.4 pouvait être un programme ArgoLusΓle traitement de la variable iΓqui fait communiquer les deux automates ArgosΓnécessite la notion d'enrichissement de la mémoire courante lors de l'évaluation des composantes d'une mise en parallèle. L'introduction de cette notion soulève de nombreuses difficultés liées à l'absence en Argos de contrainte statique imposant la définition unique d'une variable.

- **La réinitialisation d'une composante en Argos.**

  La sémantique de l'opérateur de raffinement du langage Argos fait intervenir une fonction de réinitialisation des sous-programmes opérandes de l'opérateur. D'après la syntaxe du langage ArgoLusΓune telle composante peut être constituée par un bloc Lustre. Il faut donc définir la réinitialisation d'un bloc Lustre. En ArgosΓcette réinitialisation se fait en deux étapes. La première étape consiste à calculer le programme associé à l'état courant de la composante à réinitialiser. C'est le rôle de la fonction *prog*Γelle élimine dans la forme syntaxique de l'état toutes les informations sur l'état courant des automates. La seconde étape calcule l'état initial de ce programme.

  Nous allons voir si ce principe peut être appliqué au langage Lustre. La deuxième étape a déjà été définie : l'état initial d'un programme Lustre booléen s'obtient en ajoutant un paramètre à toutes les occurrences de l'opérateur `pre` du programmeΓqui a pour valeur `true`. Il reste à savoir s'il est possible de retrouver un programme Lustre à partir d'un état quelconque de l'automate qui lui est associé.

  La réponse à cette question est négative. La raison de cette impossibilité est liée à la règle de sémantique de l'opérateur d'initialisation $exp1$ `->` $exp2$. En effetΓd'après la règle [E4]Γl'expression $exp1$ n'est pas conservée dans l'état but car elle n'est d'aucune utilité pour la suite de l'évaluation du programme.

  Ce problème peut se régler par une transformation syntaxique des équations Lustre présentes dans le programme ArgoLus. Dans chaque équationΓune expression de la forme exp`->`exp' est remplacée par `if true -> pre(false) then` exp `else` exp'

  Lors de l'évaluation de l'opérateur `->` la constante `true` va disparaître. Ce n'est pas graveΓcar la fonction de réinitialisation associe le paramètre `true` à l'opérateur `pre` restant. Par conséquentΓc'est bien exp et non exp'Γqui est utilisée pour le calcul de la valeur initiale de l'expression globale.

4.4 Un exemple de programme en ArgoLus

Cet exemple illustre l'utilisation du langage mixte ArgoLus et nous permet de mettre en valeur (cf. section 4.4.4) les apports de ce langage par rapport aux deux langages qui le constituent.

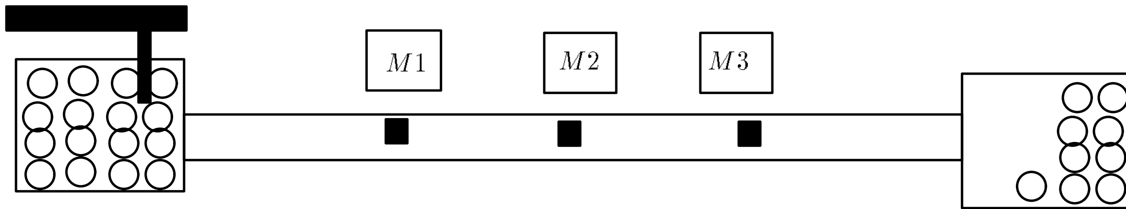


Figure 4.5: ligne de montage

4.4.1 Présentation du système à décrire

Le système décrit est une ligne de montage où s'effectuent divers traitements sur des bouteilles. Cette ligne de montage est composée des éléments suivants (cf. figure 4.5) :

- Un tapis roulant qui relie deux réserves de bouteilles ;
- Un bras manipulateur qui permet de prendre une bouteille dans la première réserve et de la poser sur le tapis ;
- Trois machines disposées le long du tapis.

Au cours du passage sur le tapis roulant, la bouteille doit subir trois types de traitement T1, T2 et T3 réalisés respectivement par les machines M1, M2 et M3. Un capteur de position en face de chaque machine détermine l'instant où une bouteille est en position pour subir le traitement.

La seule contrainte d'ordre qui doit être respectée entre les différents traitements est la suivante : le traitement T1 précède toujours les traitements T2 et T3.

Les machines M2 et M3 sont partagées entre plusieurs lignes de montage identiques. Par conséquent, elles ne sont pas toujours disponibles au moment où une bouteille est en position pour subir un traitement. Par contre, il est possible de réquisitionner une machine lorsqu'elle est disponible, afin d'assurer qu'elle reste disponible pour la ligne de montage tant qu'elle n'a pas effectué son traitement.

Le tapis roulant peut être stoppé, c'est le cas lorsqu'une machine effectue un traitement. Son sens de déroulement peut aussi être inversé, c'est le cas par exemple si le traitement T3 a été effectué avant T2.

Nous souhaitons décrire le contrôleur qui gère le lancement de chaque traitement, la réquisition des machines M2 et M3, le comportement du tapis roulant et enfin le placement d'une bouteille sur le tapis.

Pour gérer les problèmes de disponibilité des machines, nous demandons à ce contrôleur d'appliquer la stratégie suivante :

- Dès que le traitement T1 est terminé, les machines M2 et M3 sont réquisitionnées dès qu'elles sont disponibles pour la ligne. Une machine n'est réquisitionnée qu'une seule fois pour la même bouteille ;

- Maintien de la bouteille entre les positions 2 et 3Γtant que les deux traitements T2 et T3 n'ont pas été effectués. Libre choix est donné au contrôleur de bloquer la bouteille entre une de ces positions ou de la faire osciller entre celles-ci.

Lorsque les deux traitements T2 et T3 ont été effectuésΓla bouteille doit être amenée dans la seconde réserveΓet une autre bouteille doit être posée sur le tapis.

Un interrupteur général permet de démarrer la ligne de montage et de la stopper. Lorsque la ligne est démarréeΓle tapis est mis en marche de la première réserve vers la seconde. Lorsque la ligne est stoppéeΓle tapis roulant doit être arrêté. Le superviseur de la ligne doit ôter du tapis la bouteille en cours de traitementΓafin de replacer la ligne de montage dans sa position initiale.

4.4.2 Interface avec l'environnement

Les entrées du système sont :

- $onoff$: commutation de l'interrupteur général ;
- *pos_i* : présence d'une bouteille en position iΓpour $i \in \{1, 2, 3\}$;
- $finTi$: terminaison du traitement TiΓpour $i \in \{1, 2, 3\}$;
- $dispMi$: disponibilité de la machine MiΓpour $i \in \{2, 3\}$.

Ses sorties sont :

- $debTi$: démarrage du traitement TiΓpour $i \in \{1, 2, 3\}$;
- $reqMi$: réquisition de la machine MiΓpour $i \in \{2, 3\}$;
- *stop* : arrêt du tapis roulant ;
- *start* : démarrage du tapis roulant ;
- *inv* : inversion du sens de roulement du tapis roulant ;
- *poser* : placement d'une bouteille sur le tapis roulant.

4.4.3 Programmation en ArgoLus

Ce système peut être décrit en ArgoLus à l'aide d'une première opération de raffinement pour exprimer le statut de la ligne de montage : sous-tension ou hors-tension. La figure 4.6 donne l'automate de contrôle paramètre de cette opération de raffinement. Lorsque la ligne est lancéeΓ le tapis est démarré et une bouteille doit être posée dessus.

Le comportement de la machine sous-tension peut se décrire en utilisant une nouvelle fois l'opérateur de raffinementΓpour exprimer les différentes phases du traitement de la bouteille : avant T1Γpendant T1 et après T1. La figure 4.7 donne l'automate de contrôle de ce

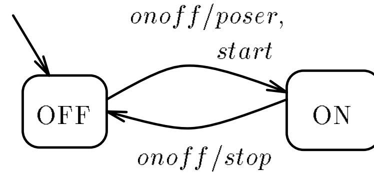


Figure 4.6: automate de contrôle du premier raffinement

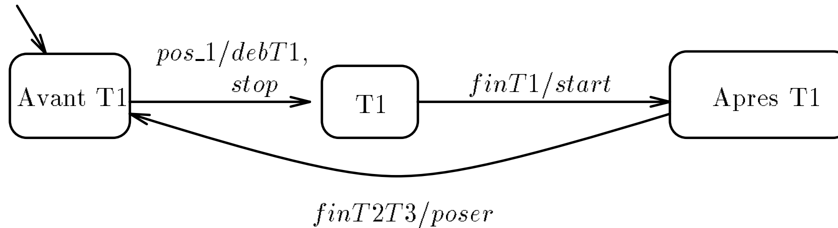


Figure 4.7: automate de contrôle du second raffinement

deuxième raffinement. La présence de l'événement interne $finT2T3$ indique la fin des traitements T2 et T3. On suppose que lorsque cet événement est émisΓle tapis est en marcheΓet va de la première réserve vers la seconde. L'état `Apres T1` est un état raffiné par une composante qui gère la réquisition et le lancement des machines M2 et M3Γainsi que l'évolution du tapis.

Cette composante peut être décrite en Argos ou en Lustre. Le style déclaratif du langage Lustre nous semble plus adapté à cette descriptionΓc'est pourquoi nous présentons une solution dans laquelle l'état `Apres T1` est raffiné par un nœud Lustre. Les ensembles d'entrées et de sorties de ce nœud sont respectivement :

- $\{pos_2, pos_3, finT2, finT3, dispM2, dispM3\}$Γ
- $\{debT2, debT3, reqM2, reqM3, stop, start, inv, finT2T3\}$.

Nous appelons ce nœud `T2T3` puisqu'il assure le bon déroulement des traitements T2 et T3.

La figure 4.8 reprend l'architecture générale du programme ArgoLus qui décrit le contrôleur de la ligne de montage. Il nous reste maintenant à décrire le nœud Lustre `T2T3`.

Le nœud Lustre `T2T3` utilise plusieurs nœuds intermédiairesΓnous donnons leur spécification avant de donner le corps du nœud principal.

- `ControleM` : contrôle une machine M d'après les entrées $pos, dispM$ et $finT$ qui l'informe respectivement de la bonne position de la bouteille sur le tapis pour la machineΓde sa disponibilité et de la fin du traitement. En fonction de ces entréesΓles valeurs des sorties $reqM$ (réquisition de la machine) et $debT$ (lancement du traitement) sont mises à jour.

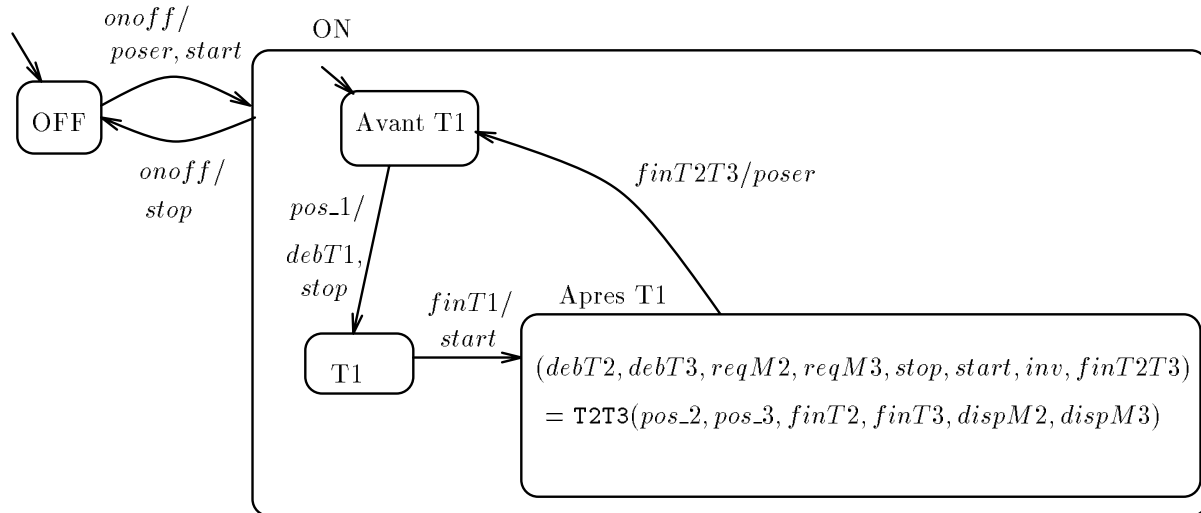


Figure 4.8: architecture générale du programme ArgoLus

- `Tapis` : contrôle le comportement du tapis d'après les informations suivantes : position de la bouteilleΓétat des traitements T2 et T3. En fonction de ces entréesΓles valeurs des sorties *inv*, *stop* et *start* (commandes du tapis) sont mises à jour.

- `Apres2` : détermine si deux signaux ont déjà été présents depuis l'instant initial jusqu'à l'instant courant (instant courant compris).

- `Apres` : détermine si un signal a déjà été présent depuis l'instant initial jusqu'à l'instant courant (instant courant compris).

Nous donnons la réalisation en Lustre des différents nœuds en commençant par le nœud principal.

```
node T2T3 (pos_2, pos_3, finT2, finT3, dispM2,dispM3:bool)
returns (debT2,debT3,reqM2,reqM3,stop,start,inv,finT2T3:bool) ;
let
     (reqM2,debT2) = ControleM(pos_2,dispM2,finT2);
     (reqM3,debT3) = ControleM(pos_3,dispM3,finT3);
     (inv,start,stop) = Tapis(pos_2,pos_3,debT2,debT3,finT2,finT3,finT2T3);
     finT2T3 = Apres2(finT2,finT3);
tel
```

Le contrôle des deux machines est assuré par deux appels au nœud `ControleM`. L'appel au nœud `Tapis` gère le comportement du tapis. La fin des deux traitements est détectée lorsqu'une occurrence du signal de fin de chaque traitement a été détectée.

Nous décrivons à présent chaque nœud utilisé dans `T2T3`.

```
node ControleM(pos,disp,fin:bool) returns (req,deb:bool);
var M_a_ete_req, M_a_fini :  bool;
let
     M_a_ete_req = Apres(req);
     T_en_cours = Apres(deb);
     req = disp and not pos
           and true -> (not pre(T_en_cours) and not pre(M_a_ete_req)) ;
     deb = pos and disp and true -> not pre(T_en_cours);
tel
```

Le traitement de la machine M peut être lancé (sortie `deb`) si les trois conditions suivantes sont réunies :

- Le traitement n'a pas déjà été effectué ;
- La machine est disponible ;
- La bouteille est en position adéquate.

La machine est réquisitionnée si elle ne l'a pas déjà étéΓet seules les deux premières conditions sont réunies.

```
node Tapis(pos_2,pos_3,debT2,debT3,finT2,finT3,finT2T3:bool)
     returns (inv,start,stop:bool);
var sens_1_vers_2 , T3_en_cours :  bool;
let
     sens_1_vers_2 = true -> if inv then not pre(sens_1_vers_2);
                           else pre(sens_1_vers_2);
                           else if start then true;
                                else pre(en_marche);
     stop = debT2 or debT3;
     start = finT2 or finT3;
     inv = pos_2 and not (true -> pre(sens_1_vers_2)) or pos_3 and
           not T3_en_cours and (true -> pre(sens_1_vers_2)) and not finT2T3;
     T3_en_cours = false -> (pre(T3_en_cours) and not finT3)
                  or (not pre(T3_en_cours) and debT3);
tel
```

Le tapis est arrêté dès qu'un traitement est lancé. Il est remis en marche dès qu'un traitement se termine. Nous supposons que l'ordre d'inversion de la direction du tapis est mémorisé lorsque le tapis est arrêté. La solution choisie fait osciller la bouteille entre les positions 2 et 3 tant qu'un des deux traitements reste à effectuer.

```
node Apres2(e1,e2:bool) returns (after2:bool);
var after_e1,after_e2 :  bool;
let
     after_e1 = Apres(e1);
     after_e2 = Apres(e2);
     after2 = after_e1 and after_e2;
tel
```

```
node Apres(e:bool) returns (after:bool);
let
     after = e or (ff -> pre(after))
tel
```

4.4.4 Apports d'ArgoLus par rapport aux langages Lustre et Argos

L'apport du langage mixte ArgoLus le plus évident est l'introduction d'une structure de contrôle hiérarchique à l'intérieur des programmes Lustre. Ce type de structuration est nécessaire pour décrire bon nombre de systèmes réactifsΓ dont celui que nous venons de présenter. Cela ne signifie pas qu'il n'est pas possible de décrire de tels systèmes en LustreΓmais seulement que leur description n'est pas aisée. Par exempleΓla description en Lustre de la ligne de montage utilise de nombreuses variables booléennes servant à coder les différents modes. Ces variables doivent être passées en paramètres de tous les nœuds et doivent être prises en compte pour la définition des sorties. Nous verrons dans la section 4.6 que le besoin en structures de contrôle hiérarchiques dans un langage déclaratifΓcorrespond à un besoin reconnu dans la communauté des langages synchronesΓpuisque un mécanisme de séquencement de tâches a été introduit dans le langage Signal [RG94].

Néanmoins l'apport du langage ArgoLus ne se limite pas à cet aspect puisqu'il permet :

- De résoudre plus facilement certains types de communication en utilisant des variables d'états plutôt que la diffusion d'événements internes. Si par exempleΓon essaie de programmer en Argos le nœud `T2T3`Γalors la description du comportement du tapis fait intervenir des communications complexes entre la composante qui détecte la fin des deux traitements et celle qui détecte si le traitement T3 est ou non en cours ;
- D'utiliser pour décrire une composante de base du programme (non structurée) soit le style déclaratif de LustreΓsoit la description en automate. Reprenons l'exemple précédentΓle nœud `after2` qui détecte la fin des deux traitements s'exprime de façon moins élégante par un automate ArgosΓcar on est obligé de distinguer quatre états qui mémorisent l'état de chaque traitement. A l'opposéΓle nœud Lustre `ControleM` s'exprime lui aussi facilement par un automate Argos (cf. figure 4.9). Par conséquentΓlibre choix est laissé au programmeur de choisir le style de description qui lui convient le mieux.

4.5 La mise en œuvre du langage ArgoLus

La solution que nous présentons consiste à traduire tout programme ArgoLus en un programme Lustre équivalentΓen ce sens que les deux programmes réagissent de la même façon à toutes les histoires d'entrées. Cette traduction permet d'utiliser le compilateur Lustre comme compilateur ArgoLus. De plusΓelle rend possible l'utilisation d'outils spécifiques à l'environnement Lustre.

La solution symétrique : traduction de tout programme ArgoLus en un programme Argos équivalentΓest aussi possible. Elle est présentée en détail dans [JLMR94a] et brièvement dans la section 4.5.3.

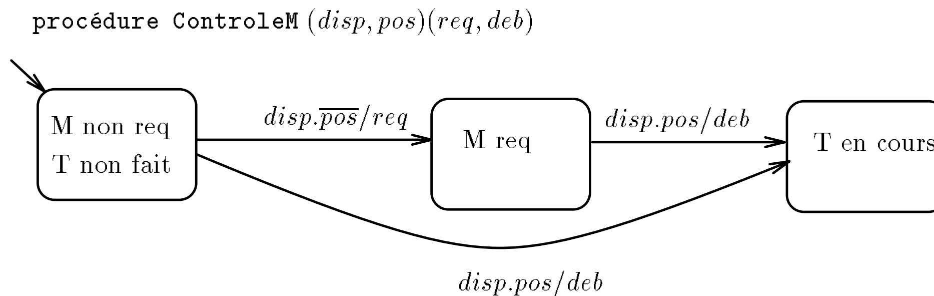


Figure 4.9: description Argos équivalente au nœud `ControleM`

La traduction des programmes ArgoLus en Lustre est basée sur :

- La traduction structurelle d'un programme Argos en un programme Lustre : les automatesΓ composantes de base des programmes ArgosΓsont traduits en nœuds Lustre ; les opérateurs Argos sont vus comme des opérateurs sur des nœuds Lustre ;
- La transformation d'un nœud Lustre en un nœud "réinitialisable". L'opérateur de raffinement du langage Argos introduit les notions d'activité et de réinitialisation d'un nœud. Ces nouvelles notionsΓpour être prises en compte de manière structurelleΓnécessitent la modification des nœuds Lustre qui apparaissent dans le programme ArgoLus.

Nous commençons par présenter la traduction d'un programme Argos en un programme Lustre. Les détails de cette traduction sont présentés dans l'annexe A. Nous ne donnons ici que les idées intuitivesΓillustrées par des exemplesΓqui permettent de comprendre les principes de la traduction. Cette présentation s'effectue en deux temps : traduction des programmes Argos sans raffinementΓpuis prise en compte de l'opérateur de raffinement. Elle se termine par l'étude du comportement de la traduction vis-à-vis du critère syntaxique de détection des problèmes de causalité en Lustre. Nous présentons ensuite la traduction d'un programme ArgoLus en un programme LustreΓpuis brièvement la traduction d'ArgoLus en Argos.

4.5.1 La traduction structurelle d'Argos en Lustre

La traduction est structurelle au sens où le programme Lustre associé à un programme Argos est obtenu en tenant compte de l'opérateur principal du programme Argos et des programmes Lustre associés aux sous-programmes opérandes de cet opérateur.

Lorsque le programme Argos est un programme principalΓle programme Lustre associé est tel que leurs deux modèles sont équivalents : ils représentent le même système réactif. Cette équivalence peut être formalisée par l'égalité des ensembles de "traces" des automates modèles. Une trace d'un automate est une séquence d'exécution dans laquelle on ne conserve que les étiquettes.

Pour faciliter la présentation de la traductionΓnous autorisons les appels de nœuds dans les programmes Lustre produits (forme avant expansion).

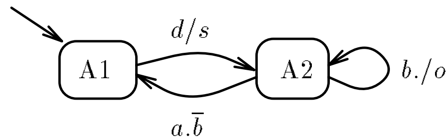


Figure 4.10: traduction d'un automate en Lustre

La traduction structurelle d'Argos sans raffinement en Lustre

Un programme Argos sans raffinement est soit un automateΓsoit obtenu par une mise en parallèle de deux sous-programmesΓsoit obtenu en déclarant un ensemble d'événements internes à un sous-programme.

Nous commençons par définir l'application de la fonction de traduction à un automateΓpuis nous définissons le nœud Lustre équivalent à un programme Argos composéΓà partir des nœuds associés aux opérandes de l'opérateur de composition.

Traduction d'un automate

L'idée à la base de cette traduction consiste à associer une variable booléenne à chaque état. Cette variable est vraie à un instant i si et seulement si dans le programme Argos l'état est actif à cet instant. Pour un état q nous notons `atq` cette variable. Elle est définie par l'intermédiaire de deux autres variables :

- `entersq` est vraie à l'instant i si et seulement si dans cet instant une transition qui atteint l'état q est déclenchée ;
- `exitsq` est vraie à l'instant i si et seulement si dans cet instant une transition qui sort de l'état q est déclenchée (même si c'est une boucle).

Les deux variables `entersq` et `exitsq` peuvent être simultanément vraies. Cela signifie qu'à l'instant i une boucle sur l'état q est déclenchée.

La relation suivante est alors vérifiée pour tout instant différent de l'instant initial :

```
atq = pre(entersq or (atq and not exitsq))
```

En effetΓon se trouve à un instant i dans l'état q si et seulement si à l'instant précédentΓsoit une transition qui entre dans q a été déclenchéeΓsoit q était déjà actif et aucune transition qui sort de qΓsans être une boucle sur qΓn'a été déclenchée.

La valeur de `atq` au premier instant d'activation dépend du statut de l'état q dans l'automate. S'il est l'état initial de l'automateΓalors la variable `atq` est vraie.

Le triplet de variables défini pour chaque étatΓconstitue l'ensemble des variables locales au nœud Lustre associé à un automate. Ces variables locales permettent de définir simplement les équations des sorties de l'automateΓpuisque comme le montre l'exemple qui suitΓil ne reste plus qu'à coder les conditions booléennes dénotées par les monômes d'événements. Le programme Argos formé d'un seul automate donné par la figure 4.10 est traduit par le programme Lustre suivant :

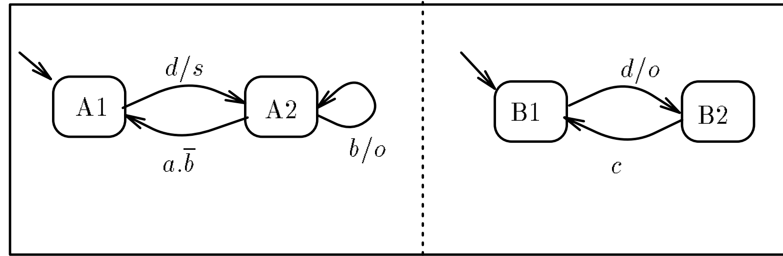


Figure 4.11: exemple illustrant le codage en Lustre de la mise en parallèle

```
node A(a,b,d :  bool) returns (s, o :  bool);
var atA1, atA2, entersA1, exitsA1, entersA2, exitsA2 :  bool;
let
     atA1 = true -> pre(entersA1 or (atA1 and not exitsA1));
     atA2 = false -> pre(entersA2 or (atA2 and not exitsA2));
     entersA1 = atA2 and a and not b;
     exitsA1 = atA1 and d;
     entersA2 = (atA1 and d) or (atA2 and b);
     exitsA2 = (atA2 and a and not b) or (atA2 and b);
     s = atA1 and a;
     o = atA2 and b;
tel;
```

La mise en parallèle

La seule difficulté pour traiter la mise en parallèle est due à la sémantique statique de LustreΓ qui impose la définition unique des sorties. Deux composantes d'un programme Argos pouvant avoir des sorties communesΓla mise en parallèle ne se traduit pas par de simples appels aux nœuds associés à ses opérandes. L'idée pour résoudre ce problème est la suivante : si les deux opérandes ont la sortie o en communΓalors cette sortie est renommée en $o1$ dans un des appelsΓ en $o2$ dans l'autreΓet o est la disjonction de ces deux variables locales.

Considérons le programme Argos donné par la figure 4.11. Le nœud associé à l'automate `A` est celui décrit précédemment. Nous supposons que celui associé à l'automate `B` se nomme `B`. Les deux composantes ont la sortie `o` en commun. Le nœud Lustre associé à ce programme Argos est le suivant :

```
node par(a,b,c,d :  bool) returns (s, o :  bool);
var o1, o2 :  bool ;
let
     (s, o1) = A(a, b,d);
     o2 = B(c, d);
     o = o1 or o2;
tel;
```

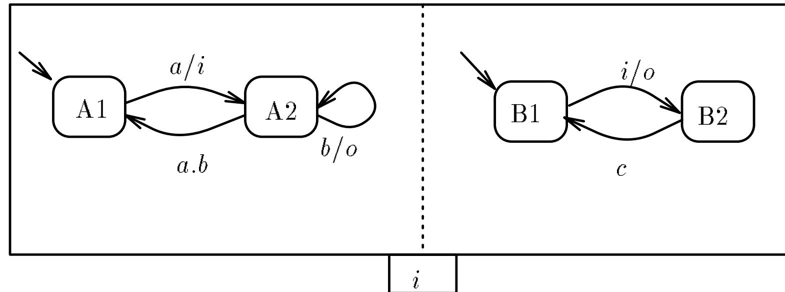


Figure 4.12: exemple illustrant le codage en Lustre de la déclaration d'événements internes

La déclaration d'événements internes

L'opérateur de déclaration d'événements internes permet en Argos de relier les occurrences d'entrée et de sortie d'un même événementΓet d'éliminer du modèle les informations portant sur cet événement. La traduction en Lustre doit exprimer ces deux effets. Pour le premierΓil se trouve qu'en Lustre une sortie et une entrée qui ont le même nom ne forment qu'une seule et unique variableΓleurs occurrences sont donc automatiquement reliées. L'élimination des informations portant sur les événements internes dans le modèleΓs'effectue en déclarant les variables associées à ces événementsΓlocales au nœud.

Considérons le programme Argos de la figure 4.12Γsi `par` est le nom du nœud Lustre associé à la mise en parallèleΓalors le nœud Lustre associé à ce programme est le suivant :

```
node int(a,b,c :  bool) returns (o :  bool);
var i:  bool;
let
    (i, o) = par(a,b,c,i);
tel;
```

Nous pouvons maintenant généraliser ces principes de traduction au langage Argos complet.

Généralisation à tout le langage Argos

Dès qu'un programme Argos est construit à l'aide d'un opérateur de raffinementΓl'activité d'un état ne se décide plus localementΓen examinant l'ensemble des informations propres à l'automate auquel il appartient. En effetΓoutre les considérations locales du type : une transition qui entre dans cet état a été déclenchée à l'instant précédentΓil faut prendre en considération des informations globales du type : l'automate auquel l'état appartient a été interrompu à l'instant précédent. Dans un tel casΓquelle que soit l'évolution locale de l'automate à l'instant précédentΓ tous ses états sont inactifs à l'instant courant.

Prenons pour exemple le programme de la figure 4.13. Supposons qu'à l'instant iΓles états actifs du programme soient `C1` et `A1`. A cet instantΓl'automate `A` est actif alors que `B` ne l'est pas. Si l'événement b est présentΓalors l'automate `A` est interrompu et `B` est lancé. Par conséquent à l'instant suivant `C1` et tous les états de `A` seront inactifsΓet `C2` et `B1` seront actifs.

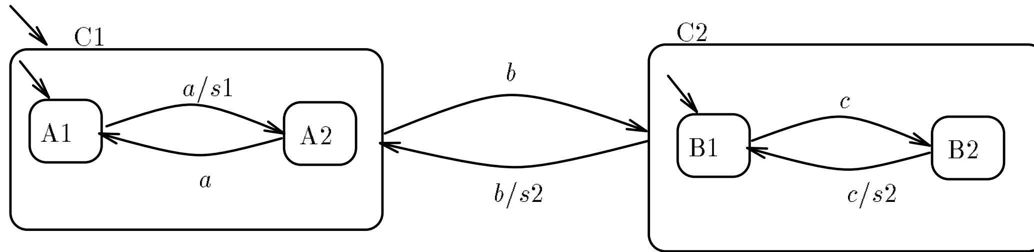


Figure 4.13: introduction du raffinement dans la traduction d'Argos vers Lustre

Pour conserver une fonction de traduction structurelle d'Argos en Lustre — un nœud Lustre est associé à chaque nœud de l'arbre abstrait du programme Argos — nous ajoutons à la liste des paramètres formels d'entrées des nœuds construitsΓtrois paramètres qui servent à mémoriser l'information globale nécessaire au codage de l'activité des états d'un automate. Ces informations sont synthétisées lors du codage d'un opérateur de raffinement. De plusΓlors de l'appel du nœud associé au programme principalΓils sont instanciés avec des valeurs adéquates. Nous donnons leur signification par rapport à leur utilisation lors du codage des états d'un automate.

- Le paramètre `initial` est un flot booléen constant qui a la valeur vraie si et seulement si l'automate est actif dans l'état initial du programme. Autrement ditΓl'automate est à l'intérieur d'états qui sont tous les états initiaux de leur automate. Par exempleΓsur le programme Argos précédentΓce paramètre doit être vrai lors du codage de l'automate `A` et faux dans le cas de `B`. Seul le paramètre `initial` est un flot booléen constant.
- Le paramètre `reset` est vrai à un instant donné dans deux situations :
  - L'automate est inactif et est lancé dans l'instant courant ;
  - L'automate est réinitialisé dans l'instant courant (une boucle est déclenchée sur un état qui le contient).
- Le paramètre `kill` est vrai à un instant donné dans deux situations :
  - L'automate est actif et est interrompu dans l'instant courant ;
  - L'automate est réinitialisé dans l'instant courant (une boucle est déclenchée sur un état qui le contient).

Les paramètres `reset` et `kill` peuvent être simultanément vrais à un instant donné. Cela signifie que l'automate est réinitialisé dans l'instant courant.

Par rapport à la traduction des programmes Argos sans raffinementΓnous identifions la notion de programme principal pour pouvoir instancier une première fois le triplet de paramètres. Pour les opérateurs de mise en parallèle ou de déclaration d'événements internesΓla seule modification engendrée par l'éventuelle présence d'opérateurs de raffinementΓest liée aux triplets de paramètres. Dans les deux casΓles nœuds construits ne font que transmettre les valeurs reçues pour ces paramètresΓaux nœuds qu'ils appellent.

Programme principal

Le nœud Lustre associé à un programme Argos principal est construit en deux temps. Tout d'abordΓle nœud Lustre associé au programme est construit sans tenir compte du fait qu'il est principal. Ce nœud a dans sa liste d'entrées le triplet de paramètres {`initial`,`reset`,`kill`}. Il est ensuite appelé en instanciant son triplet de paramètres avec les valeurs {`true`,`false`,`false`}. Ces valeurs initiales s'expliquent par la remarque suivante : tous les automates qui ne sont sous aucun raffinement sont toujours actifs. Par conséquentΓils ne peuvent être ni lancésΓni interrompusΓni réinitialisés. Dès qu'un opérateur de raffinement est rencontréΓles valeurs de ces paramètres sont modifiées.

On peut vérifier que grâce à ces valeursΓle codage d'un programme Argos sans raffinement est équivalent à celui présenté précédemment.

Le raffinement

Le codage en Lustre d'un opérateur de raffinement s'effectue en trois parties :

- Codage des états et des sorties de l'automate de contrôle du raffinement ;
- Appels des nœuds associés aux sous-programmes raffinant les états de l'automate de contrôleΓ en donnant des valeurs adéquates aux triplets de paramètres {`initial`, `reset`,`kill`} ;
- Résolution des problèmes de sorties communes soit aux sous-programmes et à l'automate de contrôleΓsoit aux sous-programmes entre eux.

Ce dernier point est résolu de manière similaire au problème des sorties communes dans une mise en parallèle. Nous ne considérons à présent que des programmes qui ne contiennent pas de tels problèmes. Par conséquentΓseuls les deux premiers points ci-dessus sont détaillés.

Si l'automate de contrôle n'est pas raffinéΓalors seule la première de ces trois parties est effectuée.

Les variables `entersq` et `exitsq` qui permettent respectivement de savoir si une transition de l'automate de contrôle qui atteint ou qui sort de l'état q est déclenchéeΓse définissent de la même façon que lors de la traduction d'un programme Argos sans raffinement en Lustre. Elles ne tiennent pas compte des informations globales. La définition de la variable `atq` qui code le statut d'un état à un instant donné doitΓelleΓêtre modifiéeΓde manière à prendre en compte ces informations globales.

Si l'état q est un état initialΓalors l'équation qui définit `atq` est :

```
atq = initial -> pre ( ( ( entersq or
                            (atq and not exitsq)
                          )
                          and not kill
                        )
                        or reset
                      )
```

L'état q est actif à l'instant initial du programme si et seulement si l'automate auquel il appartient est lui-même actif à cet instant. Cette dernière information est donnée par le paramètre `initial`. Pour tous les autres instantsΓl'état q est actif à un instant donné si et seulement si lors de l'instant précédent une des propositions suivantes est vraie :

1. Une transition de l'automate de contrôle qui entre dans l'état q (information locale) est déclenchée et l'automate n'a pas été interrompu (information globale) ;
2. L'état q était déjà actifΓaucune transition de l'automate de contrôle qui sort de qΓsauf éventuellement une boucleΓn'a été déclenchée (informations locales)Γet l'automate n'a pas été interrompu (information globale) ;
3. L'automate a été lancé ou réinitialisé (informations globales).

Chacune de ces propositions est codée en une expression booléenne construite sur les variables `entersq` et `exitsq` et les paramètres `reset` et `kill`. En conservant le même ordreΓnous avons :

1. `entersq and not (kill and not reset)`
2. `atq and not (exitsq and not enterq) and not (kill and not reset)`
3. `reset`

La disjonction de ces trois expressions booléennes se simplifie et donne l'expression argument du `pre` dans l'équation qui définit la variable `atq`.

Si l'état q n'est pas un état initialΓalors l'équation qui définit `atq` est :

```
atq = false -> pre ( ( ( entersq or
                          (atq and not exitsq)
                        )
                        and not kill
                      )
                      and not reset
                    )
```

Dans ce casΓnon seulement il est certain que l'état q n'est pas actif à l'instant initialΓmais en plusΓseules les deux premières des propositions définies dans le cas précédent assurent l'activité de l'état q à l'instant suivant. En effetΓl'état q n'étant pas l'état initial de son automateΓle fait que ce dernier ait été lancé ou réinitialisé dans l'instant courant n'affecte pas son activité à l'instant suivant.

Le codage de l'automate de contrôle étant établiΓil faut à présent appeler les nœuds Lustre associés aux sous-programmes en instanciant correctement les triplets de paramètres. Si l'état raffiné par le sous-programme n'est pas l'état initial de l'automate de contrôleΓalors le bon triplet de paramètres effectifs est {`false, entersq, exitsq or kill`}. En effet tous les automates qui sont sous l'état qΓsans être sous aucun autre opérateur de raffinementΓsont tels que :

- Ils sont nécessairement inactifs à l'instant initial (d'où le paramètre `initial` instancié par `false`) ;

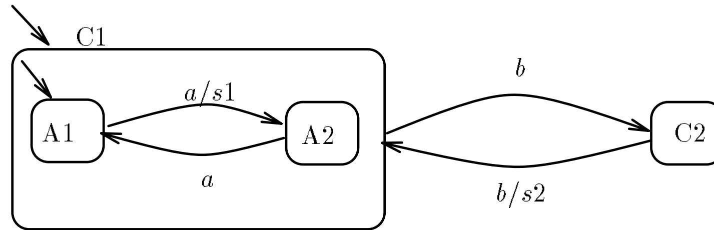


Figure 4.14: exemple illustrant la traduction d'Argos en Lustre

- Ils sont lancés si une transition qui entre dans l'état q est déclenchée (d'où le paramètre `reset` instancié par `entersq`) ;
- Ils sont interrompus dans deux situations : soit une transition qui quitte l'état q est déclenchéeΓsoit l'automate de contrôle est lui-même interrompu ou réinitialisé (d'où le paramètre `kill` instancié par `exitsq or kill`).

Dans le cas où l'état q est initialΓalors le bon triplet de paramètres effectifs est {`initial, entersq or reset, exitsq or kill`}Γpuisque tous les automates qui sont sous cet étatΓsans être sous aucun autre raffinementΓsont tels que :

- Ils sont actifs à l'instant initial si et seulement si l'automate de contrôle est lui-même actif à cet instant ;
- Ils sont lancés dans deux conditions : une transition qui entre dans l'état q est déclenchéeΓ ou l'automate de contrôle est lui-même lancé ou réinitialisé ;
- Ils sont interrompus dans deux situations : soit une transition qui quitte l'état q est déclenchéeΓsoit l'automate de contrôle est lui même interrompu ou réinitialisé (d'où le paramètre `kill` instancié par `exitsq or kill`).

Nous donnons un exemple de programme Argos construit à l'aide de l'opérateur de raffinement avec sa traduction en Lustre. Le programme Argos choisi (cf. figure 4.14) ne contient pas de problème de sorties communes à plusieurs composantes. Le programme Lustre équivalent est le suivant :

```
node main(a,b :  bool) returns (s1,s2 :  bool);
let
     (s1,s2) = raffC(a,b,true,false,false);
tel ;
```

Le neud `raffC` est défini par :

```
node raffC(a,b,initial,reset,kill :  bool) returns (s1,s2 :  bool);
var atC1,atC2,entersC1,entersC2,exitsC1,exitsC2 :  bool;
let
     atC1 = initial -> pre ( ( ( entersC1 or ( atC1 and not exitsC1 )
                               )
                               and not kill
                             )
                             or reset
                           );
     entersC1 = atC2 and b;
     exitsC1 = atC1 and b;
     atC2 = false -> pre ( ( ( entersC2 or
                               (atC2 and not exitsC2)
                             )
                             and not kill
                           )
                           and not reset
                         );
     entersC2 = atC1 and b;
     exitsC2 = atC2 and b;
     s2 = atC2 and b;
     s1 = raffA(a, initial, entersC1 or reset, exitsC1 or kill);
tel;

node raffA(a, initial,reset,kill :bool) returns (s1 :  bool);
var atA1,atA2,entersA1,entersA2,exitsA1,exitsA2 :  bool;
let
     atA1 = initial -> pre ( ( ( entersA1 or ( atA1 and not exitsA1 )
                               )
                               and not kill
                             )
                             or reset
                           );
     entersA1 = atA2 and a;
     exitsA1 = atA1 and a;
     atA2 = false -> pre ( ( ( ( entersA2 or
                                 (atA2 and not exitsA2)
                               )
                               and not kill
                             )
                             and not reset
                           );
     entersA2 = atA1 and a;
     exitsA2 = atA2 and a;
     s1 = atA1 and a;
tel;
```

Les blocs Argos

Nous n'avons pas mentionné jusqu'à présent le cas des blocs Argos. Ceux-ci ne posent aucun problème particulier. Le nœud Lustre correspondant est simplement constitué d'un appel au nœud résultat de la fonction de traduction sur le corps du bloc.

