

# THÈSE

PRÉSENTÉE À

## L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET  
INFORMATIQUE**Par Gérald Point**

POUR OBTENIR LE GRADE DE

### DOCTEUR

SPÉCIALITÉ : Informatique

---

## **AltaRica : Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement**

---

**Soutenu le :** 20 janvier 2000  
**Après avis de :** Nicolas Halbwachs Rapporteurs  
Olivier Roux . . . . .

**Devant la Commission d'examen formée de :**

André Arnold	Professeur . . . . .	<i>Président, Rapporteur</i>
Henrik Reif Andersen	Professeur . . . . .	<i>Examineur</i>
Didier Bégay	Directeur de Recherche (CNET)	<i>Examineur</i>
Jean-Pierre Braquelaire	Professeur . . . . .	<i>Examineur</i>
Nicolas Halbwachs	Directeur de Recherche (CNRS)	<i>Examineur</i>
Antoine Rauzy	Chargé de Recherche (CNRS) . .	<i>Directeur de thèse</i>
Olivier Roux	Professeur . . . . .	<i>Examineur</i>
Jean-Claude Hochon	Ingénieur (IXI) . . . . .	<i>Invité</i>
Tony Hutinet	Ingénieur (IXI) . . . . .	<i>Invité</i>
Jean Gauthier	Ingénieur (Dassault Aviation) . .	<i>Invité</i>

— 2000 —



# RÉSUMÉ

---

Les méthodes formelles et la sûreté de fonctionnement sont deux domaines connexes qui s'intéressent à l'analyse des comportements des systèmes critiques. Ces domaines adoptent des points de vue différents mais complémentaires sur les systèmes. Les méthodes formelles considèrent le point de vue fonctionnel et adoptent, en général, une approche « vérification de programme ». Dans ce domaine on cherche en général à mettre en évidence un scénario menant le programme à un bogue, ou à générer de manière automatique des programmes sûrs (par rapport à leurs spécifications). La sûreté de fonctionnement considère plutôt les aspects dysfonctionnels des systèmes. Dans ce domaine on cherche à déterminer les scénarios prépondérants qui mènent le système à une défaillance ou à évaluer des mesures probabilistes sur ses comportements (fiabilité, disponibilité, ...).

Les travaux présentés dans cette thèse ont été réalisés dans le cadre d'un projet industriel, AltaRica, qui ambitionne le rapprochement des méthodes formelles et de la sûreté de fonctionnement. Cette unification se concrétise par le développement d'un atelier d'analyse système, l'atelier AltaRica, qui fédèrera à terme un ensemble de modèles et d'outils pour l'analyse des systèmes. Cet atelier propose une représentation unique pour la description des systèmes ; celle-ci étant destinée à être compilée vers des modèles/outils existants. Ce rapport présente le formalisme supporté par cet atelier, sa forme textuelle et graphique (le langage AltaRica), certaines propriétés de sa sémantique et quelques exemples de modélisations.

L'étude des scénarios de panne est un des principaux problèmes de la sûreté de fonctionnement. Ce problème est généralement traité en utilisant le modèle des arbres de défaillances. Ce modèle ne permettant pas de prendre en compte le séquençement des pannes, cette thèse propose une solution au problème de l'obtention des scénarios de panne minimaux pour l'ordre des sous-mots.



# REMERCIEMENTS

---

Aux membres du jury. En particulier Henrik Reif Andersen, Jean-Pierre Braquelaire, Nicolas Halbwachs, Olivier Roux et Jean-Claude Hochon pour leur participation active à ma soutenance de thèse.

Aux partenaires industriels des différents projets auxquels j'ai eu l'occasion de participer. En particulier je souhaiterais remercier Jean Gauthier (Dassault Aviation) et Jean-Pierre Signoret (ELF Exploration-Production) qui ont motivé de manière constante et constructive mes travaux de recherches.

À la société IXI et à son équipe AltaRica : Sylvain Lajeunesse, Tony Hutinet, Romuald Winckell, Christophe Lansade, Hervé Antoniol, Jérôme Arrault et Nicolas Denis.

À David Sherman, Macha Nikolskaia, Jean-Michel Couvreur et Pierre Castéran pour leur gentillesse et leur aide ponctuelle. Aux membres de la CVT (Cellule de Valorisation et de Transfert). À Pierre-André Wacrenier pour les nombreuses discussions autour d'une « blanche » et cette fameuse soirée du 19 janvier 2000. Et bien entendu à Philippe Thomas qui m'a accompagné dans toutes les rigolades et les galères.

À mes maîtres et amis. À André Arnold pour ces longues discussions sur la sémantique du langage AltaRica. À Alain Griffault qui a toujours été de bon conseil tout au long de ces trois années de recherches et qui s'est énormément investi dans le projet AltaRica m'apportant ainsi son soutien moral et scientifique. À Didier Bégay qui, après avoir été mon mentor en DEA, a continué à me soutenir, malgré son absence, pendant les moments les plus difficiles de ma thèse. Je le remercie pour tous ses conseils et encouragements depuis la fin de ma maîtrise jusqu'au jour de ma soutenance de thèse (supportant à l'occasion ma mauvaise humeur passagère). À mon directeur et « grand frère », Antoine Rauzy, qui fût un exemple tant par son pragmatisme d'ingénieur-chercheur que par sa rigueur scientifique.

À ma grande famille et à Agnès qui m'ont poussé, malgré les difficultés, jusqu'au bout de ces longues années d'études.

Finalement à  $\mu$  qui fût plus un taux de réparation qu'un plus petit point fixe d'une fonction monotone.



# TABLE DES MATIÈRES

---

<b>1</b>	<b>CONTEXTE DE L'ÉTUDE</b>	<b>9</b>
1.1	L'analyse des systèmes : un domaine bipolaire . . . . .	9
1.2	Objectifs et motivations . . . . .	10
1.3	Structure du document . . . . .	12
	<b>Partie I : L'analyse des systèmes</b>	<b>13</b>
<b>2</b>	<b>INTRODUCTION</b>	<b>15</b>
2.1	Des systèmes critiques . . . . .	15
2.2	Des analyses différentes . . . . .	16
2.3	Deux approches, un objectif . . . . .	16
2.4	Une démarche commune . . . . .	17
2.5	Un aperçu de quelques travaux . . . . .	18
<b>3</b>	<b>LA MAÎTRISE DU RISQUE</b>	<b>21</b>
3.1	Les modèles booléens . . . . .	21
3.2	Des modèles pour les systèmes dynamiques . . . . .	26
<b>4</b>	<b>LA VÉRIFICATION FORMELLE</b>	<b>29</b>
4.1	Modélisation des systèmes . . . . .	29
4.2	Formulation de propriétés . . . . .	31
4.3	Vérifier les formules pour un modèle . . . . .	33
<b>5</b>	<b>LA PROGRAMMATION « SÛRE »</b>	<b>35</b>
5.1	La méthode B : de la spécification à l'implémentation du logiciel . . . . .	36
5.2	Systèmes réactifs : des langages asynchrones . . . . .	37
5.3	Systèmes réactifs : les langages synchrones . . . . .	40
<b>6</b>	<b>CONCLUSION</b>	<b>47</b>
	<b>Partie II : AltaRica</b>	<b>49</b>
<b>7</b>	<b>INTRODUCTION : LE PROJET ALTAERICA</b>	<b>51</b>
7.1	AltaRica : Positionnement industriel . . . . .	51
7.2	AltaRica : Positionnement scientifique . . . . .	52
7.3	Plan de cette partie . . . . .	53

<b>8</b>	<b>LE LANGAGE ALTARICA</b>	<b>55</b>
8.1	Les composants . . . . .	55
8.2	La hiérarchie . . . . .	61
8.3	Modèles graphiques . . . . .	72
<b>9</b>	<b>SÉMANTIQUE</b>	<b>77</b>
9.1	Systèmes de transitions interfacés . . . . .	77
9.2	Sémantique des composants . . . . .	79
9.3	Sémantique des nœuds . . . . .	81
9.4	Sémantique symbolique . . . . .	83
9.5	AltaRica et les automates temps-réel . . . . .	84
<b>10</b>	<b>QUELQUES MODÈLES DE SYSTÈMES</b>	<b>89</b>
10.1	Conception d'un ascenseur . . . . .	89
10.2	Régulation du niveau d'une cuve . . . . .	94
10.3	Réduction par bisimulation . . . . .	98
<b>11</b>	<b>SIMULATION ET ANALYSES QUALITATIVES</b>	<b>103</b>
11.1	La simulation des modèles AltaRica . . . . .	103
11.2	Analyse qualitative de séquences . . . . .	104
11.3	Vers les formules booléennes . . . . .	111
<b>12</b>	<b>CONCLUSIONS ET PERSPECTIVES</b>	<b>115</b>
12.1	Conclusions . . . . .	115
12.2	Perspectives . . . . .	115
<b>13</b>	<b>BIBLIOGRAPHIE</b>	<b>119</b>
	<b>Partie III : Annexes</b>	<b>131</b>
<b>A</b>	<b>PREUVES DES CHAPITRES PRÉCÉDENTS</b>	<b>133</b>
A.1	Preuves du chapitre 9 . . . . .	133
A.2	Preuves du chapitre 11 . . . . .	136
<b>B</b>	<b>GRAMMAIRE DU LANGAGE ALTARICA</b>	<b>143</b>



## CONTEXTE DE L'ÉTUDE

Les travaux présentés dans ce rapport ont été réalisés dans le cadre d'une convention CIFRE (Convention Industrielle de Formation à la REcherche) entre la société IXI (Ingénierie Concourante et Systèmes d'Information) et le Ministère de l'Éducation Nationale, de la Recherche et de la Technologie. Les activités de recherche de l'auteur se sont déroulées au sein de l'équipe MVTsi (Modélisation, Vérification et Test des Systèmes Informatisés) du LaBRI (Laboratoire Bordelais de Recherche en Informatique).

Cette thèse se situe dans le cadre très général de la validation des systèmes industriels. Dans ce contexte la collaboration entre la société IXI et l'équipe MVTsi ambitionne l'unification de deux approches pour l'étude des systèmes critiques.

### 1.1 L'ANALYSE DES SYSTÈMES : UN DOMAINE BIPO-LAIRE

Ce rapport a été rédigé avant le passage de l'an 2000. Sa version électronique est naturellement stockée sur un micro-ordinateur. Autant dire que l'auteur a quelques soucis pour le passage à la nouvelle année. Bien que le risque de perte de données soit certainement négligeable, l'auteur a prévu un archivage du présent rapport sur différents supports (papiers, disquettes, ...). Que de méfiance pour une personne qui a fait de l'informatique son métier.

Le problème du passage à la dernière année du millénaire a mis en évidence ce que des ingénieurs de l'industrie s'efforcent de faire depuis de nombreuses années (sauf, peut-être au début de l'informatique) : se prémunir des dysfonctionnements de leurs systèmes. Ainsi, des ingénieurs experts ont pour métier, non seulement de concevoir des systèmes (informatiques ou autre), mais aussi de garantir le bon fonctionnement de ce qu'ils ont conçu par des justifications autres que leur propre savoir-faire.

Que ce soit par leurs tailles ou les technologies impliquées, les systèmes industriels deviennent de plus en plus complexes. Malgré l'expertise des concepteurs, il est devenu déraisonnable de déployer un système industriel sans s'assurer au préalable de son bon fonctionnement.

De par leurs complexités croissantes, les systèmes industriels ne peuvent plus être étudiés « à la main » par les ingénieurs/concepteurs. De ce fait, l'analyse des comportements des systèmes s'est automatisée. Ainsi, de nombreux travaux de recherches ont porté sur la définition de modèles comportementaux et le développement d'outils logiciels assistant les ingénieurs pour la compréhension et la détection des mauvais

fonctionnements de leurs systèmes.

Historiquement il existe deux grands pôles d'activités dans le domaine de l'analyse des systèmes :

- La *sûreté de fonctionnement* regroupe un ensemble de méthodes (en général probabilistes) visant à mesurer le risque inhérent à un système. Dans ce domaine on considère que la défaillance du système est inévitable. Cette défaillance est acceptée sous certaines contraintes ; par exemple, si la défaillance ne se produit que très rarement. L'objectif de la sûreté de fonctionnement est de déterminer, dès l'étape de conception, si ces contraintes sont respectées ou non par le système.
- La *vérification formelle* (traditionnellement académique) est une branche de l'informatique qui s'efforce de garantir au mieux la correction des programmes (séquentiels, distribués, embarqués, . . . ). L'informatique a depuis longtemps migré dans l'industrie et de nos jours de nombreux programmes se trouvent « embarqués » sur des systèmes livrés à eux-mêmes (p. ex. un satellite). De tels logiciels ne peuvent être intégrés à un système sans une vérification préalable de ses exécutions (p.ex. Ariane 5).

Ces deux domaines d'activités se retrouvent dans la collaboration entre la société IXI et l'équipe MVTsi :

- La sûreté de fonctionnement est un des premiers métiers de la société IXI. Dans ce domaine la société mène des actions d'études ou de conseils pour le compte de ses clients. Par ailleurs, la société soutient des activités de recherches et de développements d'outils pour la sûreté de fonctionnement (p.ex. cette thèse) ;
- L'équipe MVTsi est orientée vers la vérification et le test des systèmes. Cette équipe, intéressée tant par les aspects liés à la modélisation qu'aux aspects algorithmiques de la vérification, s'est dotée d'outils de validation/vérification de modèles tels que MEC (mise en œuvre du modèle de Arnold et Nivat) ou TOUPIE ( $\mu$ -calcul sur les domaines finis). Par ailleurs, cette équipe a acquis une expertise dans le domaine de la modélisation formelle.

Cette collaboration entre IXI et le LaBRI, deux organismes aux domaines d'activités connexes, ne pouvait que tendre vers la convergence de la sûreté de fonctionnement et de la vérification formelle.

## 1.2 OBJECTIFS ET MOTIVATIONS

En décembre 1996, les sociétés ELF Exploration production, IXI et l'équipe MVTsi ont initié un projet qui ambitionne d'unifier à terme la sûreté de fonctionnement et la vérification formelle autour d'un seul outil : l'atelier d'analyse système ALTARICA.

La thèse présentée dans ce rapport a pour objectif de définir un cadre formel préalable au développement des outils qui seront intégrés dans l'atelier.

### 1.2.1 DE FIABEX À ALTARICA

L'atelier FIABEX[1] a été précurseur dans le domaine des ateliers système. Cet atelier dédié à l'évaluation d'architectures système a été développé dans le cadre d'un

## 1.2. OBJECTIFS ET MOTIVATIONS

consortium industriel (CNES, CEP, COGEMA, ELF, IBM, MATRA ESPACE) à partir des résultats de recherche issus du projet Européen de recherche FIABEX (projet EUREKA n° 85).

Le modèle FIABEX repose sur la description comportementale fonctionnelle et dysfonctionnelle d'une architecture système (représentation graphique unifiée Matérielle/Fonctionnelle) dont les composants élémentaires sont des processeurs de flux (décrits sous forme de bases de connaissance élémentaire). Ce modèle dynamique qualitatif est exploité grâce à un moteur d'inférence intégrant un mécanisme de gestion de la cohérence des valeurs inférées de type ATMS (Assumptions Truth Maintenance System).

L'atelier FIABEX permet à partir de la modélisation d'une architecture système de générer l'ensemble des modèles spécifiques à la sûreté de fonctionnement (AMDEC, arbres de défaillance, réseaux de Petri stochastiques, . . . ) utilisés pour l'évaluation des performances système. Son développement s'est réparti sur une dizaine d'années en fonction des besoins des partenaires (module de simulation, module de génération de modèles SdF, module d'aide au diagnostic). Ce logiciel manquait toutefois d'une formalisation robuste de son modèle comportemental. Ce manque s'est fait ressentir dès que les concepteurs ont commencé à compiler les descriptions vers les modèles pour les outils de calculs. Du fait de l'absence de sémantique formelle pour les modèles FIABEX, la correction des algorithmes de compilation ne pouvait être garantie que par l'expertise des concepteurs de FIABEX.

Dans ce contexte, le projet ALTARICA avait pour objectif initial la formalisation du modèle comportemental utilisé dans FIABEX. Cette tâche de formalisation avait pour but la définition d'un langage de modélisation (FIABEX n'en possédant pas) et dont la sémantique serait celle de l'atelier.

Malheureusement, au bout de quelques mois, nous avons été contraints d'abandonner le modèle FIABEX (du fait du refus de celui-ci par l'un des copropriétaires de l'atelier FIABEX). Il ne restait que les prémisses du langage ALTARICA et les quelques modélisations effectuées avec ce modèle. Les objectifs du projet AltaRica ont été modifiés et orientés vers le développement du successeur de FIABEX : l'atelier ALTARICA.

### 1.2.2 DE MEC À ALTARICA

MEC est l'outil mettant en œuvre le modèle de Arnold et Nivat. Cet outil a démontré que ce modèle, bien que très simple, permettait de modéliser et de vérifier un nombre important de systèmes (avec les restrictions que l'on connaît dues au nombre d'états du modèle global). Ce modèle a toujours souffert de son manque d'expressivité pratique ; il est vrai que pour un système comportant un grand nombre de composants, l'écriture de la contrainte de synchronisation était une tâche laborieuse et génératrice d'erreurs.

Le projet AltaRica a été une occasion pour étendre le modèle Arnold-Nivat afin d'accroître son pouvoir d'expression. Par ailleurs, le développement d'un vérificateur de modèle (*model-checker*) basé sur le langage AltaRica est un point « stratégique » pour la diffusion des méthodes formelles dans le milieu industriel.