Adéquation de la traduction vis-à-vis de la détection syntaxique des problèmes de causalité en Lustre

La détection en Lustre des problèmes de causalité est effectuée par un critère syntaxique. Ce critère est trop fort : il rejette des programmes pour lesquels il existe une unique hypothèse cohérente sur la valeur des variables. CependantΓil est satisfaisant du point de vue du style de programmation Lustre.

ToutefoisΓsi les programmes Lustre sont produits automatiquementΓrien n'assure que ce critère syntaxique soit toujours bien adapté. Il faut donc vérifier que les programmes Lustre obtenus à partir d'Argos ne sont pas rejetés bien que corrects.

Nous rappelons la manière dont les cycles sont détectés en Lustre. C'est le prédicat `NoCycle` qui vérifie qu'il n'existe pas dans une liste d'équations de cycle dans la définition des valeurs des identificateurs. Nous supposons ici que le programme Lustre est expanséΓce qui signifie qu'il n'y a plus d'appel de nœud dans la liste d'équations. Une liste contient un cycle si et seulement si le graphe de dépendances de la relation binaire $\mathcal{R}el$ entre identificateurs en contient un. Le couple $(idf1, idf2)$ appartient à $\mathcal{R}el$ si et seulement s'il existe une équation de la forme $idf1 = exp$ dans la listeΓet si $idf2$ est utilisé dans expΓexcepté le cas où toutes les occurrences de $idf2$ sont à l'intérieur d'une expression argument d'un opérateur `pre`.

Pour déterminer l'adéquation — ou l'inadéquation — des programmes Lustre obtenus par traduction de programmes ArgosΓnous définissons une relation de dépendance $\mathcal{R}$ entre événements internes du programme Argos. Nous établissons ensuite les liens entre le graphe de cette relation de dépendance et celui de la relation $\mathcal{R}el$ utilisé pour détecter les cycles en Lustre.

Définition 4.12 Relation $\mathcal{R}$
Soient $\mathcal{Y}$ l'ensemble de tous les événements internes d'un programme Argos, $\mathcal{T}$ l'union des ensembles de transitions de tous les automates du programme. Nous définissons une relation $\mathcal{R}$ incluse dans $\mathcal{Y} \times \mathcal{Y}$.

$$(i1, i2) \in \mathcal{R} \textit{ ssi } \exists (q, m_e/o, q') \in \mathcal{T} \textit{telle que } (i2 \in {m_e}^+ \cup {m_e}^-) \wedge (i1 \in o)$$

■

La traduction des programmes Argos en Lustre est telle que si le graphe de dépendance de $\mathcal{R}$ contient un cycleΓalors celui de $\mathcal{R}el$ en contient un aussi. La réciproque est elle aussi vraie.

La justification du lien entre les deux relations de dépendance est donnée par la forme des équations qui codent les sorties d'un programme Argos en Lustre. En effetΓsi on restreint la relation $\mathcal{R}el$ aux variables localesΓ$\mathcal{R}$ et $\mathcal{R}el$ contiennent exactement les mêmes couples.

Ce lien entre $\mathcal{R}el$ et $\mathcal{R}$ nous permet d'identifier syntaxiquement les programmes Argos qui

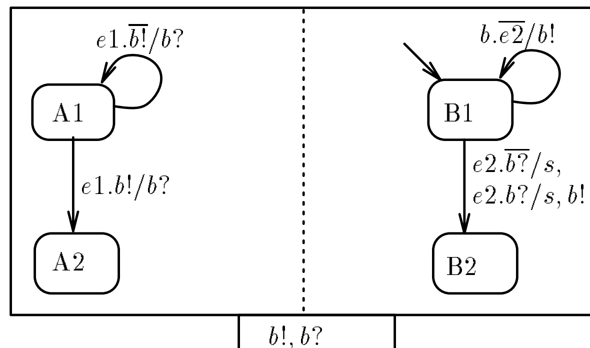


Figure 4.15: exemple de programme Argos problématique

conduisent à des programmes Lustre rejetés dès la vérification de la sémantique statique. Parmi ces programmesΓse trouvent de manière évidente les programmes Argos qui contiennent des problèmes de causalité. En effetΓles relations $\mathcal{R}$ associées à de tels programmes sont nécessairement bouclées. Si elles ne l'étaient pasΓil serait possible de définir un ordre dans la définition des valeurs des événements internes qui conduirait à une solution unique.

Se trouvent aussi parmi les programmes Argos qui conduisent à des programmes Lustre incorrectsΓdes programmes qui ne contiennent pas d'erreur de causalité.

Parmi ces programmes problématiques se trouve une classe qui en faitΓne pose pas de véritable problème. Ce sont ceux qui expriment du dialogue instantané entre composantesΓsans utiliser l'opérateur de raffinement. Le programme de la figure 4.15 est un exemple d'un tel cas.

Ce programme exprime une forme de dialogue instantané entre deux composantes : l'automate `A` sur réception de $e1$ interroge l'automate `B` pour savoir s'il se trouve ou non dans l'état `B1`. Cette interrogation s'effectue par l'événement interne $b?$. La réponse à cette interrogation se fait par l'événement interne $b!$. Le programme équivalent en utilisant l'opérateur de raffinement est donné par la figure 4.16.

Dans ce programme équivalentΓla relation $\mathcal{R}$ n'est pas bouclée car la dépendance entre $b!$ et $b?$ a disparu.

D'autres formes de programmes problématiques existent. Les plus gênants sont ceux dans lesquels la présence d'un cycle dans $\mathcal{R}$ qui ne se traduit pas par un problème de causalitéΓest due à une configuration d'états inaccessible. Le programme de la figure 4.17 est un exemple d'un tel cas. Le programme ne peut pas se trouver simultanément dans les états $A2$ et $B2$Γ par conséquent le problème de causalité sur les événements internes a et b n'est pas détecté. CependantΓla présence d'un cycle dans la définition de ces événements internes conduit au rejet du programme Lustre associé.

4.5.2 La traduction structurelle d'un programme ArgoLus en Lustre

Nous avons vu comment un programme Argos est codé structurellement en un programme Lustre. Nous allons maintenant voir comment cette fonction de traduction peut être étendue aux pro-

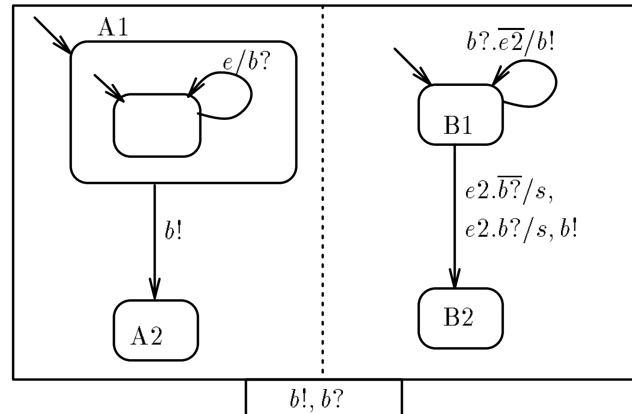


Figure 4.16: programme Argos équivalent non problématique

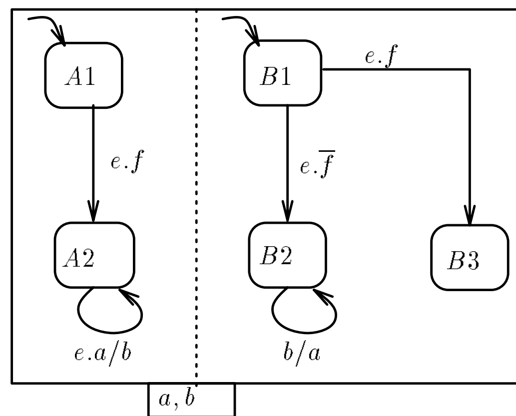


Figure 4.17: problème du à la présence d'un état inaccessible

grammes ArgoLus. Un programme ArgoLus est composé soit d'une structure d'opérateurs ArgosΓ soit d'une liste d'équations LustreΓdans lesquelles cohabitent appels de procédures Argos et appels de nœuds Lustre. Il pourrait sembler suffisant de traduire toutes les composantes Argos qui composent le programme ArgoLusΓsans modifier les nœuds Lustre. Ce n'est pas le casΓil est en effet nécessaire de modifier les nœuds Lustre qui font partie du programme ArgoLus.

Un nœud Lustre plongé à l'intérieur d'un programme Lustre est toujours actif. Ce n'est pas le cas d'un nœud Lustre plongé à l'intérieur d'un programme ArgoLus. En effetΓs'il est sous un raffinementΓson activité est conditionnée par l'activité de l'état qui le contrôle. Si l'on se réfère à la notion d'inactivité telle qu'on la conçoit en ArgosΓdire qu'un nœud Lustre est inactifΓsignifie que tout le reste du programme se comporte comme si ce nœud n'existait pas. Cela revient à considérer qu'à un instant où le nœud est inactifΓtoutes ses sorties sont fausses. Par conséquent le comportement d'un nœud dépend du fait qu'il est ou non actifΓet cette dépendance qui n'est pas prévue dans le nœud initial doit être prise en compte lors de la traduction d'un programme ArgoLus en Lustre. La prise en compte de cette possible inactivité des nœuds Lustre est l'objet de la transformation que subissent les nœuds Lustre d'un programme ArgoLus.

L'activité d'un nœudΓcomme l'activité d'un automate dans un programme ArgosΓn'est pas une information localeΓelle dépend des réactions de tous les automates à l'intérieur desquels le nœud est placé. Nous retrouvons là l'utilité des trois paramètres {`initial, reset, kill`} introduits lors de la traduction d'un programme Argos en LustreΓpour déterminer l'activité d'un automate. Ils vont être désormais utilisés pour le codage d'un automateΓmais aussi pour la transformation d'un nœud Lustre. Le résultat de cette transformation est un autre nœud Lustre qui a pour entrées supplémentaires les trois paramètres en question.

De la même manière que pour la traduction des programmes ArgosΓles programmes principaux du langage ArgoLus doivent être identifiés afin d'instancier une première fois le triplet de paramètres {`initial,reset,kill`}. Les valeurs initiales de ces paramètres sont toujours données par {`true, false,false`}Γcar tout automate ou tout nœud Lustre qui n'est sous aucun raffinement est toujours actif.

Nœud (ou bloc) Lustre

De la même manière que pour la traduction d'Argos en LustreΓnous avons introduit une variable `atq` pour signifier l'activité d'un état à l'instant courant dans le programme ArgosΓnous introduisons une variable `atSleep`Γlocale au nœud résultatΓpour signifier l'inactivité d'un nœud Lustre à l'instant courant dans le programme ArgoLus.

L'équation qui définit cette variable est la suivante :

```
atSleep = not initial -> pre ( (kill and not reset)
                              or (atSleep and not reset)
                             )
```

En effetΓl'activité du nœud à l'instant initial est une information globale précisément codée par le paramètre `initial`. Pour tous les instants suivantsΓle nœud est inactif si et seulement si à l'instant précédent une des propositions suivantes est vérifiée :

- A la fin de l'instant courantΓle nœud devient inactif et n'est pas réinitialisé;
- Le nœud est inactif et il le reste à la fin de l'état courant.

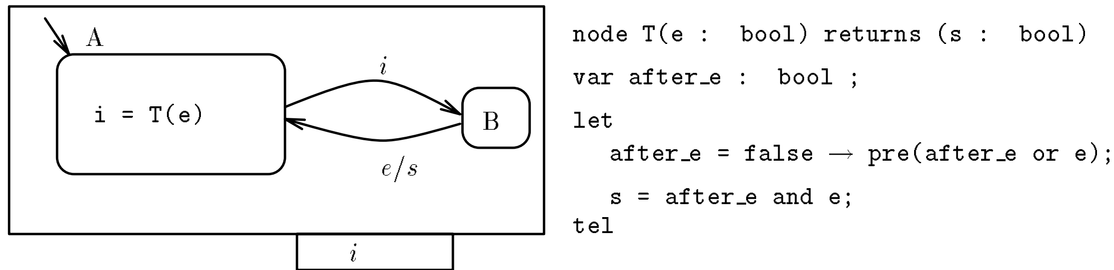


Figure 4.18: programme ArgoLus traduit en Lustre

Cette variable `atSleep` est utilisée pour modifier les équations qui constituent le corps du nœud : à chaque instant où `atSleep` est vraieΓtous les identificateurs définis par un nœud deviennent faux. Par conséquentΓune équation de la forme `id = exp` devient :

`id = if atSleep then false else exp'` (i)

Dans cette nouvelle équationΓl'expression `exp'` n'est pas égale à `exp`. En effetΓune importante conséquence de l'introduction du raffinement comme opérateur sur des nœuds Lustre est que ces derniers peuvent dorénavant être réinitialisés. Par conséquentΓsi e est l'expression qui définit la valeur d'une variable lors de la première activation du nœudΓalors à chaque fois que ce dernier est réinitialiséΓla valeur de cette variable doit de nouveau être calculée à partir de l'expression e.

En LustreΓ seul l'opérateur d'initialisation $exp1$`->`$exp2$ permet de différencier l'expression utilisée pour définir la valeur d'une variable lors de la première activation ($exp1$) de celle utilisée pour tous les instants suivants ($exp2$). D'après la remarque précédenteΓ $exp1$ doit aussi être l'expression utilisée pour calculer la valeur de la variable lorsque le nœud est réinitialisé. Par conséquent dans l'équation (i)Γ`exp'` doit prendre en compte cette possibilité de réinitialisation. De manière généraleΓ`exp'` est le résultat de la fonction `Reinit` appliquée à `exp`. Cette fonction transforme uniquement les expressions de la forme $exp1$`->`$exp2$ en

`Reinit(`$exp1$`) -> if pre(reset) then Reinit(`$exp1$`) else Reinit(`$exp2$`)`.

Considérons l'exemple du programme ArgoLus donné par la figure 4.18. Le programme Lustre obtenu par traduction est le suivant :

```
node main(e : bool) returns (s:bool);
let
    s = interne(e,true,false,false);
tel;


node interne(e,initial,reset,kill : bool) returns (s:bool);
var i : bool;
let
    (s,i) = raffAB(i,e,initial,reset,kill);
tel;
```

```
node raffAB(i,e,initial,reset,kill:bool) returns (s,i :  bool)
var atA,atB,entersA,entersB,exitsA,exitsB :bool;
let
     atA = initial -> pre ( ( ( entersA or
                                (atA and not exitsA)
                              )
                              and not kill
                            )
                            or reset
                          );
     entersA = atB and e;
     exitsA = atA and i;
     atB = false -> pre ( ( ( entersB or
                                (atB and not exitsB)
                              )
                              and not kill
                            )
                            and not reset
                          );
     entersB = atA and i;
     exitsB = atB and e;
     s = atB and e;
     i = Tm(e,initial, entersA or reset, exitsA or kill);
tel;

node Tm(e,initial,reset,kill :  bool) returns (i :  bool)
var after_e,atSleep :bool;
let
     atSleep = not initial -> pre((kill and not reset) or (atSleep and not
reset));
     after_e = if atSleep then false
            else false -> (if pre(reset) then false else pre(after_e) or
pre(e));
     i = if atSleep then false else after_e and e;
tel;
```

4.5.3 La solution symétrique : traduction d'ArgoLus en Argos

Il est aussi possible de traduire tout programme argoLus en un programme Argos. Cette solution est détaillée dans[JLMR94a]. Elle est basée sur la traduction structurelle de Lustre en Argos. Celle-ci est plus simple que la traduction inverse car ne se pose pas le problème de l'existence dans le langage traduit d'un opérateur qui n'a pas d'équivalent dans le langage cible. Ce problème se pose lors de la traduction d'Argos en Lustre avec l'opérateur de raffinement. De plus c'est la présence de cet opérateurΓqui engendre la modification des nœuds Lustre du programme

ArgoLusΓlors de sa traduction en Lustre. Lors de la traduction des programmes ArgoLus en ArgosΓles composantes Argos ne sont pas modifiées.

L'idée de la traduction structurelle de Lustre en Argos est la suivante. Tout d'abord les programmes Lustre sont modifiés syntaxiquement de telle manière qu'il n'y ait qu'un seul opérateur par équation. Il est alors possible d'associer à chaque équationΓen fonction de l'opérateur qu'elle contientΓun automate qui définit le statut de la variable définie par l'équation. Tous ces automates doivent ensuite être placés en parallèle — en utilisant l'opérateur Argos — et toutes les variables locales au programme Lustre doivent être déclarées internes.

Cette traduction permet de détecter de manière exacte les problèmes de causalité présents dans un programme Lustre booléen. En effetΓil suffit de traduire ce programme Lustre en programme Argos puis de calculer le modèle de ce programme.

4.6 Conclusion

Nous avons présenté dans ce chapitre la définition du langage mixte ArgoLus. Ce langage permet de décrire des automates booléens modèles de systèmes réactifs en combinant les avantages de la programmation déclarative flots de données et de la programmation impérative à base d'automates parallèles et hiérarchisés. Sa définition est basée sur la possibilité d'interchanger nœuds Lustre et procédures Argos à l'intérieur des programmes des deux langages. Par conséquentΓce langage introduit une structure de contrôle hiérarchique dans le langage Lustre. De plusΓil est possible de choisir pour chaque composante du système réactif à décrireΓle langage le plus adapté à sa description.

Pour mettre en œuvre la définition d'ArgoLusΓnous avons choisi de définir la traduction structurelle de tout programme ArgoLus soit en programme LustreΓsoit en programme Argos. Ces traductions permettent d'utiliser les compilateurs respectifs de ces deux langages comme compilateur ArgoLus. Elles reposent essentiellement sur la définition de traductions structurelles entre Lustre et Argos. Nous avons présenté en détail dans ce chapitreΓla traduction des programmes ArgoLus en LustreΓcar c'est celle des deux qui est la plus difficile.

L'étude des traductions structurelles entre Argos et Lustre constitue un résultat à part entière de notre travail. Elles mettent en évidence les liens entre ces deux langagesΓet répondent à l'objectif d'homogénéisation des environnements de programmation associés à ces langages. En particulierΓil devient possible d'utiliser dans l'environnement du langage ArgosΓles outils de l'environnement Lustre suivants :

- Le compilateur Lustre [Ray91]. Cette utilisation du compilateur Lustre comme compilateur de programmes Argos doit faciliter l'intégration des données non booléennes dans le langage Argos. En effetΓsi cette intégration s'effectue en conservant cette possibilitéΓalors la mise en œuvre de cette extension d'Argos sera immédiate ;
- Le système Lésar [Rat92] qui permet de prouver des programmes Lustre par la méthode des observateurs. Cet outil est basé sur une représentation et une manipulation symboliques des programmes — à l'aide de graphes de décision binaires — qui lui permettent de mettre en œuvre de manière efficace cette méthode de vérification ;

- Le compilateur sur circuit [Roc92] qui permet de produire du code pour une architecture matérielle ;
- Le simulateur de programmes Lustre.

L'utilisation de ces outils est d'autant plus intéressante que la traduction structurelle d'Argos en Lustre est telleΓqu'il est facilement possible de retrouver dans le programme Lustre obtenu par traductionΓles informations pertinentes pour le programme Argos. Cette capacité à faire de la "remontée dans le source" est un facteur positif qui favorise l'utilisation des outils Lustre à partir de programmes Argos.

La définition d'ArgoLus telle que nous l'avons donnée s'étend très facilement pour prendre en compte des données non booléennesΓà condition que celles-ci — entréesΓsorties ou variables locales — soient uniquement gérées par des nœuds Lustre et par conséquentΓne communiquent pas avec des composantes Argos. En effetΓla sémantique flots de données de Lustre que nous donnons est valide quel que soit le type des variables manipulées par le programme. La mise en œuvre du langage doit alors être défini par la traduction structurelle d'ArgoLus en Lustre.

Le travail de définition du langage ArgoLus que nous venons de présenter est à rapprocher :

- Des travaux sur la compilation sur circuits du langage Esterel restreint aux signaux purs (non accompagnés de valeurs)[Ber91]. Ces travaux ont permis de définir une traduction structurelle de ce sous-ensemble d'Esterel vers le langage Lustre. Cette traduction est assez différente de la traduction structurelle d'Argos en LustreΓcar on ne retrouve pas en Esterel une distinction aussi nette qu'en Argos entre opérateurs et objets de base. On trouve cependant dans cette traduction le fait que des signaux supplémentaires sont introduits d'après la structure du programme pour déterminer à chaque instant l'ensemble des opérateurs actifs du programme. Ces signaux sont à rapprocher des triplets de paramètres introduits dans la traduction Argos vers Lustre.
- Les travaux sur le séquencement de tâches en Signal [RG94]. Ces travaux sont une réponse possible au besoin de structure de contrôle hiérarchique à l'intérieur des langages déclaratifs pour la programmation des systèmes réactifs. La solution développée en Signal consiste à exprimer ce type de décomposition par l'introduction d'une notion d'intervalle qui permet de délimiter l'activité d'une tâche. Ces intervalles sont définis par des occurrences de signaux. Les tâches peuvent être soit suspendues lorsque leur intervalle n'est pas actifΓsoit réinitialisées à chaque nouvelle activité de l'intervalle ;
- Les travaux sur le mélange du formalisme des Grafcets et d'Esterel développés par l'équipe dirigée par C. André à l'université de Nice.

Partie II

Le modèle graphes temporisés : vérification et programmation

Chapitre 5

Le modèle graphes temporisés : vérification

Les graphes temporisés sont une forme d'automates avec variables étudiés depuis peu longtemps [ACD90]Γpour modéliser des systèmes fortement contraints par l'évolution du temps. Les intérêts de ce modèle sont multiples :

- Le phénomène d'explosion du nombre d'états est réduit par rapport au cas des automates booléens. En effetΓune des causes de ce phénomène est la présence dans les programmes de structures de contrôle à base de compteursΓpour exprimer des durées. Dans une modélisation en automates booléensΓles valeurs de ces compteurs sont mémorisées sous forme d'étatsΓd'où le problème d'explosion lorsque plusieurs compteurs évoluent simultanément. Dans un graphe temporiséΓces valeurs sont mémorisées par des variables. Il s'ensuit que les limites pratiques d'utilisation des outils de vérification basés sur les automates booléens sont repoussées par les outils qui opèrent sur les graphes temporisés ;

- Les données manipulées ne sont pas uniquement booléennesΓmais les actions effectuées sur celles-ci sont suffisamment simplesΓpour que des résultats de décidabilité permettent de définir des méthodes de vérification formelle de propriétés intéressantes. Parmi celles-ci se trouvent *les propriétés quantitatives temporelles temps réel*. Dans cette caractérisationΓ *temporelle* fait référence à l'ordre d'exécution des actions du programmeΓalors que *temps réel* fait référence à l'écoulement du temps entre ces actions. Si le système considéré est un contrôleur de passage à niveau automatiqueΓde telles propriétés permettent par exempleΓ d'exprimer que la barrière est toujours fermée une minute après avoir reçu l'ordre de se baisser.

Ces deux caractéristiques des graphes temporisés montrent que ce modèle est un bon compromis entre les automates booléens et les automates interprétés généraux.

Nous présentons dans ce chapitre les graphes temporisés et leur vérification à l'aide de la logique temporelle temps réel TCTL. La sémantique des graphes temporisés est définie en termes de *systèmes temporisés*Γqui ont eux-mêmes une sémantique définie en termes de *séquences de pas*. Sur ces séquences de pas est définie la sémantique des formules de la logique TCTL. Toute la première partie du chapitre est consacrée à la présentation de ces différentes notions.

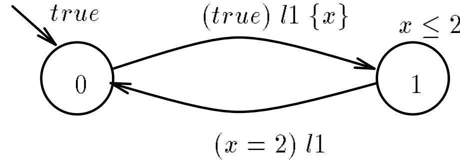


Figure 5.1: un exemple de graphe temporisé

Nous formalisons ensuite la notion de *graphes temporisés modèles de systèmes réactifs* sur laquelle nous travaillons dans cette deuxième partie. Dans le chapitre suivantΓnous montrons comment la sémantique du langage Argos peut être définie en termes d'un sous-ensemble des graphes temporisés modèles de systèmes réactifs [JMO93].

5.1 Les graphes temporisés

Les graphes temporisés sont des automates étendus avec un ensemble de variables réellesΓappelées *horloges*Γdont les valeurs croissent uniformément avec le passage du temps. Une horloge peut être mise à zéro par un arc du graphe. A tout instant la valeur d'une horloge est égale au temps écoulé depuis la dernière fois qu'elle a été initialisée. Une condition temporelle portant sur les horloges est associée à tout sommet et à tout arc du graphe. Le système peut rester dans un sommet tant que la condition associée à ce sommetΓson *invariant*Γest vérifiée par les valeurs des horloges. Un arc peut être franchi si et seulement si les valeurs courantes des horloges satisfont la condition associée à l'arc.

La figure 5.1 donne un exemple de graphe temporisé ayant une horloge x. L'invariant du sommet 0 est la condition toujours vraieΓle système peut donc y rester indéfiniment. L'arc du sommet 0 vers le sommet 1 est étiqueté par l'étiquette $l1$. Il peut être franchi à tout instant car la condition sur les horloges qui lui est associée est la condition toujours vraie. Lorsque cet arc est franchiΓl'horloge x est remise à zéro. Le sommet 1 a pour invariant la condition $x \leq 2$Γce qui indique que le système ne peut pas rester dans cet état plus de deux unités de temps. De plus lorsque l'horloge x a atteint la valeur 2Γle système peut franchir l'arc qui le ramène dans le sommet 0.

Définition 5.1 Condition temporelle
Soit $\mathcal{H}or$ un ensemble fini d'horloges prenant leurs valeurs dans $\mathbb{R}^+$. L'ensemble $\Psi(\mathcal{H}or)$ des conditions temporelles sur $\mathcal{H}or$ est défini par la grammaire suivante :

$$\psi ::= true \mid h_1 \prec c \mid h_1 - h_2 \prec c \mid \neg\psi \mid \psi \wedge \psi$$
$$\text{où } h_1, h_2 \in \mathcal{H}or, c \in \mathbb{Z} \text{ et } \prec \in \{<, \leq, >, \geq, =\}.$$

■

Une valuation v d'horloges est une fonction qui associe à chaque $h \in \mathcal{H}or$ une valeur de $\mathbb{R}^+$. L'ensemble des valuations est noté Val.

Pour chaque $\psi \in \Psi(\mathcal{H}or)$ et $v \in Val\Gamma\psi(v)$ représente la valeur de la condition ψ évaluée en v. La valuation v satisfait une condition $\psi\Gamma$si et seulement si $\psi(v)$ est vraie. On utilise alors la notation $v \models \psi$.

Soient v une valuation des horloges de $\mathcal{H}or$ et H un sous-ensemble de $\mathcal{H}or$; $v[H := 0]$ est la valuation qui associe à h la valeur 0 si h appartient à H et $v(h)$ sinon.

Soient v une valuation des horloges et $t \in \mathbb{R}^+$; $v+t$ est la valuation v' telle que $v'(h) = v(h)+t$ pour tout $h \in \mathcal{H}or$.

Définition 5.2 Les graphes temporisés
Un graphe temporisé GTest un quintuplet $(S, s_{init}, A, H, \mathcal{I}nv)$ où

- *S est l'ensemble des sommets du graphe*
- *s_{init} est le sommet initial du graphe*
- *$A \subseteq S \times \Psi(H) \times \mathcal{L} \times 2^H \times S$ est l'ensemble des arcs du graphe. $\mathcal{L}$ est un ensemble d'étiquettes quelconque. On note l un représentant de $\mathcal{L}$. Un arc est donc défini par un tuple de la forme $(s_s, \psi, l, \mathcal{R}, s_b)$. Les sommets s_s et s_b sont respectivement les sommets source et but de l'arc. ψ est la condition que doivent satisfaire les horloges du graphe pour pouvoir franchir l'arc. Enfin, $\mathcal{R}$ est l'ensemble des horloges à remettre à zéro lorsque l'arc est franchi.*
- *H est l'ensemble des horloges du graphe*
- *$\mathcal{I}nv : S \rightarrow \Psi(H)$ associe à chaque sommet du graphe son invariant. La fonction $\mathcal{I}nv$ doit vérifier la propriété suivante :*
  $\forall s \in S, v \in Val, t \in \mathbb{R}^+, v + t \models \mathcal{I}nv(s) \Rightarrow v \models \mathcal{I}nv(s)$

■

La sémantique des graphes temporisés est définie en termes de *systèmes temporisés.*

5.2 Les systèmes temporisés

Un système temporisé est un système de transitions étiquetées soit par un élément de $\mathcal{L}\Gamma$soit par un réel strictement positif. Une transition étiquetée par un réel simule le passage du tempsΓla valeur du réel définissant la quantité de temps écoulé.

Définition 5.3 Les systèmes temporisés
Un système temporisé ST est un triplet (Q, q_{init}, T) où

- *Q est l'ensemble des états du système temporisé*
- *q_{init} est l'état initial du système*

- $T \subseteq Q \times (\mathcal{L} \ \cup \ \mathbb{R}^{+*}) \times Q$ *est l'ensemble des transitions du système. Elles peuvent avoir deux formes :* (q_s, l, q_b) *ou bien* (q_s, t, q_b) *. Les états* q_s *et* q_b *sont les états source et but de la transition.* l *est une étiquette quelconque appartenant à* $\mathcal{L}$*,* t *est un réel strictement positif. Une transition étiquetée par* l *est dite* instantanée*, une transition étiquetée par* t *est dite* temporelle.

La relation de transition T *satisfait les deux propriétés suivantes :*

déterminisme des transitions temporelles *Le système ne peut évoluer, depuis un état donné et en un temps donné, que vers un seul autre état.*

$$\forall q, q', q'' \in Q, \ \forall t \in \mathbb{R}^{+*}, \ (q, t, q') \in T \ \wedge \ (q, t, q'') \in T \quad \Longrightarrow \quad q' = q''$$

additivité du temps *Les transitions doivent exprimer la continuité du temps. Si le système peut évoluer d'un état* q *vers un état* q' *en* t *unités de temps, et de* q' *vers* q'' *en* t' *unités de temps, alors il doit pouvoir évoluer de* q *vers* q'' *en* $t + t'$ *unités de temps. La réciproque doit aussi être vraie.*

$$\forall q, q' \in Q, \ \forall t, t' \in \mathbb{R}^{+*}, (\exists q'' \in Q, (q, t, q'') \in T \ \wedge \ (q'', t', q') \in T) - \ (q, t + t', q') \in T$$

■

L'ensemble des systèmes temporisés est noté $\mathcal{ST}$.

5.2.1 Sémantique d'exécution des systèmes temporisés

L'exécution d'un système temporisé est une séquence de pas où chaque pas a deux étapes :

- Le premier pas est une étape de "progression du temps". Dans cette étape le temps progresse d'une quantité finie. Cette étape peut être réduite à une progression nulle du temps.
- Dans la seconde étapeΓle système exécute au plus une action instantanée — franchissement d'une transition instantanée —.

Un nouveau pas commence lorsque la seconde étape se termine.

Afin de modéliser les exécutions d'un système temporiséΓnous définissons les notions de pas et de séquence de pas. Tout d'abordΓnous introduisons la notation : $q \stackrel{t}{\longmapsto} q'$ où q et q' sont deux états d'un système temporisé et t est un réel positif. Cette relation modélise la notion d'étapeΓ et est utilisée par la suite pour définir un pas d'exécution d'un système temporisé.

Soit (Q, q_{init}, T) un système temporiséΓ

- Pour tout $q, q' \in Q, q \stackrel{0}{\longmapsto} q'$ si $q = q'$ ou s'il existe $l \in \mathcal{L}$ telle que $(q, l, q') \in T$.
- Pour tout $q, q' \in Q$ et $t > 0$Γ$q \stackrel{t}{\longmapsto} q'$ si $(q, t, q') \in T$.

Notons que d'après ces définitionsΓpour tout état qΓ$q \stackrel{0}{\longmapsto} q$.

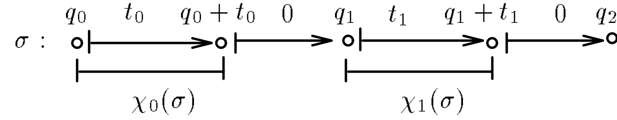


Figure 5.2: support d'une séquence

Pour tout état q et tout $t \in \mathbb{R}^+$Γ$q + t$ désigne l'état atteint par le système si dans l'état q on laisse progresser le temps de t unités de temps ; si $t = 0$Γ$q' = q$Γsinon $q \overset{t}{\longmapsto} q'$. L'état q' n'existe pas nécessairementΓmais lorsqu'il existeΓil est unique grâce à la propriété de déterminisme temporel.

Définition 5.4 Pas d'exécution d'un système temporisé
Soit $ST = (Q, q_{init}, T)$ un système temporisé, pour tout $q, q' \in Q$ et $t \in \mathbb{R}^+$, on définit :

$q \overset{t}{\triangleright} q'$ *si et seulement si* $q \overset{t}{\longmapsto} q + t$ *et* $q + t \overset{0}{\longmapsto} q'$.

On dit que $q \overset{t}{\triangleright} q'$ est un pas de q à q' de durée t. ■

Définition 5.5 Séquence de pas
Soit $ST = (Q, q_{init}, T)$ un système temporisé, une séquence de pas $\sigma = q_0 \overset{t_0}{\triangleright} q_1 \overset{t_1}{\triangleright} \ldots$ *est une suite infinie de pas. L'ensemble des séquences de pas de ST est noté Σ_{ST}.*

Pour tout $q \in Q$, on définit $\Sigma_{ST}(q) = \{q_0 \overset{t_0}{\triangleright} q_1 \overset{t_1}{\triangleright} \ldots \in \Sigma_{ST} | q_0 = q\}$, l'ensemble des séquences de pas qui ont q pour état initial. ■

Une séquence de pas est une séquence discrète. Par conséquentΓelle ne modélise pas l'exécution d'un système temporiséΓun pas $q \overset{t}{\triangleright} q'$ modélisant en fait le passage par tous les états $q + t'$ tels que $t' \leq t$. La notion de *support d'une séquence* prend en compte la densité du domaine de temps. Elle définit l'ensemble de tous les états visités par une séquence de pas donnée. Pour pouvoir définir un ordre sur cet ensemble d'états — l'ordre dans lequel les états sont visités — le support d'une séquence étiquette chaque état $q_i + t$ par le rang d'apparition de q_i dans la séquence. La définition suivante est illustrée par la figure 5.2.

Définition 5.6 Support d'une séquence
Soit $ST = (Q, q_{init}, T)$ un système temporisé, pour toute séquence de pas $\sigma = q_0 \overset{t_0}{\triangleright} q_1 \overset{t_1}{\triangleright} \ldots$ de Σ_{ST}, le support de σ, noté $\chi(\sigma)$, est défini par :

$\chi(\sigma) = \bigcup_{i \in \mathbb{N}} \ \chi_i(\sigma)$, *où pour tout* $i \in \mathbb{N}$, $\chi_i(\sigma) = \{(q_i + t, i) \mid t \in \mathbb{R}^+ \ \wedge \ t \leq t_i\}$. ■

L'ordre total associé au support d'une séquence est l'ordre de gauche droite sur la figure 5.2.

Définition 5.7 Ordre total associé au support d'une séquence
Soit $ST = (Q, q_{init}, T)$ un système temporisé, pour toute séquence de pas $\sigma = q_0 \overset{t_0}{\triangleright} q_1 \overset{t_1}{\triangleright} \ldots$ de Σ_{ST}, l'ordre total associé au support de σ est noté $\preceq_\sigma$. Il est défini par :

$\forall \ i, j \in \mathbb{N}, t, t' \in \mathbb{R}^+ \quad (q_i + t, i) \preceq_\sigma (q_j + t', j) \Longleftrightarrow i < j \vee (i = j \wedge t \leq t')$. ■

5.2.2 Les systèmes bien temporisés

Parmi les séquences de pas d'un système temporiséΓcertaines sont divergentes — le temps peut progresser au-delà de tout instant — alors que d'autres sont convergentes.

Définition 5.8 Séquence divergente
Soit $ST = (Q, q_{init}, T)$ *un système temporisé, une séquence* $\sigma = q_0 \overset{t_0}{\triangleright} q_1 \overset{t_1}{\triangleright} \ldots$ *de* Σ_{ST} *est divergente si pour tout* $t \in \mathbb{R}^+$*, il existe* $i \in \mathbb{N}$ *tel que* $t < \sum_{j \leq i} t_j$*. L'ensemble des séquences divergentes de* ST*est noté* Δ_{ST}*.* ■

On considère en général qu'un système temporiséΓpour constituer une représentation acceptable de la réalitéΓne doit pas comporter d'état à partir duquel toutes les séquences sont convergentesΓc'est-à-dire à partir duquel le temps est "bloqué".

Définition 5.9 Système bien temporisé
Soit $ST = (Q, q_{init}, T)$ *un système temporisé,* ST *est dit bien temporisé si tout préfixe fini d'une séquence de* ST *est préfixe d'une séquence divergente.* ■

Notons que cette définition n'exclut pas qu'un système bien temporisé contienne des séquences convergentes. Elle exclut les systèmes qui contiennent au moins un état accessibleΓà partir duquel toutes les séquences sont convergentes.

Nous verrons que les systèmes bien temporisés jouent un rôle particulier en ce qui concerne la définition d'une méthode d'évaluation des formules d'une logique temporelle temps réel sur les systèmes temporisés.

Par extensionΓnous dirons par la suite qu'un graphe est bien temporisé lorsque le système associé au graphe par la fonction sémantique est lui-même bien temporisé.

5.3 Sémantique des graphes temporisés en termes de systèmes temporisés

Soit $GT = (S, s_{init}, A, H, \mathcal{I}nv)$ un graphe temporisé. On lui associe le système temporisé défini comme suit :

- Les états du système sont des couples (s, v) où s est un sommet de GT et v une valuation de ses horloges ;
- L'état initial du système temporisé est le couple (s_{init}, v_{init}) où v_{init} est la valuation qui associe la valeur 0 à toutes les horloges du graphe. Nous la désignons par la suite par $\vec{0}$;
- A partir d'un état (s, v) un arc $(s, \psi, l, \mathcal{R}, s_b)$ du graphe peut être franchi si les valeurs des horloges satisfont la condition ψ. L'étiquette de la transition ainsi définie est lΓson état but est $(s_b, v[\mathcal{R} := 0])$ (cf. règle $[\mathcal{G}1]$ ci-dessous) ;

- Le système peut séjourner dans un sommet sΓle temps progressantΓtant que l'invariant $\mathcal{I}nv(s)$ de s est vérifié par les valeurs des horloges (cf. règle $[\mathcal{G}2]$ ci-dessous).

FormellementΓla sémantique d'un graphe temporisé en termes de systèmes temporisés est définie par la fonction $\mathcal{G}$.