### 1.2.3 ANALYSE QUALITATIVE DES SCÉNARIOS DE PANNE

En marge des travaux sur la sémantique du langage AltaRica, nous nous sommes intéressés à l'étude des scénarios de défaillance des systèmes. Ce sujet est un thème clas-

sique de la sûreté de fonctionnement qui n'est pas abordé par la vérification formelle (il ne s'agit pas en effet d'une démarche de vérification).

Habituellement ce problème est traité en utilisant des modèles essentiellement booléens tels que les *arbres de défaillance* (qui représentent par une formule booléenne l'ensemble des états de panne d'un système). L'approche des arbres de défaillance étudie totalement les aspects temporels (logiques) des scénarios menant le système dans un état non souhaité.

Un autre objectif de nos travaux a été d'initier une approche analogue à l'analyse qualitative des arbres de défaillance mais en nous plaçant dans le cadre de la théorie des langages (un langage représentant un ensemble de scénarios de panne). Sur ce thème, nous nous sommes principalement intéressés au codage des séquences et au calcul des *coupes* qui sont les scénarios minimaux menant à la panne du système.

### 1.3 STRUCTURE DU DOCUMENT

Ce rapport comporte deux parties. La première partie est une présentation succincte des modèles et/ou méthodes utilisés dans le domaine de la sûreté de fonctionnement ou dans le domaine de la vérification formelle. Cette partie est moins un état de l'art des méthodes qu'un instantané (non exhaustif) des méthodes rencontrées (ou recensées) au cours de cette thèse.

La seconde partie du rapport est consacrée au langage de modélisation AltaRica. Favorisant la compréhension du modèle plutôt que sa théorie nous consacrons deux chapitres à la description de ce langage. Le chapitre 8 présente de manière informelle les principaux concepts mis en œuvre dans le langage. Le chapitre 9 aborde la sémantique formelle du langage; dans ce chapitre nous montrons en particulier la compositionnalité du modèle. Le chapitre 10 illustre le langage sur quelques exemples. Finalement, au chapitre 11, nous présentons quelques considérations algorithmiques liées à la simulation du langage, puis une structure de donnée permettant le codage et l'étude qualitative des scénarios de défaillance extraits d'un modèle AltaRica.

# **Partie I**

## **L'analyse des systèmes**





## INTRODUCTION

Les systèmes industriels deviennent de plus en plus complexes. De par leurs dimensions ou leurs hétérogénéités technologiques, il est désormais impossible d'appréhender tous les fonctionnements ou les dysfonctionnements d'un système. Un des soucis majeurs de ceux qui spécifient, réalisent, opèrent et maintiennent de tels systèmes est de prouver dès les premiers stades de la conception, leur aptitude à remplir de façon sûre les missions qui leurs sont confiées. Ce souci rejoint par ailleurs une volonté généralisée d'amélioration de la productivité et de réduction des cycles de développement.

Dans ce contexte de nombreux outils mathématiques et informatiques sont proposés afin de permettre aux ingénieurs d'analyser d'une manière la plus exhaustive possible les comportements de ces systèmes dits *critiques*.

Un système est dit *critique* dès lors que les conséquences de ses dysfonctionnements ne sont plus négligeables. Suivant les systèmes, les conséquences peuvent être économiques (p. ex. la destruction du système), écologiques (p. ex. la pollution de la nappe phréatique) ou humaines (p. ex. l'explosion d'un avion en vol). Le passage de l'an 2000 a certainement été une prise de conscience collective de la criticité des systèmes (informatisés).

### 2.1 DES SYSTÈMES CRITIQUES

Les figures 2.1 et 2.2 représentent deux *systèmes* qui pourraient s'avérer critiques :

- Le premier (figure 2.1) est un système qui, par un jeu de pompes et de vannes, vide un premier réservoir pour remplir un second. Il suffit d'imaginer que le second réservoir soit le circuit de refroidissement d'une centrale nucléaire pour prendre conscience que même quelques vannes et quelques pompes peuvent être considérées comme des éléments critiques d'un système.

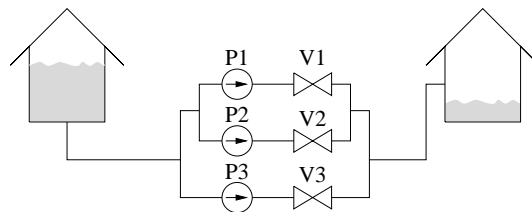


Figure 2.1: Un système de remplissage d'un tank composé de trois pompes  $P_i$  et trois vannes  $V_i$  [2].

- Le second est une petite fonction (écrite en C) qui élève son paramètre  $N$  à la puissance 3 sans utiliser la multiplication. Bien que les connaissances de l'auteur soient très limitées pour ce qui concerne le domaine de l'avionique, si ce programme est intégré à un calculateur de vol d'un 747, les passagers apprécieraient certainement que l'on soit sûr que cette fonction termine et retourne bien le résultat attendu.

```
int cube(int N) {
    int n = N, x, c = 0, d = 1, e = 6;
    if(n<0) n = -n;
    for( x=0; x != n; x++ ) {
        c += d; d += e; e += 6;
    }
    if(N<0) c=-c;
    return c;
}
```

Figure 2.2: Une fonction C calculant  $N^3$  [3].

## 2.2 DES ANALYSES DIFFÉRENTES

Les deux exemples présentés précédemment sont analysés de manières différentes :

- Dans le premier cas, l'ingénieur/concepteur cherchera à déterminer les scénarios de panne du système et, à partir de données techniques sur les composants (les pompes, les vannes, ...), il établira un bilan probabiliste sur les fréquences d'occurrences de ces scénarios. Les données techniques sont recueillies à partir du retour d'expérience ou fournies par le constructeur du composant. Le bilan sur les scénarios de panne du système permet d'évaluer la sûreté de l'installation et autorise les prises de décisions sur le choix des composants ou les mesures de sécurité supplémentaires à mettre en oeuvre. Cette analyse a pour objectif la diminution du risque inhérent au système.
- Pour la fonction C, le travail à effectuer est de prouver que l'algorithme est correct et que la mise en oeuvre en langage C respecte l'algorithme. D'un point de vue plus général, l'ingénieur aura à réaliser un travail identique sur tout le programme. Toutefois il est certain que ce travail ne prouve pas que le programme fonctionnera correctement une fois implanté dans le calculateur (il suffit d'imaginer les étapes nécessaires pour passer d'un algorithme à un logiciel embarqué – le compilateur a-t-il été vérifié? Le système d'exploitation est-il sûr?). Cette démarche vise à garantir que le système remplit bien sa fonction.

D'un point de vue plus global, ces deux types d'analyses peuvent (doivent?) être rencontrées lors de l'étude d'un même système; il est désormais fréquent de rencontrer des systèmes possédant des composantes tant logicielles que matérielles.

## 2.3 DEUX APPROCHES, UN OBJECTIF

Dans le domaine de l'analyse des systèmes critiques, deux communautés cohabitent celle de la *sûreté de fonctionnement* (SdF) et celle de la *vérification formelle* (VF). La

## 2.4. UNE DÉMARCHE COMMUNE

nuance est plutôt subtile mais elle permet de distinguer deux approches : le fonctionnel et le dysfonctionnel.

Dans le domaine de la SdF, les ingénieurs s'attachent principalement à *maîtriser* le risque inhérent à des systèmes complexes. La SdF considère que le système tombera certainement en panne un jour ; les problèmes à résoudre sont alors d'identifier : comment le système peut-il tomber en panne ? Quelle est la fréquence de cette panne ? Quelles sont les conséquences de la panne du système ? . . .

La vérification formelle est orientée vers les aspects fonctionnels du système. Ce système a une fonction à remplir. Cette fonction étant généralement très complexe, il est nécessaire de vérifier si elle est bien réalisée (ou parfois réalisable) par le système. En adoptant un point de vue inspiré de la preuve de programme, la vérification formelle s'intéresse, d'une manière générale, à tous les aspects comportementaux d'un système.

Bien que les techniques d'analyse diffèrent, les études sur les systèmes critiques ont la même finalité : avoir un certain niveau de *confiance* en le système. Il y a bien longtemps que les industriels considèrent que le *risque nul*<sup>1</sup> n'est qu'un mythe. Aujourd'hui, le mieux que l'on puisse faire est de maîtriser et gérer les risques liés à des systèmes de plus en plus complexes. En effet, il ne suffit pas de montrer qu'un système est correct d'un point de vue fonctionnel (il réalise ce pour quoi il a été conçu) mais il est nécessaire d'envisager les raisons et les conséquences de ses *dysfonctionnements*.

## 2.4 UNE DÉMARCHE COMMUNE

- *Forty-two!* yelled Loonquawl. *Is that all you've got to show for seven and a half million years' work?*

- *I checked it very thoroughly, said the computer, and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is.*

Douglas Adams

*The Hitch Hicker's Guide to the Galaxy*

Que l'on considère le point de vue fonctionnel ou dysfonctionnel, la démarche mise en œuvre pour l'analyse d'un système suit les grandes lignes présentées ci-dessous (cf. figure 2.3).

- Déterminer la portée de l'étude : que cherche-t-on à analyser ? Quels aspects (propriétés) du système veut-on étudier ?
- Trouver le modèle mathématique (ou la méthode) qui permettra l'analyse du système. Le choix du modèle, évidemment essentiel, est à la charge des ingénieurs. Ce choix est influencé par :
  - L'expressivité du modèle : est-ce que le modèle permet de décrire l'aspect étudié ?
  - Les possibilités de calcul : existe-t-il des outils informatiques permettant de traiter le problème ? Combien de temps est nécessaire pour obtenir le résultat ?

---

<sup>1</sup>À ne pas confondre avec le *zéro défaut* qui est un concept lié à la qualité et non de fiabilité. Voir [4] (pages 36 et 37) pour une comparaison entre ces deux notions.

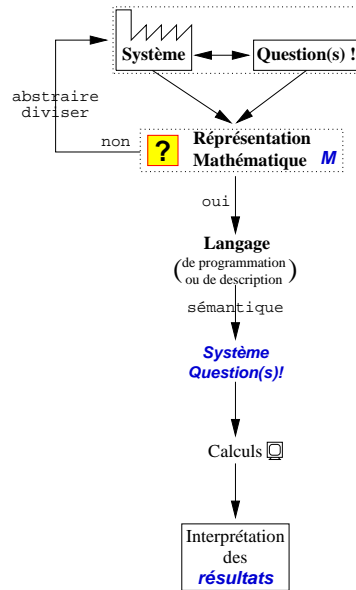


Figure 2.3: Démarche générale pour l'analyse d'un système

- L'adéquation avec l'aspect étudié : est-ce que tous les aspects requis par le modèle sont nécessaires pour la propriété étudiée? N'existe-t-il pas un modèle plus grossier qui soit suffisant?
- Le modèle  $M$  et l'outil informatique étant choisis, il ne reste plus qu'à décrire le système et les propriétés que l'on désire étudier. Cette description est réalisée dans le langage supporté par l'outil. La sémantique du langage transforme la description du système en une structure de donnée représentant le système par un modèle  $M$ . La qualité de cette transformation, totalement transparente à l'ingénieur/concepteur, est fondamentale mais délicate : en effet, c'est souvent lors de cette phase qu'apparaissent les problèmes d'explosion combinatoire qui saturent la mémoire des machines ou requièrent un temps de calcul déraisonnable.
- Les calculs terminés, l'ingénieur doit évidemment traiter les réponses de l'outil. L'interprétation des résultats est une étape délicate car il s'agit, pour l'ingénieur, d'établir d'une part, le lien entre le modèle et les résultats et d'autre part, le lien entre le modèle, la réponse et le système réel. C'est à partir de cette étape que les prises de décision seront effectuées (bien que dès la modélisation certains défauts de conception soit détectables).

## 2.5 UN APERÇU DE QUELQUES TRAVAUX

Dans la suite de ce rapport nous ne présentons qu'un petit sous-ensemble des modèles et méthodes employés pour étudier les systèmes. Il eut été bien prétentieux de résumer en quelques pages un domaine qui fait intervenir tellement de métiers (chimie, mécanique, électronique, logiciel, ...), de paramètres d'évaluation (coûts financiers, vies humaines, écologie, ...) et de travaux de recherches.

## 2.5. UN APERÇU DE QUELQUES TRAVAUX

Il nous a semblé pertinent de présenter dans un même document les points de vue et méthodes tant de la sûreté de fonctionnement que de la vérification formelle. Cette thèse se situe donc à la croisée des chemins de la sûreté de fonctionnement et de la vérification formelle. L'ambition de ce rapport est d'établir une nouvelle passerelle entre les deux communautés ; il s'agit notamment d'une excellente occasion pour présenter les travaux de l'une à l'autre.

La suite de cette partie est organisée de la manière suivante : le chapitre 3 présente des modèles probabilistes que l'on rencontre fréquemment dans l'industrie ; le chapitre 4 présente des techniques issues du monde académique pour la vérification de propriétés comportementales ; le chapitre 5 aborde des aspects orientés vers le génie logiciel en présentant des modèles et techniques pour la génération automatique de programmes vérifiés.





## LA MAÎTRISE DU RISQUE

La *maîtrise du risque* est un terme très générique pour désigner un ensemble de problématiques en *sûreté de fonctionnement*. Pour reprendre la définition de [5] (page 743), le *risque* est « une mesure d'un danger associant une mesure de l'occurrence d'un événement indésirable et une mesure de ses effets ou conséquences ». Ainsi les méthodes et outils à mettre en œuvre pour maîtriser le risque inhérent à un système dépendent fortement de l'événement non souhaité et de la mesure associée à ses conséquences. Cette dernière est en général une fonction de coût (financier, humain ou écologique).

Le risque possède deux dimensions. La maîtrise de la première est la plus naturelle car elle vise à diminuer (en général) la fréquence d'occurrence de la défaillance du système. La seconde dimension est apparue lorsque, avec le retour d'expérience, les industriels ont pris conscience qu'il était inutile de concevoir des systèmes trop sécurisés et, par voie de conséquence, trop onéreux. Les ingénieurs *fiabilistes* ont donc à jouer sur deux paramètres : la fiabilité (ou une autre mesure) du système et le coût nécessaire à l'amélioration de cette fiabilité.

Comme le ferait très justement remarquer un ingénieur fiabiliste, il existe des méthodes en sûreté de fonctionnement qui ne font appel à aucun outil informatique (ou du moins elles peuvent s'en passer). C'est le cas, par exemple, des APR (Analyse Préliminaire des Risques) ou des AMDEC (Analyse des Modes de Défaillances des composants, de leurs Effets sur le système et de leur Criticité). Bien que non formalisées mathématiquement, ces méthodes précèdent en général les études assistées par ordinateur. Elles permettent d'établir de manière détaillée une espèce de « carte routière » des fonctionnements et des dysfonctionnements du système et de déceler d'éventuels problèmes de conception<sup>1</sup>.

Dans ce chapitre, nous présentons quelques modèles liés à la première dimension du risque c'est-à-dire les mesures d'occurrences d'un événement non souhaité. Les méthodes citées dans le paragraphe précédent ne sont pas présentées. De même, nous n'abordons pas la maîtrise du coût dont l'automatisation nécessite des techniques d'optimisation combinatoire (p.ex. [7]).

### 3.1 LES MODÈLES BOOLÉENS

Dans cette section nous considérons qu'un composant, un sous-système ou le système lui-même possèdent uniquement deux états : marche et panne. Nous présentons deux

---

<sup>1</sup>Dans [6], il est conseillé de pratiquer la méthode AMDEC en groupe de travail où l'ensemble des participants à la construction du système sont présents (ingénieurs, techniciens, ...). Cette démarche vise à confronter les différents points de vue sur le système et à lever les incertitudes liées aux aspects métiers.

modèles : les diagrammes de fiabilité et les arbres de défaillance. Si ces représentations décrivent, tout deux, des formules booléennes (on parle de modèles booléens), il diffèrent principalement sur trois points :

**La méthode** qui détermine le raisonnement à suivre pour construire le modèle ;

**La représentation graphique** qui est plus ou moins induite par la démarche ;

**La sémantique** associée à la formule. Suivant les méthodes, la formule booléenne représente le bon fonctionnement (p. ex. les diagrammes de fiabilité) ou le dysfonctionnement du système (p. ex. les arbres de défaillance).

Le choix d'un type de modélisation dépend de plusieurs facteurs :

- le mode de raisonnement de l'ingénieur : il peut préférer une démarche inductive (causes/conséquences) ou déductive (conséquences/causes) ;
- les contraintes imposées par son environnement de travail : dans certains domaines (p. ex. le nucléaire) les méthodes employées sont imposées par les organismes de tutelle ou les procédures de développement mises en place dans l'entreprise.

### 3.1.1 LES DIAGRAMMES DE FIABILITÉ

Un *diagramme de fiabilité* modélise les conditions de bon fonctionnement du système ou de réalisation d'une de ses fonctions. Le système est représenté par un circuit électrique comportant un générateur  $E$ , un récepteur  $S$  et un ensemble d'interrupteurs ; le courant est supposé se propager de  $E$  à  $S$ . Chaque interrupteur modélise un élément (ou une fonction) du système.

La figure 3.1 donne un exemple de diagramme de fiabilité. Un arc  $E_1 \rightarrow E_2$  entre deux interrupteurs indique que le bon fonctionnement de  $E_2$  dépend du bon fonctionnement de  $E_1$ . Un interrupteur est ouvert lorsque l'élément associé ne remplit pas sa fonction. Le système est en bon fonctionnement si le récepteur  $S$  est alimenté ; en d'autres termes, si il existe dans le circuit un chemin de  $E$  à  $S$  ne contenant que des interrupteurs fermés ; un tel chemin est appelé un *chemin de succès*.