Définition 5.10 Sémantique des graphes temporisés
Le système temporisé associé à un graphe temporisé $GT = (S, s_{init}, A, H, \mathcal{I}nv)$ *est* $\mathcal{G}(GT) = (Q, q_{init}, T)$, *où :*

- $Q = \{(s, v)\ s \in S, v \in Val \mid v \models \mathcal{I}nv(s)\}$
- $q_{init} = (s_{init}, \vec{0})$
- T *est la relation de transition incluse dans* $Q \times \mathcal{L} \cup \mathbb{R}^+ \times Q$ *définie par les règles suivantes (on note* $q \xrightarrow{l} q'$ *un élément de* T*) :*

$$\frac{(s_s, \psi, l, \mathcal{R}, s_b) \in A,\ v \models \psi}{(s_s, v) \xrightarrow{l} (s_b, v[\mathcal{R} := 0])} \qquad [\mathcal{G}1]$$

$$\frac{v + t \models \mathcal{I}nv(s)}{(s, v) \xrightarrow{t} (s, v + t)} \qquad [\mathcal{G}2]$$

■

On vérifie aisément que cet ensemble de transitions satisfait les propriétés d'additivité et de déterminisme temporel.

Remarquons que dans la règle $[\mathcal{G}1]$Γla condition $v[\mathcal{R} := 0] \models \mathcal{I}nv(s_b)$ est implicite du fait de la définition de l'ensemble Q.

Notons enfinΓque d'après cette définitionΓles systèmes temporisés obtenus sont telsΓque si q est de la forme (s, v) et $t \geq 0$ alors $q + t$ est égal à $(s, v + t)$: la progression du temps s'effectue toujours en restant dans le même sommet.

5.4 La vérification des systèmes temporisés

5.4.1 Principe général

La méthode de vérification présentée dans cette thèse est définie dans [HNSY92]. Elle est basée sur la logique temporelle temps réel TCTL [ACD90ΓAlu91ΓHNSY92]. TCTL est une extension temps réel de la logique temporelle arborescente CTLΓprésentée dans le chapitre 1.

La sémantique de CTL est définie sur les séquences d'exécution d'un automate modèle d'un programme. Celle de TCTL est définie sur les séquences d'exécution d'un système temporisé modèle d'un graphe temporisé (qui peut être lui-même le modèle d'un programme).

La méthode de vérification présentée repose sur l'existence d'un algorithme de "model-checking". Cet algorithme permetΓétant donnés un graphe temporisé et une formule φ de la logique considéréeΓde déterminer l'ensemble des états du système temporisé modèle du grapheΓqui satisfont φ.

Parmi les algorithmes de "model-checking" existantsΓdeux approches peuvent être distinguées :

- **L'approche énumérative** consiste à construire le modèle et à évaluer la formule après une énumération exhaustive de tous les états de ce modèle. Les ensembles d'états sont représentés en extension. Cette approche n'est possible que si le modèle est fini.
- **L'approche symbolique** consiste à représenter les ensembles d'états par des prédicats et à évaluer la formule sans construire le modèle. Un des avantages de cette approche est qu'elle peut s'appliquer aussi à des modèles infinis.

Dans le cas des systèmes temporisésΓ l'approche énumérative ne peut pas être appliquéeΓ puisqu'elle impose que le modèle soit fini.

Une solution dérivée de l'approche énumérative a été définie dans [ACD90ΓAlu91]. Elle repose sur la construction d'un modèle réduit modulo une relation d'équivalence entre les états. Cette relation garantit que deux états équivalents ne peuvent pas être distingués par les formules de TCTL. Le modèle réduit étant finiΓl'évaluation des formules peut s'effectuer par application d'une méthode énumérative. L'inconvénient de cette solution est que la taille du modèle construit croît exponentiellement en fonction du nombre d'horloges du graphe temporisé ainsi que des valeurs des constantes qui apparaissent dans ses conditions temporelles.

L'intérêt de la méthode présentée est que l'algorithme proposé est symbolique : il travaille directement sur le graphe temporisé. Ici son coût ne dépend ni du nombre d'horlogesΓni des valeurs des constantes qui apparaissent dans le graphe.

L'algorithme symbolique repose d'une part sur la caractérisation des opérateurs temporels de TCTL comme des points fixes de fonctionnelles monotonesΓd'autre part sur l'utilisation de contraintes linéaires pour représenter des ensembles d'états. L'idée de la caractérisation par point fixe des opérateurs de TCTL consiste à étendre les fonctionnelles utilisées pour la logique CTL. L'une des difficultés est de passer d'opérateurs sur des séquences de pas discrètes à des opérateurs sur des séquences de pas continues. Le résultat de ce travail est la caractérisation des opérateurs temporels de TCTL par des points fixes de fonctionnelles monotones lorsque le système temporisé considéré est bien temporiséΓd'où l'importance de cette propriété pour les aspects de vérification formelle.

L'algorithme symbolique de vérification des formules de TCTL sur des graphes temporisés ainsi définiΓest implémenté dans l'outil KRONOS [NSY91ΓNSY92ΓYov93]. KRONOS implémente aussi la phase de compilation de l'algèbre de processus ATP [NRSV90ΓNS91ΓNSY92ΓNic92] vers des graphes temporisés.

5.4.2 La logique temporelle temps réel TCTL

La version de TCTL que nous présentons est celle définie dans [ACD90]. Une façon simple d'introduire le temps explicitement dans la syntaxe consiste à borner la portée dans le temps des opérateurs temporels.

Considérons l'abréviation $\exists\Diamond p$ de CTL. Cette formule est satisfaite dans un état q si et seulement s'il existe une séquence d'exécution issue de qΓtelle qu'un des états visités par la séquence satisfait p. A l'aide de la logique TCTLΓil est possible de borner dans le temps le moment où l'état qui satisfait p est rencontré le long de la séquence.

Par exempleΓla formule $\exists\Diamond_{\leq 2}p$ de TCTL permet d'exprimer qu'il existe une exécution du programme où la propriété p est vérifiée avant 2 unités de temps.

Définition 5.11 Formules de TCTL
$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi\, \exists\mathcal{U}_{\#\, c}\, \varphi \mid \ \mid \varphi\, \forall\mathcal{U}_{\#\, c}\, \varphi$

Où p est un prédicat de base sur les états, $\#$ est un symbole de relation dans l'ensemble $\{<, \leq, >, \geq, =\}$ et $c \in \mathbb{N}$. ■

Remarque 5.1

- *Les opérateurs temporels non bornés sont définis en bornant leur équivalent par ≥ 0.*
- *les opérateurs temporels bornés : $\forall\Diamond_{\#\, c}\varphi, \exists\Diamond_{\#\, c}\varphi, \exists\Box_{\#\, c}\varphi$ et $\forall\Box_{\#\, c}\varphi$ sont définis comme des abréviations à partir des opérateurs $\exists\mathcal{U}_{\#\, c}$ et $\forall\mathcal{U}_{\#\, c}$ de la façon suivante :*
  - $\forall\Diamond_{\#\, c}\varphi = (true\ \forall\mathcal{U}_{\#\, c}\ \varphi)$,
  - $\exists\Diamond_{\#\, c}\varphi = (true\ \exists\mathcal{U}_{\#\, c}\ \varphi)$,
  - $\exists\Box_{\#\, c}\varphi = (\neg\forall\Diamond_{\#\, c}\neg\ \varphi)$,
  - $\forall\Box_{\#\, c}\varphi = (\neg\exists\Diamond_{\#\, c}\neg\ \varphi)$.

■

Les formules de TCTL sont interprétées sur les états d'un système temporisé. Comme pour CTL (cf. section 1.3)Γun système temporisé satisfait une formule si et seulement si son état initial la satisfait. Nous rappelons que la relation de satisfaction d'une formule par un état dépend de la fonction d'interprétation des prédicats de base. Cette fonction est notée $\mathcal{I}$. La relation de satisfaction d'une formule par un état est notée $\models_{\mathcal{I}}$.

La définition de cette relation nécessite de définir le temps écoulé depuis le début de l'exécution d'une séquence σ jusqu'à un état $q_i + t$ où q_i est un état de la séquence de pas σ. L'état $q_i + t$ pouvant apparaître plusieurs fois dans la séquenceΓon lui associe son rang d'apparition dans la séquence.

Définition 5.12 Temps écoulé dans un état
Soit $ST = (Q, q_{init}, T)$ un système temporisé, pour toute séquence de pas $\sigma = q_0 \overset{t_0}{\triangleright} q_1 \overset{t_1}{\triangleright} \ldots$, $\pi(q_i + t, i)$, avec $t \in \mathbb{R}^+$, est le temps écoulé depuis q_0 jusqu'à la ième occurrence de $q_i + t$. La fonction π est définie par :

$$\pi(q_i + t, i) = \sum_{j<i} t_j + t$$ ■

Définition 5.13 Sémantique des formules de TCTL
Pour un système temporisé $ST = (Q, q_{init}, T)$, un état $q \in Q$, et une formule φ de TCTL, la relation de satisfaction $q \models_{\mathcal{I}} \varphi$ est définie par induction sur la structure de φ de la façon suivante :

$q \models_{\mathcal{I}} p$ *ssi* $\mathcal{I}(p, q)$
$q \models_{\mathcal{I}} \neg\varphi$ *ssi* $\neg(q \models_{\mathcal{I}} \varphi)$
$q \models_{\mathcal{I}} \varphi_1 \vee \varphi_2$ *ssi* $q \models_{\mathcal{I}} \varphi_1$ *ou* $q \models_{\mathcal{I}} \varphi_2$
$q \models_{\mathcal{I}} \varphi_1 \exists\mathcal{U}_{\#\,c}\, \varphi_2$ *ssi*
$\exists\sigma \in \Sigma_{ST}(q), \exists(q', i) \in \chi(\sigma)$ *tel que* $q' \models_{\mathcal{I}} \varphi_2 \wedge \pi(q', i)\#\, c \wedge \forall(q'', j) \preceq_\sigma (q', i)\; q'' \models_{\mathcal{I}} \varphi_1 \vee \varphi_2$
$q \models_{\mathcal{I}} \varphi_1 \forall\mathcal{U}_{\#\,c}\, \varphi_2$ *ssi*
$\forall\sigma \in \Delta_{ST}(q), \exists(q', i) \in \chi(\sigma)$ *tel que* $q' \models_{\mathcal{I}} \varphi_2 \wedge \pi(q', i)\#\, c \wedge \forall(q'', j) \preceq_\sigma (q', i)\; q'' \models_{\mathcal{I}} \varphi_1 \vee \varphi_2$

■

Afin de mieux comprendre la sémantique de ces formulesΓnous donnons le sens intuitif de quelques exemples :

- La formule $\exists\Diamond_{>3}p$ exprime qu'il existe une exécution du systèmeΓoù la proposition p est satisfaite au moins une fois par un étatΓaprès l'instant 3 ;
- La formule $\forall\Box_{\leq 5}p$ exprime que sur toutes les exécutions du systèmeΓla proposition p est vraie pour tous les états jusqu'à l'instant 5 ;
- La formule $\forall\Diamond_{\leq 3}p$ exprime que sur toutes les exécutions du systèmeΓil existe un état qui vérifie pΓavant l'instant 3.

5.5 Les graphes temporisés modèles de systèmes réactifs

Le comportement d'un système réactif peut être modélisé par un graphe temporisé si on structure les étiquettes en deux parties pour modéliser les interactions du système avec son environnement. La condition de déclenchement d'un arc est donc formée de deux conditions : la condition sur les valeurs des horloges et la condition sur le statut des entrées.

Nous utilisons pour formaliser les étiquettes du grapheΓles mêmes notations que pour les automates modèles de systèmes réactifs.

Nous notons $\mathcal{GT}$ l'ensemble des graphes temporisés qui ont des étiquettes de la forme m_e/O. Les fonctions In et Out respectivement définies en 2.3 et 2.4Γqui associent à un automate dont les étiquettes sont de cette forme ses ensembles d'entrées et de sortiesΓs'étendent sans difficulté aux éléments de $\mathcal{GT}$.

Les propriétés :

- De non recouvrement des ensembles d'entrées et de sorties ;
- De complétude et non contradiction des monômesΓ

définies respectivement en 1.4 et 1.5 s'étendent elles aussi facilement aux éléments de $\mathcal{GT}$. $\mathcal{GT}^{\emptyset}$ (resp. $\mathcal{GT}^{c}$) désigne le sous-ensemble des éléments de $\mathcal{GT}$ qui vérifient la première (resp. seconde) de ces propriétés.

Les propriétés de déterminisme et de réactivité peuvent être étendues aux graphes temporisés. IntuitivementΓces deux propriétés mises en commun signifient que pour tout sommet du grapheΓ quelles que soient les valeurs des horloges à condition qu'elles satisfassent l'invariant du sommetΓ et pour tout monôme d'événementsΓil existe un et un seul arc dans le graphe qui peut être déclenché.

Définition 5.14 Déterminisme d'un graphe temporisé
Soit $GT = (S, s_{init}, A, H, \mathcal{I}nv) \in \mathcal{GT}$; il est déterministe si et seulement si

$\forall s_s \in S, m_e \in \mathcal{M}_c^{\star}(In), v \in$ *Val telle que* $v \models \mathcal{I}nv(s_s)$,

$$|\{(s_s, \psi, m_e, O, \mathcal{R}, s_b) \in A \text{ tel que } v \models \psi\}| \leq 1$$

■

Définition 5.15 Réactivité d'un graphe temporisé
Soit $GT = (S, s_{init}, A, H, \mathcal{I}nv) \in \mathcal{GT}$; il est réactif si et seulement si

$\forall s_s \in S, m_e \in \mathcal{M}_c^{\star}(In), v \in$ *Val telle que* $v \models \mathcal{I}nv(s_s)$,

$$|\{(s_s, \psi, m_e, O, \mathcal{R}, s_b) \in A \text{ tel que } v \models \psi\}| \geq 1$$

■

$\mathcal{GT}_d$ (resp. $\mathcal{GT}_r$) désigne le sous-ensemble des éléments de $\mathcal{GT}$qui vérifient la propriété de déterminisme (resp. réactivité).

Définition 5.16 Graphe temporisé modèle de système réactif
Les graphes temporisés modèles de systèmes réactifs sont les éléments de $\mathcal{GT}$ qui vérifient les quatre propriétés ci-dessus. Ce sont donc des éléments de $\mathcal{GT}_{dr}^{\emptyset,c}$. ■

Chapitre 6

Sémantique d'Argos en termes de graphes temporisés

Dans les langages synchronesΓaucune notion intrinsèque de temps n'est définie. L'expression de contraintes temporelles s'effectue en comptabilisant les occurrences d'une entrée qui constitue une base de temps. Le langage Argos offre la structure particulière d'état temporisé pour exprimer facilement ce type de contraintes.

Cette gestion du temps a pour principales conséquences :

- Le caractère discret du temps manipulé dans les programmes ; les durées sont donc approchées (cf. section 3.2.4). Cette discrétisation du temps n'est pas surprenante puisque les langages synchrones sont des langages de programmation.
- Le caractère multiforme du temps manipulé par les programmes. On peut choisir plusieurs entrées indépendantes qui expriment des échelles de temps non corrélées.

Le besoin d'exprimer des contraintes temporelles n'apparaît pas uniquement au niveau des langages de programmation. Il apparaît aussi lorsqu'on s'intéresse aux aspects de vérification des programmes. En effetΓl'expérience montre qu'un nombre important de propriétés que l'on souhaite voir satisfaites par un programmeΓsont des propriétés qui font intervenir le temps de manière quantitative. Pour ne prendre qu'un exempleΓprouver qu'une alarme est toujours déclenchée lorsqu'une situation critique est détectéeΓest un pas intéressant dans le processus de vérification du programme. NéanmoinsΓil est encore plus satisfaisant de prouver que la durée qui sépare la détection de la situation critique du déclenchement de l'alarmeΓest toujours inférieure à une borne.

Les graphes temporisés et la méthode de vérification basée sur la logique TCTL qui accompagne ce modèle constituent une réponse possible aux besoins en vérification de propriétés quantitatives temps réel. En modélisant l'évolution du temps par des variables — et non par des états comme dans le cas des automates booléens — ils sont moins exposés au phénomène d'explosion d'étatsΓce qui augmente les possibilités de vérification. De plusΓla logique TCTL permet d'exprimer simplement des propriétés quantitatives.

L'intérêt pour les langages synchrones de posséder des sémantiques qui en font des cons-

tructeurs de graphes temporisés est donc évident. Nous présentons dans ce chapitre comment le langage Argos peut être étendu pour que le modèle d'un programme soit un graphe temporisé [JMO93].

Dans la méthode de vérification des graphes temporisésΓle temps est mono-forme et continuΓ ce qui semble s'opposer à son utilisation sur des programmes synchrones. Le temps mono-forme impose le choix d'un temps de référence dans le programme : celui sur lequel porte la propriété. Par conséquentΓil n'est pas possible de vérifier des propriétés quantitatives qui font intervenir deux bases de temps indépendantes. C'est là une limitation de la méthode. L'hypothèse d'évolution continue du temps n'est contradictoire qu'avec l'objectif d'exécution d'un programme synchrone. Dans ce cas en effetΓle temps est nécessairement discret. Si par contre seul l'aspect modélisation (ou spécification) du comportement d'un système réactif importeΓalors l'hypothèse d'évolution continue du temps est particulièrement intéressante car elle permet d'exprimer des durées temporelles qui sont exactes. Lorsque l'exécution du programme est aussi un objectif souhaitéΓla préservation sur le code exécutable des vérifications effectuées avec une interprétation continue du tempsΓsoulève un problème difficile auquel nous répondons dans ce chapitre. Notons que jusqu'à présentΓce problème lié à l'exécution des programmes n'a pas fait l'objet d'étude particulièreΓles graphes temporisés étant utilisés uniquement pour spécifier des systèmes.

Pour terminer cette introductionΓnotons que la description de graphes temporisés non bien temporisés — capable de "bloquer" le temps — à partir d'un langage synchroneΓserait contradictoire avec la philosophie de ces langagesΓdans lesquels le temps est une entrée comme une autre que l'on ne maîtrise pas.

Ce chapitre est divisé en quatre sections :

- La section 6.1 présente la sémantique d'Argos en termes de graphes temporisés ;
- La section 6.2 contient la preuve de la bonne temporisation des graphes temporisés décrits à partir d'Argos.
- La section 6.3 présente la problématique : "vérification en temps continu et exécution en temps discret"Γainsi que les résultats de préservation obtenus.
- La section 6.4 contient un exemple de modélisation en Argos temporiséΓet de propriété exprimée en TCTL.

6.1 Sémantique d'Argos en termes de graphes temporisés

6.1.1 Principe général

Nous commençons par présenter les différentes étapes suivies pour définir la sémantique d'Argos en termes de graphes temporisésΓpuis nous détaillons chaque étape.

1. Identifier syntaxiquement dans le langage les éléments qui vont engendrer des horloges dans le graphe temporisé.
2. Définir la traduction des objets de base du langage en graphes temporisés.

3. Définir la sémantique des opérateurs du langage en termes de graphes temporisés.

Une contrainte limite les choix possibles dans la première étape : les éléments pour engendrer les horloges doivent tous être reliés à la même base de temps.

Considérons à présent l'application de ce principe au langage ArgosΓétape par étape.

1. Les états temporisés sont des éléments du langage parfaitement appropriés à un codage par horloges dans le graphe : la valeur courante de la temporisation peut être remise à zéroΓelle augmente lorsqu'on reste dans l'état et ne peut pas dépasser une limite. De plus les transitions issues d'un état temporisé sont conditionnées par la valeur courante de la temporisation. La contrainte sur l'unicité de la base de temps n'est pas directement satisfaiteΓpuisque les états temporisés d'un programme n'ont pas nécessairement tous le même événement de référence. Il est donc nécessaire de désigner parmi l'ensemble des événements possibles celui qui va définir dans le graphe la base de temps. Le choix fait en Argos consiste à donner à cet événement le nom réservé χ. Les autres états temporisés sont alors considérés comme des macro-notations et sont expansés en automates classiques. Après expansionΓles seuls états temporisés du programme sont ceux dont l'événement de référence est χ. Nous simplifions dans toute la suite les figuresΓen omettant de préciser cet événement de référence dans les états temporisés.

2. Les objets de base du langage Argos sont des automates qui contiennent éventuellement des états temporisés par χ. Le graphe temporisé modèle de cet automate est défini selon les principes suivants.

   - A chaque état de l'automate est associé un état du graphe temporisé. Cet état temporisé a pour invariant la condition toujours vraie si l'état de l'automate n'est pas un état temporisé. Dans le cas contraireΓune horloge h est associée à l'étatΓet l'invariant de cet état est égal à $h \leq d$ où d est la borne de temporisation de l'état.
   - Une transition de l'automate dont les états source et but ne sont pas des états temporisés n'est pas modifiée dans le graphe temporisé. Une transition dont l'état but est un état temporisé doit remettre l'horloge associée à cet état à zéro. Une transition dont l'état source est un état temporiséΓsi elle est la transition d'expiration doit être conditionnée par $h = d$. Le statut des entrées est alors indifférent pour le franchissement de cet arcΓet l'ensemble de ses sorties est hérité de la transition. Si la transition n'est pas la transition d'expiration alors la condition sur les horloges de l'arc correspondant dans le graphe temporisé est $h < d$. La condition sur le statut des entréesΓainsi que l'ensemble des sorties associés à cet arc sont hérités de la transition.

   La figure 6.1 présente un automate temporisé (6.1(a)) et le graphe temporisé qui lui est associé (6.1(b)).

3. La fonction $\mathcal{F}_{op}$ (cf. section 2.2.4) qui définit la sémantique de chaque opérateur du langage doit prendre pour opérandes non plus des automates mais des graphes temporisés. Les principes communs aux fonctions associées aux trois opérateurs du langage Argos sont les suivants :

   - Un sommet du graphe représentant une configuration des états actifs du programmeΓl'invariant associé à un sommet est toujours égal à la conjonction des invariants associés aux états actifs ;

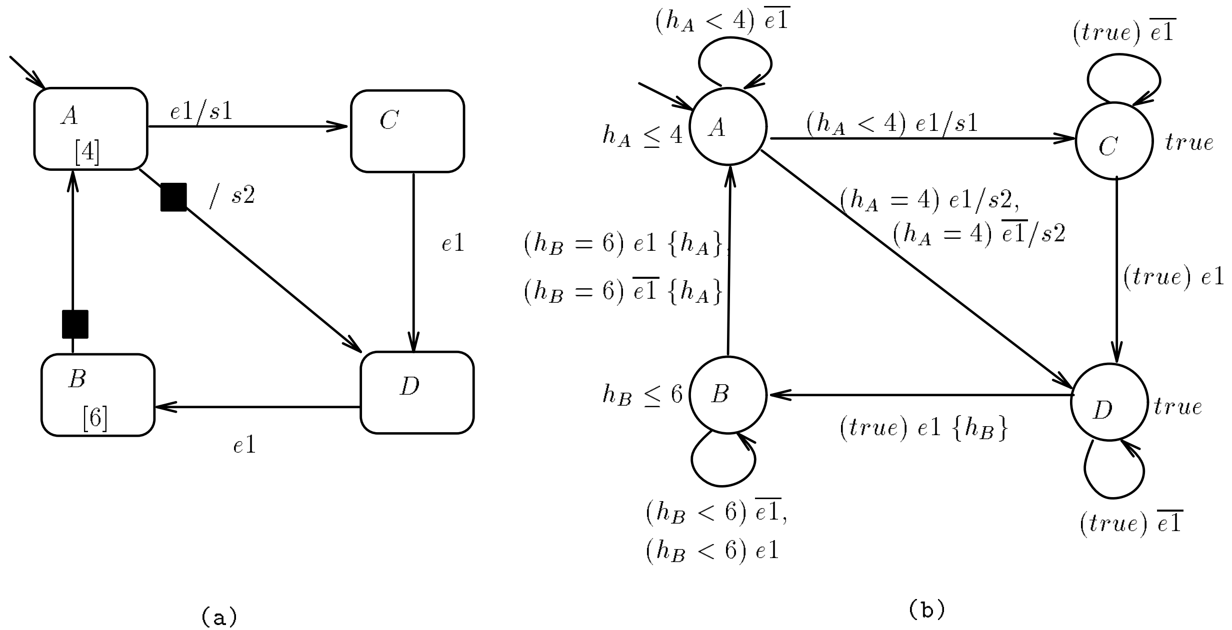


Figure 6.1: traduction d'un automate temporisé en un graphe temporisé

- Un arc du graphe représentant un ensemble de transitions des automates du programmeΓla condition associée à un arc du graphe est égale à la conjonction des conditions associées aux transitions qui le composent.

Nous présentons dans la suite la nouvelle fonction sémantiqueΓnommée $\mathcal{S}^t$Γqui associe à tout programme Argos un graphe temporisé qui est son modèle. Pour la mise en œuvre de cette fonctionΓnous utilisons la même forme de présentation que dans les chapitres précédents — algorithme de calcul des transitions du modèle—. Pour cette raisonΓn'apparaissent dans cette présentation ni la fonction de traduction d'un automate en un graphe temporiséΓni les fonctions $\mathcal{F}_{op}$ de manière explicite.

6.1.2 Exemples de programmes Argos temporisés avec leur modèle

Le comportement à décrire est le suivant :

toutes les deux unités de temps émettre la sortie $s1$ et ceci jusqu'à la première occurrence de l'entrée e. Une fois cette première occurrence de e reçue, émettre $s1$ toutes les dix unités de temps.

Un programme Argos qui décrit ce comportement est donné par la figure 6.2(a)Γle graphe temporisé modèle de ce programme est donné par la figure 6.2(b). Ce graphe est à la fois réactif et déterministe.

Si à présent le changement de fréquence d'émission de la sortie $s1$ s'effectue non pas sur occurrence d'une entrée eΓmais sur l'expiration d'un certain délaiΓpar exemple au bout de huit unités

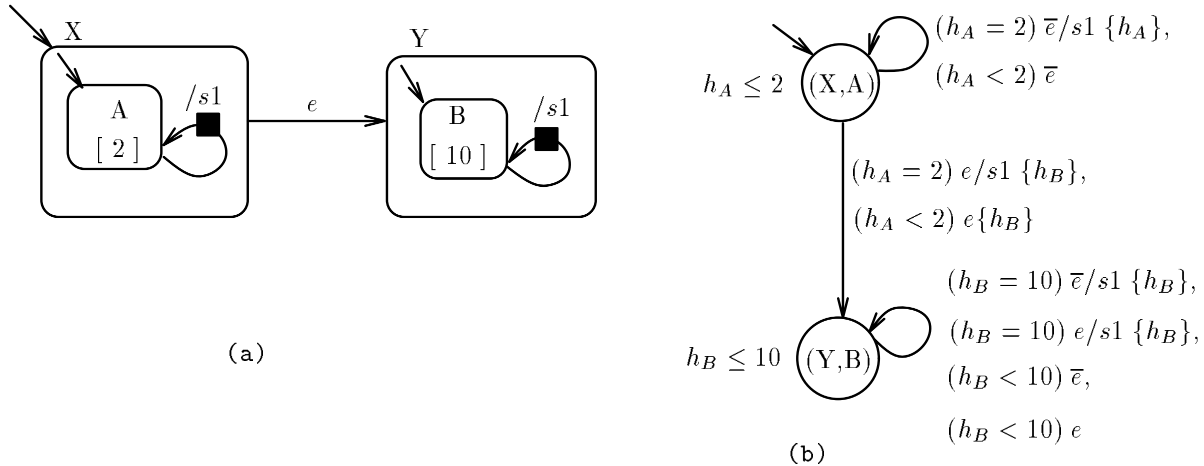


Figure 6.2: exemple de graphe temporisé décrit en Argos

de tempsΓalors le programme précédent devient celui de la figure 6.3 (a). Le graphe temporisé modèle de ce programme est donné par la figure 6.3 (b). Le comportement du programme étant uniquement dirigé par l'évolution du tempsΓce graphe n'admet pas d'entrée. Dans ce grapheΓ certains arcs ne sont jamais franchis car leur condition d'horloges est insatisfaisable. Ils font tout de même partie du modèle du programme.

Observons sur la figure 6.4 la forme du graphe temporisé décrit par un programme Argos temporisé obtenu à l'aide d'une mise en parallèle de deux automates temporisés. Dans ce grapheΓ le sommet (AΓY) est en fait inaccessible. Il fait cependant partie du modèle du programme. La présence de sommets inaccessibles dans les modèles est la cause du caractère approché de la détection des problèmes de causalité dans les programmes Argos temporisés.

6.1.3 Graphes temporisés modèles de programmes Argos

D'après les exemples qui viennent d'être présentésΓnous pouvons remarquer qu'à chaque horloge h_i du graphe est associée un entier strictement positif d_i. C'est la borne de l'état temporisé pour lequel on a introduit l'horloge.

Les graphes temporisés décrits en Argos possèdent les deux caractéristiques suivantes :

- Les invariants des sommets sont de la forme $\psi ::= true \mid h_i \leq d_i \mid \psi \wedge \psi$;
- Les conditions temporelles sur les arcs sont de la forme $\psi ::= true \mid h_i < d_i \mid h_i = d_i \mid \psi \wedge \psi$

La forme particulière de ces conditions temporelles — conjonctions de propositions de base pouvant prendre deux formes — permet d'utiliser la notation de monôme. Ces monômes sont construits sur des sous-ensembles d'horloges du graphe. Nous les notons m_h pour les différencier des monômes d'événementsΓnotés m_e. La convention d'interprétation des monômes d'horloges est la suivante :

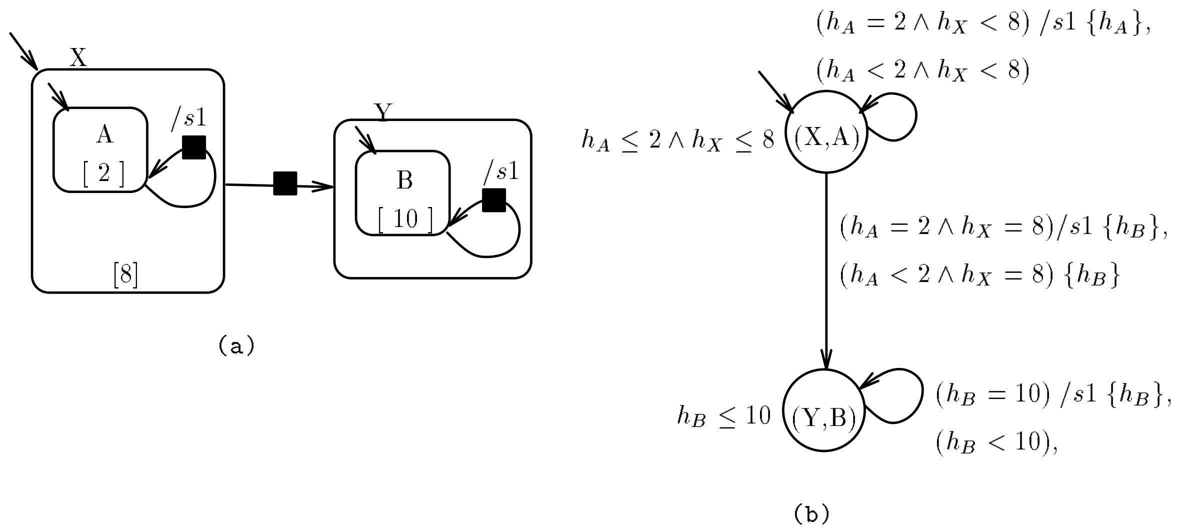


Figure 6.3: traduction de l'opérateur de raffinement

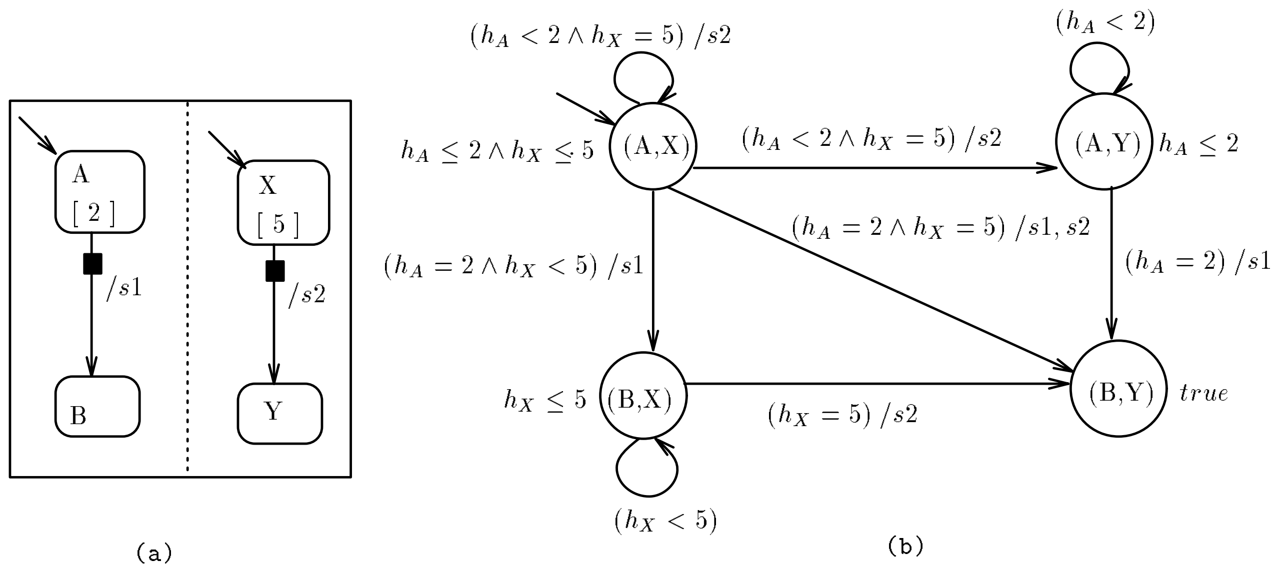


Figure 6.4: traduction de la mise en parallèle

- Si $m_h = (\emptyset, \emptyset)$ alors la condition dénotée par m_h est la condition toujours vraie ;
- Si $m_h \neq (\emptyset, \emptyset)$ alors la condition désignée est la conjonction de propositions de base de la forme $h_i = d_i$ et $h_i < d_i$. Le choix entre ces deux formes dépend du statut de h_i dans m_h :
  - Si $h_i \in {m_h}^+$ alors la condition désignée contient la proposition de base $h_i = d_i$;
  - Si $h_i \in {m_h}^-$ alors la condition désignée contient la proposition de base $h_i < d_i$.

Par exempleΓle monôme $h_1.\overline{h_2}$ désigne la condition $h_1 = d_1 \wedge h_2 < d_2$.

Les monômes d'horloges qui conditionnent les arcs issus d'un sommet s sont construits sur un ensemble d'horloges qui dépend de s. Ce sont les horloges associées aux états qui composent le sommet s. On les appelle les horloges *pertinentes* de sΓet on note cet ensemble $H(s)$. Il est défini formellement par la suite (cf. définition 6.5).

Les arcs des graphes temporisés décrits en Argos sont de la forme $(s_s, m_h, m_e/O, \mathcal{R}, s_b)$Γoù m_e et m_h sont tous les deux complets et non contradictoiresΓle premier sur l'ensemble des entrées du grapheΓle second sur l'ensemble des horloges pertinentes pour s_s. Ils constituent la condition de franchissement de l'arc.

Les invariants associés aux sommets du graphe sont eux aussi construits sur l'ensemble des horloges pertinentes du sommet considéré. Plus précisémentΓl'invariant de s est la conjonction de propositions de base de la forme $h_i \leq d_i$Γpour toutes les horloges h_i pertinentes dans le sommet s.

Dans la suite du chapitreΓon ne s'intéresse plus qu'aux graphes temporisés ayant cette forme particulière de conditions et d'invariantsΓ$\mathcal{GT}$ désigne ce sous-ensemble.

6.1.4 Nouvelles définitions du déterminisme et de la réactivité

Sur cette forme de graphes temporisésΓil est possible de donner des définitions syntaxiques des propriétés de déterminisme et de réactivité d'un graphe temporiséΓéquivalentes aux définitions générales données en 5.14 et 5.15.

Pour celaΓnous définissons la fonction $\mathcal{M}_s$ qui dans chaque sommet s associe à une valuation qui satisfait l'invariant de sΓl'unique monôme m_h de $\mathcal{M}_c^\star(H(s))$ qui est tel que $v \models m_h$.

Définition 6.1 Fonction $\mathcal{M}_s$
Le profil de la fonction $\mathcal{M}_s$ est $\{v \in Val | v \models \mathcal{I}nv(s)\} \rightarrow \mathcal{M}_c^\star(H(s))$. Soient s un sommet d'un graphe temporisé et v une valuation qui satisfait l'invariant de s ; $\mathcal{M}_s(v) = ({m_h}_v^+, {m_h}_v^-)$ où :

$${m_h}_v^+ = \{h_i \in H(s) \mid v(h_i) = d_i\} \qquad {m_h}_v^- = \{h_i \in H(s) \mid v(h_i) < d_i\}$$

Par construction, nous avons donc bien $v \models m_{h_v}$. ■

Considérons la propriété de déterminisme d'un graphe temporisé définie en 5.14. Sur les graphes temporisés modèles de programmes ArgosΓelle devient : un graphe temporisé est déterministe si et seulement si $\forall s_s \in S, m_e \in \mathcal{M}_c^\star(In), v \in Val$ telle que $v \models \mathcal{I}nv(s_s)$,

$$|\{(s_s, m_h, m_e, O, \mathcal{R}, s_b) \in A \text{ tel que } v \models m_h\}| \leq 1$$

Cette définition peut être remplacéeΓvu la forme des graphes temporisés considérésΓpar celle qui suit.

Définition 6.2 Déterminisme d'un graphe temporisé décrit en Argos
Soit $GT = (S, s_{init}, A, H, \mathcal{I}nv) \in \mathcal{GT}$ *;* GT *est déterministe ssi*

$$\forall s_s \in S, m_e \in \mathcal{M}_c^\star(In), m_h \in \mathcal{M}_c^\star(H(s_s)), \ \ |\{(s_s, m_h, m_e, O, \mathcal{R}, s_b) \in A\}| \ \leq 1$$ ■

En effetΓd'une part — 5.14 ⇐ 6.2 — pour chaque valuation v qui satisfait l'invariant d'un sommet sΓil existe un unique monôme m_{hv} de l'ensemble $\mathcal{M}_c^\star(H(s))$ tel que $v \models m_{hv}$; d'autre part — 5.14 ⇒ 6.2 — la fonction $\mathcal{M}_s$ est surjective.

Pour les mêmes raisonsΓla définition de la propriété de réactivité générale 5.15 peut être remplacée par :

Définition 6.3 Réactivité d'un graphe temporisé décrit en Argos
Soit $GT = (S, s_{init}, A, H, \mathcal{I}nv) \in \mathcal{GT}$*;* GT *est réactif ssi*

$$\forall s_s \in S, m_e \in \mathcal{M}_c^\star(In), m_h \in \mathcal{M}_c^\star(H(s_s)), \ \ |\{(s_s, m_h, m_e, O, \mathcal{R}, s_b) \in A\}| \ \geq 1$$

■

6.1.5 Syntaxe abstraite et sémantique statique

La syntaxe abstraite et la sémantique statique des programmes Argos temporisés ont été définies dans la section 3.2Γlors de la présentation des états temporisés.

La syntaxe abstraite définie dans la section 3.11 peut être légèrement simplifiée au niveau de la définition d'un automate temporiséΓpuisque nous sommes sûrs que tous les états temporisés ont χ pour événement de référence. Par conséquentΓla fonction $\mathcal{T}$ n'associe plus à chaque état temporisé un couple (d, e) — borne de la temporisation et événement de référence — mais simplement d qui est la borne de la temporisation.

La sémantique statique définie en 3.12 doit être augmentée d'une contrainte pour assurer le déterminisme des automates temporisés. Cette contrainte n'est pas nécessaire pour la définition de la macro-notation des états temporisésΓcar le déterminisme est assuré sur la forme expansée du programme. Cette nouvelle contrainte est formalisée par :

$$\forall q \in Q, m_e \in \mathcal{M}_c^\star(In), \ |\{(q, m_e/O, q') \in T\}| \leq 1.$$

Les programmes principaux sont définis par la contrainte usuelle : ensembles d'entrées et de sorties disjoints.