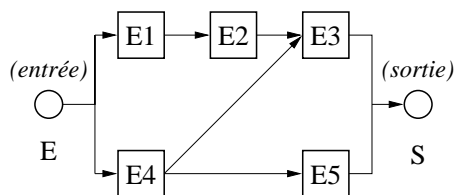


Figure 3.1: Un diagramme de fiabilité.  $E$  est l'entrée du circuit,  $S$  sa sortie et les  $E_i$  sont les interrupteurs.

La démarche proposée pour la modélisation par diagramme de fiabilité consiste en l'identification :

- De la sortie du circuit ( $S$  sur la figure 3.1) qui représente la fonction étudiée (p. ex. le refroidissement d'un moteur) ;
- Des interrupteurs (les  $E_i$  sur la figure 3.1) qui représentent les éléments du système (p. ex. les vannes, les pompes, les sources d'eau, . . . ).

### 3.1. LES MODÈLES BOOLÉENS

- Des arcs (entre les  $E_i$ ) qui représentent les dépendances de fonctionnement entre les éléments (p. ex. une pompe remplit sa fonction si la source d'eau n'est pas tarie).

Dans la mesure où la modélisation mène à un graphe dirigé sans cycle<sup>2</sup>, la transformation d'un diagramme de fiabilité en une formule booléenne est relativement aisée. Pour chaque noeud  $N$  du diagramme (hormis l'entrée  $E$ ) on effectue les opérations suivantes :

- On associe à tout noeud  $N$ , différent de  $S$ , une variable booléenne  $x_N$ . Cette variable vaut *vrai* si le composant associé à  $N$  n'est pas en panne.
- On associe une formule  $f_N$  définie par:  $f_N = x_N \wedge (f_{N_1} \vee \dots \vee f_{N_k})$  où les  $N_i$  ( $i = 1 \dots k$ ) sont les noeuds, différents de  $E$ , dont la fonction associée à  $N$  dépend (c-à-d. il existe dans le diagramme un arc  $N_i \rightarrow N$ ).

Cette formule exprime le fait que  $N$  remplit sa fonction :

1. L'élément  $N$  n'est pas en panne ( $x_N$  doit valoir *vrai* pour que  $f_N$  vaille vraie);
2. Il existe un  $N_i$  (dont dépend  $N$ ) qui fonctionne.

La formule à étudier est celle associée à  $S$  (c'est-à-dire  $f_S$ ). Pour le diagramme de la figure 3.1 nous obtenons les formules :

- $f_S = f_{E_3} \vee f_{E_5}$
- $f_{E_3} = x_{E_3} \wedge (f_{E_2} \vee f_{E_4})$
- $f_{E_5} = x_{E_5} \wedge f_{E_4}$
- $f_{E_2} = x_{E_2} \wedge f_{E_1}$
- $f_{E_1} = x_{E_1}$
- $f_{E_4} = x_{E_4}$

Le bon fonctionnement du système est donc modélisé par la formule représentée sur la figure 3.2.

#### 3.1.2 LES ARBRES DE DÉFAILLANCE

Le modèle des *arbres de défaillance* [9] est utilisé pour représenter les états de dysfonctionnement du système.

La démarche suggérée par la méthode des arbres de défaillance consiste à décrire de manière descendante les combinaisons d'événements (au sens des probabilités) qui provoquent l'événement redouté. Chaque étape de la construction consiste à raffiner un événement par un raisonnement déductif qui détermine quelles combinaisons d'événements contribuent à la réalisation de l'événement à raffiner. Les combinaisons sont exprimées à l'aide d'opérateurs booléens (pour la plupart) [10]. Ce processus est itéré jusqu'à la granularité souhaitée; les événements terminaux (les feuilles de l'arbre) sont supposés indépendants (au sens des probabilités).

<sup>2</sup>Il existe un modèle de diagrammes comportant des cycles. On les appelle des *réseaux de fiabilité*. Dans [8] il est montré comment transformer un tel réseau en une formule booléenne.

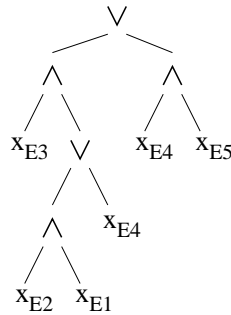


Figure 3.2: Formule booléenne  $f_S$  obtenue à partir du diagramme de fiabilité de la figure 3.1.

La description graphique des arbres de défaillance est à peu de choses près l'arbre syntaxique de la formule booléenne mais décoré de commentaires. Étant donné le nombre de composants élémentaires pouvant intervenir dans un système réel, l'arbre est inévitablement volumineux.

Les feuilles de l'arbre sont les variables de la formule et représentent la défaillance des composants élémentaires ; la variable associée au composant vaut *vrai* si celui-ci est en panne (dans la pratique, plusieurs variables peuvent être associées à un même composant, chacune représentant un « mode » de défaillance différent). La figure 3.3 représente l'arbre de défaillance équivalent<sup>3</sup> au diagramme de fiabilité de la section précédente (figure 3.1). Le lecteur remarquera qu'il s'agit de la formule de la figure 3.2 où les opérateurs ont été remplacés par leur dual ( $\wedge$  transformé en  $\vee$  et inversement) ; ceci n'est pas surprenant dans la mesure la formule associée à l'arbre représente de dysfonctionnement tandis que celle associée au diagramme de fiabilité représente le bon fonctionnement.

### 3.1.3 L'ANALYSE DES MODÈLES BOOLÉENS POUR LA SDF

Parmi les différentes méthodes d'analyse nous distinguons les plus courantes, à savoir :

- Le calcul des implicants premiers : il s'agit de déterminer les combinaisons d'événements qui inhibent le bon fonctionnement du système. Sur un système complexe, le nombre de ces combinaisons est gigantesque. Pour faciliter l'analyse des résultats, on ne s'intéresse habituellement qu'aux implicants premiers. Un implicant premier est une combinaison entraînant le dysfonctionnement qui ne contient pas dans une autre combinaison (entraînant elle aussi le dysfonctionnement). Le nombre d'implicants premiers est lui-même excessivement grand (dans [11], les auteurs donne l'exemple d'un arbre possédant plus de  $10^{11}$  implicants premiers). À l'heure actuelle les algorithmes de calcul d'implicants s'intéressent à des calculs tronqués (par rapport à un seuil de probabilité ou au nombre d'événements contenu dans la combinaison) [11, 12].
- Le calcul de la probabilité des événements : la probabilité qu'une variable soit *vraie* à un instant  $t$  étant connue, il est possible de calculer la probabilité pour que le modèle global vaille *vrai* au même instant.

<sup>3</sup>L'utilisation du terme « équivalent » est un abus de langage. En effet, le diagramme de fiabilité représente le bon fonctionnement du système tandis que l'arbre de défaillance modélise son dysfonctionnement.

### 3.1. LES MODÈLES BOOLÉENS

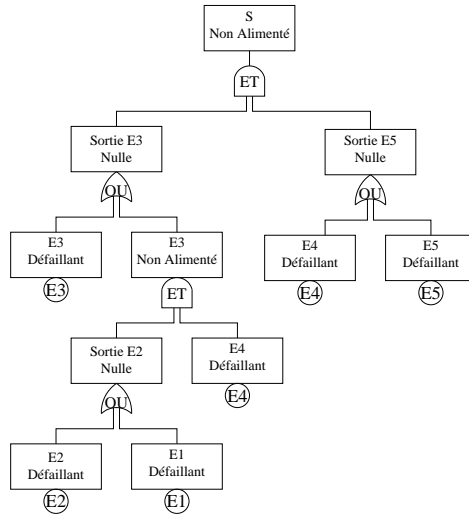


Figure 3.3: L'arbre de défaillance correspondant au diagramme de fiabilité de la figure 3.1.

Les outils supportant les modèles booléens de la SdF sont partagés en deux familles : les éditeurs de modèles et les moteurs de calcul (il existe toutefois des éditeurs intégrant complètement le moteur de calcul). Nous citerons deux des plus importants : RISK SPECTRUM/RSAT [13] et ARALIA WORKSHOP[14] (successeur de SIMTREE/ARALIA). Nous ne discuterons pas ici des interfaces graphiques des deux outils. RSAT et ARALIA [15, 11] sont les moteurs de calcul qui se distinguent tant par leur philosophie que par leur technologie. Alors que ARALIA privilégie le calcul exact, RSAT privilégie les temps de calcul en utilisant des approximations.

Le point fort d'ARALIA est le calcul exact des mesures probabilistes désirées. L'exactitude des calculs est assurée par l'utilisation de structures de données performantes, à savoir, les diagrammes binaires de décision (BDD [16, 17, 18]). À ce jour, les BDD sont considérés comme l'état de l'art pour le codage des formules booléennes. Paradoxalement, cette technologie est le talon d'Achille d'ARALIA car il est désormais bien connu que les performances des BDD sont conditionnées par un ordre fixé sur les variables de la formule [17, 19]. Il est à noter que de nombreux travaux de recherche visent à contourner ce problème par l'utilisation d'heuristique de réécriture de formule et d'ordonnement de variables (p. ex. [20]).

RISK SPECTRUM permet parfois d'aller plus loin et plus rapidement qu'ARALIA. Ces performances sont principalement due à l'utilisation d'approximations sur les calculs de probabilités et/ou la limitation du nombre d'implicants premiers stockés. L'inconvénient majeur de RISK SPECTRUM est l'obligation pour l'ingénieur de rester dans les hypothèses imposées par les approximations (faibles probabilités des événements de base, ... ).

## 3.2 DES MODÈLES POUR LES SYSTÈMES DYNAMIQUES

Les modèles présentés dans la section précédente permettent de représenter des propriétés statiques des systèmes. Ce modèles ne conviennent plus lorsque l'on considère des systèmes dynamiques dont les caractéristiques évoluent au cours du temps (prise en compte des réparations, des tests, des maintenances, . . . ). L'étude probabiliste de tels systèmes reposent sur le modèle des *processus stochastiques*.

### 3.2.1 PROCESSUS STOCHASTIQUES

Une *variable aléatoire*  $X$  est une fonction qui associe à un résultat d'une *expérience aléatoire* une certaine valeur dans  $\mathbb{R}$ . La fonction inverse  $X^{-1}$  de  $X$  nous donne l'ensemble  $E$  des résultats de l'expérience tels que  $X$  ait une certaine valeur  $x \in \mathbb{R}$ . La probabilité de  $E$  est la probabilité que  $X$  vaut  $x$ ; ainsi, l'expérience aléatoire considérée peut être étudiée par le biais de la probabilité associée à  $X$  (ou sa fonction de répartition qui est la probabilité que  $X \leq x$ ).

Un *processus stochastique* [10] est un ensemble de variables aléatoires  $X_t$  indexées par un paramètre  $t$  assimilé (en général) au temps. Pour un  $t$  donné, la variable  $X_t$  modélise une expérience aléatoire à l'instant  $t$ . Un processus stochastique modélise donc l'évolution au cours du temps du caractère aléatoire d'une expérience donnée. (Les processus de Markov sont des cas particuliers de processus stochastiques.)

Dans le cadre de la sûreté de fonctionnement, on considère un phénomène aléatoire caractérisé par un processus  $X_t$  qui dépend d'un ensemble fini de variables aléatoires  $X_t^1, \dots, X_t^n$  dont on connaît les fonctions de répartition respectives.

### 3.2.2 LES PROCESSUS DE MARKOV

Un *processus de Markov* [10] est un cas particulier de processus stochastique. Ce modèle décrit le système par un graphe d'états. Chaque sommet du graphe est un état global du système et une transition entre deux états indique que le système peut passer de l'état origine (de la transition) à l'état but. Les transitions sont étiquetées par des *taux de transitions*  $\lambda(t)$  tels que  $\lambda(t)dt$  soit la probabilité de passer de l'état à l'origine de la transition à son état but, entre les instants  $t$  et  $t + dt$ . En général, ce modèle est utilisé lorsque les  $\lambda(t)$  sont constants (le modèle est dit *markovien*). Dans ce cas les calculs sont « relativement » aisés puisqu'ils relèvent de la résolution d'un système d'équations différentielles du premier ordre et à coefficients constants.

L'inconvénient majeur de ce modèle est qu'il requière la description du graphe d'état global. Malheureusement, il est très difficile de construire ce graphe « à la main » sans introduire d'erreurs (il est toutefois possible de générer un graphe de Markov à l'aide d'un modèle de plus haut niveau). De plus des problèmes de stockage et d'approximation [2] apparaissent si l'on considère des systèmes réels. Dans [10], les auteurs conseillent son utilisation uniquement sur des petits sous-systèmes du système global.

### 3.2.3 LA SIMULATION DE MONTE-CARLO

La *méthode de Monte-Carlo* permet d'évaluer les caractéristiques d'un processus stochastique  $X_t$  (moyenne, variance, . . . ) en simulant le comportement des phénomènes décrits par les  $X_t^i$  au cours du temps.

### 3.2. DES MODÈLES POUR LES SYSTÈMES DYNAMIQUES

L'algorithme de simulation dépend complètement du comportement de  $X_t$  en fonction des  $X_t^i$ . Deux approches existent :

- L'implémentation d'un algorithme de simulation *ad hoc* construit directement à partir de la structure et des composants physiques du système;
- L'utilisation d'un algorithme générique construit à partir d'un modèle de comportement; par exemple les réseaux de Petri. Le modèle définit les règles du « jeu » nécessaires à la simulation. L'algorithme de simulation peut alors être appliqué à n'importe quel type de système (dès lors qu'il est modélisé en utilisant le formalisme choisi).

Cette méthode est fréquemment rencontrée lorsque :

- Le processus stochastique est markovien mais l'ensemble des états est trop important pour être stocké ou pour appliquer les méthodes analytiques usuelles ;
- Le processus est non markovien ce qui interdit l'utilisation des méthodes analytiques.





## LA VÉRIFICATION FORMELLE

Dans ce chapitre nous proposons un tour d'horizon des modèles et techniques utilisées pour la vérification formelle. Bien qu'il s'agisse de thèmes importants, nous n'aborderons pas les travaux qui concernent la synthèse de contrôleur (contrôler un système de manière à ce qu'il vérifie certaines contraintes), les assistants de preuve (vérification de système par la démonstration de théorèmes) ou la génération automatique de tests (extraction de séquences de tests à partir d'un modèle formel). Le lecteur pourra consulter [21, 22] pour une introduction beaucoup plus détaillée sur la vérification de modèle (appelée souvent *model-checking*).

Dans la première section nous présentons des modèles de représentation des comportements d'un système. L'objectif étant de vérifier des propriétés du système, nous décrivons dans la deuxième section les principes des formalismes permettant de décrire une propriété comportementale. Enfin, dans la troisième section, nous décrivons les principales techniques utilisées pour montrer qu'un modèle possède une propriété donnée.

### 4.1 MODÉLISATION DES SYSTÈMES

#### 4.1.1 LES SYSTÈMES DE TRANSITIONS

Le modèle des *systèmes de transitions* (ST) décrit les comportements d'un programme (ou d'une machine) par deux ensembles d'objets : les *états* et les *transitions*. Un état représente une valuation d'une certaine caractéristique du système : le compteur ordinal d'un programme, l'altitude d'un avion, le fonctionnement ou l'arrêt d'un automatisme, le niveau d'un liquide dans une cuve . . . Une transition exprime le passage d'une valuation à une autre.

Parmi les variantes des systèmes de transition on distingue :

- Les *systèmes de transition étiquetés* [22] associent à chaque transition une étiquette qui représente une action du système ou un événement. Cette étiquette est une *nom* permettant d'identifier le phénomène provoquant le changement d'état.
- Les *machines de Mealy* [23] associent à chaque transition deux étiquettes représentant un stimulus et la réaction à ce stimulus : dans un état donné, si le stimulus apparaît alors le système effectue l'action spécifiée et se retrouve dans l'état but de la transition. (Ce modèle est souvent utilisé pour représenter des circuits.)
- . . .

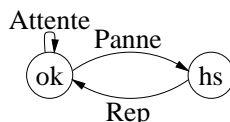


Figure 4.1: Un système de transition pour un système tombant en panne (Panne) et réparable (Rep).

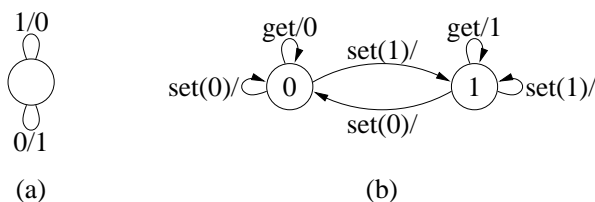


Figure 4.2: Deux machines de Mealy : (a) une porte logique NON, (b) une mémoire.

Le modèle des systèmes de transition (ainsi que ses variantes) est très fréquemment utilisé dans le domaine de la vérification car la sémantique de la plupart des langages de modélisation s'exprime dans ce modèle.

Puisqu'un système de transition décrit l'ensemble des états d'un système, il n'est pas envisageable de modéliser directement un système réel par un seul système de transition. C'est pour cette raison que toutes les méthodes formelles adoptent une approche cartésienne consistant à décomposer le système en sous-systèmes, à modéliser ces sous-systèmes en utilisant un formalisme donné puis à composer les modèles des sous-systèmes afin d'obtenir celui du système global. L'opération de composition des modèles élémentaires est essentielle et caractérise les modes de communication et d'interaction des sous-systèmes (diffusion, rendez-vous, produit synchronisé).