On considère par la suite que les états des automates sont indicés globalement par rapport au programme. Nous utilisons les notations suivantes : si j est un état temporisé du programmeΓ h_j désigne l'horloge associée à jΓet $d_j = \mathcal{T}(j)$ est la borne associée à h_j.

6.1.6 Calcul du graphe temporisé modèle d'un programme Argos

Présentation de la fonction sémantique $\mathcal{S}^t$

La fonction sémantique $\mathcal{S}^t$ associe à tout programme Argos temporisé syntaxiquement correctΓ le graphe temporisé qu'il détermine : son modèle. Le profil de cette fonction est le suivant :

$\mathcal{S}^t : \mathcal{P}^t \rightarrow \mathcal{A}^c_{dr} \cup \{\text{ERROR}\}$

La valeur d'erreur signifie que le programme contient un problème de causalité. Si le programme est principalΓet qu'il ne contient pas de problème de causalitéΓalors son modèle appartient à $\mathcal{GT}^{\emptyset,c}_{dr}$Γc'est un modèle de système réactif.

Mise en œuvre de la fonction sémantique $\mathcal{S}^t$

L'ensemble des horloges du graphe temporisé est égal à $\mathcal{H}(P^t)$Γoù $\mathcal{H}$ est la fonction définie ci-dessous.

Définition 6.4 Ensemble des horloges associées à un programme temporisé

$$
\begin{array}{lcl}
\mathcal{H}(\text{NIL}) & = & \emptyset \\
\mathcal{H}(P_1 \| P_2) & = & \mathcal{H}(P_1) \cup \mathcal{H}(P_2) \\
\mathcal{H}(\overline{P^Y}) & = & \mathcal{H}(P) \\
\mathcal{H}(\mathbf{R}_{(Q,q_{init},T,X,\mathcal{T})}(R_1,\ldots,R_n)) & = & \mathcal{H}(R_1) \cup \ldots \cup \mathcal{H}(R_n) \cup \bigcup_{i \in X} h_i
\end{array}
$$

■

Les sommets du graphe modèle d'un programme P^t mémorise la structure de P^t et les états courants des automates du programme.

Seuls les sommets accessiblesΓsans analyser la satisfaisabilité des conditions sur les horlogesΓ à partir du sommet initial font partie de l'ensemble des sommets du graphe.

L'état initial du modèle de P^t est donné par la fonction $init(P^t)$ qui s'étend sans difficulté aux programmes temporisés.

La définition de la fonction $\mathcal{I}nv$ fait intervenir la fonction de calcul des horloges pertinentes d'un sommet. InformellementΓl'ensemble des horloges pertinentes dans un sommet s est constitué des horloges associées aux états temporisés actifs dans s.

Définition 6.5 Ensemble des horloges pertinentes dans un sommet

$$
\begin{array}{lcl}
H(nil) & = & \emptyset \\
H(s_1 \| s_2) & = & H(s_1) \cup H(s_2) \\
H(\overline{s^Y}) & = & H(s) \\
H(\mathbf{R}_{(Q,q_{init},q_c,T,X,\mathcal{T})}(r_1,\ldots,r_n)) & = & H(r_c) \cup (\ si\ (q_c \in X)\ alors\ h_{q_c}\ sinon\ \emptyset)
\end{array}
$$

■

L'invariant d'un sommet s est la conjonction de propositions de base de la forme $h_i \leq d_i$Γ pour toutes les horloges h_i pertinentes dans s.

Définition 6.6 Invariant des sommets du graphe temporisé modèle de P^t

Pour tout sommet s de $\mathcal{S}^t(P^t)$, $\mathcal{I}nv(s) = \bigwedge_{h_i \in H(s)} (h_i \leq d_i)$

■

L'algorithme général de calcul des arcs du graphe s'obtient en adaptant l'algorithme 2.1 de la manière suivante :

- Pour tout sommet accessible du graphe s_sΓon calcule l'ensemble des arcs qui en sont issusΓ en énumérant toutes les conditions de franchissement possibles. Une telle condition est de la forme m_hΓm_e ; où m_e est un monôme d'événements construit sur l'ensemble des entrées du grapheΓet m_h est un monôme d'horloges construit sur l'ensemble des horloges pertinentes dans s_s. Ces deux monômes sont non contradictoires et complets.
- La fonction de calcul d'une transition `CalculTrans` est remplacée par une fonction de calcul d'un arcΓnommée `CalculArc`. Ce calcul s'effectue en connaissant le sommet source et la condition de franchissement de l'arc à définir. Il peut échouer pour cause de détection d'un problème de causalité. S'il réussitΓil fournit les ensembles de sorties et d'horloges à remettre à zéro ainsi que le sommet but associés à cet arc.

L'utilisation de cet algorithmeΓlorsqu'il ne détecte pas d'erreurΓgarantit la génération d'un graphe temporisé réactif et déterministe : à partir de tout sommet du grapheΓquelle que soit la condition de franchissementΓil existe un et un seul arc issu du sommet qui est déclenché par cette condition.

Nous présentons à présent en détail la fonction de calcul d'un arc du graphe. Elle est définie par un ensemble de règles opérationnelles de la forme :

$$\frac{Condition}{s_s \xrightarrow{m_h, m_e / O, \mathcal{R}} s_b} \quad \text{[Exemple]}$$

Une telle règle doit s'interpréter comme suit : si la $Condition$ est vérifiée alors l'arc issu de s_s déclenché par m_hΓm_e existe et admet O comme ensemble d'événements générésΓ$\mathcal{R}$ comme ensemble des horloges à remettre à zéro et s_b comme sommet but. Les conditions des règles étant deux à deux exclusivesΓla fonction échoue si et seulement si aucune règle n'est applicable.

La fonction *prog* utilisée dans le chapitre 2.7 pour déterminerΓle programme P auquel appartient un état p Γest étendue aux programmes temporisés. Par conséquentΓelle s'applique à présent à un sommet d'un graphe temporisé et a pour résultat un programme temporisé.

La mise en parallèle

Soient $I_i = \mathcal{I}(prog(s_i))$ et $H_i = H(s_i) \quad \forall i \in \{1, 2\}$;

$$\frac{s_1 \xrightarrow{m_h[H_1], m_e[I_1]/O_1, \mathcal{R}_1} s_1' \qquad s_2 \xrightarrow{m_h[H_2], m_e[I_2]/O_2, \mathcal{R}_2} s_2'}{s_1 \| s_2 \xrightarrow{m_h, m_e/O_1 \cup O_2, \mathcal{R}_1 \cup \mathcal{R}_2} s_1' \| s_2'} \quad [\mathrm{Par}^t]$$

Un arc issu d'un sommet obtenu par mise en parallèle de deux composantes est la combinaison de deux arcs, chacun issu d'une composante. Cette combinaison s'exprime par l'union des ensembles de sorties et d'horloges à réinitialiser, et la mise en parallèle des sommets buts.

Le raffinement

Soient $R_c = prog(r_c), I_c = \mathcal{I}(R_c)$ et $H_c = H(r_c)$;

$$\frac{h_{q_c} \in {m_h}^+ \qquad (q_c, \Box/O_1, q_d) \in T \qquad r_c \xrightarrow{m_h[H_c], m_e[I_c]/O_2, \mathcal{R}_2} r_c'}{\mathbf{R}_{(Q,q_{init},q_c,T,X,\mathcal{T})}(r_1, \ldots, r_n) \xrightarrow{m_h, m_e/O, \mathcal{R}} \mathbf{R}_{(Q,q_{init},q_d,T,X,\mathcal{T})}(r_1, \ldots, init(R_c), \ldots, r_n)} \quad [\mathrm{Raff1}^t]$$

$$\text{Avec} \quad \begin{array}{rcl} O & = & O_1 \cup O_2 \\ \mathcal{R} & = & H(\mathbf{R}_{(Q,q_{init},q_d,T,X,\mathcal{T})}(r_1, \ldots, init(R_c), \ldots, r_n)) \end{array}$$

$$\frac{h_{q_c} \notin {m_h}^+ \vee q_c \notin X \qquad (q_c, m_e[In]/O_1, q_d) \in T \qquad r_c \xrightarrow{m_h[H_c], m_e[I_c]/O_2, \mathcal{R}_2} r_c'}{\mathbf{R}_{(Q,q_{init},q_c,T,X,\mathcal{T})}(r_1, \ldots, r_n) \xrightarrow{m_h, m_e/O, \mathcal{R}} \mathbf{R}_{(Q,q_{init},q_d,T,X,\mathcal{T})}(r_1, \ldots, init(R_c), \ldots, r_n)} \quad [\mathrm{Raff2}^t]$$

$$\text{Avec} \quad \begin{array}{rcl} O & = & O_1 \cup O_2 \\ \mathcal{R} & = & H(\mathbf{R}_{(Q,q_{init},q_d,T,X,\mathcal{T})}(r_1, \ldots, init(R_c), \ldots, r_n)) \end{array}$$

$$\frac{h_{q_c} \notin {m_h}^+ \vee q_c \notin X \qquad \forall O_1, q_d,\ \nexists (q_c, m_e[In]/O_1, q_d) \in T \qquad r_c \xrightarrow{m_h[H_c], m_e[I_c]/O_2, \mathcal{R}_2} r_c'}{\mathbf{R}_{(Q,q_{init},q_c,T,X,\mathcal{T})}(r_1, \ldots, r_n) \xrightarrow{m_h, m_e/O_2, \mathcal{R}_2} \mathbf{R}_{(Q,q_{init},q_c,T,X,\mathcal{T})}(r_1, \ldots, r_c', \ldots, r_n)} \quad [\mathrm{Raff3}^t]$$

Pour que ces règles soient complètement définies, il faut prévoir le cas où un des sous-programme est égal à nil. D'où la règle :

$$\frac{}{nil \xrightarrow{\emptyset, \emptyset/\emptyset, \emptyset} nil} \quad [\mathrm{Nil}^t]$$

La règle [Raff1^t] s'applique lorsque les deux conditions suivantes sont réunies :

- L'état courant de l'automate temporisé dans le sommet source est un état temporisé,
- La condition temporelle de l'arc calculé indique que cet arc est celui pris lorsque le délai associé à l'horloge de l'état courant expire.

C'est l'appartenance de h_{q_c} à m_h qui détermine si ces deux conditions sont satisfaites ou non. Lorsqu'elles ne le sont pas, les règles [Raff2^t] et [Raff3^t] correspondent aux deux cas classiques distingués pour l'opérateur de raffinement : évolution de l'automate de contrôle et du sous-programme raffinant l'état courant (cf. [Raff2^t]) et évolution du sous-programme raffinant seulement (cf. [Raff3^t]).

Dans les deux premières règles ([Raff1^t] et [Raff2^t]), l'état courant de l'automate de contrôle est susceptible d'être modifié. Le sous-programme raffinant l'état quitté par cet automate est remis dans son état initial. Les horloges à réinitialiser sont celles qui sont pertinentes dans le sommet but de l'arc.

De plus, dans le cas de la règle [Raff1^t], il est important de noter que l'existence de la transition d'expiration issue de l'état temporisé q_c est assurée par la sémantique statique des programmes Argos temporisés.

La déclaration d'événements internes

Soient $I = \mathcal{I}(prog(s))$ et $H = H(s)$;

$$\frac{\begin{array}{l}\forall m_e^Y \in \mathcal{M}_c^\star(Y \cap I),\, s \xrightarrow{m_e \wedge m_e^Y, m_h[H]/O_i, R_i} s_i \\ \exists!\, m_e^Y \in \mathcal{M}_c^\star(Y \cap I) \text{ tel que } s \xrightarrow{m_e \wedge m_e^Y, m_h[H]/O_i, \mathcal{R}_i} s_i \quad {m_e^Y}^+ \subseteq O_i \quad {m_e^Y}^- \cap O_i = \emptyset\end{array}}{\overline{s^Y} \xrightarrow{m_e, m_h/O_i - Y, \mathcal{R}_i} \overline{s_i{}^Y}} \quad [\text{Int}^t]$$

La présence des horloges ne modifie pas le principe de calcul du statut des événements internes.

Remarque 6.1 *La détection des problèmes de causalité devient approchée, en ce sens que si un problème de causalité est issu d'un sommet du graphe qui, de par les conditions sur les horloges, est inaccessible, ce programme bien que correct, est considéré comme erroné par la fonction sémantique.* ■

6.2 Bonne temporisation des graphes temporisés décrits en Argos

Nous avons vu qu'un graphe temporisé peut ne pas être bien temporisé. C'est le cas si le système temporisé modèle du graphe contient un état à partir duquel le temps ne peut plus progresser. Ces graphes sont à détecter car ils ne peuvent pas modéliser des systèmes réels. D'autre part, ils sont source de problèmes lorsqu'on souhaite vérifier la satisfaction d'une propriété par le graphe.

Il est donc important de savoir s'il existe des programmes Argos temporisés qui conduisent à des graphes non correctement temporisés. Si l'examen de cette question montrait l'existence de tels programmes, il serait indispensable de définir un moyen de les distinguer, car ceux-ci peuvent être considérés comme le résultat d'erreurs de conception.

Dans [Nic92], il est montré qu'à partir de l'algèbre de processus temporisé ATP, de tels programmes existent. Une procédure de décision est définie pour déterminer si un programme conduit ou non à un système bien temporisé.

Nous allons montrer qu'à partir du langage Argos, de tels programmes n'existent pas.

Dans toute la suite, nous considérons uniquement les programmes Argos temporisés qui ne contiennent pas d'erreurs de causalité. Par conséquent, l'application de $\mathcal{S}^t$ sur ce type de programmes a toujours pour résultat un graphe temporisé.

Le plan suivi dans cette section est le suivant :

- Dans la section 6.2.1 le théorème à démontrer est formalisé. Des exemples montrent pourquoi ce résultat n'est pas immédiat ;
- Dans la section 6.2.2 deux lemmes utilisés pour prouver le théorème sont présentés et démontrés.
- Dans la 6.2.3 le schéma de la preuve du théorème est donné.

6.2.1 Théorème

Théorème 6.1 Les graphes temporisés décrits en Argos sont bien temporisés
Pour tout programme temporisé P^t, le graphe temporisés $\mathcal{S}^t(P^t)$ est bien temporisé

■

Les propriétés de réactivité et de déterminisme qui sont assurées sur les graphes modèles de programmes ArgosΓne suffisent pas pour conclure à la bonne temporisation des graphes temporisés. L'exemple de la figure 6.5 est un graphe temporisé déterministe et réactif qui a la forme d'un modèle de programme Argos — c'est un élément de $\mathcal{GT}^{\emptyset}_{dr}$— et qui n'est pas bien temporisé. En effetΓil est possible de rester dans le sommet 0 pendant deux unités de tempsΓpuis de passer dans le sommet 1 et attendre pendant cinq unités de temps. Le temps est alors bloqué : on ne peut ni rester dans le sommet 1Γni passer dans le sommet 0 à cause des invariants associés à ces sommets. Nous montrons par la suite que ce graphe temporisé ne peut pas être décrit par un programme Argos. La raison de cette impossibilité est la forme des arcs du graphe. En effetΓnous montrons que les arcs des graphes temporisés décrits en Argos satisfont les propriétés suivantes :

- Tout arc qui n'est pas une boucle sur un sommet est telΓque les horloges qui sont pertinentes dans le sommet but sans l'être dans le sommet source sont remises à zéro ;
- Tout arc qui est une boucle sur un sommet est telΓque toutes les horloges qui ont atteint leur borne lors du franchissement de cet arc sont remises à zéro.

Ces deux résultats font l'objet du premier lemme que nous présentons. Ils suffisent à éliminer le graphe précédentΓpuisque les arc qui font changer de sommet ne remettent pas à zéro les horloges pertinentes dans leur sommet but. De plusΓsi on modifie ces arcs de manière à satisfaire ces deux contraintesΓle graphe temporisé obtenu est bien temporisé.

6.2.2 Lemmes

Le premier lemme formalise les contraintes sur les arcs d'un graphe temporisés modèles de programmes Argos qui viennent d'être présentées. Le second porte sur les systèmes temporisés déduits des graphes temporisés modèles de programmes Argos. Deux propriétés sur ces systèmes peuvent être démontrées. Elles découlent des propriétés de déterminisme et de réactivité d'une partΓdes contraintes sur les horloges remises à zéro par les arcs — premier lemme — d'autre part.

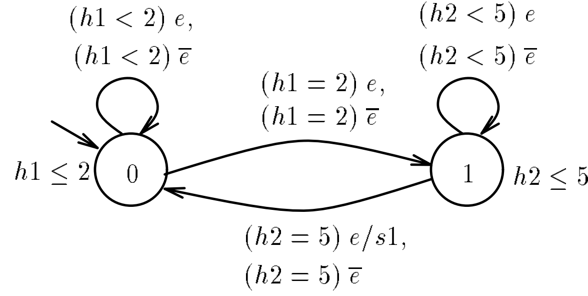


Figure 6.5: graphe temporisé déterministe et réactif mais non bien temporisé

Lemme 6.1 *Soient P^t un programme temporisé et $GT = (S, s_{init}, A, H, \mathcal{I}nv)$ le graphe temporisé modèle de P^t alors,*

$$\forall \ s_s \xrightarrow{m_h, m_e/O, \mathcal{R}} s_b \in A, (H(s_b) \setminus H(s_s) \subseteq \mathcal{R}) \wedge (s_s = s_b \implies {m_h}^+ \subseteq \mathcal{R}) \qquad \blacksquare$$

La preuve de ce lemme se fait par induction sur la structure du sommet source de l'arc. Nous développons uniquement les cas de la mise en parallèle et du raffinement. La déclaration d'événements internes se traite de façon similaire à la mise en parallèle. Le cas de base est donné par $s = nil$

Démonstration 6.1

- $s = nil$

  Le résultat est immédiat puisque $nil \xrightarrow{\emptyset,\emptyset/\emptyset,\emptyset} nil$

- $s_s = s_1 \| s_2$

  L'arc considéré est de la forme $s_1 \| s_2 \xrightarrow{m_h, m_e/0, \mathcal{R}} s'_1 \| s'_2$*, où* m_h *appartient à* $\mathcal{M}_c^\star(H(s_s))$. *D'après [Par^t], il est le résultat de la combinaison de deux arcs de la forme :*

  $s_i \xrightarrow{m_h[H(s_i)], m_e[\mathcal{I}(prog(s_i))]/Oi, \mathcal{R}_i} s'_i$*, avec* $i \in \{1, 2\}$.

  De plus, $\mathcal{R}$ *est l'union de* $\mathcal{R}_1$ *et* $\mathcal{R}_2$*. L'hypothèse d'induction appliquée à ces deux arcs nous permet d'écrire :*

  $(H(s_i) \setminus H(s'_i) \subseteq \mathcal{R}_i) \wedge (s_i = s'_i \implies m_h[H(s_i)]^+ \subseteq \mathcal{R}_i) \ \ \forall i \in \{1, 2\}$.

  Par définition de la fonction de calcul des horloges pertinentes, $H(s_1 \| s_2) = H(s_1) \cup H(s_2)$.

  En utilisant cette égalité, les hypothèses d'induction, plus l'égalité $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$*, nous obtenons le résultat souhaité.*

- $s_s = \mathbf{R}_{(Q, q_{init}, q_c, T, X, \mathcal{T})}(r_1, \ldots, r_n)$

  L'ensemble $\mathcal{R}$ *dans le cas du raffinement dépend de la règle utilisée (parmi les trois possibles) pour construire l'arc considéré. Si cette règle est [Raff1^t] ou [Raff2^t], le résultat est immédiat car :*

– *$\mathcal{R}$ est égal à $H(s_b)$, par conséquent $H(s_b) \setminus H(s_s) \subseteq \mathcal{R}$;*
– *De plus, si $s_s = s_b$ alors étant donné que m_h est construit sur $H(s_s)$, nous avons nécessairement ${m_h}^+ \subseteq \mathcal{R}$.*

Si la règle appliquée est [Raff3^t], $\mathcal{R}$ est l'ensemble $\mathcal{R}_2$ des horloges remises a zéro par l'arc $r_c \xrightarrow{m_h[H(r_c)], m_e[\mathcal{I}(prog(r_c))]/O_2, \mathcal{R}_2} r'_c$.

L'hypothèse d'induction appliquée à cet arc nous permet d'écrire :

$(H(r'_c) \setminus H(r_c) \subseteq \mathcal{R}_2) \wedge (r_c = r'_c \implies m_h[H(r_c)]^+ \subseteq \mathcal{R}_2)$.

En détaillant le calcul des horloges pertinentes dans s_s et s_b nous obtenons :

$H(s_s) = H(r_c) \cup (si\ (q_c \in X)\ alors\ h_{q_c}\ sinon\ \emptyset)$

$H(s_b) = H(r'_c) \cup (si\ (q_c \in X)\ alors\ h_{q_c}\ sinon\ \emptyset)$

Par conséquent, $H(s_b) \setminus H(s_s) = H(r'_c) \setminus H(r_c)$.

L'application de l'hypothèse d'induction et l'égalité $\mathcal{R} = \mathcal{R}_2$ nous permettent de conclure $H(s_b) \setminus H(s_s) \subseteq \mathcal{R}$.

Nous savons que $h_{q_c} \notin {m_h}^+$ (condition d'application de [Raff3^t]). Par conséquent, m_h étant un monôme construit sur $H(s_s)$, nous avons $m_h[H(r_c)]^+ = {m_h}^+$.

Puisque $s_s = s_b \implies r_c = r'_c$, nous pouvons appliquer la deuxième partie de l'hypothèse d'induction et conclure.

■

La première partie du lemme suivant met en évidence le fait que dans chaque état q du système temporiséΓpour chaque monôme d'événements m_eΓil existe une unique transition issue de q déclenchée par m_e. Cette première partie se déduit des propriétés de déterminisme et de réactivité des graphes temporisés modèles de programmes Argos.

La seconde partie montre que toutes les transitions instantanées du système vérifient les contraintes suivantes :

- Toute transition instantanée qui n'est pas une boucle sur un état est telle que la valuation des horloges dans l'état but diffère de celle dans l'état source par un ensemble d'horloges remises à zéro. Les horloges pertinentes dans le sommet associé à l'état but sans l'être dans celui associé à l'état but se trouve nécessairement dans cet ensemble ;
- Toute boucle instantanée sur un état est telleΓque chaque horloge qui est pertinente dans le sommet associé à l'étatΓet qui a atteint sa borneΓest remise à zéro.

Cette seconde partie se déduit du premier lemme.

Lemme 6.2 *Soient P^t un programme temporisé et $ST = (Q, q_{init}, T)$ le système temporisé $\mathcal{G}(\mathcal{S}^t(P^t))$;*

$$\forall (s_s, v_s) \in Q, m_e \in \mathcal{M}_c^\star(In),\ \exists!\ (s_s, v_s) \xrightarrow{m_e/O} (s_b, v_b) \in T$$
$$\wedge\ \forall (s_s, v_s) \xrightarrow{m_e/O} (s_b, v_b) \in T$$
$$(v_b = v_s[H := 0])\ \wedge\ (H(s_b) \setminus H(s_s) \subseteq H)\ \wedge\ (s_s = s_b\ \Rightarrow\ \{h_i \in H(s_s) \mid v(h_i) = d_i\} \subseteq H)$$

■

Démonstration 6.2
Soit $GT = (S, s_{init}, A, H, \mathcal{I}nv)$ le graphe temporisé égal à $\mathcal{S}^t(P^t)$.

Soient $(s_s, v_s) \in Q$ et $m_{h_{v_s}}$ le monôme égal à $\mathcal{M}_{s_s}(v_s)$. Nous rappelons que la fonction $\mathcal{M}_s$ (cf. définition 6.1) associe à une valuation qui satisfait l'invariant d'un sommet s, le monôme complet et non contradictoire sur les horloges pertinentes de s qui est satisfait par la valuation. GT étant déterministe et réactif, nous savons que :

$$\forall m_e \in \mathcal{M}_c^\star(In), \exists!\ s_s \xrightarrow{m_{h_{v_s}}, m_e/O, \mathcal{R}} s_b \in A$$

Cet arc remplit les conditions d'application de la règle [$\mathcal{G}$1] qui est la seule règle de construction des transitions instantanées. De plus, $v_b \models \mathcal{I}nv(s_b)$ car d'une part d'après le lemme 6.1, toutes les horloges pertinentes dans s_b sans l'être dans s_s sont remises à zéro par l'arc, elles ont donc la valeur 0 dans v_b, et par suite satisfont nécessairement les propositions de base qui les concernent dans $\mathcal{I}nv(s_b)$. D'autre part, les valeurs des autres horloges pertinentes dans s_b ne sont pas modifiées, par conséquent, puisqu'elles satisfaisaient $\mathcal{I}nv(s_s)$, elles satisfont aussi $\mathcal{I}nv(s_b)$. Nous pouvons donc déduire la première partie du lemme :

$$\forall m_e \in \mathcal{M}_c^\star(In), \exists!\ (s_s, v_s) \xrightarrow{m_e/O} (s_b, v_b) \in T.$$

De plus, toujours d'après la règle [$\mathcal{G}$1], $v_b = v_s[\mathcal{R} := 0]$. Par conséquent, $\mathcal{R}$ est un candidat possible pour jouer le rôle de l'ensemble H du lemme. Il remplit la condition $v_b = v_s[H := 0]$. D'après le lemme 6.1 $H(s_b) \setminus H(s_s) \subseteq H$ est aussi vérifiée. Si on considère en plus, la définition de $m_{h_{v_s}}$, nous posons l'égalité : $m_{h_{v_s}^+} = \{h_i \in H(s_s) | v_s(h_i) = d_i\}$, ce qui permet de déduire la dernière partie du lemme. ■

6.2.3 Les étapes de la démonstration du théorème 6.1

Tout d'abordΓrappelons qu'un graphe est bien temporisé si et seulement si le système temporisé qu'il définit est bien temporisé (cf. définition 5.9).

La démonstration que nous présentons est constructiveΓen ce sens que pour tout état du système temporisé nous exhibons une séquence d'exécution divergente issue de cet état.

Etant donnés un programme P^t quelconque et $ST = (Q, q_{init}, T)$ le système temporisé déduit de P^t(par application de $\mathcal{S}^t$ puis de $\mathcal{G}$)Γnous effectuons dans l'ordre indiqué les étapes suivantes :

(1) Définition d'une séquence de pas type entre états du système. Chaque pas est paramétré par un réel positif. Il est noté $(s, v) \overset{t}{\rightsquigarrow} (s', v')$. La séquence de pas est paramétrée par un état (s, v) du système ST— état initial de la séquence — et par un monôme d'événements m_e appartenant à $\mathcal{M}_c^\star(In)$ — monôme déclencheur des transitions instantanées de la séquence —. Nous notons cette séquence type $\sigma_{m_e}^{(s,v)}$. Etant donné un système temporisé déduit d'un

programme ArgosΓl'ensemble des séquences types construites en suivant cette définition est noté Γ_{ST}.

(2) Preuve de l'appartenance de cette séquence de pas à l'ensemble des séquences d'exécution possibles du systèmeΓquels que soient les paramètres de la séquence. Cela revient à prouver que chaque pas de la séquence est un pas d'exécution du systèmeΓou encore à montrer que $\Gamma_{ST} \subseteq \Sigma_{ST}$.

(3) Preuve de la divergence de cette séquence d'exécution. Cela revient à prouver que $\Gamma_{ST} \subseteq \Delta_{ST}$.

Ces trois étapesΓune fois démontréesΓprouvent bien que tout système temporisé obtenu à partir d'Argos est correctement temporisé. En effetΓtout préfixe fini du système peut être prolongé par une séquence typeΓdivergente en prenant pour premier état de la séquence le dernier état du préfixeΓet un monôme d'événements quelconque parmi $\mathcal{M}_c^\star(In)$.

Nous détaillons ici la première et la seconde étape. Nous donnons uniquement l'intuition de la troisièmeΓqui est détaillée dans l'annexe B.

Nous utilisons par la suite la notation suivante : pour tout sommet s d'un graphe temporisé tel que $\mathcal{I}nv(s) \neq true$Γ$\overline{\mathcal{I}nv(s)}$ désigne la condition obtenue en remplaçant les inégalités larges de $\mathcal{I}nv(s)$ par des inégalités strictes.

(1) Définition de $\sigma_{m_e}^{(s,v)}$

IntuitivementΓcette séquence type est construite de la manière suivante. Son premier état est donné par son paramètre (s, v). Les états suivants sont construits en fonction des états qui les précèdent. Dès qu'un état est formé d'un sommet dont l'invariant est $true$Γla séquence est une succession infinie d'incrémentation des horloges. Dans les autres cas la séquence est construiteΓsoit en laissant passer le temps jusqu'à ce qu'une des horloges ait atteint sa borne supérieure (cas 2 dans la définition ci-dessous)Γsoit s'il n'est plus possible d'augmenter la valeur des horlogesΓen effectuant une transition instantanée déclenchée par m_e(cas 3 dans la définition ci-dessous).

Définition 6.7 Séquence type

$\sigma_{m_e}^{(s,v)}$ *est de la forme* $(s_0, v_0) \overset{t_0}{\leadsto} (s_1, v_1) \overset{t_1}{\leadsto} \ldots \overset{t_{i-1}}{\leadsto} (s_i, v_i) \overset{t_i}{\leadsto} (s_{i+1}, v_{i+1}) \ldots$*, où* $t_i \in R^+$

Elle est définie par :

$(s_0, v_0) \;= (s, v)$

si $\mathcal{I}nv(s_i) = true$ **** cas 1 ****

alors

$t_i = 1$

$s_i = s_{i-1}$

$v_i = v_{i-1} + 1$

sinon *si* $v_i \models \overline{\mathcal{I}nv(s_i)}$ **** cas 2 ****

alors

$t_i = t_{min}$

$s_i = s_{i-1}$

$v_i = v_{i-1} + t_{min}$

avec $t_{min} = min(\{d_{i-1} - v_i(h_{i-1}) | h_{i-1} \in H(s_{i-1})\})$

sinon **** cas 3 ****

$t_i = 0$

$s_i = s'$

$v_i = v'$

avec $(s_{i-1}, v_{i-1}) \xrightarrow{m_e/O} (s', v') \in T$

■

Notons que dans le cas 3Γl'existence de la transition instantanée $(s_i, v_i) \xrightarrow{m_e/O} (s', v')$ est assurée par le lemme 6.2. De plusΓtous les états atteints par la séquence de pas appartiennent nécessairement à QΓcar quel que soit le cas utilisé pour calculer l'état suivantΓl'implication suivante est vérifiée : $\forall i \geq 0 \; v_{i-1} \models \mathcal{I}nv(s_{i-1}) \implies v_i \models \mathcal{I}nv(s_i)$. Puisque l'état initial de la séquence est par définition un état du système temporiséΓla valuation v_0 satisfait l'invariant de s_0Γce qui assure le résultat souhaité.

(2) Preuve de l'appartenance de la séquence type à l'ensemble des séquences d'exécution possibles du système

Il nous faut montrer que chaque pas de la séquence $\sigma_{m_e}^{(s,v)}$ définit un pas d'exécution du système. Cela revient à prouver que :

$$\forall i \in \mathbb{N}, (s_i, v_i) \overset{t_i}{\leadsto} (s_{i+1}, v_{i+1}) \implies (s_i, v_i) \overset{t_i}{\triangleright} (s_{i+1}, v_{i+1}).$$

cas 1 : $\mathcal{I}nv(s_i) = true$

Dans ce casΓle pas défini est $(s_i, v_i) \overset{1}{\leadsto} (s_i, v_i + 1)$.

$\forall v_i, (s_i, v_i) \xrightarrow{1} (s_i, v_i + 1) \in T$ (par application de $[\mathcal{G}2]$).

Par conséquentΓ$\forall v_i, (s_i, v_i) \overset{1}{\longmapsto} (s_i, v_i + 1)$. De plus $\forall (s, v) \in Q, (s, v) \overset{0}{\longmapsto} (s, v)$.

Nous en concluons que $(s_i, v_i) \overset{1}{\triangleright} (s_i, v_i + 1)$Γpuisque

$(s_i, v_i) \overset{1}{\longmapsto} (s_i, v_i + 1)$ et $(s_i, v_i + 1) \overset{0}{\longmapsto} (s_i, v_i + 1)$.

cas 2 : $\mathcal{I}nv(s_i) \neq true \land v_i \models \overline{\mathcal{I}nv(s_i)}$

Dans ce casΓle pas défini est $(s_i, v_i) \overset{t_{min}}{\leadsto} (s_i, v_i + t_{min})$Γoù $t_{min} = min(\{d_i - v_i(h_i) | h_i \in H(s_i)\})$.

OrΓ$(s_i, v_i) \xrightarrow{t_{min}} (s_i, v_i + t_{min}) \in T$ puisque $[\mathcal{G}2]$ peut être appliquée. En effetΓpar définition de t_{min} nous sommes sûrs que $v_i + t_{min} \models \mathcal{I}nv(s_i)$.

Par conséquentΓ$\forall v_i$ telle que $v_i \models \overline{\mathcal{I}nv(s_i)}, (s_i, v_i) \xmapsto{t_{min}} (s_i, v_i + t_{min})$.

Nous en concluons que $(s_i, v_i) \overset{t_{min}}{\triangleright} (s_i, v_i + t_{min})$Γpuisque

$(s_i, v_i) \xmapsto{t_{min}} (s_i, v_i + t_{min})$ et $(s_i, v_i + t_{min}) \xmapsto{0} (s_i, v_i + t_{min})$.

cas 3 : $\mathcal{I}nv(s_i) \neq true \wedge v_i \not\models \overline{\mathcal{I}nv(s_i)}$

Dans ce casΓle pas défini est $(s_i, v_i) \overset{0}{\rightsquigarrow} (s', v')$Γoù (s', v') est l'état but de la transition issue de (s_i, v_i) déclenchée par m_e.

Puisque cette transition fait toujours partie du système (cf. propriété 6.2)Γnous avons :

$(s_i, v_i) \xmapsto{0} (s', v')$.

Nous en concluons que $(s_i, v_i) \overset{0}{\triangleright} (s', v')$Γpuisque $(s_i, v_i) \xmapsto{0} (s_i, v_i)$ et $(s_i, v_i) \xmapsto{0} (s', v')$.

(3) Preuve de la divergence de toutes les séquences $\sigma_{m_e}^{(s,v)}$

Soient P^t un programme Argos temporisé et ST le système temporisé déduit de P^t (en passant par l'intermédiaire du graphe temporisé) ; nous montrons que $\Gamma_{ST} \subseteq \Delta_{ST}$. Soit Θ_{ST} le sous-ensemble de Γ_{ST} des séquences qui passent par un état (s_i, v_i) tel que l'invariant de s_i est $true$; l'inclusion de Θ_{ST} dans Δ_{ST} est immédiateΓpuisque tous les pas qui suivent l'état (s_i, v_i) ont une durée égale à 1.

Il nous reste à montrer que $(\Gamma_{ST} \setminus \Theta_{ST}) \subseteq \Delta_{ST}$. Tout d'abordΓnotons que si $(\Gamma_{ST} \setminus \Theta_{ST})$ est non videΓl'ensemble H des horloges du graphe temporisé défini par $\mathcal{S}^t(P^t)$ est nécessairement non vide.

Une séquence $\sigma_{m_e}^{(s,v)}$ de $\Gamma_{ST} \setminus \Theta_{ST}$ ne peut avoir qu'une des deux formes suivantes :

$$\sigma_{m_e}^{(s,v)} = (s_0, v_0) \overset{t_0}{\rightsquigarrow} (s_0, v_1) \overset{0}{\rightsquigarrow} (s_2, v_2) \overset{t_2}{\rightsquigarrow} \ldots (s_{2i}, v_{2i}) \overset{t_{2i}}{\rightsquigarrow} (s_{2i}, v_{2i+1}) \overset{0}{\rightsquigarrow} \ldots$$

ou

$$\sigma_{m_e}^{(s,v)} = (s_0, v_0) \overset{0}{\rightsquigarrow} (s_1, v_1) \overset{t_1}{\rightsquigarrow} (s_1, v_2) \overset{0}{\rightsquigarrow} \ldots (s_{2i}, v_{2i}) \overset{0}{\rightsquigarrow} (s_{2i}, v_{2i+1}) \overset{t_{2i+1}}{\rightsquigarrow} \ldots$$

En effetΓd'après les règles de construction d'une séquence type et le lemme 6.2Γles cas 2 et 3 alternent nécessairement. Le choix entre ces deux formes dépend uniquement de l'état initial de la séquence. La séquence est de la première forme si $v_0 \models \overline{\mathcal{I}nv(s_0)}$Γelle est de la deuxième forme sinon.

La preuve de la divergence des séquences types de cette forme est présentée dans l'annexe B. Elle est entièrement basée sur le résultat suivant : pour chaque pas de la séquenceΓil existe parmi les $|H| + 1$ pas de durée non nulle qui le suiventΓun pas de durée supérieure à $\frac{1}{|H|}$Γoù $|H|$ est le nombre d'horloges du graphe temporisé. La divergence de la séquence est une conséquence immédiate de ce résultat.

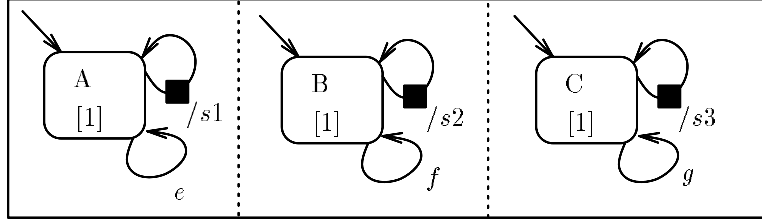


Figure 6.6: propriété des séquences

Considérons un exemple pour illustrer ce résultat. Soit le programme Argos donné par la figure 6.6. Le graphe temporisé modèle de ce programme admet un seul sommetΓdont l'invariant est la condition $h_A \leq 1 \wedge h_B \leq 1 \wedge h_C \leq 1$. Les états du système temporisé modèle de ce graphe sont caractérisés par les valeurs respectives des trois horloges. On les désigne par le triplet des valeurs de h_A, h_B et h_C.

La séquence de pas suivante est une séquence d'exécution du système.

$$(0;0;0) \overset{0,2}{\triangleright} (0,2;0,2;0,2) \overset{0}{\triangleright} (0,2;0;0,2) \overset{0,3}{\triangleright} (0,5;0,3;0,5) \overset{0}{\triangleright} (0;0,3;0,5) \overset{0,2}{\triangleright} (0,2;0,5;0,7)$$

Le deuxième (resp. quatrième) pas est constitué uniquement d'une transition instantanée (boucle sur l'état B (resp. A)).

L'état $(0,2;0,5;0,7)$ est donc un état accessible du système temporisé. Appliquons à partir de cet état la séquence typeΓnous avons :

$$(0,2;0,5;0,7) \overset{0,3}{\rightsquigarrow} (0,5;0,8;1) \overset{0}{\rightsquigarrow} (0,5;0,8;0) \overset{0,2}{\rightsquigarrow} (0,7;1;0,2) \overset{0}{\rightsquigarrow} (0,7;0;0,2) \overset{0,3}{\rightsquigarrow} (1;0,3;0,5) \overset{0}{\rightsquigarrow}$$

$$(0;0,3;0,5) \overset{0,5}{\rightsquigarrow} (0,5;0,8;1) \overset{0}{\rightsquigarrow} (0,5;0,8;0) \overset{0,2}{\rightsquigarrow} (0,7;1;0,2) \ldots$$

L'ensemble des quatre pas de durée non nulle qui suivent le premier pas est formé de {0,2; 0,3; 0,5; 0,2}. Dans cet ensembleΓse trouve un élément plus grand que $\frac{1}{3}$. Nous prouvons dans l'annexe B qu'il ne peut pas en être autrement.

Cette preuve est basée sur le fait que dans l'ensemble des $|H|+1$ pas de durée non nulle qui suivent le pas courant se trouvent nécessairement un pas obtenu en laissant augmenter jusqu'à sa borne une horloge — cas 3 dans la séquence type — déjà remise à zéro — cas 2 dans la séquence type — par les pas de cet ensemble. Cette horloge ayant déjà été remise à zéroΓla distance entre sa valeur courante et sa borne est nécessairement plus grande que $\frac{1}{|H|}$Γd'où le résultat souhaité.