#### 4.1.2 LES SYSTÈMES DE TRANSITIONS TEMPORISÉS

Alors que les systèmes de transition sont étiquetés uniquement par des noms (d'actions ou d'événements), les *systèmes de transitions temporisés* [24] utilisent deux ensembles disjoints d'étiquettes : des noms d'événements et des délais (des réels positifs ou nuls). Une transition étiquetée par un délai  $d$  modélise le fait que le temps s'écoule de  $d$  unités de temps. Ce modèle suppose d'une part l'instantanéité des actions (ou des événements) et d'autre part la cohérence des délais avec la notion du temps : déterminisme, continuité et non évolution pour un délai nul.

Dans la pratique, les systèmes de transition temporisés servent de sémantique à un modèle de plus haut niveau, l'*automate temporisé* (appelé aussi automate d'Alur et Dill [24]). Les automates temporisés sont dotés d'*horloges* qui peuvent être remises à zéro lorsqu'une action est effectuée par l'automate. Sous certaines conditions, l'automate peut, soit changer d'état par une action soit rester dans le même état en laissant le temps s'écouler pendant une durée  $d$ ; dans ce cas, les horloges sont toutes incrémentées de  $d$ .

Le modèle des automates temporisés a été étendu (en particulier dans UPPAAL [25]) afin de prendre en compte des notions de priorités (*urgent channel*), d'invariants d'états, et d'états *engagés* (*committed states*) desquels le système est « engagé » à sortir immédiatement (c-à-d. dans un délai nul).

## 4.2. FORMULATION DE PROPRIÉTÉS

Dans le modèle d'Alur et Dill les horloges avancent à la même vitesse. Le modèle des automates *hybrides* considèrent l'hypothèse contraire [26, 27]. Ainsi à chaque état est associée une équation différentielle spécifiant les variations des variables d'horloges.

### 4.1.3 LES ALGÈBRES DE PROCESSUS

Les *algèbres de processus* considèrent un ensemble de processus élémentaires qui sont composés à l'aide d'opérateurs. L'algèbre la plus connue est CCS (*Calculus of Communicating Systems*) définie par Milner [28]. Le cadre algébrique de CCS donne une assez bonne intuition de ce qu'est un processus : un processus élémentaire (p.ex. une action) ou deux processus qui sont composés pour en former un troisième. Les algèbres de processus contiennent au minimum les opérations de composition nécessaires pour décrire un programme séquentiel (p.ex. séquençement d'instructions, choix et récursion). Afin de prendre en compte la notion de parallélisme, Milner a introduit une opération de *composition parallèle* de deux processus. Cette opération définit les interactions possibles entre deux processus s'exécutant en parallèle et comment ces interactions influencent les comportements futurs des processus. Les différentes versions de CCS se distinguent principalement par la sémantique de cet opérateur (p.ex. ACCS, SCCS, CBS sont les versions asynchrones, synchrones et à diffusion de CCS).

$$\begin{aligned}V0 &= ((\text{set}0+\text{get}0).V0)+(\text{set}1.V1) \\V1 &= ((\text{set}1+\text{get}1).V1)+(\text{set}0.V0) \\M &= V0\end{aligned}$$

Figure 4.3: Expression de la mémoire (initialisée à 0) de la figure 4.2(b) sous la forme d'un processus  $M$ . Un processus de la forme  $a.P$  peut exécuter  $a$  puis se comporter comme  $P$ . Un processus de la forme  $P_1 + P_2$  se comporte soit comme  $P_1$  soit comme  $P_2$ .

## 4.2 FORMULATION DE PROPRIÉTÉS

Avant d'étudier une propriété d'un système il est nécessaire de la formaliser. Dans cette section nous présentons des objets mathématiques permettant de décrire des propriétés comportementales d'un système. Nous verrons en particulier que ces objets sont liés au modèle du système (p.ex. un système de transition).

### 4.2.1 LES LOGIQUES TEMPORELLES

Les logiques temporelles [29] sont très nombreuses mais à l'heure actuelle deux d'entre elles servent de références (et sont certainement les plus utilisées) : la logique temporelle linéaire LTL (*Linear Temporal Logics*), et la logique temporelle arborescente CTL (*Computation Tree Logic*).

Les logiques temporelles se distinguent par l'ensemble d'objets  $O$  dont elles décrivent les propriétés. Les formules de LTL expriment des propriétés de suites d'éléments d'un ensemble  $M$ . Pour la logique CTL, les propriétés portent sur des arbres dont les noeuds appartiennent à l'ensemble  $M$ . Les éléments de  $O$  sont donc de la forme  $(M, S)$  où  $S$  est une structure décrivant un objet construit à partir des éléments de  $M$ . Ces

logiques considèrent de plus un ensemble  $P$  de propositions élémentaires qui sont associées aux éléments de  $M$  par une application  $V : M \rightarrow 2^P$  (appelée *interprétation*): pour tout  $x \in M$ ,  $V(x)$  est l'ensemble des propositions élémentaires vérifiées par  $x$ . (Par exemple, si l'on considère que  $M$  est l'ensemble des saisons {hiver, printemps, été, automne} et si  $P$  contient deux propositions  $p_1 = \text{« il fait chaud »}$  et  $p_2 = \text{« il fait froid »}$  alors l'application  $V$  pourrait être  $V(\text{hiver}) = \{p_2\}$ ,  $V(\text{printemps}) = \{p_1, p_2\}$ ,  $V(\text{été}) = \{p_1\}$ ,  $V(\text{automne}) = \{p_1, p_2\}$ .)

Les formules de la logique considérée sont construites à partir de l'ensemble  $P$ , d'opérateurs de la logique propositionnelle (négation, conjonction, disjonction) et d'opérateur temporels (p.ex. dans LTL, l'opérateur  $X$  fait référence à l'instant – logique – suivant). La valeur de vérité d'une formule  $F$  pour un objet  $o \in O$  dépend de l'interprétation  $V$  (pour les saisons, on pourrait considérer qu'en automne il ne fait pas chaud:  $V(\text{automne}) = \{p_2\}$ ). Un *modèle* d'une formule  $F$  est un objet de  $O$  qui satisfait la formule  $F$  pour une certaine interprétation  $V$ . Les modèles de  $F$  sont donc des objets de la forme  $(M, S, V)$ .

Dans le cadre de la vérification de système, l'ensemble des objets  $O$  est l'ensemble des comportements possibles du système. Or nous possédons des modèles qui décrivent cet ensemble des comportements (cf. 4.1). Un système de transition décrit les changements d'états du système et donc, de manière implicite, l'ensemble de ses comportements. À partir d'un système de transition il nous est possible de générer l'ensemble  $O$ . La figure 4.4 illustre notre propos pour le système de transition de la figure 4.1.

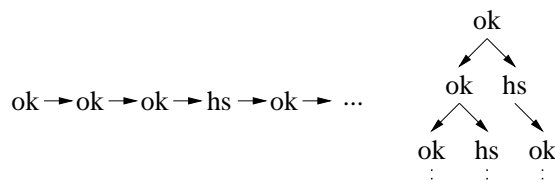


Figure 4.4: Des comportements générés par le système de transition de la figure 4.1. À gauche, le cas linéaire, où l'on considère un comportement comme une suite. À droite, le cas arborescent, où l'on considère l'arbre des branchements issus d'un état (ici ok).

Le principe de la vérification de modèle peut s'exprimer sous la forme: vérifier qu'un système satisfait une propriété consiste à vérifier que les comportements (linéaires ou arborescents) induits par le système de transition (qui modélise le système), vérifient la formule exprimant la propriété.

### 4.2.2 QUELLES PROPRIÉTÉS PEUVENT-ÊTRE EXPRIMÉES?

Les propriétés exprimables dépendent clairement de la logique utilisée. En général on distingue quatre familles de propriétés (le lecteur pourra se reporter à [21] pour plus de détails):

**L'atteignabilité:** il s'agit de vérifier qu'une situation peut se produire. En d'autres termes, on cherche à montrer qu'un état (ou un ensemble d'états) du modèle est accessible par un chemin du système de transition.

**La sûreté:** les propriétés de sûreté expriment qu'une situation  $F$  n'arrivera pas, ou plutôt que  $\neg F$  est toujours vérifiée. Dans les cas les plus simples, les propriétés

### 4.3. VÉRIFIER LES FORMULES POUR UN MODÈLE

de sûreté peuvent s'exprimer par la négation d'une propriété d'atteignabilité. Toutefois, dans certaines situations, la propriété  $F$  peut faire référence à des instants passés ou futurs [21].

**Les propriétés de vivacité :** elles expriment le fait que si une certaine condition est remplie alors une situation finira par se produire ; par exemple « si le contact est coupé alors le moteur finira par s'arrêter ». (Il ne faut pas confondre les propriétés de vivacités et la vivacité: un système est vivace s'il ne possède pas de blocage.)

**L'équité :** les conditions d'équité expriment qu'une propriété doit être vérifiée infiniment souvent. Le terme « infiniment » ne signifie pas « toujours » mais de temps en temps comme un comportement « récurrent ». Ces propriétés concernent évidemment des comportements infinis.

#### 4.2.3 D'AUTRES FORMALISMES...

Les logiques temporelles sont les formalismes les plus utilisés pour spécifier des propriétés comportementales. Il existe d'autres formalismes tels que les automates de Büchi ou le  $\mu$ -calcul. Toutefois ils sont en général utilisés comme outils théoriques pour le calcul et non comme de réels langages de spécification de propriétés. Nous citerons tout de même deux cas d'utilisation de ces formalismes pour la description de propriétés :

- Les automates de Büchi [30] permettent d'intégrer une propriété comportementale dans la description du système. Un outil bien connu qui utilise ce principe est SPIN/PROMELA [31] (dans le domaine du protocole). Alors que de nombreux outils utilisent deux données, le modèle du système et la formule à vérifier, SPIN permet d'intégrer la propriété directement dans le modèle du système. Un point important à souligner est que toutes les formules de LTL peuvent s'exprimer par un automate de Büchi. (SPIN permet d'ailleurs cette traduction afin de permettre d'intégrer une formule LTL dans la description du système).
- Le  $\mu$ -calcul [32] permet de spécifier des propriétés par des plus petits (ou plus grands) points fixes de fonctions monotones (c-à-d. des solutions d'équations de la forme  $x = f(x)$ ). Le  $\mu$ -calcul est la base de l'outil MEC [33]. Il permet de caractériser des ensembles d'états ou de transitions possédant certaines propriétés (les fonctions considérées portent donc sur des ensembles finis d'états ou de transitions).

### 4.3 VÉRIFIER LES FORMULES POUR UN MODÈLE

Dans la section précédente nous avons montré qu'une formule d'une logique temporelle permet de caractériser une propriété de comportements d'un système. Si l'on considère le modèle des systèmes de transition, un comportement (linéaire ou arborescent) peut être généré à partir de tous les états du système de transition. À quelques exceptions près (p.ex. [33]), on considère qu'un système de transition vérifie une certaine propriété  $F$  si les comportements générés depuis l'ensemble des états initiaux vérifient  $F$ . Dans le cas de la logique linéaire, on considère que le système vérifie la propriété  $F$  si toutes les suites d'états issues de l'état initial vérifient  $F$  (une quantification universelle est donc implicitement considérée). Pour CTL, le système vérifie la propriété si l'arbre issu de l'état initial vérifie  $F$  (c'est le point de vue de SMV [34]).

Les formalismes pour la modélisation d'un système et de ses propriétés étant établis, de nombreux travaux ont été menés sur l'algorithmique à mettre en œuvre pour vérifier qu'un modèle de système vérifie une propriété donnée. Ces travaux ont pour principal objectif de contourner les phénomènes d'explosion combinatoire du nombre d'états du modèle.

Dans [35, 36] un algorithme linéaire est donné pour la vérification de propriétés spécifiées en logique de Dicky [22]. La complexité de cet algorithme (implanté dans MEC [33]) est linéaire par rapport à la taille du système de transition modélisant le système global. Bien que l'algorithme soit efficace, la vérification d'un modèle se heurte en général au stockage en mémoire du système de transition.

Pour contourner ce problème des techniques de représentation symbolique ont été élaborées [34, 37]. Alors que les techniques du paragraphe précédent considèrent le système de transition en extension, les techniques symboliques traitent le modèle en intention : plutôt que de traiter les états et les transitions de manière individuelle, les algorithmes symboliques manipulent des ensembles d'états et des ensembles de transitions. L'algorithmique des méthodes symboliques repose sur l'exploitation de structures de données partagées telles que les BDD [18] (ou les arbres partagés [37]). Il est à noter que ces techniques ont été surtout employées avec succès sur des logiques dont les formules sont exprimables par des points fixes (p.ex. CTL). Les fonctions considérées pour ces points fixes sont des fonctions caractéristiques d'ensembles d'états [38]. Ces fonctions étant booléennes, elles sont alors codées par des BDD (une structure qui autorise les opérations ensemblistes usuelles).

Une autre technique algorithmique [39, 40] consiste à arrêter les calculs dès lors que l'on connaît le résultat. Tout en construisant le système de transition global, ces techniques vérifient si la propriété est falsifiée ; dans ce cas la construction est arrêtée, la réponse « non » est retournée accompagnée d'un contre-exemple. Ces techniques, dites « à la volée », ont été améliorées en intégrant la prise en compte d'informations sur les dépendances entre les événements possibles dans le système [41]. Le principe de ces méthodes (d'*ordre partiel*) consiste à réduire l'ensemble des comportements à examiner pour vérifier la valeur de vérité de la propriété. En effet, si  $a$  et  $b$  sont deux actions indépendantes consécutives alors les séquences  $a b$  et  $b a$  conduisent au même état du système. Ainsi, plutôt que de vérifier la propriété sur les deux séquences, on n'examinera qu'une d'entre elles.

À l'heure actuelle les techniques à *la volée* ou *symboliques* sont les plus répandues. Nous citerons toutefois des travaux sur l'utilisation de procédures de décisions sur les formules booléennes pour une logique dérivée de LTL [42].



## LA PROGRAMMATION « SÛRE »

Le processus de création d'un système comporte de nombreuses étapes entre l'expression des besoins et l'implémentation. La qualité d'un processus de développement dépend tant de la qualité de réalisation de chaque étape que de la qualité du passage d'une étape du processus à la suivante. Il va de soi que la qualité du processus peut être sérieusement dégradée si les étapes se succèdent sans une cohérence globale.

Suivant cette démarche des travaux ont été réalisés dans le domaine du développement des systèmes critiques. Les efforts menés ont principalement porté sur l'intégration des étapes de spécification formelle et d'implémentation. L'objectif pratique visé par ces travaux est le passage « automatique » d'une spécification validée à une implémentation.

Ce chapitre aborde deux thèmes : les méthodes formelles pour le génie logiciel, et la programmation de systèmes temps-réel (nous présentons les points de vue asynchrone puis synchrone).

**Le génie logiciel et les méthodes formelles.** Si le génie logiciel a pour ambition d'améliorer la qualité du logiciel, il est toutefois surprenant de constater que les méthodes établies dans ce domaine (par de longues années d'expériences) survolent (voire éludent) la question de la vérification formelle des programmes.

Pour des raisons évidentes liées à la fiabilité et à la sûreté, ces techniques sont maintenant utilisées pour les logiciels dits *critiques* : les systèmes embarqués (p.ex. un compteur électrique [43]), les logiciels de contrôle de systèmes critiques (p.ex. la conduite automatique de rames de métro [44]).

**Les systèmes réactifs et/ou temps-réel.** Il est inutile de rappeler qu'il est très difficile de donner une définition simple de ce qu'est un système temps-réel ; en conséquence il est tout aussi difficile de trouver une distinction honnête entre les systèmes réactifs et les systèmes temps-réel.

Dans [45], le lecteur pourra trouver une réflexion sur le temps-réel menée par un groupe de travail du CNRS. Les systèmes temps-réel sont souvent présentés par rapport à un problème spécifique (respect de contraintes de temps, réactivité, distribution, ...). Si l'on se réfère à [45, 46] les problèmes liés au temps réel sont tellement nombreux qu'il n'existe pas aujourd'hui d'outil mettant en œuvre, de manière automatique, l'ensemble du processus de réalisation d'un système temps-réel (d'autant plus que les architectures pour leur implantation sont très variées : réseaux, robotique, ...). Nous présentons ici quelques travaux qui concernent principalement la programmation « sûre » (c-à-d. vérifiée) du contrôle des systèmes réactifs.

## 5.1 LA MÉTHODE B : DE LA SPÉCIFICATION À L'IMPLEMENTATION DU LOGICIEL

Les méthodes formelles se caractérisent par l'expression des comportements d'un système dans un modèle mathématique. En première approximation, nous pouvons distinguer deux approches pour une telle formalisation :

- La première considère un modèle de machine abstraite [47, 24, 48, 49, 50, 23, 51]. Le système (ou le programme) étudié est alors décrit avec ce modèle. L'avantage de cette méthode est qu'elle permet de construire des algorithmes d'analyse, génériques et efficaces, reposant uniquement sur le modèle de machine abstraite. Du point de vue de la modélisation, l'inconvénient majeur de cette méthode est que la description du système est entièrement contrainte par l'expressivité du modèle abstrait et de la notation utilisée.
- L'autre approche consiste à rester à un niveau de formalisation plus élémentaire et à décrire le système non plus en terme de machine abstraite mais en terme de formules d'un langage mathématique donné (ce langage permet en général de décrire une machine abstraite). Des exemples de tels langages sont la logique de Hoare [52], les langages ensemblistes tels que Z [53] ou VDM [54, 55].

Ainsi un système est considéré comme un ensemble d'objets manipulés à l'aide d'opérations ; les objets et les opérations sont alors décrits sous forme mathématique. Ces méthodes permettent des raisonnements nettement plus abstraits sur le système (en particulier sur le traitement des données qui est un aspect souvent délicat pour les techniques basées sur les machines à états) et donc beaucoup plus indépendants de l'implémentation.