6.3 Vérification en temps continu et exécution en temps discret

6.3.1 Contexte et présentation de la problématique

La figure 6.7 sert de support à la présentation qui suit. Nous venons de montrer comment exprimer la sémantique du langage synchrone Argos en termes de graphes temporisés. La sémantique des graphes temporisés est elle même exprimée en termes de systèmes temporisés.

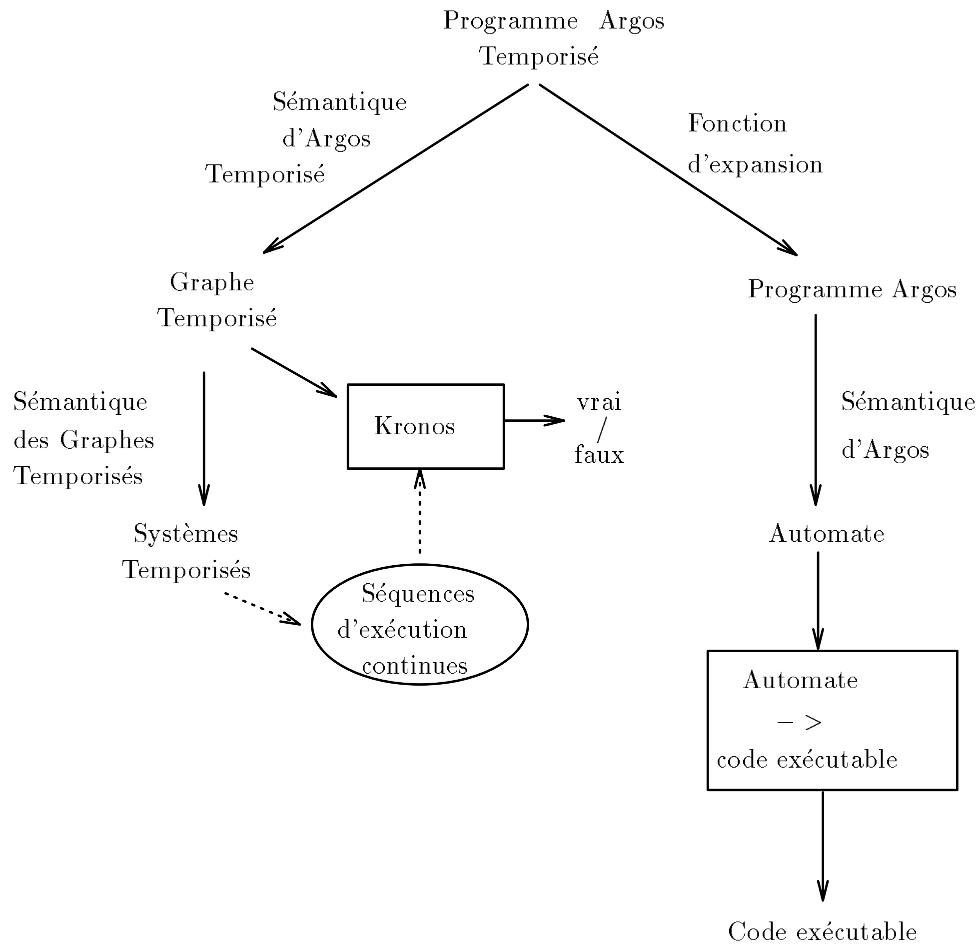


Figure 6.7: contexte d'étude

Sur ces systèmes la notion de séquences d'exécution est définie. Ces séquences sont interprétées sur un domaine temporel continu. La sémantique des formules de la logique TCTL est définie sur ces séquences continues. L'algorithme de "model-checking" — implémenté par exemple par KRONOS — opère sur le graphe temporisé en se référant de façon implicite à l'ensemble des séquences continues du système temporisé modèle du graphe.

Cette approche est adéquate dans un processus de modélisation dans lequel aucun objectif d'exécution n'est présent. Dans ce casΓles délais exprimés à l'aide des états temporisés du programme Argos sont réellement des durées et non des distances entre occurrences (cf. section 3.2.4).

Si à présentΓon est aussi intéressé par la production à partir du programme temporiséΓd'un code exécutableΓalors il est indispensable de s'interroger sur la validité des vérifications effectuées sur un modèle continu. La solution la plus naturelle pour obtenir ce code exécutable consiste à expanser la macro-notation des états temporisés puis à appliquer la sémantique d'Argos sur le programme obtenu et enfin à appliquer un traducteur d'automate en code exécutable. Avec cette solutionΓle principe du "WYPIWYE" (What You Prove Is What You Execute) [Ber89] est loin d'être respecté (cf. figure 6.7).

Une autre solution consiste à définir une notion d'exécution des systèmes temporisés et à étudier la préservation des propriétés effectuées sur le modèle continu par rapport à cette notion d'exécution. C'est cette démarche que nous présentons à présent.

Elle utilise une partie des résultats présentés dans [Nic92]. Les résultats de préservation obtenus ne sont intéressants que si la notion d'exécution attachée aux systèmes temporisésΓ coïncide avec l'exécution souhaitée d'un programme temporisé. Cette exécution souhaitée est entièrement définie par la fonction d'expansion des états temporisés. Il nous faut donc comparer la notion d'exécution définie à partir des systèmes temporisésΓet pour laquelle des résultats de préservation existentΓà la référence obtenue par expansion de la macro-notation des états temporisés.

6.3.2 Discrétisation des séquences d'exécution d'un système temporisé

Nous avons défini en 5.4 la notion de pas d'exécution d'un système temporisé. Nous en donnons à présent une définition équivalente qui évite l'intermédiaire de la relation $\longmapsto$.

Définition 6.8 Pas d'exécution d'un système temporisé (nouvelle définition)
Soit $ST = (Q, q_{init}, T)$ un système temporisé, pour tout $q, q' \in Q$ et $t \in \mathbb{R}^+$, on définit :

si $t > 0$ alors $q \overset{t}{\triangleright} q'$ si et seulement si $(q, t, q') \in T$ ou $(q, t, q'') \in T \wedge (q'', l, q') \in T$

si $t = 0$ alors $q \overset{0}{\triangleright} q'$ si et seulement si $(q, l, q') \in T$ ou $q' = q$ ∎

Cette définition et celle du support d'une séquence (cf. définition 5.6) peuvent être modifiées de manière à obtenir une notion de séquence d'exécution discrète. Il suffit pour cela de remplacer toutes les conditions d'appartenance de t à $\mathbb{R}^+$ par des conditions d'appartenance de t à $\mathbb{N}$.

Le résultat de préservation obtenu dans ce cadre (cf. [Nic92]) est le suivant : toute propriété de sûreté satisfaite sur l'ensemble des séquences continues est aussi satisfaite sur l'ensemble des séquences discrètes.

Ce résultat est lié au faitΓ d'une part que l'ensemble des séquences continues est un sur-ensemble de l'ensemble des séquences discrètes ; d'autre part que les propriétés de sûreté sont des propriétés quiΓpour être satisfaitesΓdoivent l'être pour toutes les séquences d'exécution de l'ensemble considéré. Par conséquentΓen vérifiant une propriété de sûreté sur l'ensemble des séquences continuesΓon la vérifie aussi sur l'ensemble des séquences discrètes.

Par ailleursΓil semble possible d'adapter l'algorithme d'évaluation d'une formule de TCTL de manière à ce qu'il prenne en compte uniquement les séquences d'exécution discrètes. Cette adaptation permettrait de prouver les propriétés — de sûreté et de vivacité — directement sur l'ensemble des séquences d'exécution discrètesΓet fournirait une notion de satisfaction en temps discret d'une formule. De tels outils de preuve n'existent pas actuellement.

Cette notion de séquence d'exécution discrète n'a un intérêt pour notre problématique que s'il est possible de la comparer avec l'exécution d'un programme temporisé telle qu'elle est définie par la fonction d'expansion. C'est à cet objectif de comparaison que nous cherchons à répondre à présent.

Parmi l'ensemble des séquences d'exécution discrètesΓseules celles paramétrées par des durées nulles ou égales à 1 sont réellement intéressantesΓcar tout pas de durée supérieure à 1 se décompose en une succession de pas de durée 1. Pour cette raisonΓnous considérons dans la suite uniquement les séquences d'exécution discrètes telles que chaque pas est de durée nulle ou égale à 1.

Cet ensemble peut être représenté par un automate dont la relation de transition modélise un pas d'exécution. Nous notons $A^{\triangleright}_{ST}$ l'automate qui contient toutes les séquences d'exécution discrètes d'un système temporisé. L'ensemble des états de cet automate est infini.

Etant donnés un programme Argos temporisé P^t et ST le système temporisé modèle de P^t (par l'intermédiaire du graphe temporisé) ; une séquence d'exécution discrète de ST peut être interprétée comme une abstraction d'une exécution de P^t. Cette abstraction ne retient à chaque réaction du programme que la présence (pas de durée 1) ou l'absence (pas de durée 0) de l'événement χ qui marque l'écoulement d'une unité de temps dans l'environnement.

Considérons un exemple. La figure 6.8 contient :

- Un programme Argos temporisé en `(a)` ;
- Le programme Argos obtenu par expansion de l'état temporisé en `(b)` ;
- L'automate modèle de `(b)` (les boucles sans sortie sur les états ne sont pas représentées) en `(c)` ;
- Le graphe temporisé modèle du programme temporisé en `(d)`.

Prenons l'exécution du programme (c'est un chemin dans l'automate `(c)`) suivante :

$(A, 0) \xrightarrow{a.\overline{\chi}} (B, 0) \xrightarrow{\chi.\overline{a}} (B, 1) \xrightarrow{\chi.\overline{a}} (B, 2)$

Cette exécution s'abstrait dans la séquence de pas discrets suivante — on note (s, i) un état du système où i est la valeur de l'horloge h_B — :

$(A, 0) \overset{0}{\triangleright} (B, 0) \overset{1}{\triangleright} (B, 1) \overset{1}{\triangleright} (B, 2)$

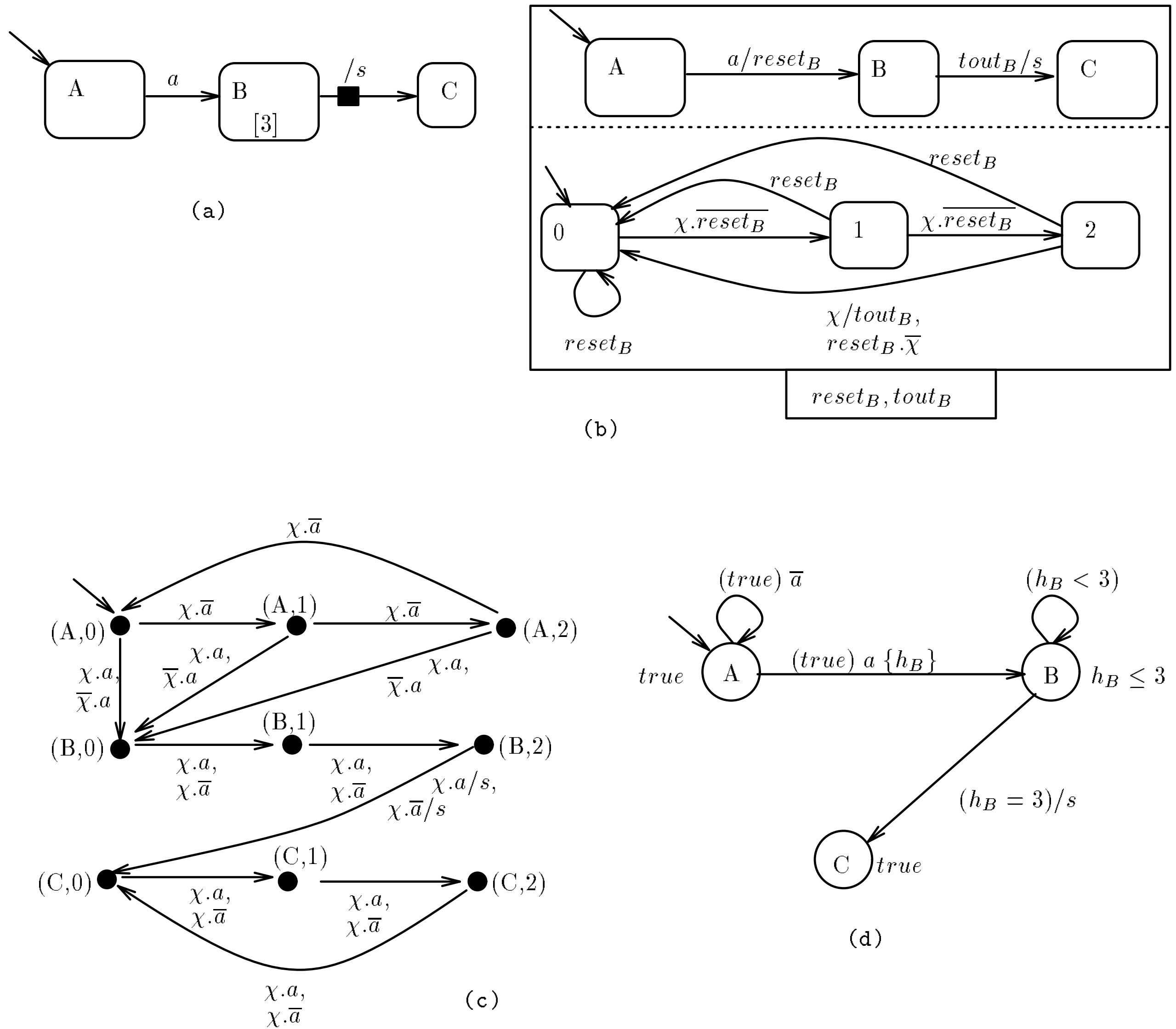


Figure 6.8: exemple complet

D'après la sémantique d'un graphe temporisé en termes de système temporisé (rappelée ci-dessous)Γcette séquence est une séquence d'exécution discrète du modèle du graphe temporisé de la figure 6.8(d).

Définition 6.9 Sémantique des graphes temporisés
Le système temporisé défini par un graphe temporisé $GT = (S, s_{init}, A, H, \mathcal{I}nv)$ *est le tuple* $\mathcal{G}(GT) = (Q, q_{init}, T)$*, où :*

- $Q = \{(s, v)\ s \in S, v \in Val \mid v \models \mathcal{I}nv(s)\}$
- $q_{init} = (s_{init}, \vec{0})$
- T *est la relation de transition incluse dans* $Q \times \mathcal{L} \cup \mathbb{R}^+ \times Q$ *définie par les règles suivantes (on note* $q \xrightarrow{l} q'$ *un élément de* T*) :*

$$\frac{(s_s, \psi, l, \mathcal{R}, s_b) \in A,\ v \models \psi}{(s_s, v) \xrightarrow{l} (s_b, v[\mathcal{R} := 0])} \quad [\mathcal{G}1] \qquad \frac{v + t \models \mathcal{I}nv(s)}{(s, v) \xrightarrow{t} (s, v + t)} \quad [\mathcal{G}2]$$

■

Certaines séquences d'exécution discrètes du modèle du graphe ne correspondent à aucune exécution du programmeΓc'est le cas par exemple de :

$(A, 0) \overset{0}{\triangleright} (B, 0) \overset{1}{\triangleright} (B, 1) \overset{1}{\triangleright} (B, 2) \overset{1}{\triangleright} (B, 3) \overset{0}{\triangleright} (C, 3)$

La recherche d'un résultat d'inclusion ne nous permettrait pas d'établir de résultat de préservation. Le problème vient de l'abstraction des étiquettes dans la séquence de pas discrets. En effetΓil ne faut pas oublier que la sémantique d'une formule de TCTL est paramétrée par la fonction d'interprétation des prédicats sur les étatsΓqui elle-même dépend de prédicats d'observation des étiquettes (de type `after` ou `enable`). Les étiquettes ayant disparu des séquences d'exécution discrètes du système temporiséΓaucun résultat de préservation ne peut être établi.

Pour remédier à ce problèmeΓnous définissons une notion de pas d'exécution concret qui conserve toute l'information sur les étiquettes.

6.3.3 Pas d'exécution concret et résultat de préservation

Nous donnons cette définition pour les systèmes temporisés modèles de graphes temporisés produits à partir d'Argos. Par conséquent les étiquettes des transitions instantanées du système sont de la forme m_e/O où m_e est un monôme complet et non contradictoire sur l'ensemble des entrées du programme (excepté χ). Dans la définition qui suitΓon note χ le monôme $(\{\chi\}, \emptyset)$ (seul χ est présent)Γet $\overline{\chi}$ le monôme $(\emptyset, \{\chi\})$ (seul χ est absent). Cette définition est inspirée de la définition 6.8 d'un pas d'exécution.

Définition 6.10 Pas concret d'exécution d'un système temporisé
Soit $ST = (Q, q_{init}, T)$ *un système temporisé. Pour tout* $q, q' \in Q$*, on définit :*

$q \overset{\chi \bullet m_e/O}{\triangleright_c} q'$ *si seulement si*

- $(q, 1, q') \in T \wedge O = \emptyset \wedge m_e \in \mathcal{M}_c^{\star}(In) \qquad (i)$

- *ou* $(q, 1, q'') \in T \wedge (q'', m_e/O, q') \in T$ (ii)

$q \overset{\overline{\chi}\bullet m_e/O}{\triangleright_c} q'$ *si et seulement si*

- $(q, m_e/O, q') \in T$ (iii)
- *ou* $q' = q \wedge O = \emptyset \wedge m_e \in \mathcal{M}_c^\star(In)$ (iv)

■

- Le cas (i) correspond à un pas d'exécution dans lequel seule l'étape de progression du temps — de une unité — est effectuée. Un tel pas peut se faire quel que soit le statut des entrées dans l'environnement du système ;
- Le cas (ii) correspond à un pas d'exécution en deux étapes : progression du temps d'une unité puis déclenchement d'une transition instantanée ;
- Le cas (iii) correspond à un pas d'exécution dans lequel seule l'étape de déclenchement d'une transition instantanée est effectuée (c'est un pas de durée nulle) ;
- Le cas (iv) correspond à un pas d'exécution dans lequel aucune des deux étapes n'est effectuée.

Nous notons $A_{ST}^{\triangleright_c}$ l'automate qui modélise l'ensemble des séquences d'exécution concrètes d'un système temporisé ST.

Connaissant la forme particulière des graphes temporisés modèles de programmes ArgosΓla sémantique des graphes temporisés en termes de systèmes temporisésΓet la définition d'un pas concretΓil est possible de définir une sémantique des graphes temporisés directement en termes de pas concrets (sans passer par l'intermédiaire du système temporisé).

Définition 6.11 Sémantique des graphes temporisés en termes de pas concrets
L'automate $A_{ST}^{\triangleright_c}$ *qui synthétise toute les séquences d'exécution concrètes du système temporisé défini par un graphe temporisé* $GT = (S, s_{init}, A, H, \mathcal{I}nv)$ *est le tuple* $\mathcal{G}'(GT) = (Q, q_{init}, T)$*, où :*

- $Q = \{(s, v)\ s \in S, v \in \mathit{Val} \mid v \models \mathcal{I}nv(s)\}$
- $q_{init} = (s_{init}, \vec{0})$
- *T est la relation de transition définie par les règles suivantes :*

$$\frac{v + 1 \models \mathcal{I}nv(s), m_e \in \mathcal{M}_c^\star(In)}{(s, v) \overset{\chi.m_e}{\triangleright_c} (s, v + 1)} \qquad [\mathcal{G}1']$$

$$\frac{(s_s, m_h, m_e/O, \mathcal{R}, s_b) \in A, v + 1 \models \mathcal{I}nv(s), v + 1 \models m_h}{(s_s, v) \overset{\chi.m_e/O}{\triangleright_c} (s_b, v[\mathcal{R} := 0])} \qquad [\mathcal{G}2']$$

$$\frac{(s_s, m_h, m_e/O, \mathcal{R}, s_b) \in A, v \models m_h}{(s_s, v) \overset{\overline{\chi}.m_e/O}{\triangleright_c} (s_b, v[\mathcal{R} := 0])} \qquad [\mathcal{G}3']$$

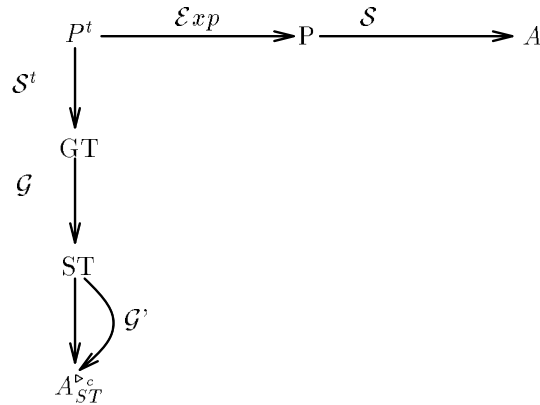


Figure 6.9: schéma des notations

$$\frac{m_e \in \mathcal{M}_c^\star(In)}{(s,v) \overset{\overline{\chi}.m_e}{\triangleright_c} (s,v)} \qquad [\mathcal{G}4']$$

■

Chacune de ces règles s'explique comme une combinaison de règles de sémantique des graphes temporisés en termes de systèmes temporisés et d'une règle de production d'un pas concret.

- La règle [$\mathcal{G}$1'] est la combinaison de [$\mathcal{G}$2] et (i).
- La règle [$\mathcal{G}$2'] est la combinaison de [$\mathcal{G}$2]Γ[$\mathcal{G}$1] et (ii).
- La règle [$\mathcal{G}$3'] est la combinaison de [$\mathcal{G}$1] et (iii).
- La règle [$\mathcal{G}$4'] s'obtient directement à partir de (iv).

La séquence suivante est un exemple de séquence d'exécution concrète déduite du système temporisé modèle du graphe temporisé de la figure 6.8(d).

$(A,0) \overset{a.\overline{\chi}}{\triangleright_c} (B,0) \overset{\chi.\overline{a}}{\triangleright_c} (B,1) \overset{\chi.\overline{a}}{\triangleright_c} (B,2) \overset{a.\chi/s}{\triangleright_c} (C,3)$Γelle correspond à la séquence d'exécution discrète $(A,0) \overset{0}{\triangleright} (B,0) \overset{1}{\triangleright} (B,1) \overset{1}{\triangleright} (B,2) \overset{1}{\triangleright} (C,3)$

De même pour $(A,0) \overset{a.\overline{\chi}}{\triangleright_c} (B,0) \overset{\chi.\overline{a}}{\triangleright_c} (B,1) \overset{\chi.\overline{a}}{\triangleright_c} (B,2) \overset{a.\chi}{\triangleright_c} (B,3) \overset{a.\overline{\chi}/s}{\triangleright_c} (C,3)$ qui correspond à

$(A,0) \overset{0}{\triangleright} (B,0) \overset{1}{\triangleright} (B,1) \overset{1}{\triangleright} (B,2) \overset{1}{\triangleright} (B,3) \overset{0}{\triangleright} (C,3)$

Il est possible à présent de comparer l'ensemble des exécutions d'un programme temporisé avec les séquences d'exécution concrètes du système temporisé associé au programme.

Théorème 6.2 *En suivant les notations de la figure 6.9, et en posant* $A = (Q, q_{init}, T)$ *et* $A_{ST}^{\triangleright_c} = (Q', q'_{init}, T')$*, nous avons*

si $q_{init} \xrightarrow{m_e{}^1/O^1} q_1 \xrightarrow{m_e{}^2/O^2} q_2 \xrightarrow{m_e{}^3/O^3} \dots \xrightarrow{m_e{}^i/O^i} q_i \dots$ *est une exécution de A alors*

$\exists q'_1, q'_2, q'_3, \dots q'_i \dots \in Q'$ *tels que* $q'_{init} \overset{m_e{}^1/O^1}{\triangleright_c} q'_1 \overset{m_e{}^2/O^2}{\triangleright_c} q'_2 \overset{m_e{}^3/O^3}{\triangleright_c} \dots \overset{m_e{}^i/O^i}{\triangleright_c} q'_i \dots$

est une exécution de $A^{\triangleright_c}_{ST}$ *(donc une séquence d'exécution concrète de ST).*

■

Cette propriété nous permet d'établir le résultat de préservation suivant : toute propriété de sûreté satisfaite sur l'ensemble des séquences d'exécution discrètes — avec des pas abstraits — l'est aussi sur l'ensemble des exécutions du programme.

Ce résultat est dû aux faits suivants :

- L'évaluation des prédicats d'étiquettes de type `after` et `enable` peut s'effectuer sur la séquence concrète. La propriété assure que tout prédicat d'étiquette satisfait sur un état q'_iΓ l'est aussi pour l'état q_i correspondant dans l'exécution du programme. De plusΓ l'abstraction des séquences d'exécution du système temporisé ne modifiant pas les étatsΓ l'implication ci-dessus est préservée.
- La propriété met en évidence une relation d'inclusion de l'ensemble des abstractions des exécutions du programme dans l'ensemble des exécutions discrètes du système temporisé.

Concrètement ce résultat de préservation signifieΓsi on lui adjoint le résultat de préservation de l'ensemble des séquences continues vers les séquences discrètesΓque toute propriété de sûreté satisfaite sur l'ensemble des séquences continues du système temporiséΓl'est aussi sur l'ensemble des exécutions du programme.

La preuve de la propriété 6.2 est immédiate si le lemme suivant est prouvée :

Lemme 6.3
En suivant les notations de la figure 6.9, et en posant $A = (Q, q_{init}, T)$ *et* $A^{\triangleright_c}_{ST} = (Q', q'_{init}, T')$, *nous avons*

$\exists \mathcal{R}el$ *incluse dans* $Q \times Q'$ *telle que :*

- $(q_{init}, q'_{init}) \in \mathcal{R}el$
- $\forall (q_1, q'_1) \in \mathcal{R}el \; q_1 \xrightarrow{m_e/O} q_2 \in T \implies q'_1 \overset{m_e/O}{\triangleright_c} q'_2 \in T'$ *avec* $(q_2, q'_2) \in \mathcal{R}el$

■

Pour prouver cette propriété nous exhibons une relation $\mathcal{R}el$ entre états de Q et Q' et nous prouvons qu'elle satisfait les deux conditions requises. Pour cela nous rappelons les formes des états de Q et de Q'.

L'ensemble Q contient les états de l'automate modèle du programme obtenu par expansion des états temporisés de P^t. Ce programme expansé est de la forme $\overline{P\|_{j\in\mathcal{X}(P^t)}C_j}^{Y}$Γoù P a une

structure identique à celle de P^t. Les éléments de Q sont donc de la forme : $\overline{p\|_{j\in\mathcal{X}(P^t)} c_j}^Y$Γoù p est un état de P. Rappelons que l'ensemble Y contient tous les événements internes de type $reset_i, tout_i$ rajoutés pour chaque état temporisé d'indice i du programme P^t.

L'ensemble Q' contient les états de l'automate qui synthétise toutes les séquences d'exécution discrètes du système temporisé ST. Ces éléments sont donc de la forme (s, v) où s est un état de P^t.

Les structures de P et P^t étant identiquesΓleurs états peuvent être comparésΓaux informations portées par les automates près — dans P^t les automates sont temporisés alors qu'ils ne le sont pas dans P —. Nous parlons dans la suite d'égalité entre un état s de P^t et un état p de P.

La définition de la relation $\mathcal{R}el$ entre Q et Q' nécessite de connaîtreΓétant donné un état d'un programme temporisé ArgosΓl'ensemble des états temporisés actifs dans cette configuration.

Définition 6.12 Ensemble des états temporisés actifs

$$
\begin{array}{lcl}
\mathcal{A}ctifs(s_1\|s_2) & = & \mathcal{A}ctifs(s_1) \cup \mathcal{A}ctifs(s_2) \\
\mathcal{A}ctifs(\mathbf{R}_{(Q,q_{init},q_c,T,X,\mathcal{T})}(r_1,\ldots r_n)) & = & (\{q_c\} \cap X) \cup \mathcal{A}ctifs(r_c) \\
\mathcal{A}ctifs(\overline{s^Y}) & = & \mathcal{A}ctifs(s) \\
\mathcal{A}ctifs(nil) & = & \emptyset
\end{array}
$$

■

Nous pouvons à présent définir la relation $\mathcal{R}el$. Informellement deux états q et q' sont mis en relation par $\mathcal{R}el$ si et seulement si d'une part ils représentent la même configuration dans P que dans P^t; et d'autre partΓpour chaque état temporisé actifΓles valeurs du compteur et de l'horloge associés à cet état coïncident.

Par exempleΓl'état (CΓ1) de l'automate de la figure 6.8`(c)` est mis en relation avec tous les états de $A_{ST}^{\triangleright c}$ de la forme (CΓv). En effet dans la configuration du programme dénotée par C aucun état temporisé n'est actifΓpar conséquent la valeur de l'unique horloge h_B est indifférente. L'état (BΓ2) estΓluiΓmis en relation avec l'état du même non dans le système temporisé.

Définition 6.13 La relation $\mathcal{R}el$

$(\overline{p\|_{j\in\mathcal{X}(P^t)} c_j}^Y, (s, v)) \in \mathcal{R}el$ *ssi :*

$s = p$

$\forall j \in \mathcal{A}ctifs(s) \;\; c_j = v(h_j)$ ■

L'appartenance du couple des états initiaux à l'ensemble des couples de la relation $\mathcal{R}el$ est immédiate : les états initiaux des compteurs sont tous égaux à 0 et la valuation initiale est celle qui associe 0 à chaque horloge.

Il nous reste à prouver l'implication entre les ensembles de transitions. Cette preuve se trouve dans l'annexe C.

6.4 Un exemple d'application à la robotique

L'exemple de vérification formelle que nous présentons est le résultat d'une collaboration avec l'action INRIA de l'unité de recherche Rhône-Alpes nommée BIPΓdirigée par Monsieur Bernard Espiau. Le thème de travail de cette équipe est le développement d'un environnement de CAO pour la programmation et la validation d'algorithmes de pilotage et/ou de contrôle-commande sur un ensemble de véhicules dans un environnement temps réel. Ce système se nomme ORC-CAD [SECK93ΓKap].

La pierre angulaire du système ORCCAD est le concept de *tâche-robot*. C'est l'association d'une loi de commande invariante et d'un comportement codé par un automate évoluant selon l'occurrence de signaux de synchronisation ou d'événements relatifs à des exceptions. La tâche-robot est décomposée en un ensemble de *tâches-modules* communicantes. Ces tâches-modules sont périodiques. On leur associe des propriétés temporelles : période d'activation et durée d'exécution. La communication entre deux taches modules s'effectue selon six protocoles prédéfinis qui se différencient par le caractère bloquant ou non des échanges.

Une tâche-robot est donc définie par les propriétés temporelles de ses tâches-modules et par les protocoles que celles-ci utilisent. Ces informations peuvent être visualisées sous forme d'un schémaΓappelé schéma de synchronisationΓcomme le montre la figure 6.10. La tâche-robot représentée est constituée de trois tâches-modules. Toutes les communications entre ces tâches sont de type *asynchrone-synchrone* ce qui signifie d'une part que le consommateur se bloque en attente du message ; d'autre part que le producteur actualise le message sans attendre la réponse du consommateur. La tâche-module nommée A a une période d'activation de 10 millisecondes et un temps d'exécution de 5 millisecondes.

Le fonctionnement attendu de cette tâche-robot est le suivant. A t_0 les trois tâches-modules sont activées mais seule A peut commencer son exécutionΓB et C étant bloquées en attente d'un message. Au bout de 5 millisecondes A se termine et envoie son message vers B qui se trouve débloquée et peut commencer à s'exécuter. A $t_0 + 7$ΓB termine son exécution et débloque C. A $t_0 + 8$, la tâche-module B est réactivée et se bloque de nouveau en attente du message en provenance de A. Les tâches-modules A et C sont ellesΓrespectivement réactivées aux instants $t_0 + 10$ et $t_0 + 20$.

Divers types de propriétés doivent être vérifiées au niveau de la tâche-robot dont l'absence de *blocage temporel*. Une tâche-robot contient un blocage temporel si la période d'activation d'une de ses tâches-modules n'est pas respectéeΓce qui signifie que la tâche-module est réactivée alors qu'elle n'a pas encore fini l'exécution associée à la période précédente.

L'exemple de tâche-robot donnée par la figure 6.10 contient un blocage temporel. En effetΓ comme montré par le diagramme temporel de la figure 6.11Γla seconde activation de B — à $t_0 + 16$ — survient alors que la tâche est encore en train de s'exécuter.

La vérification formelle de l'absence de blocage temporel à l'intérieur d'une tâche-robot est un des objectifs de la thèse d'Eduardo Castillo [Cas].

Une modélisation de la tâche-robot à l'aide du langage Argos temporisé constitue une solution pour répondre à ce problème. L'idée consiste à modéliser l'évolution de la tâche-robot par un programme Argos temporiséΓpour pouvoir vérifier sur le graphe temporisé ainsi décritΓune propriété exprimant l'absence de blocage.

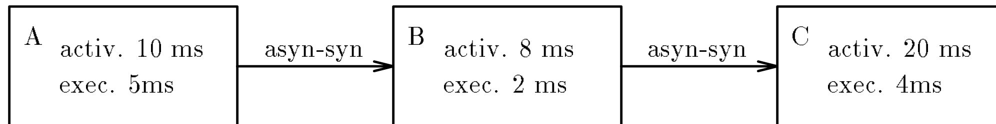


Figure 6.10: exemple de tâche-robot

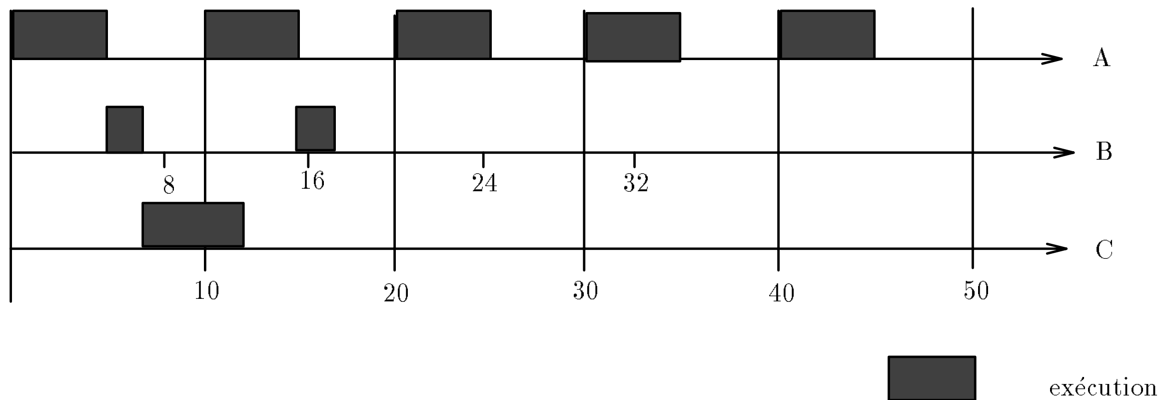


Figure 6.11: blocage temporel

Nous présentons dans la section 6.5 la modélisation en Argos de la tâche-robot de la figure 6.10Γpuis dans la section 6.6Γl'expression de la propriété d'absence de blocage par une formule TCTL. Pour terminer nous présentons dans la section 6.6.2 des extensions possibles à ce travail.

Dans le chapitre 7Γqui présente le travail d'expérimentation réalisé au cours de cette thèseΓ nous montrons comment le système KRONOS peut être utilisé pour vérifier cette propriété.

6.5 La modélisation en Argos d'une tâche-robot

La tâche-robot de la figure 6.10 est modélisée par trois composantes en parallèle. Chaque composante modélise le comportement d'une tâche-module. Elle s'obtient par la mise en parallèle :

- D'une composante chargée de réactiver la tâche-module à la fin de sa période d'activation : c'est un état temporisé par la durée d'une périodeΓqui émet à chaque expiration un ordre d'activation (cf. figure 6.12 pour la tâche-module A) ;
- D'une composante qui modélise les changements d'état de la tâche-module. Celle-ci peut être :
  - soit bloquée en attente d'un message (la tâche-module A ne se trouve jamais dans cet état là)Γ
  - soit en train de s'exécuterΓ
  - soitΓalors qu'elle a terminé son exécutionΓen attente d'un ordre de réactivation.

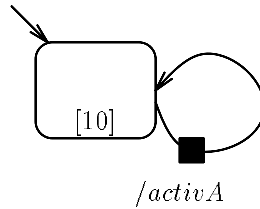


Figure 6.12: modélisation des périodes d'activation (cas de la tâche-module A)

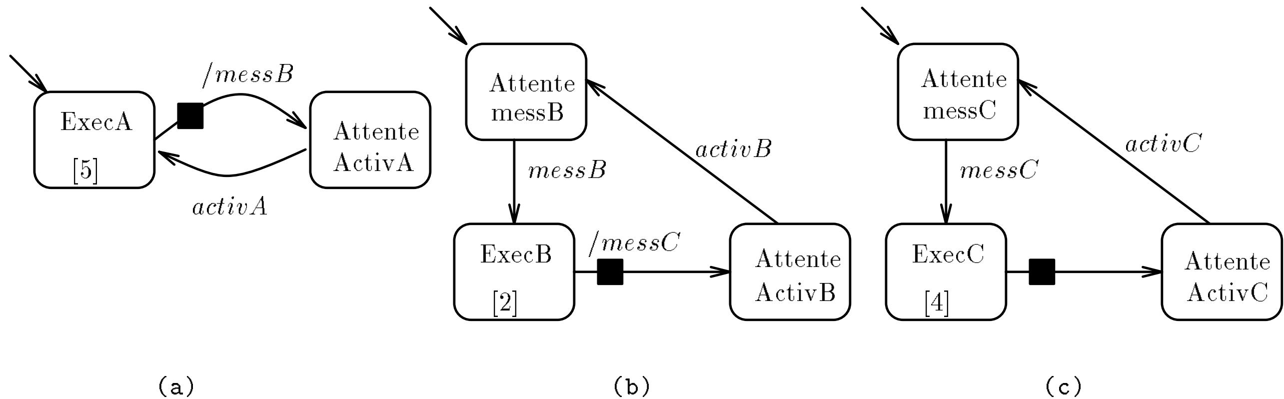


Figure 6.13: modélisation des changements d'état des tâches-modules

La figure 6.13 (`a`) modélise les changements d'état de A. Cette tâche-module commence par s'exécuter pendant cinq millisecondes puis produit son message et attend d'être réactivée. Les figures 6.13 (`b`) et (`c`) modélisent les changements d'état de B et C. Ces deux tâches-modules commencent par être bloquées en attente d'un messageΓpuis s'exécutent et se placent ensuite en attente d'être réactivées.

FinalementΓle programme Argos temporisé obtenu comme modèle de la tâche-robot est donné par la figure 6.14. On remarque que tous les événements qui apparaissent dans le programme sont déclarés internes. En effetΓ seule l'évolution du temps rythme le fonctionnement de la tâche-robot.

6.6 La vérification de l'absence de blocage temporel

6.6.1 Expression de la propriété par une formule TCTL

Une tâche-robot ne contient pas de blocage temporel si et seulement si chaque tâche-module qui la compose est réactivée alors qu'elle se trouve en attente d'être réactivée.

Pour pouvoir exprimer facilement la propriété en une formule TCTLΓnous ajoutons au programme ci-dessusΓ une composante par tâche-moduleΓ qui filtre l'ordre d'activation et ne le

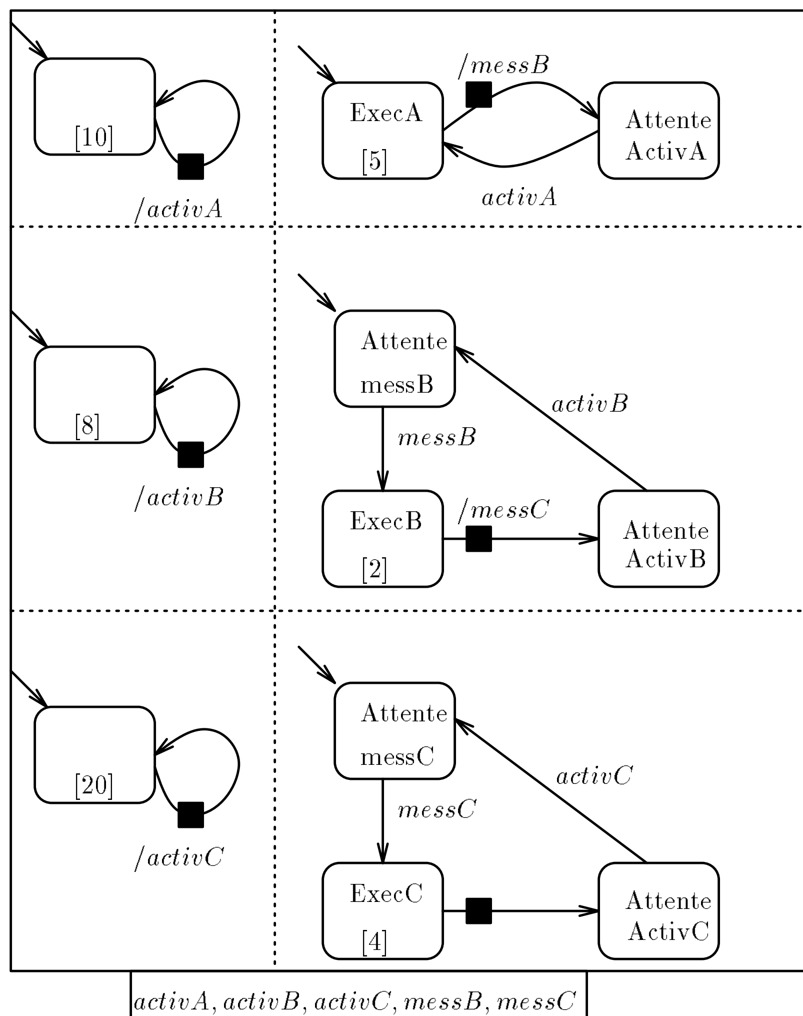


Figure 6.14: modélisation de la tâche-robot en Argos temporisé

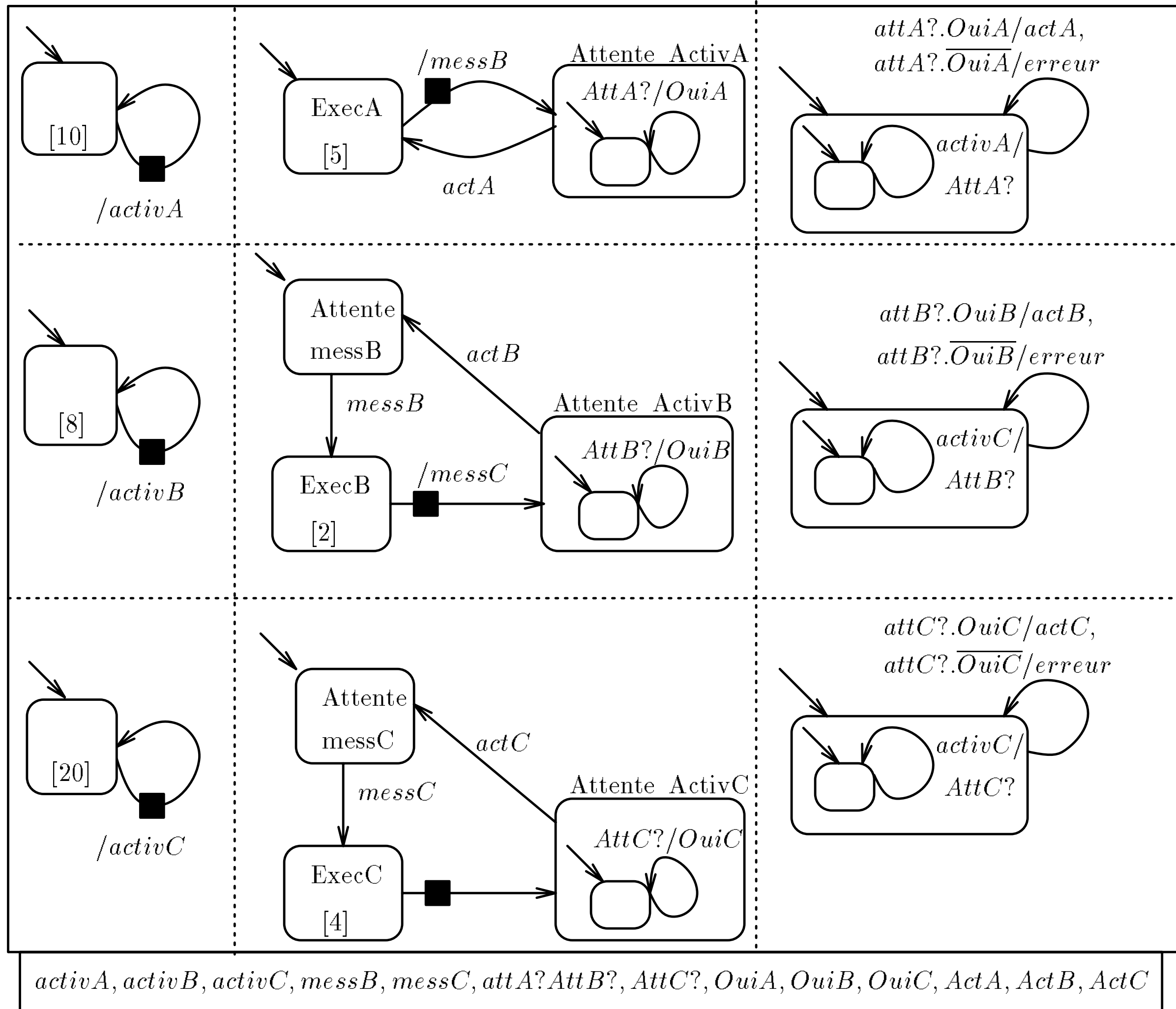


Figure 6.15: observation des blocages temporels

retransmet que si la tâche-module est dans l'état d'attente de cet ordre. Dans le cas contraireΓ un message d'erreur est généré qui peut être utilisé dans la formule TCTL.

Cette composante supplémentaire doit communiquer avec la composante qui modélise les changements d'état de la tâche-module : sur réception de l'ordre d'activationΓil interroge cette composante pour savoir si elle se trouve dans le bon étatΓsi c'est le casΓl'ordre d'activation est retransmisΓsinon un message d'erreur est généré. La figure 6.15 est la description du programme argos temporisé qui modélise la tâche-robotΓet qui génère une sortie d'erreur en présence d'un blocage temporel.

Ce programme peut ête compilé sous forme d'un graphe temporisé qui comprend 18 sommets et gère 6 horloges. La formule TCTL qui traduit l'absence de blocage temporel dans la tâche-robot est la suivante : $\forall\Box\neg\texttt{after}(\texttt{obs}(erreur))$. Elle exprime le fait queΓquelle que soit la séquence d'exécution issue de l'état initialΓaucun état rencontré le long de la séquence ne vérifie

la propriété d'être après une étiquette où la sortie d'erreur est émise.

6.6.2 Extensions

Automatisation de la génération du programme Argos temporisé modèle d'une tâche-robot

Nous venons de présenter un exemple de vérification formelle de l'absence de blocage temporel dans une tâche-robot. Cet exemple met en évidence l'intérêt d'une modélisation en Argos temporisé. CependantΓcet intérêt serait encore plus grandΓsi cette modélisation était automatique à partir des caractéristiques de la tâche-robot.

Cette extension est facilement envisageable dans le cas particulier des tâches-robots qui n'utilisent que des communications de type asynchrone-synchrone (ce sont les plus couramment utilisées). En effetΓla modélisation présentée ci-dessus est suffisamment structurelle pour que l'on puisse en dériver une méthode automatique.

D'autres types de synchronisation sont possibles dans le système ORCCADΓcomme par exemple :

- Synchrone-Synchrone : la tâche-module la plus rapide est bloquée jusqu'à ce que la plus lente soit prête.
- Asynchrone-Asynchrone : chaque tâche-module évolue librement et n'est jamais bloquée en attente d'une communication.
- Synchrone-Asynchrone : le producteur est bloqué jusqu'à que le consommateur demande un message. Le consommateur peut évoluer avec ou sans message.

Tous ces protocoles de communication ont été codés en Esterel [MSBB99]. Plus précisément chaque protocole est décomposable en deux parties : le producteur et le consommateur. Des producteurs et des consommateurs types sont codés en Esterel qu'il convient d'assembler selon une table de correspondance pour obtenir le protocole souhaité.

La même démarche est envisageable en Argos et constituerait un moyen de produire automatiquement à partir des caractéristiques de la tâche-robotΓle programme Argos temporisé qui la modélise. Il s'ensuit que la vérification formelle de l'absence de blocage temporel d'une tâche-robot devient complètement automatique et peut être intégrée dans le système ORCCAD.

Vérification formelle d'autres types de propriétés temporelles

Nous nous sommes intéressés dans l'exemple ci-dessus au problème du blocage temporel. D'autres propriétés temporelles à l'intérieur d'une tâche-robot s'avèrent intéressantes à prouver. Par exempleΓ pour définir une stratégie de répartition des tâches-modules sur une architecture multi-processeursΓil est intéressant de prouver que :

- deux tâches-modules n'ont aucun instant d'exécution en communΓ

- deux tâches-modules s'exécutent simultanément

Ces propriétés pourraient elles aussi être vérifiées à partir d'une modélisation de la tâche-robot en Argos temporisé. Il suffirait pour cela d'utiliser une composante d'observation adéquate pour mettre en évidence la propriété souhaitée.

6.7 Conclusion

Dans ce chapitre nous avons défini la sémantique d'Argos avec états temporisés en termes de graphes temporisés. Par conséquentΓce langage peut être utilisé à présent comme un moyen pour décrireΓdans un formalisme de haut niveauΓdes graphes temporisés complexes. L'intérêt de ce modèle réside dans les techniques de vérification formelle qui lui sont associées :

- Il est moins sensible au phénomène d'explosion du nombre d'étatsΓce qui repousse les limites pratiques d'utilisation des outils de vérification basés sur les automates booléens ;
- Il permet d'exprimer des propriétés quantitatives temporelles temps réel.

Le langage Argos est le premier des langages synchrones dont la sémantique est définie en termes de graphes temporisés. Le travail présenté dans cette thèse peut donc servir de modèle à des extensions similaires à partir d'autres langages synchrones.

Les graphes temporisés décrits à l'aide d'Argos ont une forme particulière liée à la sémantique des états temporisés. En étudiant l'introduction dans le langage Argos d'autres structures de temporisationΓd'autres formes de graphes pourraient être décrites.

Nous avons montré que les graphes temporisés qu'il est possible de décrire à l'aide d'Argos sont tous bien temporisésΓpropriété importante lorsqu'on souhaite effectuer des preuves formelles sur un graphe temporisé.

Nous avons aussi mis en évidence un résultat de préservation important lorsqu'on ne s'intéresse pas uniquement à la modélisation d'un graphe temporiséΓmais aussi à son exécution. En effetΓil faut s'interroger sur la cohérence de preuves faites avec une interprétation continue du tempsΓpar rapport à une exécution au cours de laquelle le temps est discret. Ce problème n'avait jusqu'à présent pas fait l'objet d'étude particulièreΓle modèle des graphes temporisés n'étant utilisé que pour spécifier des systèmes. Le résultat mis en évidence est que pour une certaine classe de propriétés — les propriétés de sûreté — leur satisfaction en temps continu est préservée lors d'une exécution en temps discret.

Chapitre 7

Réalisations et expérimentations

Nous présentons dans ce chapitre les réalisations et les expérimentations qui accompagnent le travail théorique des chapitres précédents.

Les réalisations concernent la compilation du langage Argos. Les aspects d'intégration dans un logiciel existant représentent une caractéristique importante du travail réalisé.

Les expérimentations portent sur trois outils de vérification, mettant en œuvre les trois méthodes auxquelles nous nous sommes intéressés : technique des observateurs, simulation comportementale et utilisation d'une logique temporelle. Ces expérimentations sont à l'origine des résultats présentés dans le chapitre 1, sur l'utilisation d'une certaine classe d'outils pour la vérification formelle des systèmes réactifs. Elles ont aussi donné lieu à des idées de modifications pour augmenter les fonctionnalités des outils, ou pour les rendre plus performants en tenant compte des particularités des modèles de programmes Argos.

7.1 Réalisations

Les réalisations que nous avons effectuées concernent la compilation du langage Argos. Nous sommes partis d'un algorithme énumératif directement inspiré de la définition d'Argos donnée dans la section 2.2. Nous avons amélioré les performances en temps du compilateur en mettant en œuvre un algorithme symbolique de compilation (cf. [MV92]). L'étude de la technique de compilation du langage Esterel, nous a permis d'améliorer encore ces performances en adaptant la technique des *potentiels* à la compilation du langage Argos.

Nous présentons tout d'abord la structure du compilateur Argos avant les différentes modifications que nous lui avons apportées.

7.1.1 Le compilateur Argos

Commençons par présenter l'environnement Argonaute — environnement du langage Argos — afin de mieux comprendre les interactions entre ses différentes composantes, dont le compilateur. Cet environnement est présenté par la figure 7.1, dans laquelle le compilateur apparaît dans un

cadre grisé. Un éditeur syntaxique d'Argos [Sch94] permet d'éditer un programme en intégrant des aspects graphiques — comme l'édition des automates — et des aspects textuels. Cet éditeur a été réalisé en utilisant le logiciel d'aide à la création d'éditeurs syntaxiques Grif [QV86]. Il est par ailleurs possible d'éditer un programme Argos à l'aide de n'importe quel éditeur textuelΓce langage possédant une syntaxe entièrement textuelle. Le format de cette syntaxe se nomme *targos*. L'éditeur syntaxique sauvegarde les programmes sous un format nommé *argos*Γqui contient à la fois des informations graphiques et sémantiques. Ce format est peu lisible et ne peut pas être utilisé pour l'édition des programmes Argos. Le compilateur est constitué :

- d'un traducteur du format *targos* vers le format *argos* ;
- d'un analyseur du format *argos* ;
- d'un analyseur sémantique ;
- d'un générateur d'automates.

Le compilateur Argos est entièrement écrit en CΓil représente environ 12000 lignes de code. Il produit des automates sous différents formats selon les objectifs du programmeur. Ceci permet d'utiliser dans Argonaute des outils développés dans d'autres environnements. Le *format OC* [Pa93] est un format d'automates interprétés qui sert essentiellement comme format intermédiaire pour produire du code exécutable. C'est le format de sortie des automates modèles de programmes EsterelΓLustre et Signal. Il existe différents traducteurs du format OC en code exécutable (C ou ADA). Ce format permet aussi d'utiliser SAHARA [Ghe92] un simulateur graphique de systèmes réactifsΓdéveloppé dans l'environnement Esterel. Il est aussi le format d'entrée de Oc2Rep [Gir94]Γrépartiteur de code sur une architecture multi-processeurs. Le *format SA* est un format d'automates qui permet d'utiliser les systèmes Autograph [Roy90] — interface graphique de visualisation d'automates — et Auto [dSV89ΓVer87] — outil de vérification formelle basé sur la réduction et la comparaison d'automates — tous deux développés dans l'environnement du langage Esterel. Le *format AUT* est un autre format d'automates qui permet d'utiliser Aldébaran [Fer88ΓMou92ΓKer]Γoutil de vérification formelle à base de réduction et de comparaison d'automates.

L'analyseur sémantique vérifie essentiellement le déterminisme des automates. Pour faciliter cette vérificationΓles automates sont saturésΓce qui signifie que les monômes des transitions sont complétés sur l'ensemble des entrées pertinentes pour chaque état de l'automate[1].

Le générateur d'automates met en œuvre l'algorithme 2.1. La fonction `CalculTrans` est implémentée par une fonction récursive quiΓen fonction du type de l'opérateur principal du programmeΓappelle une fonction spécifique. Nous détaillons l'algorithme de la fonction utilisée pour la déclaration d'événements internesΓcar c'est cette fonction qui subit par la suite les principales modifications. Nous l'appelons `CalculTransI`. Pour présenter cet algorithmeΓnous utilisons les mêmes notations que celles utilisées pour l'algorithme 2.1.

[1]par rapport à la définition formelle d'Argos présentée dans la section 2.2, les monômes manipulés ne sont pas complets sur toutes les entrées du programme. Cette différence ne porte pas à conséquence.

Algorithme 7.1 Le calcul de la transition issue de $\overline{p^Y}$ déclenchée par m_e

Sol_OK $\leftarrow$ 0 {nombre d'hypothèses cohérentes}
Pour chaque $m_Y \in \mathcal{M}_c^\star(Y)$ faire
    Si (CalculTrans($p, m_e \wedge m_Y$).resultat = OK) faire
        $O \leftarrow$ CalculTrans($p, m_e \wedge m_Y$).sorties
        Si $((m_Y^+ \subseteq O) \wedge (m_Y^- \cap O = \emptyset))$ faire
            Sol_OK $\leftarrow$ Sol_OK + 1
            $p_temp \leftarrow$ CalculTrans($p, m_e \wedge m_Y$).but
            $O_temp \leftarrow$ 0
    Sinon { détection d'un problème de causalité dans le sous-programme }
        Exit(erreur)
Si (Sol_OK = 1) faire
    $p_b \leftarrow \overline{p_temp^Y}$
    $O \leftarrow O_temp \setminus Y$
Sinon { détection d'un problème de causalité concernant un élément de Y }
    Exit(erreur)

■

Chaque monôme m_Y construit sur l'ensemble YΓexprime une hypothèse sur le statut des événements internes. On calcule la réaction du sous-programme opérande en prenant cette hypothèse pour les statuts des événements internes. On vérifie a posteriori la cohérence de cette hypothèse. Si c'est le casΓalors on incrémente Sol_OK et on mémorise la transition ainsi calculée. Lorsque toutes les hypothèses ont été effectuéesΓon vérifie l'existence d'une unique hypothèse cohérente. Dans un tel casΓla transition recherchée se déduit facilement de la transition mémorisée.

Le coût en temps de cet algorithme croît exponentiellement en fonction du nombre d'événements internes contenus dans l'ensemble YΓpuisqu'on essaie tous les monômes construits sur Y.

7.1.2 Algorithme symbolique de compilation du langage Argos

Les graphes de décision binairesΓou *bdd* pour *Binary Decision Diagrams*Γsont apparus récemment comme une solution novatrice — compacité et canonicité de la représentationΓet efficacité des opérateurs — pour le calcul booléen. Les bdd sont une représentation canonique des expressions booléennes sous forme de graphes de décision binaires orientés et acycliques. Cette représentation et les opérateurs associés résultent des travaux relatés dans [Sha38ΓAke78ΓBry86]. Shannon a défini une représentation canonique des fonctions booléennes modulo un ordre donné des variables — *arbre de Shannon* —. Ackers a défini la négation en temps constant mais la représentation n'était plus canonique. C'est Bryant qui a défini une nouvelle forme canonique plus compacte — élimination des nœuds redondants et partage des nœuds isomorphes — ainsi que le calcul des opérateurs booléens préservant cette canonicité. La figure 7.2 donne le bdd associé à la formule $(a \wedge b) \vee (c \wedge d)$ pour l'ordre a, b, c, d des variables.

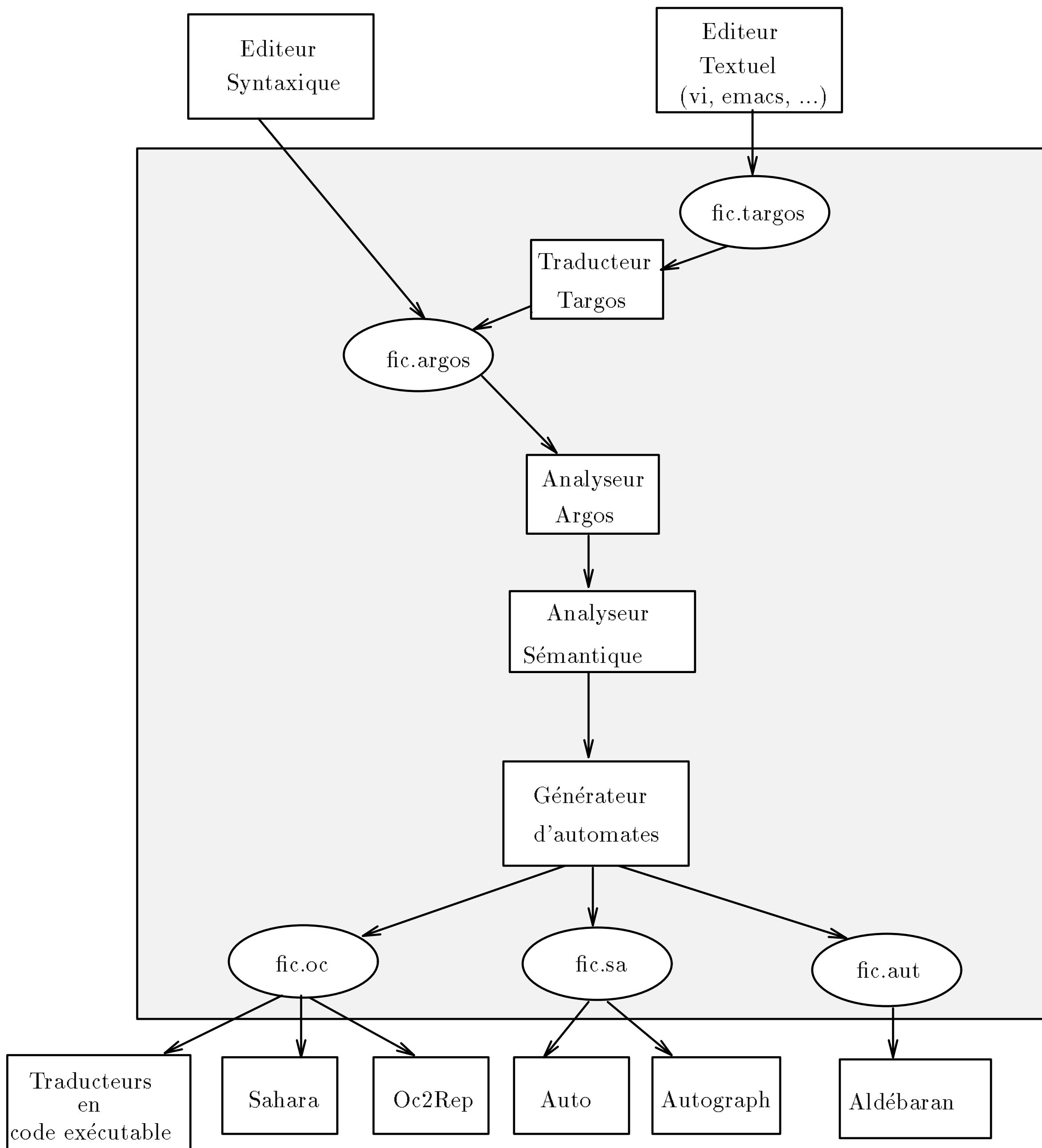


Figure 7.1: environnement Argonaute et compilateur Argos (en grisé)

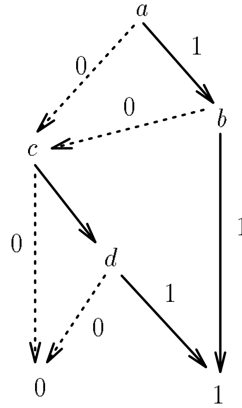


Figure 7.2: bdd représentant l'expression $(a \wedge b) \vee (c \wedge d)$

Les opérations effectuées sur les bdd se ramènent à des parcours de graphes binaires. Par conséquentΓle nombre de nœuds d'un bdd est une bonne mesure pour évaluer le coût de ces opérations. Le coût en temps des opérations booléennes $(\wedge, \vee, \Rightarrow)$ est proportionnel au produit des tailles des bdd opérandes. Les opérations de comparaison et de négation se font en temps constant. Le nombre maximum de nœuds d'un bdd est égal à $\frac{2^n}{n}$Γoù n est le nombre de variables de la fonction booléenne représentée. CependantΓdans la pratique la taille des bdd est polynômiale en nΓd'où l'intérêt de cette représentation. Une étude précise des coûts des opérations de manipulation de bdd peut être trouvée dans [Cou91]. Les travaux autour des bdd se sont multipliés depuis quelques années et actuellement plusieurs "package" sont développésΓpar exemple ceux de Berkeley [TLS$^+$90]Γde Carnegie Mellon University [BBR90] ou de Bull [CBM90ΓMad90]. Nous avons utilisé celui développé à Grenoble [Rat92].

Ce "package" est développé en C++. Nous nous servons des opérations booléennes $\wedge$ et $\vee$. A notre demandeΓune opération supplémentaire a été implémentée. Elle permetΓconnaissant le nombre de variables manipulées par la fonction booléenne représentée par un bddΓde calculer le nombre de solutions de cette fonction. Nous l'appelons `NbSol`.

Nous détaillons à présent comment la fonction `CalculTransI` a été modifiée de manière à tirer profit d'une représentation à l'aide de bdd.

Nous supposons dans un premier temps que le programme contient une seule déclaration d'événements internes.

Le nouvel algorithme de `CalculTransI` utilise les fonctions suivantes :

- `Construire_bdd` prend en paramètre un état global p et un ensemble d'événements Y. Le résultat est le bdd représentant la fonction booléenne égale à la conjonction de toutes les équations qui définissent le statut des événements de Y dans l'état global p. Pour obtenir ce résultatΓle bdd égal à la partie droite d'une équation concernant l'événement αΓs'obtient par un parcours de l'ensemble des transitions susceptibles d'être déclenchées — issues d'un état actif dans p —. A chaque fois que α fait partie des sorties de la transitionΓle monôme déclencheur de la transition est codé par un bdd. La disjonction de tous ces monômes

constitue le bdd égal à la partie droite. L'équation s'obtient en posant l'égalité entre cette partie droite et la variable α.

- `Solution` prend en paramètre un bdd qui admet nécessairement une unique solution. Le résultat est un monôme qui représente cette solution. Il s'obtient par un simple parcours du bdd.

Algorithme 7.2 Un nouvel algorithme pour `CalculTransI`(p, Y)

```
bdd_res ← Construire_bdd(p,Y)        { Construction du bdd équivalent au système}
bdd_res = bdd_res ∧ bdd_m_e      { bdd_m_e est le bdd équivalent à m_e}
Si (NbSol(bdd_res,|Y|) = 1) faire
   m_Y ← Solution(bdd_res)      {Extraction de la solution sous forme de monôme }
   O_temp ← CalculTrans(p, m_e ∧ m_Y).sorties
   p_temp ← CalculTrans(p, m_e ∧ m_Y).but
   p_b ← overline(p_temp^Y)
   O ← O_temp \ Y
Sinon
   Exit(erreur)
```

■

Examinons à présent le coût de ce nouvel algorithme. Nous appelons $M(n)$ le nombre moyen de nœuds d'un bdd représentant une fonction à n variables. Le coût moyen des opérations $\wedge$ et $\vee$ est donc de l'ordre de $O(M(n)M(m))$ où n et m sont les nombres de variables associés aux bdd opérandes. Dans ce nouvel algorithmeΓc'est la fonction `Construire_bdd` qui est critique au niveau du temps d'exécution. En effetΓles fonctions `NbSol` et `Solution` sont des simples parcours du bdd opérandeΓla borne supérieure de leur coût est donc de l'ordre de $O(M(s))$Γoù s est la somme du nombre des événements internes et des entrées du programme. La conjonction effectuée entre `bdd_res` et `bdd_`m_e est bornée supérieurement par un coût de l'ordre de $O(M(s)^2)$. La fonction `Construire_bdd` se décompose en deux phases :

1. Calcul de toutes les équations associées aux événements internes ;
2. Conjonction de toutes les équations.

La borne supérieure du coût de la seconde phase est de l'ordre de $y.O(M(s)^2)$Γoù y est le nombre d'événements internes utilisés par le programme. C'est aussi le nombre d'équations. En effetΓdans le pire des cas tous les événements (entrées + événements internes) sont concernés par chaque équation. La borne supérieure du coût de la première phase est $y^2.O(M(s)^2)$ car il faut construire y équationsΓet pour chaque équation il faut coder tous les monômes déclencheurs de transitions qui génèrent l'événement interne considéré. Or la borne supérieure du coût du codage d'un monôme est $s.O(M(s)^2)$. Les autres phases de la construction d'une équation ontΓellesΓun coût borné supérieurement par $O(M(s)^2)$.

Par conséquentΓla borne supérieure du coût de la fonction `Construire_bdd` est de l'ordre de $y^2.O(M(s)^2)$. Il n'existe pas de résultat théorique sur le coût de la fonction M. CependantΓ

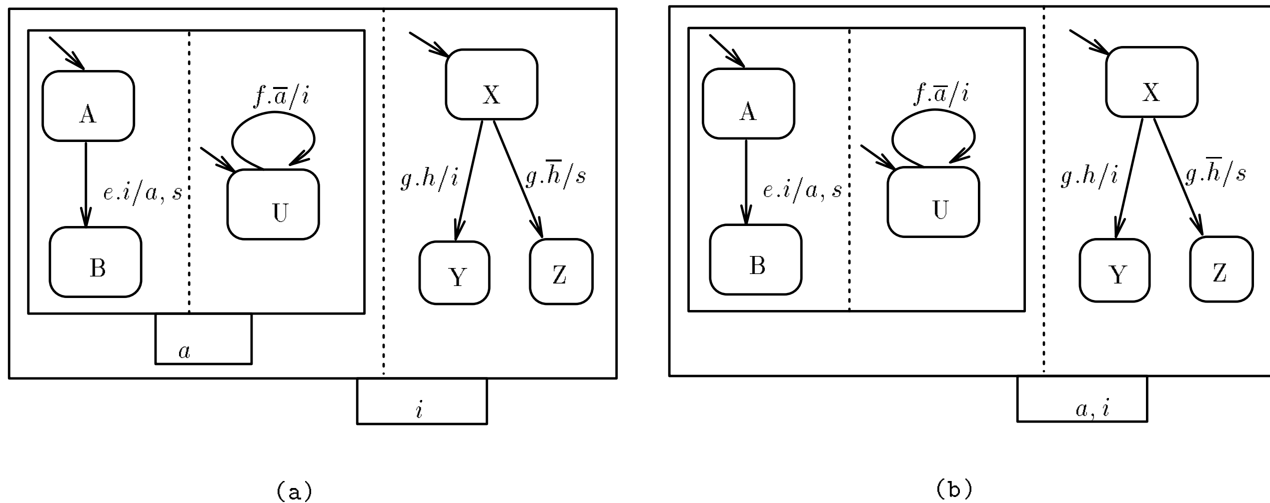


Figure 7.3: déclarations multiples d'événements internes

l'expérience montre qu'il ne croît pas exponentiellementΓd'où le gain par rapport à l'algorithme initial.

Cet algorithme n'est valable que si le programme ne contient qu'une déclaration d'événements internes. En effetΓsupposons que le programme à compiler soit celui donné par la figure 7.3 (a). Les systèmes d'équations définissant les statuts de a et i dans la configuration initiale ne peuvent être résolus indépendammentΓil faut résoudre globalement le système d'équations $i = f.\overline{a} \vee g.h$ et $a = e.i$. Ce qui revient à faire comme si le programme contenait une seule déclaration d'événements internesΓc'est-à-dire remplacer le programme de la figure 7.3 (a) par celui de la figure 7.3 (b). Ce remplacement n'est pas toujours sans conséquenceΓpar exemple le programme de la figure 7.4 contient un problème de causalité au niveau de la déclaration de a et b. Si on remonte cette déclaration au niveau de celle de iΓle problème de causalité disparaît. Nous avons montré que le fait de remonter les déclarations d'événements internes d'un programme au niveau de celle qui est la plus externeΓne peut qu'éliminer des problèmes de causalité. Par conséquentΓ nous avons choisi de remonter toutes les déclarations d'événements internes du programmeΓavant de le compiler.

7.1.3 Adaptation de la méthode des potentiels du langage Esterel

L'amélioration que nous présentons à présent s'inspire de la méthode des potentiels utilisée dans le compilateur du langage Esterel. Une description de cette méthode peut être trouvée dans [Gon88ΓBG92]. Nous décrivons directement la méthode adaptée à la compilation d'Argos.

On se ramène ici aussi à un programme Argos ne contenant qu'une seule déclaration d'événements internes.

Calculer la réaction d'un programme Argos face à une configuration d'entrées et dans une configuration donnéesΓconsiste à déterminer parmi l'ensemble des transitions susceptibles d'être déclenchées — issues d'un état actif dans la configuration — lesquelles le sont véritablement.

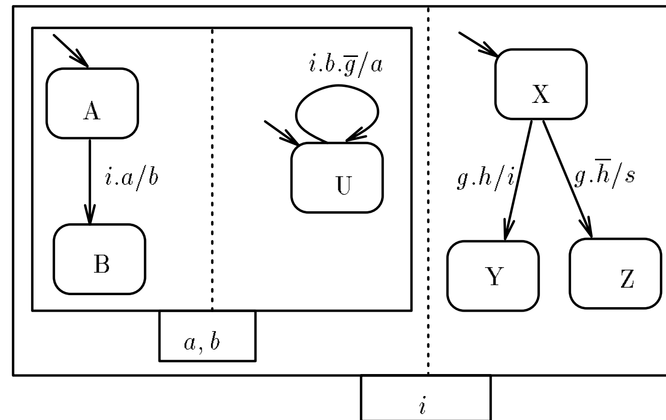


Figure 7.4: effets de la remontée des déclaration d'événements internes

Pour celaΓil ne suffit pas de connaître le statut des entrées du programme car le déclenchement de certaines transitions peut être conditionné par le statut d'événements internes. Le principe de la méthode que nous présentonsΓconsiste à calculer de manière incrémentale la réaction du programmeΓen déterminant à chaque pas un ensemble de transitions dont on est sûr qu'elles sont déclenchées : on connaît le statut de tous les événements qui apparaissent dans son monôme déclencheur.

Au départΓseul le statut des entrées est connu. Un pas est constitué des phases suivantes.

1. Elimination des transitions dont la condition de déclenchement ne peut pas être satisfaite.
2. Calcul de l'ensemble des événements internes qui peuvent encore être émis par la réaction. Cet ensemble constitue le *potentiel* du programme.
3. Déduction de l'absence d'un certain nombre d'événements internes : ceux qui ne peuvent plus être émis.
4. Détermination des transitions qui sont sûres d'être déclenchées car leur condition de déclenchement ne fait intervenir que des événements dont on connaît le statut.
5. Déduction de la présence d'un certain nombre d'événements internes : ceux qui sont émis par les transitions nécessairement déclenchées.

On redémarre alors pour un nouveau pasΓavec un ensemble d'événements dont on connaît le statutΓenrichi par rapport au pas précédent. Le calcul se termine lorsqu'un pas ne permet de calculer le statut d'aucun événement interne supplémentaire. Deux cas peuvent alors se produire : soit tous les événements internes ont alors un statut connuΓdans ce cas la réaction du programme est calculée. L'expérience montre que c'est le cas le plus courant ; soit il reste des statuts indéterminés et dans ce casΓil faut calculer le système d'équations qui régit les communications du programme en tenant compte des statuts déjà déterminésΓpuis le résoudre symboliquement.

Le nombre de pas de cette phase d'analyse de la réaction du programme par la méthode des potentiels est borné supérieurement par le nombre d'événements internes du programme. En effetΓà chaque pas le statut d'au moins un événement supplémentaire est identifié. Un pas ne dépend pas du nombre d'événements internesΓmais plutôt du nombre d'automates et de leur degré de branchement. Cette première phase a donc un coût en temps qui est linéaire par rapport au nombre d'événements internes du programme.

Remarque 7.1 *Par rapport à la méthode utilisée dans le cas du langage Esterel, le problème que nous avons à résoudre en Argos est plus simple. En effet, d'une part le langage Esterel gère des signaux valués ce qui rend plus complexe la méthode ; d'autre part, il n'y a pas d'équivalent en Argos à la notion de séquence présente en Esterel. Cette notion de séquence se traduirait en Argos par une notion d'état* transitoire, *c'est-à-dire des états qui ne correspondent pas à des points d'arrêt entre deux réactions du programme. Cette notion de séquence en Esterel engendre une complexité plus grande qu'en Argos. En effet, dans le cas d'Argos, la phase d'élimination des transitions dont on est sûr qu'elles ne sont pas déclenchées — première phase — ne tient compte que des statuts des événements déjà connus ; alors qu'en Esterel, il faut aussi tenir compte des états buts des transitions dont on est déjà sûres qu'elles sont déclenchées. Par conséquent, si en Argos on peut raisonner uniquement sur l'ensemble des étiquettes des transitions pour déterminer lesquelles sont déclenchées, en Esterel il faut aussi considérer les états buts de ces transitions.* ■

Nous venons de présenter successivement deux améliorations du compilateur Argos. Nous allons à présent comparer les performances des différentes versions du compilateur sur des exemples précis. Le tableau suivant met en évidence les temps de calcul des transitions issues d'un état du modèleΓdans le cas :

- du compilateur avant introduction des bdd ;
- de celui après introduction des bdd ;
- du compilateur mettant en œuvre la méthode des potentiels.

Les programmes exemples sont caractérisés par le nombre d'événements internes qu'ils utilisent. Les temps d'exécution sont donnés en secondes.

	prog 1 11 ev. int.	prog 2 26 ev. int.
avant bdd	60	∞
après bdd	0	0Γ36
avec potentiels	0	0Γ1

Rappelons que pour obtenir le temps de compilation d'un programmeΓles coûts donnés dans ce tableau sont à multiplier par le nombre d'états du modèle du programme.

7.1.4 Mise en œuvre de la sémantique d'Argos temporisé en termes de graphes temporisés

Nous présentons à présent la manière dont la sémantique des programmes Argos temporisés a été mise en œuvre dans le compilateur existant.

Tout d'abord, il est important de noter que la structure du compilateur est conçue de telle façon qu'il est facile d'ajouter un nouveau format de sortie d'automates. En effet, l'écriture du modèle d'un programme s'effectue à l'aide de différents points d'écritures paramétrés par le format de sortie choisi par l'utilisateur. Ce choix s'exprime par un mécanisme d'option au moment de l'appel du compilateur. Ces points d'écriture sont dédiés à l'écriture :

- de l'entête du fichier de sortie ;
- d'un état du modèle ;
- d'une transition du modèle ;
- de la fin du fichier.

Nous utilisons comme format de graphes temporisés, celui utilisé en entrée par l'outil de vérification formelle Kronos. Une description de ce format peut être trouvée dans [OS92].

Les premières modifications engendrées par les états temporisés concernent les formats *targos* et *argos* du compilateur, puisque nous avons dû y intégrer cette nouvelle structure syntaxique. Le traducteur du format *targos* vers *argos*, ainsi que l'analyseur *argos* ont dû être modifiés en conséquence. Les contraintes de sémantique statique concernant les états temporisés sont vérifiées soit au niveau du traducteur si l'on utilise la syntaxe textuelle d'Argos, soit au niveau de l'éditeur syntaxique.

D'après la section 6.1.6, une fonction de calcul d'un arc connaissant son sommet but et sa condition de déclenchement doit remplacer la fonction de calcul d'une transition `CalculTrans`. Deux différences existent entre ces deux fonctions :

- La fonction `CalculArc` prend un paramètre d'entrée supplémentaire qui est la condition sur les valeurs des horloges ;
- La fonction `CalculArc` fournit un résultat supplémentaire qui est l'ensemble des horloges remises à zéro par cet arc.

La solution que nous présentons permet d'utiliser la fonction de calcul d'une transition pour calculer un arc du graphe, ce qui limite les modifications à apporter au compilateur. Pour cela, on rajoute un paramètre de sortie à cette fonction, qui n'est utilisé que dans le cas où l'on produit à partir du programme un graphe temporisé. Pour ne pas avoir à modifier les paramètres d'entrée de la fonction, on simule les conditions sur les valeurs des horloges par des conditions sur le statut d'entrées fictives.