La méthode B [56, 57] repose sur la seconde approche. Cette méthode a été développée par J.R. Abrial (après ses travaux sur la notation Z [53, 58]) afin de renforcer l'approche formelle dans le domaine du génie logiciel. Cette méthode (supportée par l'atelier B<sup>1</sup>) couvre le processus de développement depuis la spécification formelle du système jusqu'à son implémentation. De ce point de vue cette méthode est très séduisante puisqu'elle évite (en partie) de « jongler » entre les outils de développement et les outils de vérification.

En partant de la spécification formelle du système, le processus de développement B consiste à raffiner la spécification en diminuant le niveau d'abstraction ; ce processus itératif doit converger vers l'implémentation : le code source du programme. À chaque étape du processus la spécification formelle courante est validée par un mécanisme de preuve (automatique, semi-automatique ou dirigé). De même, le passage d'une étape à la suivante est validé par un ensemble d'obligations de preuve afin de garantir que le raffinement conserve bien les propriétés établies dans les spécifications précédentes.

La dernière étape du processus est un raffinement d'une spécification formelle à un code exécutable ; ce dernier raffinement nécessite lui aussi des obligations de preuve. L'atelier B est doté d'un compilateur vers les langages C et ADA.

Nous ne nous étendrons pas plus sur ce genre de techniques qui méritent bien plus que quelques lignes. Le lecteur intéressé par un tour d'horizon sur ces méthodes pourra consulter le livre de Monin [3] ou les rapports de la NASA [59, 60].

---

<sup>1</sup>[www.b-core.com](http://www.b-core.com)

## 5.2 SYSTÈMES RÉACTIFS : DES LANGAGES ASYNCHRONES

Les langages asynchrones présentés ici considèrent que les réponses du système à un événement nécessitent un temps non nul. Du point de vue des systèmes réactifs, cette hypothèse pose d'importants problèmes puisqu'il est nécessaire de prendre en compte l'effet de l'occurrence d'un événement pendant l'exécution d'une tâche.

Dans cette section nous décrivons sommairement deux langages basés sur l'hypothèse asynchrone : ELECTRE et les STATECHARTS (de STATEMATE).

### 5.2.1 LE LANGAGE ELECTRE

#### 5.2.1.1 Le langage

Le langage ELECTRE [61, 62, 63] permet la programmation du noyau réactif d'applications temps-réel. Le principe de fonctionnement de telles applications est représenté sur la figure 5.1 (d'après [45]). Une application temps-réel est constituée d'un ensemble de tâches (appelées *modules* dans la terminologie ELECTRE) qui s'exécutent en parallèle. Ces tâches sont pilotées (c-à-d. démarrées, interrompues ou reprises) par un *exécutif temps-réel* ; ce dernier est un système informatique offrant des facilités pour l'implémentation et la manipulation de tâches concurrentes (et concourantes) sur une architecture en général distribuée et/ou embarquée (p.ex. un calculateur de vol multi-processeur). Un programme ELECTRE spécifie le comportement *externe* de l'ensemble des tâches par rapport aux événements pouvant survenir (de l'environnement ou des tâches) : le programme spécifie, en fonction des événements quand démarrer, interrompre ou reprendre un module. Chaque module d'un programme ELECTRE est considéré comme un code séquentiel ne comportant pas de point de synchronisation (c-à-d. d'attente d'un événement).

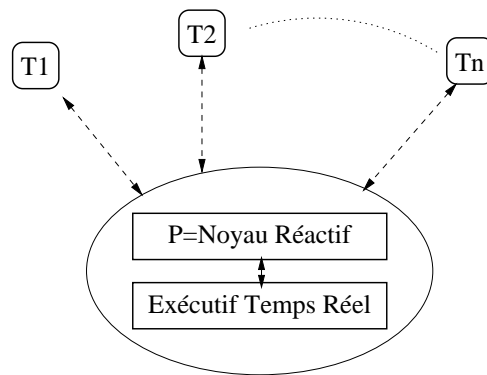


Figure 5.1: Architecture d'une application temps-reel

Le langage est doté d'opérateurs pour composer les modules (p.ex. parallélisme) mais aussi pour définir des *structures d'événements* qui représentent des activations de modules sur l'occurrence d'un événement. Au niveau des modules, il est possible de spécifier le mode de reprise : à l'état initial, au point de préemption ou non préemptible. Différentes propriétés d'événements peuvent être prise en compte : fugacité, mémorisation (unique ou multiple).

Nous n'entrerons pas plus dans les détails du langage. Il ne servirait à rien de décrire un langage qui est syntaxiquement simple mais difficile à appréhender en quelques paragraphes.

### 5.2.1.2 Compilation

L'étape de compilation [64] d'un programme ELECTRE consiste à générer un code exécutable (utilisant les primitives de l'exécutif) qui respecte les spécifications du programme.

Lorsqu'un événement survient certaines tâches sont en cours d'exécution. Un programme ELECTRE spécifie donc ce qui doit être entrepris lorsque cette situation se présente : préemption, démarrage ou reprises de modules, mémorisation de l'événement . . .

À tout programme ELECTRE, la compilation associe un automate à file appelé *FIFFO-automate* [65, 66] (FIFFO pour *First In First Fireable Out*). La file est nécessaire pour mémoriser les événements qui doivent être pris en compte mais qui ne peuvent l'être au moment de leur occurrence. L'automate associé au programme est *réactif* (c-à-d. il est complet par rapport à l'ensemble des événements pouvant survenir) et *déterministe*. Malgré sa réactivité, il est clair que la réaction du système à un événement n'est pas nécessairement immédiate (l'événement peut être mémorisé et traité ultérieurement) ; par contre, l'action du programme sur les modules est supposée instantanée (lorsque le programme doit préempter une tâche, celle-ci est immédiatement interrompue). Cette hypothèse sur le modèle doit d'ailleurs être vérifiée lors de l'implantation.

### 5.2.1.3 La vérification

La compilation d'un programme ELECTRE consiste en la construction d'un automate à file. La vérification de programme ELECTRE passe par l'utilisation de MEC [35, 33] lorsque le programme est compilé vers un produit synchronisé [49, 22] ou VALET [67] pour la compilation vers un automate hybride [27]. La thèse de Boisieu [68] montre une nouvelle fois les difficultés de la vérification pour une application réelle (mais ce problème n'est pas une spécificité du langage ELECTRE).

### 5.2.1.4 Environnement de développement

Comme nous le soulignons en introduction de ce chapitre, les problèmes liés au temps réel sont nombreux ; en conséquence, un langage de programmation et son compilateur sont nécessaires mais pas suffisants en regard des exigences formulées dans [45]. C'est sans nul doute pour cette raison qu'ELECTRE fait partie d'un projet<sup>2</sup> plus vaste s'intéressant à plusieurs problèmes du temps réel et dont la finalité est l'obtention d'un ensemble d'outils permettant la mise en oeuvre d'applications temps-réel de pilotage pour des installations industrielles.

<sup>2</sup><http://www.ircyn.ec-nantes.fr>

## 5.2.2 STATEMATE : STATECHARTS, ACTIVITY-CHARTS ET MODULE-CHARTS

### 5.2.2.1 Le langage

Les *statecharts* ont été définis par David Harel au début des années 80 [69, 70]. Ils représentent un des premiers formalismes graphiques pour la modélisation de systèmes réactifs. Son originalité ne provient pas de la description graphique des changements d'états du système (un automate suffit) mais plutôt de la représentation graphique du parallélisme, de la concurrence et de la hiérarchie.

Un *statechart* est un automate dont les états peuvent contenir un ou plusieurs automates (représentant ainsi un sous-système ou une sous-fonction du système global) : lorsque le système entre dans un tel état alors le sous-système est démarré (ou repris, s'il a été interrompu). À tout instant plusieurs automates peuvent être actifs. Ceux-ci peuvent réagir à des événements provenant soit de l'environnement soit d'autres automates. Le mécanisme de communication des événements est la diffusion (*broadcast*) ; ainsi, lorsqu'un événement survient, l'ensemble des automates actifs perçoivent cet événement. Toutes ces notions sont représentées graphiquement offrant ainsi, à l'oeil expert, une vue synthétique du système.

Un *statechart* représente un modèle des actions et réactions du système : sa partie contrôle. Afin de faire le lien entre l'abstraction et la « réalité » [70], Harel a introduit la notion d'*activité* qui se distingue de la notion d'*action*. Une *action* représente, en quelque sorte, une réaction instantanée à un événement ; elle est diffusée immédiatement dans le système qui peut alors réagir (en chaîne) à ce nouvel événement. Une *activité* est une action qui dure. Les activités sont comme les modules d'ELECTRE et les *statecharts* de Harel possèdent aussi les actions de démarrage, d'interruption ou de reprise d'une activité. L'outil basé sur ce modèle est STATEMATE qui, en plus des *statecharts*, intègre deux autres représentations graphiques : les *activity-charts* pour les activités (description de ce qui se passe lorsqu'une activité est démarrée ou suspendue) et les *module-charts* pour la description de la partie physique/structurelle du système (appelés *diagrammes de déploiement* en UML [71]).

### 5.2.2.2 Les « problèmes » sémantiques

Lors de la conception du formalisme des *statecharts*, Harel [72] avoue s'être focalisé en premier lieu sur les notations graphiques au détriment de la sémantique. Ceci a conduit à des difficultés pour définir formellement la sémantique de ce langage. Dans [72], Harel établit un bilan sur une décennie de recherche sur les *statecharts*. En particulier il souligne l'existence d'un grand nombre de sémantiques pour ce formalisme (dans [73], l'auteur énumère et compare une vingtaine de sémantiques différentes). La sémantique « officielle » des *statecharts* est celle de STATEMATE décrite dans [74] puis améliorée dans [75].

Malgré les polémiques sur leur sémantique, les *statecharts* ont séduit et font références dans toutes les démarches pour la représentation graphique des comportements des systèmes [76, 77, 71].

### 5.2.2.3 L'atelier STATEMATE

L'atelier STATEMATE a été développé rapidement après la conception du modèle des *statecharts* et a motivé la définition rapide d'une sémantique simulable de manière effi-

cace [75, 72]. Commercialisé par i-Logix<sup>3</sup>, il semble avoir reçu un écho favorable de la part de l'industrie et a bénéficié d'un retour d'expérience permettant son amélioration et une meilleure adéquation aux besoins industriels. Comme pour tout produit commercialisé, il est très difficile d'obtenir un point de vue objectif sur les capacités de ce logiciel. Toutefois, il semble évident que son utilisation dans un cadre industriel a dû contribuer à son amélioration depuis [70].

STATEMATE est un atelier de développement. Il est doté d'outils de validation du programme réalisé. Cette validation ne semble assurée que par le biais d'un simulateur (graphique). À notre connaissance, STATEMATE ne possède pas d'outil de vérification formelle basé sur les statecharts. Toutefois, des travaux sont régulièrement publiés sur la traduction des statecharts vers des formalismes pour lesquels il existe des outils de vérification (p.ex. [78]).

STATEMATE permet la génération de code C, Ada, VHDL ou VERILOG à partir des spécifications données dans l'atelier. Le lecteur intéressé par le sujet pourra consulter [79].

### 5.3 SYSTÈMES RÉACTIFS : LES LANGAGES SYNCHRONES

Cette section est dédiée à une famille de langages de programmation conçus pour la conception, la vérification et l'implantation de systèmes réactifs. Dans [80], Halbwachs caractérise ces systèmes de la manière suivante :

**Le parallélisme :** ils peuvent être constitués de plusieurs parties s'exécutant en parallèle qui interagissent de manière à remplir la fonction associée au système.

**Le temps-réel :** ils sont en général soumis à une sollicitation continue d'un environnement. Le système doit d'une part être en mesure de réagir à tout instant à un stimulus provenant de son environnement et d'autre part, de répondre à ce stimulus avant une nouvelle sollicitation de son environnement.

**Le déterminisme :** le comportement du système est complètement défini par deux paramètres : la sollicitation de l'environnement et l'instant auquel elle se produit. Puisque nous sommes dans le cadre de la programmation de système, le déterminisme est une hypothèse de travail plus que raisonnable car elle facilite la validation (les comportements sont reproductibles) et la vérification du système.

**La criticité<sup>4</sup> :** un système réactif est bien souvent une composante particulière d'un système plus grand. La contrainte de réactivité imposée à ce sous-système provient en général de sa position critique par rapport à l'ensemble du système (p. ex. un contrôleur de vol d'un avion). Ainsi ces parties du système global requièrent en général une vérification formelle la plus exhaustive possible.

**Une mise en œuvre logicielle et/ou matérielle :** Pour des raisons physiques, de performance ou de coût, un système réactif peut être décomposé en une partie logicielle et une partie matérielle (implantation dans un circuit).

<sup>3</sup>voir <http://www.ilogix.com>

<sup>4</sup>Ce vocable provient de la sûreté de fonctionnement et plus précisément de la méthode AMDEC (Analyse des Modes et Défaillance, de leurs Effets et de leur Criticité) [6]. Cette notion définit une propriété d'une défaillance d'un composant du système. Ce concept très subjectif s'exprime en fonction de la probabilité d'occurrence de la défaillance et de la gravité de ses conséquences. Il est clair que cette propriété est aussi fonction du « jugement de l'ingénieur » [2].

### 5.3. SYSTÈMES RÉACTIFS : LES LANGAGES SYNCHRONES

Concevoir un langage (et un environnement) de programmation capable de répondre à ces contraintes semble tenir de la gageure ; d'autant plus que, parmi tous ces critères, le temps-réel est le plus difficile à appréhender [46]. Toutefois des solutions existent. Les langages présentés dans les paragraphes qui suivent répondent en partie à ces exigences. Ces langages sont ARGOS, ESTEREL, LUSTRE et SIGNAL. Nous nous sommes restreint à ceux-ci pour les raisons suivantes :

- Si l'on se réfère à [80], il existe bien d'autres langages qui ont adopté (totalement ou partiellement) l'hypothèse synchrone mais ce sont généralement des langages de description (destinés à la modélisation et/ou la vérification) ou spécialisés dans un domaine particulier (p.ex. la synthèse de circuit).
- Ils sont représentatifs des paradigmes de programmation existants : impératifs, déclaratifs et graphiques.
- Leurs concepteurs se sont regroupés afin de conjuguer leurs efforts pour résoudre des problèmes généralement communs aux quatre langages (p.ex. la vérification).

#### 5.3.1 L'HYPOTHÈSE SYNCHRONE ET LE TEMPS MULTI-FORME

##### 5.3.1.1 L'hypothèse synchrone

Les langages que nous présentons maintenant ont un point commun : ils sont fondés sur l'hypothèse *synchrone*. Cette hypothèse défendue par Berry [81, 82, 83] pour le développement d'ESTEREL est la suivante : la réaction d'un système réactif à son environnement est instantanée. En d'autres termes, l'ensemble des actions effectuées par le système pour répondre à son environnement s'effectue dans un intervalle de temps *nul*. Le lecteur trouvera la justification de cette hypothèse dans les articles cités précédemment.

Dans la pratique, cette hypothèse est considérée comme remplie à partir du moment où le système réagit « infiniment » plus vite que son environnement. Une montre peut ainsi être considérée comme un système réactif par rapport à son environnement : l'utilisateur.

##### 5.3.1.2 Le temps multi-forme

Une autre hypothèse des langages synchrones est de considérer qu'il n'existe pas de temps universel (d'où le flou souligné par Dzierzowski sur le terme « temps-réel » [46]). En effet, ces langages considèrent que *courir 100 mètres* et *courir pendant 10 secondes* sont, conceptuellement, deux contraintes de temps de même nature. Ainsi les langages synchrones considèrent que le temps est un stimulus comme les autres qui provient de l'environnement (de la même manière qu'un circuit est cadencé par une horloge externe) et que l'unité de temps est l'intervalle qui sépare deux occurrences de ce stimulus. Cette hypothèse n'empêche en rien la création d'une horloge unique qui cadence globalement le système.

Dans la pratique, cette hypothèse permet de dissocier la partie conception de la partie implémentation en reportant sur cette dernière les problèmes liés aux temps de réaction. Un programme synchrone est ainsi un programme portable puisqu'il est indépendant des contraintes temporelles dues à l'architecture matérielle et/ou logicielle prévue pour l'implémentation.

### 5.3.2 À CHACUN SON STYLE

#### 5.3.2.1 Un langage graphique : ARGOS

ARGOS [84, 85] est un langage inspiré des statecharts [70, 74]. Maraninchi considère que l'automate permet d'exprimer de manière concise et lisible les comportements d'un système. Toutefois, il est bien connu que si un système réel ne peut être modélisé par un seul automate, ses composants élémentaires du système possèdent, au contraire, un nombre d'états réduit et sont donc représentables graphiquement par un automate.

C'est ainsi que ARGOS, en s'inspirant des représentations graphiques des statecharts, permet de représenter un système sous la forme d'une hiérarchie d'automates à entrée/sorties (appelés aussi machines de Mealy [86, 23]). ARGOS se distingue des statecharts par plusieurs points :

- Il est basé sur l'hypothèse *synchrone* ;
- Il est syntaxiquement plus restrictif que les statecharts. (En particulier il n'autorise ni les transitions au travers de la hiérarchie et ni l'historique c'est-à-dire la mémorisation de l'état lorsque le système quitte un macro-état.) ;
- Il peut être considéré comme une algèbre de processus. (Les processus élémentaires sont des automates et les opérateurs sont la composition parallèle et le raffinement d'un état).

Tout comme les statecharts, les événements sont diffusés (*broadcast*) dans tout le système. Il est toutefois possible de réduire la portée des événements.