L'idée est la suivante : on associe à chaque état temporisé un événement particulier. On note cet événement $tout_i$ si i est l'indice de l'état temporisé. Lors de l'analyse du format *argos* chaque transition d'expiration, jusque là identifiée par un caractère spécial, est remplacée par

une transition qui admet pour condition de déclenchement le monôme $tout_i$ si i est l'indice de son état source. Une table est crééeΓqui met en relation cet événement spécial avec un nom d'horloge et un entier strictement positif qui est la borne de l'horloge.

La phase de saturation qui permet de manipuler par la suite des monômes complets doit être modifiée pour tenir compte de la particularité de ces événements fictifs. En effetΓceux-ci ne doivent pas être traités comme les autres événementsΓen raison de la priorité donnée à la transition d'expirationΓsur les autres transitions issues de l'état temporisé.

Une fois ces événements fictifs ajoutés et correctement traités par la phase de saturationΓ l'algorithme 2.1 de calcul des transitions du modèles peut être utilisé :

- en modifiant la fonction `CalculTrans` pour qu'elle fournisse comme résultat l'ensemble des horloges remises à zéro ;
- en interprétant correctement lors des points d'écritureΓle statut des événements fictifs dans le monôme déclencheur. La règle à appliquer est la suivante : si un événement $tout_i$ est supposé présent alors l'arc est conditionné par l'égalité de l'horloge h_i à sa borne ; s'il est absentΓalors l'arc est conditionné par le fait que la valeur de l'horloge h_i est strictement inférieure à sa borne.

Remarquons que les améliorations apportées au compilateur concernant le calcul des communications profitent à la compilation des programmes Argos temporisés en graphes temporisés.

7.1.5 Autres extensions

Deux autres extensions au compilateur Argos ont été réalisées. Nous ne les détaillons pasΓmais présentons uniquement leurs intérêts.

- **Modification de la sémantique du raffinement**. L'opérateur de raffinement tel que nous l'avons présenté n'est pas symétriqueΓen ce sens que lorsqu'un programme est interrompu il a le droit de réagirΓalors que lorsqu'il est lancé il n'a pas cette possibilité. La sémantique du raffinement peut facilement être modifiée afin d'intégrer cette nouvelle possibilité. CependantΓil est nécessaire de restreindre l'utilisation de cet opérateur par des contraintes statiques pour éliminer des comportements aberrants. L'ancien opérateur de raffinement peut alors être défini comme une macro-notation. Pour pouvoir utiliser ce nouvel opérateur de raffinementΓet ainsi mieux maîtriser ses effetsΓnous avons intégré cette nouvelle sémantique au sein du compilateur Argos.
- **Compilation d'Argos temporisé en automates interprétés**. Pour pouvoir exécuter les programmes Argos contenant des états temporisésΓnous avons aussi intégré au compilateur une option permettant de compiler de tels programmes en automates interprétés — format OC —.

7.2 Expérimentations

Comme l'indique son titre "étude d'un environnement de programmation et de vérification des systèmes réactifsΓmulti-langages et **multi-outils**"Γune des motivations du travail effectué lors de cette thèse est la connexion au plus grand nombre d'outils. De faitΓune part de notre travail a consisté à étudier les fonctionnalités d'outils pour mettre en évidence leur intérêt pour la programmation et la vérification des systèmes réactifs. La connexion de ces outils à l'environnement du langage Argos a été testée sur des exemples concrets. La mise en œuvre de ces tests n'a pas toujours été immédiateΓcertains outils demandant de "rentrer dans un moule" mal adapté à nos situations. Elle a dans certains cas été à l'origine d'une réflexionΓmenée en collaboration avec les équipes concernéesΓsur les outils testés. Ces réflexions ont conduit soit à la modification des fonctionnalités de l'outilΓsoit à l'amélioration de ses performances grâce à la prise en compte des particularités des automates modèles de programmes Argos.

Ce sont ces différents aspectsΓparfois très concretsΓqu'il nous a semblé intéressant de présenter dans ce chapitreΓpour apporter des exemples de réponses aux problèmes auxquels on peut être confronté dans des situations de vérification formelle.

Nous avons choisi de présenter les expérimentations effectuées autour de trois outils. Ce sont trois outils de vérification développés au sein du projet Spectre. Ils mettent en œuvre des techniques de vérification différentes :

- vérification comportementale avec AldébaranΓ
- vérification à l'aide d'une logique temporelle avec KronosΓ
- vérification de programmes Lustre par la technique des observateurs à l'aide de Lésar.

Remarquons que les connexions à ces trois outils sont le fruit du travail formel présenté dans les différents chapitres qui précèdentΓpuisque :

- Aldébaran et Kronos travaillent tous deux par comparaison syntaxique des étiquettes du programme et de la propriétéΓd'où la nécessité d'appliquer les résultats présentés dans le chapitre 1 ;
- Kronos permet de vérifier des programmes Argos grâce à la définition de la sémantique d'Argos temporisé en termes de graphes temporisés ;
- Lésar permet de vérifier des programmes Argos ou ArgoLus grâce aux traductions de ces langages vers le langage Lustre.

7.2.1 Aldébaran

Aldébaran est un outil de vérification de spécifications comportementales. Il est actuellement principalement connecté (cf. [FGM+92]) au langage Lotos qui permet de modéliser l'évolution de systèmes parallèles répartis. De nombreux protocoles de communication ont été vérifiés à l'aide d'AldébaranΓà partir de modélisation en Lotos. Un exemple d'une telle expérience se trouve dans [Mou92]. Les étiquettes des modèles de programmes Lotos sont des noms d'action et les

algorithmes implémentés dans le système Aldébaran sont basés sur la comparaison syntaxique des étiquettes de l'automate modèle du programme avec celles de la propriété.

Différentes relations de comparaison entre le modèle d'un programme et sa propriété sont mises en œuvre dans AldébaranΓdont la simulation telle que nous l'avons présentée en 1.14 — relation simplifiée —.

Les possibilités d'utilisation d'Aldébaran se trouvent limitées par les deux points suivants :

- l'impossibilité de stocker les automates modèles de programmes à cause du phénomène d'explosion du nombre d'états inhérent aux systèmes parallèles. Pour repousser les limites d'utilisation de l'outilΓdes algorithmes permettant de ne pas stocker entièrement cet automate ont été définis et implémentés [Mou92] ;
- le coût des algorithmes de comparaisonΓen partie dû à leur caractère énumératif : les transitions du modèle du programme sont analysées une par une. Pour améliorer les performances de l'outilΓdes méthodes basées sur une représentation symbolique du modèle ont été définies et mises en œuvre [Ker]. Cette représentation permet d'analyser les transitions sous forme d'ensemble.

Dès la conception du compilateur ArgosΓsa connexion au format d'entrée d'Aldébaran a été prévueΓgrâce au format de sortie AUT. CependantΓles utilisations d'Aldébaran sur des programmes modélisés en Argos sont restées marginalesΓdu fait de l'inadéquation de la comparaison par égalité syntaxique à la vérification des systèmes réactifs.

Le travail présenté dans le chapitre 1 permet d'éliminer cette difficultéΓen définissant une fonction de transformation de la propriété qui permet de se ramener à des comparaisons par égalité syntaxique. Nous avons testé ce principe sur divers exemplesΓdont le contrôleur de feux de voitures présenté dans la section 2.3.

Parallèlement à ce travailΓnous avons étudié la possibilité d'étendre Aldébaran en paramétrant ces algorithmes par une relation de comparaisonΓdépendante du contexte d'utilisation. Pour celaΓdeux stratégies opposées sont envisageables :

- La relation de comparaison est implémentée par une fonction externe. L'outil Aldébaran appelle cette fonction lorsqu'il a besoin de comparer deux étiquettes. Cette solution suppose la résolution de problèmes d'édition de liens et/ou de communication complexes pour ne pas détériorer les performances et la souplesse d'utilisation du logiciel Aldébaran.
- La relation de comparaison est intégrée au système Aldébaran. Etant donné que ni la forme des étiquettesΓni la relation de comparaison ne peuvent être connues a prioriΓcette solution nécessite la définition de langages de description des étiquettes et des relations de comparaison.

Ces deux stratégies sont toutes les deuxΓpour des raisons différentesΓcomplexes à mettre en œuvre. Remarquons que si l'objectif est uniquement d'implémenter une connexion Argos/Aldébaran alors la deuxième solution se simplifie puisque la forme des étiquettes et la relation de comparaison sont connues. Les langages de description ne sont donc plus utiles et on peut définir un exécutable d'Aldébaran dédié à la vérification des systèmes réactifs modélisés par des programmes Argos.

Une troisième stratégie existe par ailleursΓnous l'avons définie dans le chapitre 1Γelle consiste à transformer la propriété à vérifier en tenant compte du modèleΓpour se ramener à une vérification par comparaison syntaxique. Cette solution nous semble la plus intéressante car peu difficile à mettre en œuvreΓet permettant l'utilisation de différents outilsΓdont AldébaranΓsans aucune modification.

7.2.2 Kronos

Le système Kronos [HNSY92ΓYov93] est un outil de vérification symbolique de formules de la logique TCTL à partir de graphes temporisés. Il peut aussi être utilisé comme un outil de vérification de formules de la logique CTL sur des systèmes de transitions étiquetéesΓpuisque la logique TCTL est une extension de CTL et un graphe temporisé sans horloge n'est rien d'autre qu'un système de transitions étiquetées. EnfinΓune extension de Kronos aux systèmes hybrides est en cours d'étude [Oli].

Nous avons utilisé Kronos pour vérifier divers exemples : le contrôleur de feux de voiture présenté dans la section 2.3Γdifférents schémas de synchronisation (cf. section 6.4)Γun jeu de réflexes [JMO93]Γun "scheduler" de tâchesΓ...

Pour effectuer ces expérimentationsΓnous avons dû prendre en considération le fait que le prédicat `after` qui permet de transformer des informations portées par les étiquettes en information d'état n'est pas mis en œuvre par Kronos.

Cette impossibilité d'utiliser le prédicat `after` dans les formules de la logique ne constitue par dans notre cas — modélisation du graphe à l'aide d'un langage synchrone — un obstacle à l'utilisation de KronosΓet ceci pour les raisons suivantes :

- Il est toujours possible en Argos — ceci est vrai pour n'importe quel langage synchrone — de transformer le programme et la formuleΓde manière à exprimer la propriété à l'aide du prédicat `enable`. OrΓce prédicat est plus facile à mettre en œuvre car il ne nécessite pas de modifier le modèle. Il est mis en œuvre par Kronos ;
- Toute propriété de sûreté peut se ramenerΓgrâce à l'utilisation de composantes supplémentaires mises en parallèle avec le programmeΓà une propriété d'accessibilité d'un état d'erreur. Cette technique est illustrée par la suite sur l'exemple du schéma de synchronisation présenté dans la section 6.4 ;
- Toute propriété de vivacitéΓsi on la borne dans le tempsΓest transformée en une propriété de sûreté. D'où un intérêt supplémentaire au point précédent ;

Reprenons l'exemple consistant à prouver l'absence de blocage temporel dans un schéma de synchronisation. Nous avons vu dans la section 6.4 qu'une solution pour cette preuve consiste à prouver la satisfaction de la formule $\forall \Box \neg \texttt{after}(\texttt{obs}(\mathit{erreur}))$ sur le graphe temporisé modèle du programme de la figure 6.15. Pour cela Kronos ne peut être directement utilisé à cause de la présence d'un prédicat `after`.

Kronos propose à l'utilisateur pour exprimer les propriétésΓsoit d'utiliser le prédicat `enable`Γ soit de nommer directement dans la formule des sommets du graphe. Nous utilisons sur l'exemple cette deuxième possibilité. La figure 7.5 présente le nouveau programme sur lequel nous allons

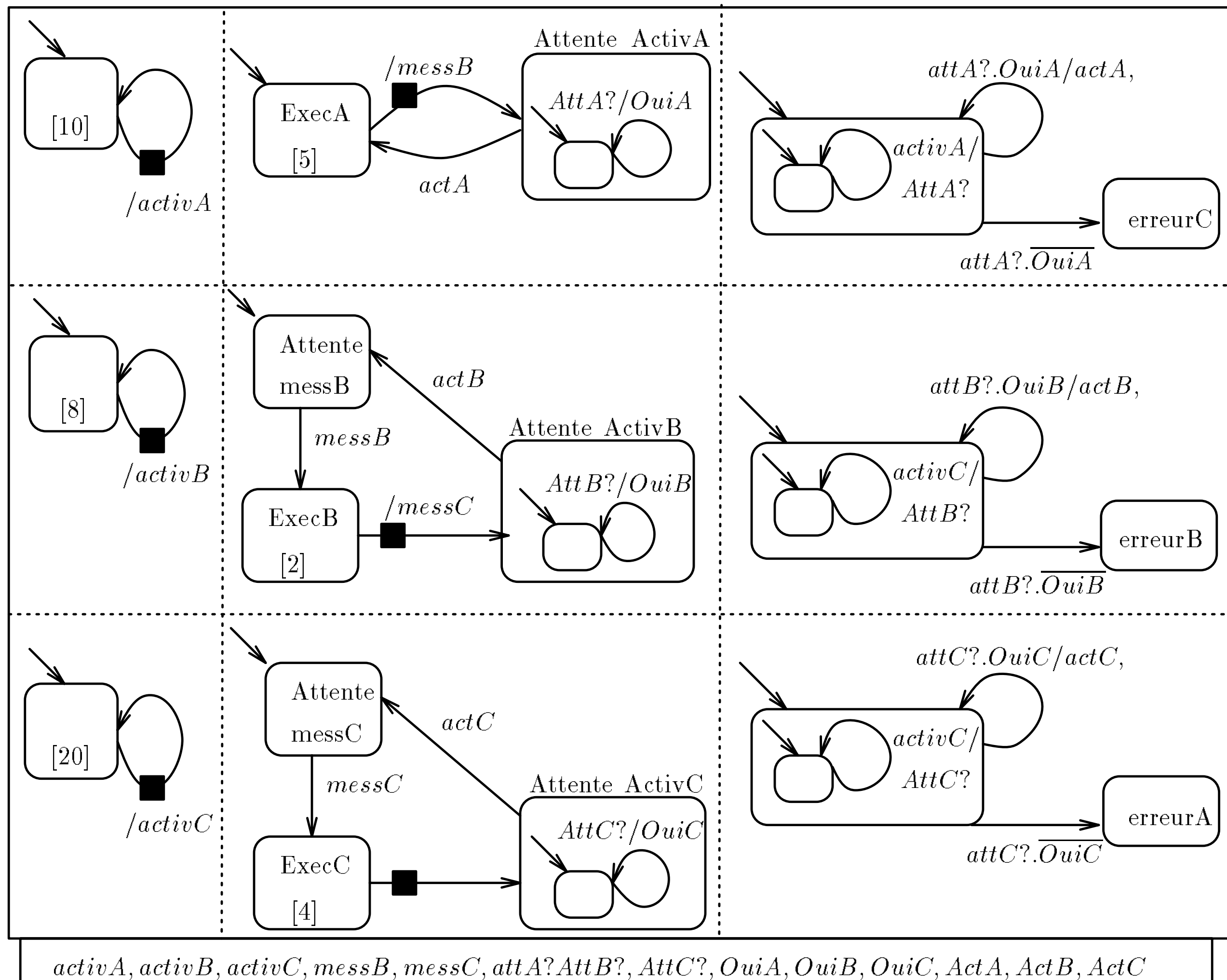


Figure 7.5: observation des blocages temporels

vérifier la nouvelle formule. L'absence de blocage temporel revient à vérifier que dans le graphe temporisé modèle de ce nouveau programmeΓaucun sommet dans lequel un des trois états d'erreur est actifΓn'est accessible.

Il est facile venant du langage Argos de connaître l'ensemble des indices de ces sommets. En effetΓchaque sommet du graphe temporisé modèle d'un programme Argos représente une configuration des états des automates du programme. Si l'on note $I = \{i_1, i_2, \ldots, i_n\}$ l'ensemble des indices de ces sommetsΓalors la formule TCTL qui peut être utilisée pour prouver l'absence de blocage temporel à l'aide de KRONOS est la suivante : $\forall\Box\neg(state(i_1)$ or $state(i_2)\ldots$ or $state(i_n))$.

Les diverses expériences faites en utilisant la connexion Argos/Kronos ont montré certaines limites aux performances du système Kronos. En effetΓles performances de l'outil sur des graphes de taille modeste : une dizaine d'états et une centaine de transitionsΓselon la forme de la propriété

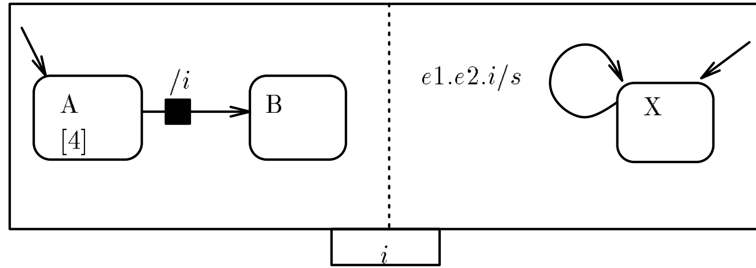


Figure 7.6: non minimalité des graphes temporisés produits à partir d'Argos

vérifiée ne sont pas celles attendues. Nous avons cherché à expliquer la cause de ces "contre-performances". Notre première hypothèse a été d'incriminer le fait que les graphes temporisés produits à partir du langage Argos ne sont pas minimaux en ce qui concerne leur nombre de transitions. Considérons pour illustrer ce point le programme Argos de la figure 7.6.

A partir de l'état `(A,X)` du graphe temporisé sont issus les arcs :

- `((A,X)`$\Gamma v = 4\Gamma e1.\overline{e2}\Gamma$`(B))` ;
- `((A,X)`$\Gamma v = 4\Gamma \overline{e1}.e2\Gamma$`(B))` ;
- `((A,X)`$\Gamma v = 4\Gamma \overline{e1}.\overline{e2}\Gamma$`(B))`.

Si l'on s'intéresse à une propriété d'accessibilité d'un état d'erreur alors ces trois transitions sont équivalentes et pourraient être remplacées par une seule. Nous avons testé cette hypothèse sur un exempleΓen remplaçant dans tout le programme chaque "faisceau" de transitions par une seule. Le temps d'exécution de Kronos n'ayant pas été sensiblement modifié par cette modificationΓnous avons abandonné cette hypothèse.

L'idée à présent en cours d'expérimentation est liée à la forme particulière des conditions sur les horloges des graphes temporisés produits à partir d'Argos. Ce sont des conjonctions de conditions de la forme $v < k$ et $v = k$ où k est une constante entière. La prise en compte de cette particularité dans les algorithmes du système Kronos semble constituer un moyen d'améliorer ses performances. Cette forme particulière de conditions temporelles est liée à la structure de contrôle utilisée pour générer les horloges dans le graphe temporisé. Ce sont des structures de type "chiens de garde temporisés". Tout langage qui utilise ce type de structures pour générer des graphes temporisés aboutit à des graphes qui manipulent cette forme particulière de conditions temporelles. Il est par conséquent très intéressant pour le système Kronos d'améliorer les performances de ses algorithmes pour ce type de conditions temporelles.

7.2.3 Lésar

Le système Lésar est un système de vérification formelle des programmes Lustre basée sur la méthode des observateurs [Rat92]. Il fait suite à une première étude sur ce thème [Glo89]. Par rapport à cette étude qui a jeté les bases de la vérification formelle à partir de programmes

LustreΓune méthode originale à base d'une représentation symbolique du programme à l'aide de graphes de décision binaires a été définie et implémentée.

Pour utiliser ce système sur un programme LustreΓun "squelette" de contrôle boléen est extrait du programme. Il est montré que cette abstraction booléenne du programme préserve les résultats de satisfaction d'une propriété de sûreté.

Lustre étant un langage déclaratif flot de donnéesΓles observateurs sont eux aussi exprimés sous cette forme. Un observateur Lustre est un nœud qui possède un seul flot de sortie qui est booléen. Ce flot de sortie est faux à un instant donné si une erreur dans le comportement du programme est détectée.

Par exempleΓpour vérifier qu'un programme qui admet deux flots booléens de sorties `o1` et `o2`Γest tel que ses deux flots sont en exclusion mutuelleΓl'observateur suivant peut être utilisé.

```
node observateur(o1,o2 :  bool) returns (ok :  bool);
let
ok = not (o1 and o2);
tel
```

Le programme entrée de l'outil de vérification est constitué d'un appel au nœud décrivant le programme à vérifier et d'un appel à l'observateurΓtoutes les sorties autres que `ok` étant déclarées locales. La propriété est vérifiée si et seulement si pour tous les flots d'entrées possiblesΓla sortie `ok` est toujours vraie. Pour faciliter la tâche du programmeurΓune bibliothèque de nœuds "observateurs" les plus utiles a été définie.

Nous avons utiliser le système Lésar pour vérifier le bon fonctionnement des anti-brouillard dans le programme Lustre du contrôleur de feux de voitures présenté dans la section 4.1.5. L'observateur vérifie la satisfaction de l'implication suivante à chaque instant : `codes and ab_on => anti_brouillard`.

Si à présentΓnous souhaitons vérifier cette même propriété sur le programme Lustre obtenu par traduction du programme Argos du contrôleur de feux de voitures (cf. figure 2.21)Γalors l'observateur est plus complexe car les sorties du programme Argos correspondent aux ordres d'extinction et d'allumage et non pas à l'état des lampes. De plusΓon n'accède pas aussi facilement à l'état du bouton associé aux anti-brouillard. Ce dernier point peut en fait être résolu par la remarque suivante : la traduction Argos vers Lustre telle que nous l'avons définieΓassocie une variable à l'état `bouton_bas` de l'automate Argos qui modélise le comportement du bouton. Il suffit de retrouver dans le programme Lustre le nom de cette variable pour pouvoir l'utiliser dans l'observateurs. On peut imaginer pour faciliter cette recherche que lors de la traduction structurelle d'Argos en Lustre une table des correspondances entre noms d'état et noms de variable soit générée.

Bilan et perspectives

7.3 Bilan

A l'origine de ce travail deux objectifs avaient été fixés :

- Etudier la faisabilité d'un mélange au niveau source des langages Argos et Lustre pour répondre au besoin d'un formalisme mixte impératif/déclaratif ;
- Etudier une extension du langage Argos pour qu'il définisse un jeu de constructeurs de graphes temporisés. L'intérêt de cette extension est lié aux possibilités intéressantes de vérification formelle qui accompagnent cette forme d'automate avec variables.

Ces deux axes de recherche ont pour point commun le langage Argos. Les diverses expérimentations réaliséesΓsoit pour la définition du langage mixteΓsoit pour l'extension d'Argos aux graphes temporisésΓont conduit à effectuer un travail spécifique sur ce langage et son environnement.

Nous présentons ici le bilan des travaux effectués :

- En liaison avec la définition du langage mixte ArgoLus ;
- En liaison avec la définition de la sémantique d'Argos en termes de graphes temporisés ;
- Spécifiques au langage Argos.

7.3.1 La définition du langage ArgoLus

C'est à partir d'études de casΓque les besoins d'un formalisme mixte impératif/déclaratif sont apparus. Celui qui semblait le plus important était d'introduire une structure de contrôle hiérarchique dans les langages déclaratifs. Dans le langage impératif ArgosΓcette structure de contrôle est introduite grâce à l'opérateur de raffinement. Le principe de cet opérateur consiste à lancer et à interrompre des composantesΓen tenant compte de l'activité des états d'un automate de contrôle. Notre première idée de mélange a été d'utiliser cet opérateur pour contrôler l'activitéΓ non plus de composantes ArgosΓmais de composantes Lustre. Il nous a alors fallu donner un sens à ce mélange syntaxique. L'idée la plus naturelle était de concevoir la composante Lustre comme la description d'un automate. On obtient ainsi un programme Argos dont on peut calculer le modèle.

De cette première forme de mélange est née une vision unificatrice des langages Argos et Lustre. Ils peuvent tous les deux être vus comme offrant :

- Un moyen de construire des transformateurs à mémoire de séquences infinies d'entrées en séquences infinies de sorties ;
- Des moyens de composer les transformateurs simples ainsi définisΓet de les faire communiquer pour en décrire de plus complexes.

En ArgosΓles transformateurs simples sont décrits par des automates. Ils peuvent être composés par les opérateurs de mise en parallèle et de raffinementΓet communiquent par un mécanisme basé sur l'introduction d'événements locaux émis par des composantes et reçus par d'autres.

En LustreΓles transformateurs simples sont décrits par des ensembles d'équations — appelés nœuds — qui définissent les liens permanents entre les entrées et sorties du programme. Ces nœuds peuvent être composés en décrivant un réseau qui relie les sorties d'un nœud aux entrées d'un autre. C'est par cette liaison que les nœuds communiquent entre eux.

De cette vision unificatrice des deux langages est née le langage ArgoLus. C'est une union des moyens de description des transformateurs simples d'une partΓdes moyens de composition et de communication d'autre part.

Pour formaliser cette idée intuitiveΓnous avons cherché à définir la sémantique d'ArgoLus comme une union des sémantiques des deux langages qui le constituent. Pour celaΓles deux sémantiques ont été définies dans un style commun — en reformulant la sémantique d'Argos dans le style utilisé pour définir celle de Lustre — puis nous avons intégré dans chaque sémantique la possibilité d'importer des composantes écrites dans l'autre langage.

ParallèlementΓil nous fallait trouver un moyen de mettre en œuvre la définition de ce langage. Pour réutiliser l'existantΓnous nous sommes orientés vers une solution à base de traduction des programmes ArgoLus soit en programmes ArgosΓsoit en programmes LustreΓafin d'utiliser les compilateurs de ces langages. C'est lors de l'étude de ces deux traductionsΓque les traductions structurelles entre Argos et Lustre ont été définies.

La définition de ces traductions structurelles constitue un résultat à part entière de notre travail. En effetΓelles permettent d'une part de mettre en évidence les liens et les différences entre les deux langagesΓd'autre part d'homogénéiser leur environnement.

7.3.2 La sémantique d'Argos en termes de graphes temporisés

Plus que la définition de la sémantique d'Argos en termes de graphes temporisésΓqui n'a pas soulevé de problèmes majeursΓce sont les interrogations que cette sémantique a fait naître qui ont été à l'origine de réflexions intéressantes :

- Quelles bonnes propriétés ont les graphes temporisés modèles de programmes Argos ?
- Quelles garanties a-t-onΓlorsqu'on effectue des preuves sur un modèle où le temps est interprété de manière continueΓque le résultat obtenu est préservé lors de l'exécution du programmeΓqui s'effectue dans notre contexte avec une vision discrète du temps ?

Les réponses que nous apportons à ces deux problèmes sont formalisées par deux théorèmes qui expriment les résultats suivants :

- Les graphes temporisés modèles de programmes Argos sont tous bien temporisés. Cette propriété est importante car les graphes qui ne la satisfont pas ne peuvent pas modéliser des systèmes réels. De plusΓla méthode de vérification qui analyse ces graphes n'est valable que sur les graphes bien temporisés ;
- La satisfaction des propriétés de sûreté sur le graphe temporisé interprété en temps continu est préservée lors de l'exécution du programme.

Il est important de noter que le problème d'exécution en temps discret n'avait jusqu'alors pas été étudiéΓles graphes temporisés n'étant utilisés que pour spécifier des systèmes et non en tant que modèles de programmes censés être exécutés. OrΓdans un objectif de spécification d'un systèmeΓavoir une interprétation continue du temps est plus conforme à la réalitéΓil est donc justifié que des outils de vérification travaillent avec une telle interprétation.

Cette réflexion sur la problématique “vérification en temps continu et exécution en temps discret”Γnous a permis de définir un nouveau champ d'application au langage Argos en tant que langage de spécification de systèmes — ou de problèmes — n'ayant a priori pas de liens avec les systèmes réactifs. Une telle expérienceΓnon décrite dans ce documentΓa été menée en collaboration avec le laboratoire LABRI de Bordeaux. Elle concerne un problème d'ordonnancement de tâches qui est par nature fortement indéterministe. En introduisant des entrées supplémentaires qui permettent de gérer cet indéterminismeΓle langage Argos peut être utilisé pour spécifier le problème et répondre à certaines questionsΓpar des preuves formelles effectuées sur le graphe temporisé modèle du programme Argos.

Par ailleursΓ c'est en effectuant des expériences de vérification formelle sur le modèle des graphes temporisés à l'aide de l'outil KronosΓque nous nous sommes trouvés confrontés au problème de l'inadéquation d'un grand nombre d'outils de preuve existantsΓpour la vérification des systèmes réactifs. Depuis longtemps en faitΓcette inadéquation était apparueΓque ce soit pour les outils à base de logique temporelle — comme Kronos — ou pour ceux de type simulation comportementale — comme Aldébaran —Γsans véritablement savoir si cette inadéquation s'expliquait dans les deux cas par les mêmes raisons. L'analyse de ce problème nous a non seulement permis d'en trouver la cause communeΓà savoir le besoin lors de la vérification des systèmes réactifs de relations de comparaison d'étiquettes plus complexes que l'égalité syntaxiqueΓmais aussi d'y remédier sans pour autant modifier les outils existants. En effetΓla solution que nous proposons consiste à modifier la propriété pour se ramener à une comparaison par égalité syntaxique. Cette solution a été mise en évidence grâce à une présentation de ces deux méthodes de vérification plus générale que leurs présentations usuelles. Cette généralisation a permis d'identifier le cas implémenté par les outils comme un cas particulier auquel on peut toujours se ramener par transformation de la propriété. Le cadre formel que nous avons définiΓainsi que la fonction de transformation de la propriétéΓne sont pas spécifiques aux systèmes réactifs et doivent pouvoir trouver d'autres applications.

7.3.3 Les extensions apportées au langage Argos et l'optimisation de son compilateur

Le langage Argos est le point commun aux deux axes de travail que nous venons de présenter. Pour les mener à bienΓde nombreuses études de cas sur la programmation en Argos et la vérification des systèmes réactifs ont été réalisées. Celles-ci ont mis en évidence le manque de facilité en Argos pour exprimer des situations courantes dans les systèmes réactifs. Pour répondre à ces besoinsΓnous avons défini deux extensions au langage Argos : les variables de contrôle et les états temporisés. La première permet de décrire plus facilement des automates à structure régulière et certaines formes de communication. La seconde permet de limiter le temps de stationnement dans un état et donc d'intégrer facilement dans les programmes des contraintes temporelles.

Les expériences de programmation de systèmes réactifs faites en Argos ont aussi révélé la faiblesse des performances en temps du compilateur. L'un des rôles du compilateur d'un langage synchrone est de calculer les communications qui existent à l'intérieur du programmeΓet cette phase est critique pour les performances du compilateur. Comme nous l'avons vu dans la présentation informelle du langage ArgosΓle calcul de ces communications repose sur la résolution de systèmes d'équations booléennes. Pour optimiser cette résolutionΓnous sommes passés d'une résolution de type énumérative à une résolution symboliqueΓà l'aide d'un codage du système sous formes de graphes de décision binaires ("binary decision diagrams" ou "bdd" en anglais). Cette première optimisationΓqui théoriquement ne se traduit pas en un gain au niveau temps d'exécution du compilateurΓconduit en pratique à des résultats très satisfaisants. Ces résultats ont encore été améliorés par l'adaptation à la compilation du langage Argos de la méthode des potentielsΓutilisée pour le calcul des communications dans le langage Esterel. L'objectif de cette adaptation est de réduire le nombre d'inconnues des systèmes d'équations booléennes utilisés pour calculer les communications présentes dans un programme. Le temps d'exécution de cette phase de réduction augmente linéairement par rapport au nombre d'événements internes du programme. Si le système d'équations ainsi obtenu n'est pas entièrement résoluΓla méthode de résolution symbolique peut être utilisée pour finir le calcul — elle l'est alors sur un système plus petit que celui initialement calculé —.

7.3.4 Interactions entre ces différents travaux

Remarquons pour terminer que ces différents travaux ne sont pas indépendants :

- Le travail effectué sur le langage Argos est bénéfique au langage ArgoLus et à l'extension d'Argos en termes de graphes temporisés. En particulierΓl'optimisation du compilateur Argos profite directement à la mise en œuvre du langage ArgoLus — par l'intermédiaire de la traduction d'ArgoLus en Argos — et à celle de la sémantique d'Argos en termes de graphes temporisés.

- La traduction structurelle d'Argos en Lustre permet d'utiliser le compilateur Lustre comme compilateur Argos. Des expériences doivent être effectuées pour comparer l'efficacité de ce dernier par rapport au véritable compilateur Argos. De plusΓcette traduction a permis de définir un critère syntaxique de détection des problèmes de causalité trop fort mais qui semble adapté au style de programmation Argos. Ce critère syntaxique pourrait être utilisé pour une nouvelle optimisation du compilateur Argos.

- L'extension de la sémantique d'Argos en termes de graphes temporisés et la traduction des programmes ArgoLus en Argos permettent d'utiliser les possibilités de vérification formelle liées aux graphes temporisés à partir des programmes ArgoLus. Dans ces programmesΓla gestion des contraintes temporelles doit être effectuée par les états temporisés du langage Argos ;

7.4 Perspectives

Les perspectives ouvertes par ce travail sont nombreuses. Tout d'abordΓl'élaboration d'un environnement de programmation synchrone et de vérification de systèmes réactifs multi-langages et multi-outils est un projet à long terme qui ne fait que débuter. DeuxièmementΓla collaboration menée avec le projet de l'INRIA Rhône-Alpes BIP est pleine de promesses et doit non seulement se poursuivreΓmais aussi servir d'exemple à d'autres types de collaborations. Celles-ci doivent permettre de valoriser l'approche synchrone pour la vérification formelle de systèmes dans des domaines où elle est encore peu utilisée. EnfinΓl'extension du langage Argos qui en fait un jeu de constructeurs d'une certaine forme de graphes temporisésΓouvre de nouvelles perspectives à ce langageΓen tant que jeu de constructeurs de diverses formes d'automates.

Nous évoquons ici ces trois directions possibles pour la poursuite de ce travail.

7.4.1 Le développement d'un atelier de programmation synchrone et de vérification multi-outils et multi-langages

La mise en œuvre d'un tel atelier est un projet à long termeΓqui peut s'organiser selon deux axes :

- La poursuite de l'étude des langages impératifs synchrones basés sur les automatesΓet de leur sémantique.

  Le travail sur la sémantique des langages impératifs à base d'automates doit être poursuiviΓ notamment en définissant de nouveaux constructeurs. En particulierΓla définition d'un opérateur de *raffinement généralisé* permettant d'associer l'activité d'une composante non plus à un état d'un automateΓmais par exemple à une configuration d'étatsΓdonnerait la possibilité de décrire plus facilement certains systèmes réactifs.

  Les efforts effectués pour améliorer l'efficacité des compilateurs de ce type de langages – et plus généralement des langages synchrones impératifs — doivent être maintenus. A cet effetΓla définition d'un codage systématique d'un programme Argos sous forme d'un unique système d'équations booléennesΓreprésentés par des graphes de décision binairesΓ permettrait de réaliser un *compilateur symbolique* d'ArgosΓqui pourrait alors être à la base d'un outil de *vérification symbolique* des programmes Argos.

  En dernier lieuΓle langage ArgosΓafin de devenir un véritable langage de programmation des systèmes réactifsΓau même titre qu'EsterelΓLustre et SignalΓdoit être étendu aux entrées et sorties accompagnées de valeurs.

- La généralisation de l'aspect multi-langages dans le cadre de l'approche synchrone.

La définition du langage mixte ArgoLus montre la faisabilité du mélange au niveau source de deux langages synchrones. Cette définition ne constitue qu'une étape dans l'élaboration d'une plate-forme de programmation et de vérification des systèmes réactifs tirant tous les bénéfices du multi-langages dans le cadre de l'approche synchrone. La définition de cette plate-forme nécessite en particulier l'étude des points suivants :

- *Etude du mélange Esterel-Argos.* Bien qu'ayant tous les deux un style impératifΓces deux langages sont suffisamment différents pour que leur mélange au niveau source ait un réel intérêt.
- *Les apports du multi-langages dans la technique de vérification par observateurs.* La technique de vérification par observateurs est une méthode de vérification propre aux langages synchronesΓtrès adaptée à la vérification d'un certain type de propriétés. C'est une méthode qui jusqu'à présent est considérée comme étant *mono-langage* en ce sens que le programme et la propriété sont exprimés dans le même langage. Il nous semble intéressant d'étudier les apports du multi-langages dans cette techniqueΓ principalement pour deux raisons : d'une partΓl'expérience montre que le style d'un langage n'est pas nécessairement adapté à la fois à la description du programme et à celle de l'observateur. Or l'expression d'une propriété est un problème intrinsèquement difficile car il ne faut pas se tromper sur la propriété à vérifier. Par conséquentΓ toute technique qui permet de faciliter cette expression est intéressante ; d'autre partΓ l'utilisation du même langage pour décrire le programme et la propriété nous semble être propice à la duplication dans la propriétéΓdes erreurs faites dans le programme.
- *La définition de traductions structurelles entre les différents langages synchrones.* Pour queΓface à un système réactif donnéΓle seul critère de choix du langage de programmation soit l'adéquation du styleΓil est indispensable d'uniformiser les fonctionnalités offertes par les divers environnements. Une façon d'atteindre cet objectif consiste à définir les traductions structurelles de chaque langage synchrone vers tous les autres.

Ce travail sur les aspects multi-langages doit être réalisé en liaison avec celui portant sur le codage systématique d'Argos en système d'équations booléennes. En effetΓl'existence d'un tel codage pour tous les langages synchrones constituerait une solution au problème de leur intégration. De plusΓle système d'équations booléennes pourrait constituer une forme intermédiaire commune à tous les langagesΓa partir de laquelle des outils communs seraient définis. Un tel codage a été défini pour Lustre restreint aux données booléennes dans [Rat92] ; et plus récemment pour Esterel restreint aux données booléennes.

7.4.2 Valorisation de l'approche synchrone dans de nouveaux domaines

La collaboration avec le projet BIP doit être poursuivie et servir de modèle à d'autres collaborations dans des domaines non liés directement à l'approche synchrone. L'idée pour mener à bien ce transfert de compétences est d'adopter une stratégie "boîte noire"Γc'est-à-dire automatiser les phases de modélisation du problème en Argos temporisé et d'expression de la propriété dans la logique souhaitéeΓpour rendre l'utilisation de l'outil de vérification Kronos transparente à l'utilisateur non spécialiste du domaine. Ce type de stratégie est envisageable dans le cas du système ORCCADΓpour la vérification des schémas de synchronisation. D'autres parties de ce système pourraient bénéficier d'une démarche similaire. Plus généralementΓce travail de modélisation et d'expression de propriétés peut trouver des applications dans d'autres domainesΓcomme

celui des systèmes distribuésΓoù une première expérience sur des problèmes d'ordonnancement a été menée avec le laboratoire LABRI de Bordeaux.

7.4.3 Argos : jeu de constructeurs de diverses formes d'automates

Le langage Argos a été initialement défini pour constituer un jeu de constructeurs de haut niveau d'automates booléens complexesΓbien adapté à la description de la partie contrôle d'un système. L'expérience menée autour des graphes temporisés montre qu'il peut aussi être étendu de manière à définir un jeu de constructeurs de cette forme d'automate avec variables. Elle nous conduit à considérer le langage Argos comme un langage offrant des primitives de haut niveau pour construire des automates complexesΓet ceci de façon quasi-indépendante de la forme des étiquettes des transitions de cet automate. D'autres formes de modèles peuvent donc être atteintes par ce langage. En particulierΓle domaine des systèmes hybrides pourrait bénéficier avantageusement de l'existence d'un tel langage de haut niveau pour décrire les modèles.

Bibliographie

[ACD90] R. AlurΓC. Courcoubetis et D. Dill. Model-checking for real-time systems. Dans *Proceedings of the fifth annual IEEE symposium on Logics In Computer Science*Γ pages 414–425ΓPhiladelphieΓPAΓUSAΓjuin 1990.