ARGOS est basé sur la notion d'automate et permet la description de comportements non déterministes (c'est bien là un des avantages des automates). Dans [87], Maraninchi propose, pour la compilation de programmes réactifs, une sémantique qui prend en compte et rejette ce non-déterminisme.

Dés lors que l'on définit un langage graphique, il est tout à fait naturel de doter celui-ci d'un outil graphique permettant sa saisie. ARGONAUTE [76] est un logiciel graphique qui a été développé pour l'édition de modèles ARGOS.

#### 5.3.2.2 Un langage impératif : ESTEREL

ESTEREL fut le premier langage mettant en œuvre l'hypothèse synchrone [88, 83]. C'est un langage impératif basé sur un jeu d'instructions assez limité (une quinzaine d'instructions). Le langage a été étendu par de nouvelles intructions mais exprimées en fonction des instructions de base ; ainsi les sémantiques (opérationnelles ou dénotationnelle) d'ESTEREL reposent uniquement sur celles des instructions de base. De même qu'il est possible d'écrire des programmes non-déterministes en ARGOS, il est aussi possible de décrire en ESTEREL des systèmes qui ne sont pas corrects ; la sémantique du langage met en œuvre les outils nécessaire à la détection des programmes non cohérents.

Il est à noter que seuls ESTEREL et ARGOS possèdent un mécanisme de préemption. Ce mécanisme apparaît souvent comme un pré-requis à la définition d'un langage de programmation de systèmes réactifs [89, 46, 90].

#### 5.3.2.3 Le paradigme flots de données : LUSTRE et SIGNAL

Ces deux langages sont inspirés des réseaux de Kahn [48]. Kahn a abordé la programmation parallèle en adoptant une approche *flots de données*. Il considère des réseaux

### 5.3. SYSTÈMES RÉACTIFS : LES LANGAGES SYNCHRONES

d'opérateurs (des fonctions d'arité quelconque) interconnectés qui s'exécutent en parallèle. Les opérateurs sont activés lors de l'arrivée de données sur leurs entrées ; Kahn n'impose aucune contrainte sur ces instants d'arrivée (une hypothèse raisonnable si l'on considère que la finalité de ces réseaux était la programmation sur des architectures parallèles). En ce sens, les réseaux de Kahn sont asynchrones.

LUSTRE [91, 92, 93] et SIGNAL [94] sont basés sur l'hypothèse synchrone. À chaque entrée d'un opérateur est associée une horloge qui est un ensemble (ordonné) d'instants (discrets). Une entrée (ou une sortie) prend sa  $n^{\text{ième}}$  valeur au  $n^{\text{ième}}$  instant de son horloge.

Le modèle LUSTRE considère l'existence d'une horloge de *base*. Celle-ci est la plus rapide du système ; toutes les autres horloges sont obtenues par un échantillonnage de l'horloge de base (à l'aide d'opérateurs du langage). Une description LUSTRE consiste en l'écriture d'une hiérarchie de *nœuds* qui possèdent des entrées et des sorties. Le flot de donnée d'une sortie est décrit de manière unique (une seule équation) en fonction des flots des entrées. Ainsi un nœud est une liste d'équations de la forme  $S = f(E_1, \dots, E_n)$  où  $S$  est une variable représentant une sortie et les  $E_i$  sont les entrées du nœud. Cette équation n'est pas une affectation mais bien une égalité mathématique qui exprime l'égalité des horloges et des valeurs de  $S$  et de  $f(E_1, \dots, E_n)$ .

En dehors de sa syntaxe, SIGNAL se distingue de LUSTRE par le fait que les flots de données d'une sortie ne s'expriment pas par une fonction des entrées mais par une relation. Il est possible de décrire des flots de sortie ayant une horloge plus rapide que les entrées.

Finalement, on notera que les langages flots de données ont généralement une sémantique beaucoup plus simple que celle des langages impératifs [80].

#### 5.3.3 LA COMPILATION

Bien que leurs mécanismes de communication diffèrent (la diffusion pour ESTEREL et ARGOS, les flots de données synchrones pour LUSTRE et SIGNAL), ces langages servent à décrire un même objet mathématique : une machine de Mealy. Cette machine représente la partie contrôle du programme réactif. Suivant l'architecture d'implantation visée, les sémantiques des langages associent à un programme : soit une seule machine de Mealy (pour le cas de la compilation vers un code séquentiel), soit un ensemble de machines (réparties sur une architecture distribuée). Le dernier cas est particulièrement difficile à traiter puisqu'il est consisté à générer des processus s'exécutant en parallèle de telle manière à ce que l'exécution du système ainsi formé soit déterministe et synchrone (il ne faut pas oublier que les langages synchrones sont indépendants de l'architecture d'implantation).

Les langages réactifs cités dans cette section ont des sémantiques respectives rigoureuses qui ont été définies durant les années 80. Depuis, une partie des travaux sur ces langages ont porté sur l'amélioration des techniques de compilation pour différentes cibles : le code séquentiel (dans un langage hôte, le C par exemple), le code distribué et la synthèse de circuit.

Nous ne présenterons pas ici les techniques utilisées. Le lecteur intéressé pourra consulter [80] pour une présentation générale puis [95, 96, 97, 98, 99, 100, 101, 102] pour la génération de code séquentiel, [103, 94, 104] pour la génération de code distribué et [105, 106, 107, 108, 109] pour la synthèse de circuits.

### 5.3.4 UN AUTRE POINT COMMUN : LES PROBLÈMES

#### 5.3.4.1 Détecter les incohérences

Les langages synchrones ont des problèmes spécifiques dus à l'hypothèse synchrone. De par leur sémantique, les langages synchrones permettent de décrire des phénomènes incohérents ou non-déterministes. Pour les langages basés sur le mécanisme de diffusion (ESTEREL, ARGOS), les incohérences proviennent du fait qu'une réaction du système à un stimulus est une réaction *en chaîne* (c'est un problème bien connu de ce mécanisme [110]). Pour le paradigme flots de données, le problème provient des horloges : un programme définit un ensemble de contraintes, de dépendances entre les horloges ; ces contraintes peuvent posséder zéro, une ou plusieurs solutions alors que, par hypothèse, il est nécessaire qu'il n'y ait qu'une seule solution.

À des fins de compilation, les concepteurs des langages synchrones se sont restreints aux programmes *corrects*. Étant donné la complexité des systèmes réels, il n'était pas possible de demander à un ingénieur de vérifier « à la main » la cohérence de son programme. C'est pour cette raison que les compilateurs des langages synchrones ont été outillés de manière à détecter les incohérences.

#### 5.3.4.2 Vérifier l'hypothèse synchrone

Pour citer une nouvelle fois [80], l'hypothèse synchrone est réaliste si, au niveau de l'implantation, le programme réagit à un stimulus de son environnement avant l'occurrence d'un nouveau stimulus. Dans la pratique, les langages synchrones sont dotés d'outils permettant le calcul du temps de réponse maximal du programme. Il s'avère que la compilation génère en général du code très performant [101] ; de plus le temps de réponse maximal du programme est facilement calculable à partir de l'automate généré par le compilateur et des durées des instructions du langage hôte.

Toutefois, ce problème reste très délicat lorsque l'on s'occupe de la distribution du code ou de la synthèse de circuit. Pour ce dernier cas, il est nécessaire de déterminer les délais de latence dans le système afin de calculer la fréquence d'horloge optimale pour le cadencement du circuit synthétisé [111].

### 5.3.5 UNE GAMME D'OUTILS

Les langages synchrones ont été conçus par des équipes de recherches différentes. Ces équipes ont mutualisé leurs efforts et ont fait en sorte de partager leurs outils. Cette gamme d'outils comporte bien évidemment les compilateurs, des environnements graphique tels que AUTOGRAPH [112] (visualisation d'automates) ou SAHARA [113] (simulateur graphique/générateur d'interfaces) ainsi que des outils de vérification formelle (p. ex. LESAR [114] ou AUTO [112]).

Il est à noter que l'interconnexion entre ces outils est assurée par un ensemble de formats communs [115, 116].

### 5.3.6 TRAVAUX CONNEXES

Dans la littérature sur les langages synchrones on peut trouver quelques réflexions sur l'intégration des différents formalismes [117]. Par exemple le couplage des automates hiérarchisés de ARGOS avec LUSTRE [118] (ces travaux semblent à l'origine des automates de modes [119] qui associent à tout état de l'automate un ensemble d'équations LUSTRE). Il semblerait que la même démarche soit initiée pour le langage SIGNAL et

### 5.3. SYSTÈMES RÉACTIFS : LES LANGAGES SYNCHRONES

les statecharts de STATEMATE [120, 78]. Enfin, dans [121], on trouvera une formalisation du couplage d'une partie de ESTEREL avec CSP [122] appelé CRP (Communicating Reactive Process) : ce modèle consiste à faire communiquer des programmes réactifs par *rendez-vous* (on retrouve une démarche analogue, visant à coupler les approches synchrones et asynchrones, dans [123]).

Nous ne mentionnerons pas les expériences industrielles menées autour des langages synchrones<sup>5</sup>. Toutefois nous tenons à citer [124], une expérience intéressante sur l'utilisation de ESTEREL dans un cadre industriel. La démarche employée est surprenante puisque le langage ESTEREL est utilisé non pas pour la synthèse du circuit mais pour la génération d'un modèle de référence de DSP (Digital Signal Processor) servant à valider (par tests) le système réel construit à l'aide du langage VHDL [125].

---

<sup>5</sup>Le lecteur pourra consulter le site ESTEREL : <http://www.inria.fr/meije>.





## CONCLUSION

*Ultimately, the systems we specify are physical objects, and mathematics cannot prove physical properties. We can prove properties only of a mathematical model of the system; whether or not the system correctly implements the model must remain a question of law and not of mathematics.*

Leslie Lamport  
*Comm. ACM, 1989*

Comme l'a écrit Lamport, l'analyse des systèmes permet uniquement de montrer qu'un *modèle du système* (et non le système lui-même) possède une bonne propriété. Il n'existe pas, en effet, de méthode ou de modèle « miracle » qui garantisse que le système réponde à ses spécifications.

Lors d'un projet de développement d'un système critique plusieurs méthodes d'analyse doivent être mises en œuvre. La multiplication des modèles utilisés entraîne inévitablement des problèmes organisationnels, entre autres : la communication entre les équipes de développement. En effet, les différentes tâches du projet (spécification, conception, implantation) sont de plus en plus souvent menées en parallèle par des équipes distinctes. Lorsque les équipes utilisent des méthodes d'analyse différentes, elles sont confrontées à l'exploitation des résultats d'équipes n'utilisant pas la même méthode.

Dans ce contexte le projet AltaRica a l'ambition de fédérer tant les domaines d'activité (sûreté de fonctionnement et vérification formelle) que les modèles. À cette fin, le projet se propose de :

**Fédérer les outils** d'analyse afin d'exploiter au mieux l'existant et d'autoriser l'étude des systèmes selon des points de vue différents (sûreté de fonctionnement ou vérification formelle).

**Fédérer les modèles** en un seul langage de description afin d'améliorer la traçabilité des modèles et la communication entre les équipes de conception ;

**Développer les compilateurs** du langage AltaRica vers des outils existants ;

**Proposer un atelier** logiciel de modélisation et d'analyse des modèles AltaRica. Cet atelier utilise le langage AltaRica comme modèle de description et établit la connection avec les outils par le biais de compilateurs.

La seconde partie de ce rapport a pour sujet principal le langage AltaRica. Ce langage est, d'une certaine manière, la pierre d'angle du projet car :

- Il définit la base formelle des descriptions de l'atelier AltaRica ;
- De son acceptation par les ingénieurs dépend, en partie, la réussite du projet.

# **Partie II**

# **AltaRica**





## INTRODUCTION : LE PROJET ALTARICA

Les travaux présentés dans ce rapport ont débuté lors du lancement du projet AltaRica en décembre 1996. Dans ce chapitre nous présentons les motivations tant industrielles que scientifiques de ce projet.

### 7.1 ALTARICA : POSITIONNEMENT INDUSTRIEL

Le domaine de la sûreté de fonctionnement s'est doté de nombreux outils pour l'analyse des systèmes. Ces outils de calcul supportent en général un modèle spécifique : arbres de défaillances [11], réseaux de Petri stochastique [126], graphe de Markov, . . . L'existence de ces outils souligne l'utilité de ces modèles comme fondement mathématiques et algorithmiques des logiciels de calculs. Les modèles cités précédemment ont l'inconvénient d'être de bas niveau. En effet, il existe une certaine distance (une forte abstraction) entre le modèle du système et le système lui-même. Une telle réalité a plusieurs conséquences :

- Les membres des équipes de conception/validation doivent posséder une parfaite connaissance des modèles/outils utilisés afin de permettre d'une part la communication entre les membres d'une même équipe et d'autre part, la communication entre les différentes équipes d'un même projet.

Le travail de modélisation tenant plus de l'art que de la science, le « savoir-modéliser » ne peut s'acquérir que par une longue expérience. Un chef de projet est dès lors confronté au problème de la constitution d'une équipe d'ingénieurs expérimentés dans les formalismes utilisés.

- D'un point de vue organisationnel, la traçabilité des modèles au cours des différentes phases d'un projet, nécessite la mise en place de procédures rigoureuses de conception et de documentation des modèles. La multiplication des formalismes/outils implique inévitablement la multiplication des procédures et des nomenclatures imposées aux équipes.

Afin de remédier à ces problèmes récurrents, des ateliers logiciels, orientés sûreté de fonctionnement, ont été développés (p. ex. FIABEX [1], FIGARO [127] ou SOFIA [128]). Ces ateliers (qui ont nécessité d'énormes développements) se caractérisent par trois aspects :

1. Une représentation graphique unique du système (description architecturale du système par des blocs fonctionnels ou matériels);

2. L'intégration d'outils d'analyse supportant des modèles spécifiques; par exemple les arbres de défaillances, les AMDEC, . . .
3. Peu (ou pas pour FIABEX) de formalisation des descriptions lors de leur conception.

Le dernier point est problématique. En effet, l'absence de sémantique formelle lors de la conception de ces logiciels engendre d'énormes difficultés pour la formalisation, *a posteriori*, des modèles de l'atelier et l'introduction de nouveaux algorithmes (p.ex. de compilation vers un autre modèle) nécessite, en général, l'expertise des concepteurs (de l'atelier).

Dans ce contexte, les objectifs industriels du projet AltaRica sont :

1. le développement d'un atelier logiciel graphique (successeur de FIABEX) supportant un unique formalisme et intégration des outils d'analyse existants (ou futurs); par exemple ARALIA [15];
2. la formalisation des descriptions de l'atelier par un langage textuel et/ou graphique (cf. chapitre 8) dont la sémantique est clairement définie (cf. chapitre 9);
3. le maintien d'un partenariat d'industriels autour du projet afin de motiver et de valider les actions de recherches menées;
4. profiter de la synergie du partenariat AltaRica afin de motiver le rapprochement de la sûreté de fonctionnement et de la vérification formelle.

## 7.2 ALTARICA : POSITIONNEMENT SCIENTIFIQUE

### 7.2.1 POSITIONNEMENT DU LANGAGE

Le langage AltaRica peut être considéré comme l'aboutissement de deux modèles issus de travaux de recherches de l'équipe MVTsi<sup>1</sup> du LaBRI :

1. Le modèle Arnold-Nivat [49] qui considère un système comme un ensemble de systèmes de transition synchronisés. Ce modèle, mis en œuvre dans l'outil MEC [35], a montré d'une part, sa capacité à capturer un éventail assez large de problèmes de modélisation et d'autre part, son manque d'expressivité pratique due, paradoxalement, à sa simplicité.
2. Le modèle des automates à contraintes [129] qui exprime le modèle Arnold-Nivat sous forme de contraintes, offre une plus grande souplesse dans la description des systèmes. L'expérimentation de ce modèle dans TOUPIE [130] a permis de valider cette extension et de montrer la faisabilité d'un encodage symbolique du produit synchronisé exprimé sous forme de contraintes.

Si le positionnement d'AltaRica par rapport aux travaux de l'équipe MVTsi est relativement naturel, il est toutefois assez délicat de justifier l'existence d'un nouveau langage alors qu'il en existe déjà tant (dont la plupart sont dotés d'outils dédiés).

L'existence même du langage est principalement justifiée par des raisons historiques. Initialement, ce langage devait être le support sémantique de l'atelier FIABEX.

<sup>1</sup>Modélisation, Vérification et Tests des Systèmes Informatisés

### 7.3. PLAN DE CETTE PARTIE

Lors de cette première phase du projet nous aurions pu envisager l'utilisation d'un langage existant ; par exemple, LUSTRE pour ses aspects flots de données (fréquents dans les systèmes de nos partenaires) ou les STATECHARTS pour l'utilisation d'automates hiérarchiques. Cette solution a été écartée pour les raisons techniques suivantes :

- Il eut été difficile d'adapter un langage existant aux besoins présents et futurs des partenaires du projet ;
- Dans la mesure où la problématique était la définition d'une sémantique propre des descriptions de FIABEX, le langage de modélisation (considéré comme un ensemble de règles syntaxiques) est accessoire (contrairement à un langage de programmation où le style est un critère de sélection).

#### 7.2.2 POSITIONNEMENT DU PROJET

La coordination scientifique du projet AltaRica a été attribuée à l'équipe MVTsi du LaBRI dont les travaux de recherches portent sur la vérification formelle (*model-checking* ou tests) des systèmes informatiques. Le projet a été motivé par les besoins d'ingénieurs de la sûreté de fonctionnement, un domaine qui est connexe à la vérification formelle mais dont les modèles et les méthodes diffèrent. À notre connaissance, AltaRica est un des rares projets (voire le seul) qui ambitionnent d'offrir, à terme, les deux approches. La réussite de ce projet est toutefois conditionnée par la résolution de problèmes spécifiques à la sûreté de fonctionnement qui ne sont pas traités par les méthodes de la vérification formelle :

- l'analyse qualitative des modèles consiste à déterminer les séquences de défaillances des composants élémentaires qui mènent le système dans un état non souhaité (cf. chapitre 11) ;
- la compilation vers des modèles de plus bas niveau, en particulier les arbres de défaillance, pour lesquels il existe des outils performants ;
- résoudre, ou du moins contourner, les problèmes d'explosion combinatoire (spécifiques aux systèmes industriels) qui sont exagérés lorsque l'on modélise les aspects dysfonctionnels des systèmes.

### 7.3 PLAN DE CETTE PARTIE

La suite de ce document est organisée de la manière suivante. Dans le chapitre 8 (page 55) nous présentons de manière informelle le langage de modélisation AltaRica. Nous consacrons le chapitre 9 (page 77) à la sémantique du langage ainsi qu'à ses propriétés (en particulier sa compositionnalité). Le chapitre 10 (page 89) illustre le langage par quelques modélisations de systèmes. Quelques algorithmes pour l'étude l'analyse des séquences critiques d'un modèle sont proposés au chapitre 11 (page 103).





## LE LANGAGE ALTA RICA

Le langage AltaRica permet la description de systèmes constitués de composants qui interagissent de deux manières. Le premier mécanisme d'interaction repose sur l'éventuel synchronisme des actions des composants; pour cela nous nous sommes fortement inspiré du *produit synchronisé* du modèle de Arnold et Nivat [49, 131] (implémenté dans MEC [35, 33]). Le second mode d'interaction est basé sur la coordination des flux des composants: chaque composant possède un ensemble de flux qui dépendent de son état et qui sont visibles (et éventuellement contraints) par son environnement. Ce mécanisme est inspiré du modèle des *automates à contraintes* de Rauzy et Brlek [129, 132] (qui ont traité ce modèle à l'aide de TOUPIE [133, 130]). Un modèle similaire [134] fut proposé par Fribourg et Peixoto dans le cadre de la programmation logique. Ces modèles d'interactions sont associés à une description hiérarchique du système: un sous-système (appelé *nœud*) peut intégrer et contrôler (en fonction de son propre état) un ensemble de composants ou de sous-systèmes.

Ce chapitre présente de manière informelle les principales notions mises en œuvre dans le langage de modélisation AltaRica. Pour chaque notion nous tentons de donner sa sémantique intuitive en termes de systèmes de transitions. Dans un premier temps nous présentons les éléments de base d'une description AltaRica, à savoir les composants (section 8.1). Dans un deuxième temps nous introduisons la hiérarchie (les nœuds) et les mécanismes de description des interactions entre les composants (section 8.2). La dernière section de ce chapitre propose une syntaxe graphique minimale pour les descriptions en langage AltaRica.

### 8.1 LES COMPOSANTS

Un système est constitué de *composants*. Dans la terminologie AltaRica un composant est situé au plus bas niveau dans la hiérarchie décrivant le système.

#### 8.1.1 NOTIONS DE BASE

Un composant est une machine abstraite dont l'état est modifié par certains phénomènes. Ceux-ci proviennent soit de l'environnement soit du composant lui-même; dans le premier cas on les appelle, en général, des *événements* et dans le second cas, des *actions* du composant. Cette distinction reste toutefois informelle et n'est pas prise en compte dans la sémantique du langage (cf. le chapitre 9). Dans la suite, nous utiliserons généralement le terme "événement" mais le lecteur notera bien qu'il n'y a aucune

autre sémantique attachée à ce vocable que celle d'un phénomène modifiant l'état du composant.

Un composant AltaRica peut être équipé de *flux* (appelés parfois *observations*) qui l'informent partiellement sur l'état de son environnement. En sens inverse, ces mêmes flux lui permettent de transmettre une information vers l'extérieur. Dans une modélisation un flux peut être utilisé pour décrire aussi bien un port de connexion (pour un protocole de communication) que les bornes d'un dipôle dans un circuit électrique. L'ensemble composé des événements et des flux d'un composant est appelé l'*interface* du composant. Un composant interagit avec son environnement par le biais de son interface.

D'un point de vue syntaxique, un événement d'un composant est simplement un nom (une étiquette dont l'interprétation est laissée à la charge du modélisateur). Les observations sont quant à elles représentées par des variables dites de *flux*, qui prennent leurs valeurs dans un domaine spécifié par le modélisateur (les booléens, les entiers ou un ensemble de chaînes de caractères).

Les états d'un composant sont décrits de manière implicite par un ensemble de variables (dites *d'état*). Un état du composant est alors une valuation de ces variables. Par la suite nous désignerons par *état d'un composant*, une valuation de ses variables d'états et par *configuration*, l'association d'un état et d'une valuation des variables de flux. Alors que la première notion décrit l'état interne d'un composant, la seconde prend en compte la vision de son environnement par le composant. Par exemple, si l'on considère un interrupteur, son état est sa position (ouvert ou fermé) tandis que sa configuration est son état et une valuation de ses bornes (cette dernière dépend de son environnement).

Un composant observe son environnement par l'intermédiaire de ses flux. Réciproquement un composant est autorisé à donner une vue partielle de son état à son environnement. Le modèle AltaRica suppose qu'il existe une relation de dépendance  $A(\vec{s}, \vec{t})$  entre l'état du composant et la valeur de ses flux. Cette relation est représentée par une contrainte sur les variables du composant; cette contrainte est appelée l'*assertion* du composant et elle est supposée vérifiée par toutes les configurations.

Les événements modifient l'état d'un composant; ces changements d'états sont appelés *transitions*. Le langage permet une description implicite de l'ensemble des transitions d'un composant par un ensemble de macro-transitions. Une *macro-transition* est de la forme  $G(\vec{s}, \vec{f}) \mid - e \rightarrow \vec{s}' := Succ(\vec{s}, \vec{f})$  où  $G(\vec{s}, \vec{f})$  est une condition, appelée *garde*, qui porte sur les valeurs des variables d'états et de flux,  $e$  est un nom d'événement et  $\vec{s}' := Succ(\vec{s}, \vec{f})$  est un vecteur d'affectations des variables d'états  $\vec{s}$ . La sémantique intuitive d'une macro-transition est la suivante: l'événement  $e$  modifie l'état du composant si sa configuration vérifie  $G$ ; de plus, après l'occurrence de  $e$ , chaque variable d'état reçoit la valeur spécifiée par  $Succ(\vec{s}, \vec{f})$ , évaluée en utilisant les valeurs des variables à l'instant qui précède l'occurrence de  $e$ . Le modèle AltaRica suppose que toutes les configurations du composant satisfont son assertion; cette dernière représente donc un invariant puisque les configurations origine et but d'une transition doivent vérifier l'assertion.

#### Exemple 8.1

*Dans cette exemple nous donnons la description d'un compteur (de 0 à N). Ce compteur peut être incrémenté, décrémenté ou remis à zéro par des événements appelés, respectivement, Inc, Dec et Raz. La valeur du compteur est transmise à son environnement par un flux Val. L'état du compteur est modélisé par une variable `compte` prenant ses valeurs dans l'intervalle  $[0, N]$ .*

## 8.1. LES COMPOSANTS

La description de ce compteur est donnée ci-dessous (*//* indique le début d'un commentaire). Elle commence par la spécification du nom du modèle de composants, `node Compteur`, et se termine par `edon`. La description consiste ensuite en la spécification des composantes du modèle: l'interface (les variables de flux et les événements), les variables d'état, les transitions et l'assertion.

```

const N = 2;                                // Déclaration d'une constante (globale)

node Compteur                                // Un modèle de compteur
  flow Val : [0,N];                          // La variable de flux
  event Inc, Dec, Raz;                       // Les événements
  state compte : [0,N];                     // La variable d'états
  trans compte < N |- Inc -> compte := compte+1; // Les transitions
      compte > 0 |- Dec -> compte := compte-1;
      true |- Raz -> compte := 0;
  assert compte = Val;                       // L'assertion
edon
  
```

Le lecteur notera que l'assertion de ce modèle impose que l'état du compteur soit toujours égal à son flux; l'environnement peut donc connaître exactement la valeur du compteur par l'intermédiaire de `Val`.

Le modèle des composants AltaRica est événementiel dans le sens où les changements d'état sont uniquement provoqués par des occurrences d'événements (ou d'actions). L'évolution d'un composant soumis à une séquence d'événements est représentée sur la figure 8.1. Cette figure montre que l'occurrence d'un événement modifie uniquement l'état. La configuration du composant étant dépendante de son état, les flux sont mis à jour de manière instantanée en fonction du nouvel état.

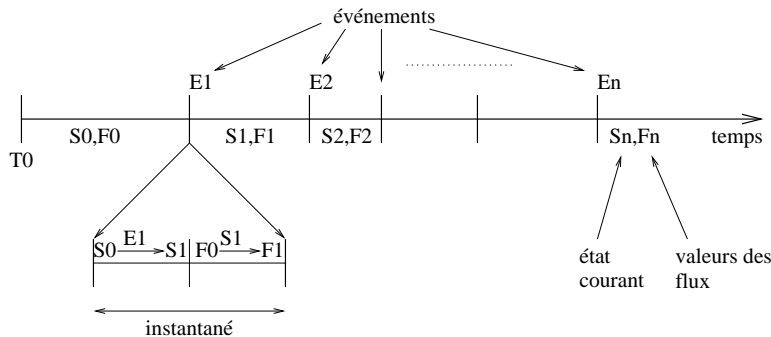


Figure 8.1: Un composant évolue en fonction des événements. La modification des flux n'a lieu qu'après un changement d'état et se produit de manière instantanée.

### Exemple 8.2 (suite de l'exemple 8.1)

Intuitivement, la sémantique du modèle `Compteur` est le graphe des configurations représenté à la figure 8.2. À chaque état sont associés une valuation des variables d'états

(ici compte) et, en italique, les valuations des variables de flux (ici val). Les transitions entre les états portent une double étiquette: le nom de l'événement provoquant le changement d'état et la valeur du flux lors de l'occurrence de l'événement.

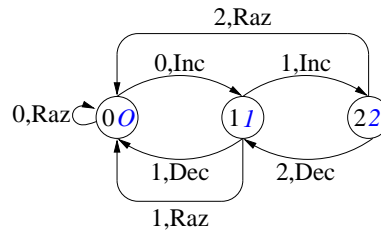


Figure 8.2: Le graphe des configurations du compteur pour  $N=2$ .

Par la suite, lorsque l'ensemble des valuations des variables de flux associé à un état est le produit cartésien des domaines de ces variables, nous noterons cet ensemble par #.

Sur l'exemple précédent la relation entre l'état du compteur et son flux *Val* est fonctionnelle (ici il s'agit de l'identité sur les entiers). Les modèles de composants comportent en général plusieurs flux. La valeur d'un flux donné n'est pas nécessairement une fonction de l'état interne car il peut être contraint par les autres flux du composant. L'exemple suivant illustre notre propos sur un modèle d'interrupteur.

### Exemple 8.3

Dans cette exemple nous considérons un interrupteur qui ouvre ou ferme un circuit. Ce composant comporte deux flux, *E1* et *E2*, décrivant ses connexions au circuit. Lorsque l'interrupteur est fermé, il transmet le courant (qu'il provienne de *E1* ou de *E2*). L'événement *Pression* modélise une action d'un utilisateur sur le bouton de l'interrupteur. Le modèle est le suivant:

```
node Interrupteur
  flow  E1, E2 : bool;  //bool est le domaine des booléens
  event  Pression;
  state  ouvert : bool;
  trans  true |- Pression -> ouvert := not ouvert;
  assert (ouvert = false) => (E1=E2);
edon
```

L'assertion spécifie que si l'interrupteur est fermé (`ouvert = false`) alors ses bornes doivent être égales. Le graphe des configurations de ce modèle est représenté sur la figure 8.3. Sur cette figure les états sont les valuations de la variable `ouvert`; `T` représente la valeur `true` et `F` la valeur `false`. Nous avons regroupé les transitions qui ont le même événement et les mêmes états origine et but. Les transitions sont étiquetées par un ensemble de valeurs de flux. Cet ensemble est omis s'il s'agit de toutes les valuations possibles des variables de flux.

## 8.1. LES COMPOSANTS

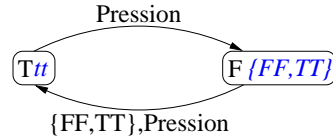


Figure 8.3: Graphe des configurations de l'interrupteur.

Le graphe des configurations présenté dans l'exemple précédent montre effectivement les changements d'états en fonction des événements. Si l'on considère maintenant l'interrupteur dans son environnement (un circuit), le graphe donné à la figure 8.3 n'est pas totalement correct puisqu'il ne tient pas compte des éventuelles évolutions de l'environnement. En effet, supposons que la configuration de l'interrupteur soit *ouvert*=*F*, *E1*=*E2*=*T* (l'interrupteur est fermé et il existe un courant passant d'une borne à l'autre) et supposons que le circuit évolue de manière à rendre nulle la tension à une borne de l'interrupteur (p. ex. la source d'électricité tombe en panne) alors la configuration de l'interrupteur doit devenir *ouvert*=*F*, *E1*=*E2*=*F* (l'interrupteur est toujours fermé et les deux flux doivent donc être égaux). Afin de prendre en compte cet aspect, le modèle de composant AltaRica suppose l'existence d'un événement particulier  $\epsilon$ , qui ne modifie pas l'état du composant mais qui autorise les changements de configurations. Dans la pratique, une macro-transition  $true \mid - \epsilon \rightarrow$  est implicitement ajoutée à la description; sa garde est vérifiée pour toutes les configurations et elle ne modifie aucune variable d'état.

Si maintenant nous prenons en compte les transitions étiquetées  $\epsilon$  nous obtenons, pour l'interrupteur de l'exemple 8.3, le graphe des configurations de la figure 8.4.

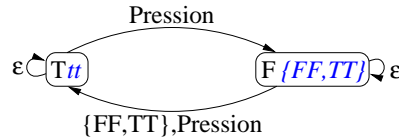


Figure 8.4: Graphe des configurations de l'interrupteur. Des transitions étiquetées par  $\epsilon$  indiquent les évolutions de l'environnement alors que le composant ne change pas d'état.

### 8.1.2 PRIORITÉS

Le modèle AltaRica intègre la notion de *priorité* (statique). Le concept de priorité est un mécanisme bien connu pour l'ordonnancement d'activités dans les systèmes concurrents (p. ex. les processus sous UNIX). Ce mécanisme consiste à définir une relation d'ordre sur les actions du système et si, dans un état donné du système, deux actions sont possibles alors la plus grande pour l'ordre est choisie (la plus prioritaire).

Les priorités permettent, en général, d'augmenter le pouvoir d'expression du modèle sous-jacent. On peut trouver la notion de priorité dans différents modèles : les réseaux de Petri [135, 136], les expressions de chemins (*path expressions*) de COSY [137, 138, 139]. Certains modèles utilisent un mécanisme qui s'apparente à la notion de priorité; c'est le cas de la notion de *canaux urgents* dans le modèle d'automates temporisés de UPPAAL [25] ou encore le mécanisme de préemption de ESTEREL [89]. Pour ces deux derniers cas, le mécanisme est plutôt celui des priorités dynamiques: l'ordre

sur les événements est fonction de l'état du système. Comme cela est présenté au chapitre 9, l'utilisation des priorités dans le langage AltaRica introduit implicitement une quantification existentielle sur les gardes des transitions.

L'utilisation des priorités pour la modélisation des composants a un intérêt limité mais ce mécanisme permet parfois de simplifier l'écriture du modèle. L'utilité des priorités apparaît lorsque l'on modélise des processus concurrents; nous reviendrons par conséquent sur cette notion lorsque nous présenterons la hiérarchie dans AltaRica (section 8.2).

Exemple 8.4

Dans cet exemple nous illustrons le mécanisme des priorités par un composant qui observe deux flux booléens et qui réagit lorsqu'ils deviennent vrais (true); la réaction du composant est différente suivant le flux. La description de ce composant est la suivante:

```

node Observateur
  flow f1, f2 : bool;
  event ReactionF1, ReactionF2;
  state mode : { observation, modeF1, modeF2 };
  trans mode = observation and f1 = true | -
    ReactionF1 -> mode := modeF1;
    mode = observation and f2 = true | -
    ReactionF2 -> mode := modeF2;
edon
  
```

Le graphe des configurations de l'observateur est présenté à la figure 8.5. Sur ce graphe on remarquera que si les flux sont vrais simultanément un choix de réaction doit être effectué. Supposons que pour des raisons liées au système modélisé, ce choix n'ait pas de sens et que, dans cette situation, seul ReactionF1 doit avoir lieu. Il faut alors imposer cette contrainte en spécifiant que ReactionF1 est prioritaire à ReactionF2. Pour cela nous spécifions dans la description: event ReactionF1 priority 1, ReactionF2 priority 0 qui indique que ReactionF1 (resp. ReactionF2) est au niveau de priorité 1 (resp. 0). Si l'on applique maintenant les priorités, la transition étiquetée TT,ReactionF2 n'existe plus dans le modèle.