[AD90] R. Alur et D. Dill. Automata for modeling real-time systems. Dans M.S. PatersonΓ éditeurΓ*Proceedings of ICALP 90*Γpages 322–335. Springer VerlagΓ1990.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*ΓC-27(6)Γ 1978.

[Alu91] R. Alur. *Techniques for automatic verification of real-time systems.* ThèseΓDepartment of Computer ScienceΓStanford UniversityΓCAΓUSAΓ1991.

[AS86] B. Alpern et F. Schneider. Recognizing safety and liveness. Rapport technique 86-727ΓDepartment of Computer ScienceΓCornell UniversityΓjanvier 1986.

[BB88] T. Bolognesi et E. Brinksma. Introduction to the iso specification language lotos. *Computer Networks and ISDN Systems*Γ14:25–29Γ1988.

[BB91] A. Benveniste et G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE, Vol. 79, No. 9*Γseptembre 1991.

[BBR90] K. S. BraceΓR. E. Bryant et R. L. Rudell. Efficient Implementation of a BDD Package. Dans *Proc. of The 27th Design Automation Conference*Γpages 40–45Γ OrlandoΓFLΓjuin 1990.

[BCG87] G. BerryΓP. Couronné et G. Gonthier. Programmation synchrone des systèmes réactifs Γle langage ESTEREL. *Technique et Science Informatique*Γ4:305–316Γ1987.

[BD88] S. Budkowski et P. Dembinski. An introduction to estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems*Γ14:3–24Γ1988.

[Ber89] G. Berry. Real time programming: Special purpose languges or general purpose languages. *presented at the 11th IFIP World Congress, San Francisco, CA*Γ1989.

[Ber91] G. Berry. A hardware implementation of pure ESTEREL. Dans *ACM Workshop on Formal Methods in VLSI Design, Miami*Γjanvier 1991.

[BG92] G. Berry et G. Gonthier. The Esterel synchronous programming language: DesignΓ semanticsΓimplementation. *Science Of Computer Programming*Γ19(2):87–152Γ 1992.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*ΓC-35(8)Γ1986.

[BS91] F. Boussinot et R. De Simone. The esterel language. *Proceedings of the IEEE, Vol. 79, No. 9*Γseptembre 1991.

[Cas] E. Castillo Castaneda. *Principes, techniques et outils de simulation, vérification et exécution des actions robotiques.* ThèseΓInstitut National Polytechnique de GrenobleΓFranceΓen préparation.

[CBM90] O. CoudertΓC. Berthet et J.-C. Madre. Formal boolean manipulations for the verification of sequential machines. Dans L. J.M. CleasenΓéditeurΓ*IMEC-IFIP Internationnal Workshop on Applied Formal Methods For Correct VLSI Design*ΓNorth HollandΓ1990.

[CES83] E. M. ClarkeΓE. A. Emerson et A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal-logic specifications : a practical approach. Dans *ACM symposium on Principles of Programming Languages*Γpages 117–126Γ1983.

[CES86] E. M. ClarkeΓE. A. Emerson et A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal-logic specifications. Dans *ACM Transactions on Programming Languages and Systems*Γvolume 8Γ1986.

[CHPP87] P. CaspiΓN. HalbwachsΓD. Pilaud et J. Plaice. LustreΓa declarative language for programming synchronous systems. Dans *14th Symposium on Principles of Programming Languages*Γjanvier 1987.

[Cou91] O. Coudert. *SIAM : une boite à outil pour la preuve formelle de systèmes séquentiels.* ThèseΓEcole Normale Supérieure des TélécommunicationsΓoctobre 1991.

[CPS89] R. CleavelandΓJ. Parrow et B. Steffen. *The Concurrency Workbench*Γvolume 407. BerlinΓjuin 1989.

[dSV89] R. de Simone et D. Vergamini. Aboard auto. Rapport Technique 111ΓINRIAΓ1989.

[EC82] E. A. Emerson et E. M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. Dans *Science of Computer Programming*Γvolume 2Γpages 241–266Γ1982.

[Fer88] J.C. Fernandez. *Aldébaran, un système de vérification par réduction de processus communicants.* ThèseΓUniversité Joseph FourierΓGrenobleΓ1988.

[Fer93] J. Cl. Fernandez. Abstract interpretation and verification of reactive systems. Dans *Static Analysis.* LNCS 724 Springer VerlagΓseptembre 1993.

[FGM+92] J.Cl. FernandezΓH. GaravelΓL. MounierΓA. RasseΓC. Rodriguez et J. Sifakis. A tool box for the verification of lotos programs. Dans *14th International Conference on software Engineering*Γmai 1992.

[GBBG85] P. Le GuernicΓA. BenvenisteΓP. Bournai et T. Gauthier. Signal: A data flow oriented language for signal processing. Rapport de rechercheΓIRISA report 246ΓIRISAΓRennesΓFranceΓ1985.

[GGaCLM91] P. Le GuernicΓT. Gautier et M. Le Borgne anf C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE, Vol. 79, No. 9*Γseptembre 1991.

[Ghe92] G. Gherardi. *Sahara: un environnement de mise au point graphique pour les programmes Esterel.* ThèseΓUniversity of NiceΓFranceΓdécembre 1992.

[Gir94] A. Girault. *Sur la répartition de programmes synchrones.* ThèseΓInstitut National Polytechnique de GrenobleΓFranceΓ1994.

[Glo89] A.C. Glory. *Vérification de propriétés de programmes flots de données synchrones.* ThèseΓUniversité Joseph FourierΓGrenobleΓ1989.

[GMP+90] N. GhezalΓS. MatiatosΓP. PiovesanΓY. Sorel et M. Sorine. Un environnement de programmation pour multiprocesseur de traitement du signal. Rapport de recherche 1236ΓINRIAΓ1990.

[Gon88] G. Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones; Application à* ESTEREL. ThèseΓUniversité Paris VIΓFranceΓ1988.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. Dans *5th Conference on Computer-Aided Verification (Elounda, Greece).* LNCS 697ΓSpringer VerlagΓjuin 1993.

[Har87] D. Harel. Statecharts : A visual approach to complex systems. *Science of Computer Programming*Γ8:231–275Γ1987.

[HCRP91a] N. HalbwachsΓP. CaspiΓP. Raymond et D. Pilaud. Programmation et vérification des systèmes réactifs à l'aide du langage flot de données synchrone LUSTRE. *Technique et Science Informatique*Γ10(2)Γ1991.

[HCRP91b] N. HalbwachsΓP. CaspiΓP. Raymond et D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE, Vol. 79, No. 9*Γseptembre 1991.

[HL92] M. Hennessy et H. Lin. Symbolic bisimulation. Rapport technique 1/92ΓSchool of Cognitive and Computing SciencesΓ1992.

[HLR92] N. HalbwachsΓF. Lagnier et C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*Γ29(6/7)Γ1992.

[HLR93] N. HalbwachsΓF. Lagnier et P. Raymond. Synchronous observers and the verification of reactive systems. Dans M. NivatΓC. RattrayΓT. Rus et G. ScolloΓéditeursΓ *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, Twente.* Workshops in ComputingΓSpringer VerlagΓjuin 1993.

[HNSY92] T. HenzingerΓX. NicollinΓJ. Sifakis et S. Yovine. Symbolic model-checking for real-time systems. Dans *LICS'92.* IEEE Computer Society PressΓjuin 1992.

[HP85] D. Harel et A. Pnueli. On the development of reactive systems. Dans *Logic and Models of Concurrent Systems, NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*Γvolume 13. NATO ASI series FΓSpringer VerlagΓ1985.

[HPSS86] D. HarelΓA. PnueliΓJ.P. Schmidt et R. Sherman. On the formal semantics of statecharts. Dans *Symposium on Logic in Computer Science (LICS)*Γpages 54–64Γ1986.

[JLMR94a] M. JourdanΓF. LagnierΓF. Maraninchi et P. Raymond. Embedding declarative subprograms into imperative constructs. Rapport technique Spectre-Lustre L20ΓVerimagΓ1994.

[JLMR94b] M. JourdanΓF. LagnierΓF. Maraninchi et P. Raymond. A multiparadigm language for reactive systems. Dans *5th International Conference on Computer Languages*ΓToulouseΓmai 1994.

[JM92] M. Jourdan et F. Maraninchi. Argos programming examples: The multi-functionΓsix digit lcd watch circuit with alarmΓthree stopwatch modes and event counter. Rapport technique Spectre 44ΓLGI/IMAGΓ1992.

[JM94a] M. Jourdan et F. Maraninchi. A modular state/transition approach for programming reactive systems. Dans *Worshop on Language, Compiler, and Tool Support for Real-Time Systems*ΓOrlandoΓjuin 1994.

[JM94b] M. Jourdan et F. Maraninchi. Studying synchronous communication mechanisms by abstractions. Dans *IFIP Working Conference on Programming Concepts, Methods and Calculi*ΓSan MiniatoΓItalyΓjuin 1994. Elsevier Science Publishers.

[JMO93] M. JourdanΓF. Maraninchi et A. Olivero. Verifying quantitative real-time properties of synchronous programs. Dans *5th International Conference on Computer-aided Verfication*ΓEloundaΓjuin 1993. LNCS 697ΓSpringer Verlag.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. Dans *IFIP 74*. North HollandΓ1974.

[Kap] K. Kapellos. *Environnement de programmation des applications robotiques réactives*. ThèseΓEcole des Mines de ParisΓFranceΓen préparation.

[Ker] A. Kerbrat. *Methodes symboliques pour la verification de processus communicants*. ThèseΓUniversité Joseph FourierΓGrenobleΓFranceΓen préparation.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*ΓSE-3(2):125–143Γ1977.

[Lam83] L. Lamport. What good is temporal logic? Dans R. E. A. MasonΓéditeurΓ*Information Processing 83: proceedings of the* 9th *IFIP World Computer Congress*Γpages 657–668. Elsevier Science PublishersΓ1983.

[Lec89] V. Lecompte. *Vérification automatique de programmes Esterel*. ThèseΓEcole Nationale des Mines de Paris à Sophia-AntipolisΓFranceΓ1989.

[Mad90] C. Madre. *PRIAM : un outil de vérification formelle de circuit intégrés digitaux*. ThèseΓEcole Normale Supérieure des TélécommunicationsΓjuin 1990.

[Maf93] O. Maffeïs. *Ordonnancements de graphes de flots synchrones ; Application à la mise en œuvre de* SIGNAL. ThèseΓUniversité Rennes 1Γ1993.

[Mar90] F. Maraninchi. *Argos : un langage graphique pour la conception, la description et la validation des systèmes réactifs.* ThèseΓUniversité Joseph FourierΓGrenobleΓ 1990.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. Dans *CONCUR*. LNCS 630ΓSpringer VerlagΓaoût 1992.

[Mil80] R. Milner. A calculus of communication systems. Dans *LNCS 92*. Springer VerlagΓ 1980.

[Mil89] R. Milner. Communication and concurrency. Prentice Hall InternationalΓ1989.

[Mou92] L. Mounier. *Méthodes de Vérification de Spécifications Comportementales : étude et mise en œuvre.* ThèseΓUniversité Joseph FourierΓGrenobleΓ1992.

[MSBB99] M. MejiaΓD. SimonΓP. Belmans et J.J. Borrelly. Mécanismes de synchronisation dans un système robotique réparti. Dans *Séminaire Franco-Brésilien sur les systèmes informatiques répartis*Γseptembre 1899.

[MV92] F. Maraninchi et M. Vachon. An experience in compiling a mixed imperative/declarative language for reactive systems. Dans *International Workshop on Compiler Construction (poster session)*. Springer VerlagΓLNCS 641Γoctobre 1992.

[Nic92] X. Nicollin. *ATP : une algèbre pour la spécification et l'analyse des systèmes temps réel.* ThèseΓInstitut National Polytechnique de GrenobleΓFranceΓ1992.

[NRSV90] X. NicollinΓJ.L. RichierΓJ. Sifakis et J. Voiron. ATP: an algebra for timed processes. Dans *IFIP Working Conference on Programming Concepts and Methods*Γ avril 1990.

[NS91] X. Nicollin et J. Sifakis. An overview and synthesis on timed process algebras. Dans K. G. LarsenΓéditeurΓ*Proceedings of the* 3rd *workshop on Computer-Aided Verification*ΓDanemarkΓjuillet 1991.

[NSY91] X. NicollinΓJ. Sifakis et S. Yovine. From ATP to Timed Graphs and Hybrid Systems. Dans J.W. de BakkerΓC. HuizingΓW.P. de Roever et G. RozenbergΓ éditeursΓ*LNCS 600, proceedings of REX Workshop "Real-Time: Theory in Practice"*. MookΓThe Nederlands. Springer VerlagΓjuin 1991.

[NSY92] X. NicollinΓJ. Sifakis et S. Yovine. Compiling real-time specifications into timed automata. *IEEE Transactions on Software Engineering, special issue on Specification and Analysis of Real-Time Systems*Γ1992.

[Oli] A. Olivero. *Modélisation et analyse de systèmes temporisés et hybrides.* ThèseΓ Institut National Polytechnique de Grenoble.

[OS92] A. Olivero et S.Yovine. *Kronos: a Tool for Verifying Real-time Systems. User's Guide and Reference Manual.* VERIMAGΓGrenobleΓFranceΓ1992.

[Pa93] J.P. Paris et al. Les formats communs des langages synchrones. Rapport de recherche Rapport Technique 157ΓINRIAΓ1993.

[Pnu77] A. Pnueli. The temporal logic of programs. Dans *18th Symp. on the Foundations of Computer Science*. IEEEΓ1977.

[Pnu92] A. Pnueli. How vital is liveness? Verifying timing properties of reactive and hybrid systems. Dans *CONCUR'92, Stony Brook*. LNCS 630ΓSpringer VerlagΓaoût 1992.

[PS91] A. Pnuelli et M. Shalev. What is in a step: On the semantics of statecharts. *Lecture Notes en Computer Science, Theoretical Aspects of Computer Science*Γ 526Γ1991.

[QS81] J. Queille et J. Sifakis. Specification and verification of concurrent systems in CESAR. Dans *LNCS 137: Proceedings of 5th International Symposium in Programming*Γpages 337–351. Springer VerlagΓ1981.

[QV86] V. Quint et I. Vatton. Grif: an Interactive System for Structured Document Manipulation. Dans J.C. van VlietΓéditeurΓ*Text Processing and Document Manipulation, Proceedings of the International Conference*Γpages 200–213. Cambridge University PressΓ1986.

[Rat92] C. Ratel. *Définition et réalisation d'un outil de vérification formelle des programmes Lustre : le système* LÉSAR. ThèseΓInstitut National PolytechniqueΓ GrenobleΓ1992.

[Ray91] P. Raymond. *Compilation efficace d'un langage déclaratif synchrone : le générateur de code Lustre-V3*. ThèseΓInstitut National Polytechnique de GrenobleΓ FranceΓ1991.

[RG94] E. Rutten et P. Le Guernic. Sequencing data flow tasks in signal. Dans *Worshop on Language, Compiler, and Tool Support for Real-Time Systems*ΓOrlandoΓjuin 1994.

[Roc92] F. Rocheteau. *Extension du langage Lustre et application à la conception de circuits: Le langage Lustre-V4 et le système Pollux*. ThèseΓInstitut National Polytechnique de GrenobleΓFranceΓ1992.

[Rod88] C. Rodriguez. *Spécification et validation de systèmes en* XESAR. ThèseΓInstitut National PolytechniqueΓGrenobleΓmai 1988.

[Roy90] V. Roy. *Autograph, Un outil de visualisation pour les calculs de processus*. ThèseΓ Université de NiceΓFranceΓ1990.

[RRSV87] J.-L. RichierΓC. RodriguezΓJ. Sifakis et J. Voiron. XESAR*: A Tool for Protocol Validation. User's Guide*. LGI-IMAGΓGrenobleΓFranceΓ1987.

[Sch94] P. Schaar. Un environnement de programmation pour le langage graphique argos. Mémoire de diplôme d'Ingénieur CNAM en InformatiqueΓConservatoire National des Arts et MétiersΓ1994.

[SECK93] D. SimonΓB. EspiauΓE. Castillo et K. Kappellos. Computer-aided design of a generic robot controller handling reactivity and real-time control issues. Dans *IEEE Transactions on Control Systems Technology, vol. 1, no 4*Γdécembre 1993.

[Sha38] C. E. Shannon. A symbolic analysis of relay and switching circuits. *AIEE Transactions*Γ57Γ1938.

[TLS+90] H. J. TouatiΓB. LinΓH. SavojΓK. Brayton et A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machine using BDD's. Dans *IEEE Internationnal Conference on Computer-Aided Design*Γ1990.

[Ver87] D. Vergamini. *Vérification de réseaux d'automates finis par équivalences observationnelles : le système Auto*. ThèseΓUniversité de NiceΓFranceΓ1987.

[Yov93] S. Yovine. *Méthode et outils pour la vérification symbolique de systèmes temporisés*. ThèseΓInstitut National Polytechnique de GrenobleΓmai 1993.

Partie III

Annexes

Annexe A

La traduction structurelle d'Argos vers Lustre

Nous présentons dans cette annexe la traduction structurelle des programmes Argos en programmes Lustre.

Nous supposons dans la suite l'existence d'un ordre sur les événements utilisés dans les programmes Argos (par exemple l'ordre lexicographique). Cet ordre est utilisé pour le calcul des fonctions $\mathcal{I}$ et $\mathcal{O}$ qui ont désormais pour résultats non plus des ensembles d'événements mais des listes ordonnées. Nous définissons sur les listes ordonnées les opérations suivantes :

- **L'union ensembliste** : L1 $\cup$ L2 définit une liste ordonnée qui contient une seule fois l'ensemble des éléments de L1 et L2
- **Le renommage** : L$\{(\alpha_1, \beta_1), \ldots, (\alpha_n, \beta_n)\}$ définit une liste ordonnée obtenue en renommant dans L les occurrences des α_i par β_i.
- **L'élimination** : L$\backslash Y$Γ où Y est un ensemble d'événementsΓ définit une liste ordonnée obtenue en éliminant de L les occurrences des événements présents dans Y.

La fonction de traduction est notée $\mathcal{T}_{A2L}$. Elle transforme un programme Argos principal en un nœud Lustre équivalentΓen raisonnant par induction sur la structure du programme Argos. Un nœud Lustre est défini par un quintuplet <nameΓInΓOutΓLocΓEqs>. Nous nommons le nœud pour pouvoir effectuer des appels (le nœud résultat n'est pas sous forme expansée). Nous utilisons une fonction `NewName` pour générer à chaque appel un nouveau nom de nœud.

Programme principal Argos

$\mathcal{T}_{A2L}((In, Out, Ps)) = < main, I \cup \{\texttt{initial}, \texttt{reset}, \texttt{kill}\}, O, \emptyset, Eqs >$

Eqs est formée de la seule équation suivante :

$O = name(I\{(initial, \texttt{true}), (reset, \texttt{false}), (kill, \texttt{false})\})$

Où $name$ est le nom du nœud défini par $\mathcal{T}_{A2L}(Ps)$.

La mise en parallèle

Le programme Argos à traduire est de la forme $P1 \| P2$. Les éléments qui composent les nœuds résultats de la traduction de $P1$ et $P2$ sont nommés.

- $\mathcal{T}_{A2L}(P1) = < n1, I1, O1, L1, Eqs^1 >$
- $\mathcal{T}_{A2L}(P2) = < n2, I2, O2, L2, Eqs^2 >$

$\mathcal{T}_{A2L}(P1 \| P2) = < \mathtt{NewName}(), I1 \cup I2, O1 \cup O2, \{\alpha_1, \alpha_2 | \forall \alpha \in O1 \cap O2\}, Eqs >$

Eqs est formée des trois équations suivantes :

$$
\begin{array}{l}
O1\{\{(\alpha, \alpha_1) \quad \forall \alpha \in O1 \cap O2\}\} = n1(I1); \\
O2\{\{(\alpha, \alpha_2) \quad \forall \alpha \in O1 \cap O2\}\} = n2(I2); \\
\alpha = \alpha_1 \ \mathtt{or} \ \alpha_2 \quad \forall \alpha \in O1 \cap O2;
\end{array}
$$

La déclaration d'événements internes

Le programme Argos à traduire est de la forme $\overline{P^Y}$. Les éléments qui composent le nœuds résultat de la traduction de P sont nommés.

- $\mathcal{T}_{A2L}(P1) = < n, I, O, L, Eqs >$

$\mathcal{T}_{A2L}(\overline{P^Y}) = < \mathtt{NewName}(), I \setminus Y, O \setminus Y, Y, O = n(I) >$

Le raffinement

Le programme Argos à traduire est de la forme $\mathbf{R}_{(Q, q_{init}, T)}(R1, \ldots, Rn)$. Nous appelons $\mathcal{N}$ l'ensemble des indices d'états de l'automate de contrôle qui sont raffinés par un sous-programme différent de NIL.

Les éléments qui composent les nœuds résultats des traductions des sous-programmes raffinant des états de $\mathcal{N}$ sont nommés.

- $\mathcal{T}_{A2L}(Ri) = < n_i, I_i, O_i, L_i, Eqs_i > \forall i \in \mathcal{N}$

$\mathcal{T}_{A2L}(\mathbf{R}_{(Q, q_{init}, T)}(R1, \ldots, Rn)) = < \mathtt{NewName}(), In \cup \bigcup_{i \in \mathcal{N}} I_i, Out \bigcup_{i \in \mathcal{N}} O_i, Loc, Eqs >$

L'ensemble Loc contient les variables de la forme `atq, entersq` et `exitsq` qui servent à coder l'évolution de l'automate de contrôle. De plusΓcet ensemble contient aussi les variables introduites pour résoudre les problèmes de sorties communes. Le problème des sorties communes se règle en appliquant la stratégie de renommage suivante : toutes les sorties sont renommées en utilisant l'indice 0 si ce sont des sorties de l'automate de contrôle et l'indice i si ce sont des sorties du sous-pro gramme Ri ; chaque sortie α du nœud peut alors être définie comme la disjonction de toutes les variables α_i utilisées dans la liste d'équations. Si cette sortie n'est émise que par une seule composanteΓalorsΓcette disjonction est réduit à un seul élément.

$Loc = \{\mathtt{atq}, \mathtt{exitsq}, \mathtt{entersq} | q \in Q\} \cup \{\alpha_0 | \alpha \in Out\} \bigcup_{i \in \mathcal{N}} \{\alpha_i | \alpha \in Out_i\}$.

La liste d'équations Eqs se compose :

- Des équations qui codent l'activité des états de l'automate de contrôle :

$\forall q \in Q$

```
entersq = (atq'_1 and m_1) or ... or (atq'_n and m_n)
        avec {(atq'_1, m_1), ..., (atq'_n, m_n)} = {(atq', m)|(q', m/o, q) ∈ T}
exitsq = (atq and m_1) or ... or (atq and m_n)
        avec {m_1, ..., m_n} = {m|(q, m/o, q') ∈ T}
Si q = q_init Alors
atq = initial->pre ( ( ( entersq or
                             (atq and not exitsq)
                           )
                           and not kill
                         )
                         or reset
                       )
Sinon
atq = false->pre ( ( ( entersq or
                           (atq and not exitsq)
                         )
                         and not kill
                       )
                       and not reset
                     )
```

- Des équations qui codent les valeurs des sorties de l'automate de contrôle :

$\forall \alpha \in Out$ | $\alpha_0 = (\texttt{atq}_1 \texttt{ and } m_1) \texttt{ or } \ldots \texttt{ or } (\texttt{atq}_n \texttt{ and } m_n)$
avec $\{(\texttt{atq}_1, m_1), \ldots, (\texttt{atq}_n, m_n)\} = \{(\texttt{atq}, m) | (q, m/o, q') \in T \wedge \alpha \in o\}$

- Des équations qui codent les valeurs des sorties des sous-programmes raffinant s :

$\forall i \in \mathcal{N} \mid O\{\{(\alpha, \alpha_i) | \alpha \in Out_i\}\} = name_i(I_i);$

- Des équations pour gérer le problème des sorties communes :

$\forall \alpha \in Out \bigcup_{i \in \mathcal{N}} Out_i$ |
Si $\alpha \in Out$ alors
$\alpha = \alpha_0 \texttt{ or } \alpha_{i1} \texttt{ or} \ldots \texttt{ or } \alpha_{in}$, avec $\{i1, \ldots, in\} = \{i | \alpha \in Out_i\};$
sinon
$\alpha = \alpha_{i1} \texttt{ or} \ldots \texttt{ or } \alpha_{in}$, avec $\{i1, \ldots, in\} = \{i | \alpha \in Out_i\};$

Annexe B

Preuve de la divergence des séquences types

La séquence type est définie en 6.7. Pour simplifier la preuve de la divergence de cette séquenceΓ nous raisonnons sur une abstraction où seuls les pas de durée non nulle sont conservés.

Définition B.1 Abstraction d'une séquence entre états d'un système temporisé
Pour toute séquence $\sigma_{m_e}^{(s,v)} \in \Gamma_{ST} \setminus \Theta_{ST}$, *la séquence abstraite* $\rho(\sigma_{m_e}^{(s,v)})$ *est définie par :*

`Si`	$\sigma_{m_e}^{(s,v)} = (s_0, v_0) \overset{t_0}{\leadsto} (s_0, v_1) \overset{0}{\leadsto} (s_2, v_2) \overset{t_2}{\leadsto} \ldots (s_{2i}, v_{2i}) \overset{t_{2i}}{\leadsto} (s_{2i}, v_{2i+1}) \overset{0}{\leadsto} \ldots,$
`alors`	$\rho(\sigma_{m_e}^{(s,v)}) = (s_0, v_0) \overset{t_0}{\leadsto} (s_2, v_2) \overset{t_2}{\leadsto} \ldots (s_{2i}, v_{2i}) \overset{t_{2i}}{\leadsto} \ldots$
`Si`	$\sigma_{m_e}^{(s,v)} = (s_0, v_0) \overset{0}{\leadsto} (s_1, v_1) \overset{t_1}{\leadsto} (s_1, v_2) \overset{0}{\leadsto} \ldots (s_{2i}, v_{2i}) \overset{0}{\leadsto} (s_{2i}, v_{2i+1}) \overset{t_{2i+1}}{\leadsto} \ldots,$
`alors`	$\rho(\sigma_{m_e}^{(s,v)}) = (s_1, v_1) \overset{t_1}{\leadsto} (s_3, v_3) \overset{t_3}{\leadsto} \ldots (s_{2i+1}, v_{2i+1}) \overset{t_{2i+1}}{\leadsto} \ldots$

■

Propriété B.1 Propriété des séquences abstraites

Si $\sigma_{m_e}^{(s,v)}$ *est une séquence de* $\Gamma_{ST} \setminus \Theta_{ST}$ *alors* $\rho(\sigma_{m_e}^{(s,v)}) = (s_0, v_0) \overset{t_0}{\leadsto} (s_1, v_1) \overset{t_1}{\leadsto} (s_2, v_2) \ldots$ *est telle que :*

$\forall i \in \mathbb{N}$, *si* $\tau = \{t_i, t_{i+1}, \ldots, t_{i+|H|}\}$ *alors* $\exists t \in \tau$ *tel que* $t \geq \frac{1}{|H|}$

■

Démonstration B.1 *Etant donné* $\rho(\sigma_{m_e}^{(s,v)}) = (s_0, v_0) \overset{t_0}{\leadsto} (s_1, v_1) \overset{t_1}{\leadsto} \ldots$, *nous définissons,* H_{min}^i *l'ensemble des horloges qui au* i^{eme} *pas sont les plus proches de leur borne.*

Définition B.2
Soit $\rho(\sigma_{m_e}^{(s,v)}) = (s_0, v_0) \overset{t_0}{\leadsto} (s_1, v_1) \overset{t_1}{\leadsto} \ldots$ *une séquence abstraite, alors*

$H^i_{min} = \{h_j \in H(s_i) | d_j - v_i(h_j) = t_i\}$ *(t_i par construction de la séquence type est la distance minimum des horloges à leur borne (cas 2))*

■

En utilisant le lemme 6.2, les relations suivantes entre valuations des états de la séquence abstraite peuvent être établies.

$$\forall i > 0,$$
$$\begin{aligned} v_i(h_j) &= 0 \text{ ssi } (h_j \in H(s_i) \setminus H(s_{i-1})) \text{ ou } h_j \in H^i_{min} \\ &= v_{i-1}(h_j) + t_{i-1} \text{ sinon} \end{aligned}$$

Considérons à présent l'ensemble τ de la propriété B.1, il contient au moins deux éléments puisque $|H|$ est différent de 0. Par conséquent, il peut toujours être considéré comme l'union de deux ensembles, comme indiqué ci-dessous.

$\tau = \{t_i, \ldots, t_{i+|H|-1}\} \cup \{t_{i+|H|}\}$

Si $\exists t \in \{t_i, \ldots, t_{i+|H|-1}\}$ tel que $t \geq \frac{1}{|H|}$ alors la propriété est démontrée.

Dans le cas contraire, nous allons montrer que $t_{i+|H|}$ est nécessairement supérieur à $\frac{1}{|H|}$. Pour cela nous distinguons deux cas, selon qu'il existe ou pas dans $H^{i+|H|}_{min}$ une horloge qui ait déjà été remise à zéro.

Supposons qu'il existe $h_j \in H^{i+|H|}_{min}$ telle que $h_j \in H^i_{min} \cup H^{i+1}_{min} \cup \ldots \cup H^{i+|H|-1}_{min}$; alors $v_i(h_j) < (|H| - 1) * \frac{1}{|H|}$. En effet, h_j est remise à zéro, puis incrémentée au plus $|H| - 1$ fois d'une valeur inférieure à $\frac{1}{|H|}$.

Par conséquent, $t_{i+|H|} = d_j - v_{i+|H|}(h_j) > \frac{1}{|H|}$, puisque le délai d'une horloge est au minimum égal à 1.

Supposons à présent que $H^{i+|H|}_{min} \cap (H^i_{min} \cup H^{i+1}_{min} \cup \ldots \cup H^{i+|H|-1}_{min}) = \emptyset$.

Nous allons montrer que cette hypothèse est en contradiction avec :

$\forall t \in \{t_i, \ldots, t_{i+|H|-1}\}, t < \frac{1}{|H|}$.

En effet, nous pouvons déduire de $H^{i+|H|}_{min} \cap (H^i_{min} \cup H^{i+1}_{min} \cup \ldots \cup H^{i+|H|-1}_{min}) = \emptyset$, le fait que : $H^i_{min} \cap H^{i+1}_{min} \cap \ldots \cap H^{i+|H|-1}_{min} \neq \emptyset$, puisque ces $|H|$ ensembles contiennent tous au moins un élément pris dans un ensemble qui en contient au plus $|H| - 1$ ($H^{i+|H|}_{min}$ contient au moins un élément, et celui-ci ne fait pas partie des ensembles précédents).

Par conséquent, $\exists h_j \in H^m_{min} \cap H^n_{min}$, avec $m, n \in [i, i + |H| - 1]$.

On en déduit $v_n(h_j) < (|H| - 1) * \frac{1}{|H|}$ et $t_n > \frac{1}{|H|}$, ce qui termine la preuve. ■

Annexe C

Preuve du théorème 6.2

Cette annexe contient la preuve de l'implication des ensembles de transitions dans le théorème 6.2. Ce qui revient à prouver vu la forme des états que :

$$\forall(\overline{p\|_{j\in\mathcal{X}(P^t)}c_j}^{Y}, (s,v)) \in \mathcal{R}el,$$

$$\overline{p\|_{j\in\mathcal{X}(P^t)}c_j}^{Y} \xrightarrow{m_e/O} \overline{p'\|_{j\in\mathcal{X}(P^t)}c'_j}^{Y} \Rightarrow (s,v) \overset{m_e/O}{\triangleright_c} (s',v')$$

avec$(s'=p') \wedge \forall j \in \mathcal{A}ctifs(s')\ \ c'_j = v'(h_j)$

Deux cas particuliers se démontrent très simplement. Ce sont les cas où

- $p = p' \wedge \forall j \in \mathcal{X}(P^t)\ c_j = c'_j \wedge O = \emptyset \wedge \chi \in {m_e}^-$
- $p = p' \wedge \forall j \in \mathcal{X}(P^t)\ c'_j = c_j + 1 \wedge O = \emptyset \wedge \chi \in {m_e}^+$

Dans le premierΓ la règle [$\mathcal{G}$4'] nous assure que $(s,v) \overset{m_e}{\triangleright_c} (s,v)$Γ ce qui suffit à démontrer l'implication.

Dans le secondΓ les conditions d'application de la règle [$\mathcal{G}$1'] sont assurées par $\forall j \in \mathcal{X}(P^t) c'_j = c_j + 1 \wedge \forall j \in \mathcal{A}ctifs(s)\ \ c_j = v(h_j)$Γ d'où $(s,v) \overset{m_e}{\triangleright_c} (s, v+1)$Γ ce qui permet de démontrer l'implication.

La preuve de cette implication dans tous les autres cas s'effectue par induction sur la structure du programme temporisé. Le cas de base est donné par le programme égal à NILΓil se démontre immédiatement. Nous ne présentons ici que le cas de l'opérateur de mise en parallèle.

Commençons par remarquer que

$$\overline{p1\|p2\|_{j\in\mathcal{X}({P^t}_1\|{P^t}_2)}c_j}^{Y} = \overline{p1\|_{j\in\mathcal{X}({P^t}_1)}c_j}^{Y1}\|\overline{p2\|_{j\in\mathcal{X}({P^t}_2)}c_j}^{Y2}$$

où $Y1$ et $Y2$ forment une partition de Y selon que les états temporisés concernés font partie de ${P^t}_1$ ou ${P^t}_2$.

D'après la règle [Par] de la fonction sémantique $\mathcal{S}$Γ

$$\overline{p1\|_{j\in\mathcal{X}(P^t{}_1)}c_j}^{Y1}\|\overline{p2\|_{j\in\mathcal{X}(P^t{}_2)}c_j}^{Y2}i \xrightarrow{m_e/O} \overline{p1'\|_{j\in\mathcal{X}(P^t{}_1)}c'_j}^{Y1}\|\overline{p2'\|_{j\in\mathcal{X}(P^t{}_2)}c'_j}^{Y2} \Longrightarrow$$

$$\overline{p1\|_{j\in\mathcal{X}(P^t{}_1)}c_j}^{Y1} \xrightarrow{m_e[I1]/O1} \overline{p1'\|_{j\in\mathcal{X}(P^t{}_1)}c'_j}^{Y1} \wedge \overline{p2\|_{j\in\mathcal{X}(P^t{}_2)}c_j}^{Y2} \xrightarrow{m_e[I2]/O2} \overline{p2'\|_{j\in\mathcal{X}(P^t{}_2)}c'_j}^{Y2}$$

avec $O = O1 \cup O2$.

L'appartenance du couple $(\overline{p1\|p2\|_{j\in\mathcal{X}(P^t{}_1\|P^t{}_2)}c_j}^{Y}, (s1\|s2, \mathtt{v}))$ à la relation $\mathcal{R}el$ nous permet de déduire Γsi on note $\mathtt{v}_1$ la valuation $\mathtt{v}$ restreinte aux horloges de $P^t{}_1$Γl'appartenance du couple

$(\overline{p1\|_{j\in\mathcal{X}(P^t{}_1)}c_j}^{Y1}, (s1, \mathtt{v}_1))$ à la relation $\mathcal{R}el_1$ associée à $P^t{}_1$.

Le même type de raisonnement s'applique au deuxième opérande de la mise en parallèle.

Par conséquentΓil est possible d'appliquer l'hypothèse d'induction. On en déduit

$(s1, \mathtt{v}_1) \overset{m_e[I1]/O1}{\triangleright_c} (s1', \mathtt{v}'_1) \wedge (s2, \mathtt{v}_2) \overset{m_e[I2]/O2}{\triangleright_c} (s2', \mathtt{v}'_2)$

avec $s1' = p1' \wedge \forall j \in \mathcal{A}ctifs(s1')\ c'_j = \mathtt{v}'_1(h_j) \wedge s2' = p2' \wedge \forall j \in \mathcal{A}ctifs(s2')\ c'_j = \mathtt{v}'_2(h_j)$

Il faut à présent étudier les conséquences de l'existence de ces deux pas concrets sur les graphes temporisés modèles de $P^t{}_1$ et $P^t{}_2$Γet ce grâce à la fonction $\mathcal{G}$'.

Les deux cas particuliers traités ci-dessus correspondent respectivement aux cas où les deux pas concrets proviennent tous les deux soit de l'application de [$\mathcal{G}$4'] (premier cas)Γ soit de l'application de [$\mathcal{G}$1'] (second cas). Il faut maintenant étudier les 4 autres cas possiblesΓ qui sont :

- Les deux pas concrets proviennent de l'application de [$\mathcal{G}$3'].
- Les deux pas concrets proviennent de l'application de [$\mathcal{G}$2'].
- Un des deux pas concrets provient de l'application de [$\mathcal{G}$2']Γl'autre de [$\mathcal{G}(1)$'].
- Un des deux pas concrets provient de l'application de [$\mathcal{G}$3']Γl'autre de [$\mathcal{G}(4)$'].

Seuls ces quatre cas sont possibles car soit χ appartient à $m_e[I1]^+$ et à $m_e[I2]^+$Γ soit il appartient à $m_e[I1]^-$ et à $m_e[I2]^-$. Nous ne détaillons que le premierΓles autres se traitent de façon similaire.

Si $(s1, \mathtt{v}_1) \overset{m_e[I1]/O1}{\triangleright_c} (s1', \mathtt{v}'_1)$ est obtenue par application de [$\mathcal{G}$3']Γalors il existe nécessairement dans l'ensemble des arcs du graphe temporisé modèle de $P^t{}_1$Γun arc $(s1, m_{h1}, m_e[I1], O1, \mathcal{R}1, s1')$ tel que $v'_1 = v_1[\mathcal{R}1 := 0]$ et $v_1 \models m_{h1}$.

Le même raisonnement est valable pour le second pas concretΓil existe donc dans le graphe temporisé modèle de $P^t{}_2$ un arc $(s2, m_{h2}, m_e[I2], O2, \mathcal{R}2, s2')$ tel que $v'_2 = v_2[\mathcal{R}2 := 0]$ et $v_2 \models m_{h2}$.

D'après la règle [Par^t] (c.f. 6.1.6)Γla présence de ces deux arc permet de conclure à la présence de l'arc $(s1\|s2, m_{h1} \bullet m_{h2}, m_e, O, \mathcal{R}1 \cup \mathcal{R}2, s1'\|s2')$ dans le graphe temporisé modèle de $P^t{}_1\|P^t{}_2$.

La règle [$\mathcal{G}$3'] peut s'appliquer car :

$v_1 \models m_{h1} \wedge v_2 \models m_{h2} \quad \Longrightarrow \quad v \models m_{h1} \bullet m_{h2}\Gamma$

On obtient donc le pas concret suivant :

$(s1\|s2, v) \overset{m_e/O}{\triangleright_c} (s1'\|s2', v')$ avec $v' = v[\mathcal{R}1 \cup \mathcal{R}2 := 0]$.

En utilisant les résultats intermédiaires suivants :

- $s1' = p1' \wedge s2' = p2'$
- $\forall j \in \mathcal{A}ctifs(s1') \;\; c'_j = \mathtt{v}'_1(h_j) \;\wedge\; \forall j \in \mathcal{A}ctifs(s2') \;\; c'_j = \mathtt{v}'_2(h_j)$
- $v'_1 = v_1[\mathcal{R}1 := 0] \;\wedge\; v'_2 = v_2[\mathcal{R}2 := 0]$
- $v' = v[\mathcal{R}1 \cup \mathcal{R}2 := 0]$

on peut conclure à l'appartenance du couple $(\overline{p'1\|p'2\|_{j\in\mathcal{X}(P^t{}_1\|P^t{}_2)}c_j}^{Y}, (s1'\|s2', v'))$ à la relation $\mathcal{R}el$.