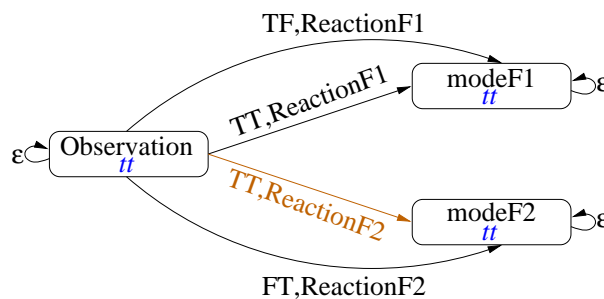


Figure 8.5: Le graphe des configurations pour l'observateur. Un état représente la valeur de mode, et les transitions sont étiquetées par des couples (f1, f2) de valeurs des flux.

Il est à noter que, pour cet exemple, les priorités ne sont pas nécessaires: pour

## 8.2. LA HIÉRARCHIE

la réaction au flux  $f2$  il suffit d'ajouter un condition sur la valeur de  $f1$  à la macro-transition étiquetée  $ReactionF2$ :

```
mode = observation and f2 = true and not f1 |-
  ReactionF2 -> mode := modeF2;
```

## 8.2 LA HIÉRARCHIE

Un *noeud* AltaRica est un modèle de sous-système qui intègre et contrôle d'autres sous-systèmes. Un noeud possède son propre comportement et peut ainsi modifier les interactions entre ses sous-composants en fonction de son état interne. Un système est décrit par un noeud particulier appelé *Main*.

Un noeud AltaRica est une "boite noire". Considéré depuis son environnement, il se comporte comme un composant et interagit par l'intermédiaire de son interface (formée de flux et d'événements); la hiérarchie intrinsèque du noeud est totalement masquée à son environnement.

D'un point de vue interne, un noeud peut être considéré comme l'union d'un contrôleur et d'un ensemble de composants qui interagissent sous la tutelle du contrôleur. Ce dernier a pour rôle de contraindre les interactions des sous-composants du noeud. Lorsqu'il n'impose aucune contrainte, l'ensemble des composants du noeud s'exécutent de manière fortement asynchrone, c'est-à-dire qu'un seul composant à la fois peut changer d'état.

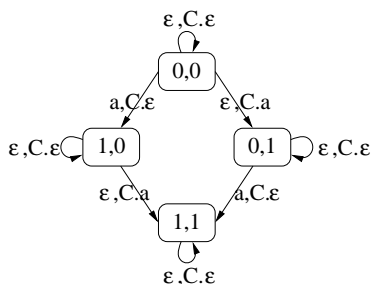
### Exemple 8.5

Dans cet exemple, nous considérons que le système (noeud *Main*) utilise un composant *C* de modèle Composant qui peut effectuer une action *a*. Le système peut lui aussi agir, indépendamment de *C*, par une action *a*. La description AltaRica de ce système ainsi que le graphe de ses configurations sont donnés ci-dessous.

```
node Composant
  event a;
  state etat : { 0, 1 };
  trans etat = 0 |- a -> etat := 1;
edon
```

```
node Main
  // La description du contrôleur
  event a;
  state etat : { 0, 1 };
  trans etat = 0 |- a -> etat := 1;
```

```
// Déclaration des sous-composants
sub C : Composant;
edon
```



Le lecteur aura certainement noté que sur le graphe des configurations de cet exemple, les transitions ne sont pas étiquetées par des événements mais par des couples d'événements  $(e_1, e_2)$  où  $e_1$  est un événement de *Main* et  $e_2$  est un événement de *C* (l'identificateur de l'événement est préfixé par le nom du sous-composant). Il s'agit

de vecteurs de synchronisation qui décrivent des événements globaux du système et qui indiquent quels événements élémentaires les composent. Un événement est supposé instantané par conséquent les événements élémentaires contenus dans un vecteur sont simultanés. Dans cet exemple, les vecteurs sont de la forme  $(a, C.\epsilon)$ ,  $(\epsilon, C.a)$  ou  $(\epsilon, C.\epsilon)$  ce qui décrit le fait que si l'un des deux processus (Main ou C) effectue une action alors l'autre ne change pas d'état (asynchronisme fort). Les vecteurs de synchronisation sont décrits plus en détail dans la section 8.2.2.

Le contrôle exercé par un noeud porte sur les interfaces de ses sous-composants. Syntaxiquement, la visibilité d'un noeud est réduite à ses sous-composants directs (visibilité sur un seul niveau hiérarchique). Le contrôle s'exprime par deux contraintes. La première est une contrainte de coordination des flux des sous-composants et la seconde contraint les combinaisons possibles d'événements.

Afin d'illustrer les concepts introduits par la suite, nous considérons le circuit décrit dans l'exemple suivant.

#### Exemple 8.6

Le système est représenté sur la figure 8.6.

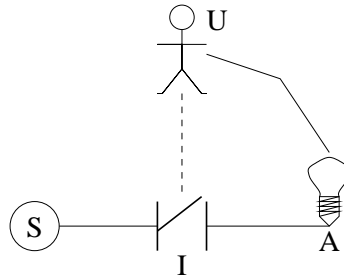


Figure 8.6: Un système composé d'un circuit comportant une source d'électricité S, un interrupteur I et une ampoule A. Un utilisateur U peut agir sur l'interrupteur.

```

/*
 * La description de l'exemple 8.3 à laquelle nous avons ajouter un flux ex-
 * portant l'état.
 */
node Interrupteur
  flow  estFerme :  bool;
  ...
  assert ouvert = not estFerme;
edon

/*
 * La source comporte un flux courant modélisant le courant qu'elle génère.
 * Sa valeur nominale est true. Cette source est supposée parfaite et alimente
 * donc constamment le circuit (d'où l'assertion).
 */
node Source

```

## 8.2. LA HIÉRARCHIE

```
flow courant: bool;
assert courant = true;
edon

/*
 * L'ampoule reçoit le courant par un flux booléen courant. Deux événements
 * sont utilisés pour modéliser ses passages d'éteinte à allumée (et vice versa).
 * Un flux allumee permet à l'environnement d'observer l'état de la lampe
 * (allumée ou non).
 */
node Ampoule
flow courant: bool;
   allumee: bool;
event S_Allume, S_Eteint;
state allumee_: bool;
trans allumee_ and not courant |-
      S_Eteint -> allumee_ = false;
      not allumee_ and courant |-
      S_Allume -> allumee_ = true;
assert allumee = allumee_;
edon

/*
 * L'utilisateur observe l'ampoule par un flux ampouleAllumee. Si l'ampoule est étein-
 * te alors il l'allume et inversement, si elle est allumée alors il l'éteint.
 */
node Utilisateur
flow ampouleAllumee: bool;
event Allume, Eteint;
trans ampouleAllumee |- Eteint ->;
      not ampouleAllumee |- Allume ->;
edon

node Main
sub
  S: Source;
  I: Interrupteur;
  A: Ampoule;
  U: Utilisateur;
  ...
edon
```

---

### 8.2.1 COORDINATION DES FLUX

Un noeud AltaRica peut observer et contraindre les flux de ses sous-composants. Nous avons vu à la section 8.1 que les flux d'un composant ne dépendent que de son état et/ou de son environnement. De manière analogue, les flux d'un noeud ainsi que ceux

de ses sous-composants dépendent de l'état du noeud ou de son environnement.

L'assertion décrite dans un noeud peut porter sur les variables de flux des sous-composants; il est ainsi possible de contraindre, en fonction de l'état du contrôleur et de ses flux, les combinaisons des flux des sous-composants.

Exemple 8.7

*Si l'on considère le circuit de l'exemple 8.6, nous exprimons les liens S-I, I-A et A-U dans l'assertion du noeud Main:*

```
node Main
sub
  ...
assert
  S.courant = I.E1;
  I.E2 = A.courant;
  U.ampouleAllumee = A.allumee;
  I.E2 => I.estFerme;
edon
```

*Les trois premières assertions (qui spécifient des égalités de flux) représentent les liens (en trait continu) sur le schéma de la figure 8.6. La dernière assertion modélise l'orientation du courant. En effet la modélisation de l'interrupteur ne tient pas compte du sens de circulation du courant; dans une certaine mesure il est relativement réutilisable. Toutefois l'utilisation d'un tel modèle d'interrupteur implique que le sens de propagation du courant soit modéliser au niveau hiérarchique supérieure; ceci est réalisé à l'aide de la dernière assertion qui spécifie le sens  $E1 \rightarrow E2$ . Cette orientation s'explique simplement:*

- *Si l'on considère le graphe des configurations de l'interrupteur hors de son environnement (voir la figure 8.4 page 59), on peut remarquer que, si le composant est ouvert c'est-à-dire si la variable d'état ouvert vaut true alors toutes les valuations des variables de flux sont possibles. En conséquence, lorsque l'interrupteur est ouvert le flux E2 à destination de l'ampoule est autorisé valoir true. Dans cette situation l'ampoule serait autorisée à s'allumer ce qui est en contradiction avec l'ouverture du circuit.*
- *La contrainte  $I.E2 \Rightarrow I.estFerme$  imposée par le noeud Main interdit que l'interrupteur soit ouvert alors que son flux E2 vaut true (dans ce cas l'ampoule est alimentée). De ce fait, l'ensemble des configurations de l'interrupteur dans l'état ouvert =true n'est plus que { (ouvert = true, E1 = true, E2 = false), (ouvert = true, E1 = false, E2 = false)}: lorsque l'interrupteur est ouvert l'ampoule n'est plus alimentée.*

*Si l'on s'intéresse maintenant au graphe des configurations du noeud Main nous obtenons celui de la figure 8.7.*

---

Les assertions (des noeuds ou des composants) sont de simples expressions booléennes qui doivent être vérifiées par chaque configuration. La notion de contrôle n'apparaît pas dans une expression booléenne. Si il est clair que le contrôleur contraint les flux des sous-composants, il est important de noter que la réciproque est tout aussi vraie: les flux des sous-composants contraignent le contrôleur. Cette remarque est illustrée par l'exemple suivant.

## 8.2. LA HIÉRARCHIE

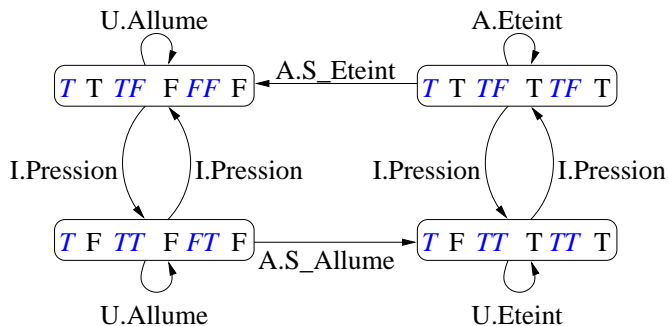


Figure 8.7: Graphe des configurations du circuit. Une configuration est de la forme  $\langle C_S, C_I, C_A, C_U \rangle$  où  $C_X$  est une configuration de  $X$ .

### Exemple 8.8

Nous reprenons l'exemple 8.5 (page 61) dans lequel le composant  $C$  exporte la valeur de son état par un flux  $f$ . Le noeud  $\text{Main}$  spécifie quant à lui que son état doit toujours être égal au flux  $f$  de  $C$ . Le graphe des configurations du système est représenté à la figure 8.8. Sur le graphe des états nous pouvons constater que le système est bloqué dans un des deux états autorisés par les contraintes de  $\text{Main}$  et de  $C$ ; le noeud  $\text{Main}$  ne peut pas changer d'état par l'action  $a$  car il entrerait dans un état violant les assertions.

```

node Composant
  flow f: { 0, 1 };
  event a;
  state etat: { 0, 1 };
  trans etat = 0 |- a -> etat: = 1;
  assert etat = f;
edon

node Main
  event a;
  state etat: { 0, 1 };
  trans etat = 0 |- a -> etat: = 1;
  sub C: Composant;
  assert C.f = etat;
edon
    
```

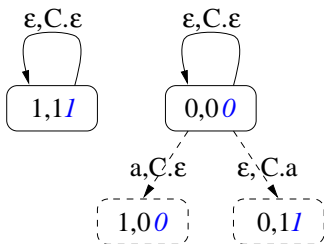


Figure 8.8: Les configurations sont des vecteurs  $\langle \text{etat}, C.\text{etat}, C.f \rangle$ . Les transitions en pointillés sont interdites par les assertions.

### 8.2.2 SYNCHRONISATIONS

Le modèle AltaRica est inspiré du modèle de Arnold et Nivat (voir [49, 22]). L'hypothèse fondamentale que présuppose ce modèle est: “[...] *dans un système global, tous les systèmes composants exécutent leurs transitions simultanément: il est possible de découper le temps en intervalles de telle sorte que pendant chacun de ces intervalles chaque composant exécute une et une seule transition.*” L’hypothèse fondatrice du modèle Arnold-Nivat suppose donc un fort synchronisme entre les composants du système. Paradoxalement, et bien qu’il s’inspire de ce modèle, le langage AltaRica est destiné à la modélisation de systèmes aux comportements asynchrones. Un exemple typique de tels comportements est l’occurrence de défaillances qui sont, par essence, imprévisibles et peuvent donc se produire à tout moment. Dans [22, 33] les auteurs montrent que l’on peut modéliser des systèmes asynchrones malgré l’hypothèse de fort synchronisme; les modélisations de tels systèmes requièrent l’utilisation de l’événement  $\epsilon$  que nous avons introduit précédemment.

Dans l’exemple 8.5 nous avons montré que si aucune contrainte n’est imposée par le contrôleur du noeud et qu’un sous-composant exécute une action (différente de  $\epsilon$ ) alors les autres doivent effectuer l’action  $\epsilon$ . Les contraintes de synchronisation qu’un contrôleur peut imposer aux sous-composants sont des *vecteurs de synchronisation* analogues à ceux nécessaires au produit synchronisé du modèle Arnold-Nivat.

#### Exemple 8.9

*Revenons maintenant sur le graphe des configurations de l’exemple 8.7 (figure 8.7). Deux critiques s’imposent :*

- *L’utilisateur peut allumer (ou éteindre) continuellement l’ampoule. Lorsqu’il a allumé l’ampoule il n’y a aucune raison pour qu’il recommence avant de l’avoir éteinte.*
- *Pour allumer (ou éteindre) l’utilisateur doit presser l’interrupteur mais sur ce graphe l’action de l’utilisateur ne se produit pas en même temps que la pression sur le bouton.*

*La description de cet exemple est clairement incomplète: il manque la spécification du synchronisme entre les actions de l’utilisateur et la pression sur l’interrupteur. Pour spécifier que ces actions doivent être simultanées nous spécifions dans le noeud Main deux vecteurs de synchronisation:*

```
node Main
  ...
  sync
    <U.Allume,I.Pression>; 1
    <U.Eteint,I.Pression>;
edon
```

*La sémantique intuitive de cette contrainte est la suivante:*

<sup>1</sup>L’événement  $\epsilon$  n’existe pas dans la syntaxe du langage; il est ajouté implicitement. Les composants qui n’interviennent pas dans un vecteur de synchronisation sont supposés faire  $\epsilon$  (p. ex. le vecteur  $\langle U.Allume,I.Pression \rangle$  correspond à  $\langle Main.\epsilon, S.\epsilon, U.Allume,I.Pression,A.\epsilon \rangle$ ).

## 8.2. LA HIÉRARCHIE

- Si U.Allume a lieu alors l'événement I.Pression doit avoir lieu.
- Si U.Eteint a lieu alors l'événement I.Pression doit avoir lieu.
- Si I.Pression a lieu alors un des événements U.Allume ou U.Eteint doit avoir lieu.

Le graphe des configurations qui prend en compte la contrainte de synchronisation est représenté à la figure 8.9.

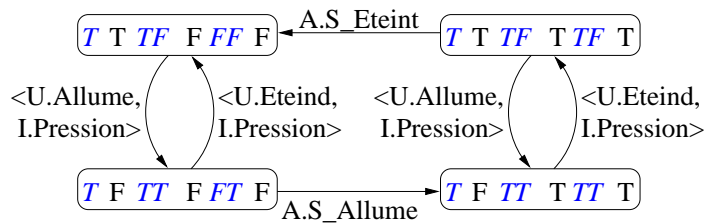


Figure 8.9: Le graphe du circuit prenant en compte la synchronisation de l'utilisateur et de l'interrupteur.

Le modèle du système de l'exemple précédent n'est toujours pas convaincant puisque l'utilisateur a la possibilité d'éteindre avant même que l'ampoule n'ait eu le temps de s'allumer. Afin de résoudre ce problème nous allons indiquer dans le noeud *Main* que les actions de l'ampoule sont "plus rapides" que celles de l'utilisateur. Nous utilisons naturellement le mécanisme des priorités.

### Exemple 8.10

Nous ajoutons dans le modèle *Main* deux événements:

- *CtrlUtilisateur* qui doit se produire en même temps que les actions de l'utilisateur; il est donc synchronisé avec U.Allume ou U.Eteint.
- *CtrlAmpoule* qui doit se produire en même temps que les actions de l'ampoule; il est synchronisé avec A.S\_Allume ou A.S\_Eteint.

Puis nous spécifions que l'événement *CtrlAmpoule* est plus prioritaire que *CtrlUtilisateur*. Il s'en suit que, si le modèle a le choix entre une action de l'utilisateur (contrôlée via *CtrlUtilisateur*) et une action de l'ampoule (contrôlée via *CtrlAmpoule*), il choisit la plus prioritaire spécifiée par le contrôleur c'est-à-dire celle de l'ampoule. Ceci est représenté sur le graphe des configurations de la figure 8.10.

```
node Main
event
  CtrlUtilisateur priority 0,
  CtrlAmpoule priority 1;
trans
  true |- CtrlUtilisateur ->;
  true |- CtrlAmpoule ->;
sync
  <CtrlUtilisateur, I.Pression, U.Allume>;
```

```

<CtrlUtilisateur,I.Pression,U.Eteint>;
<CtrlAmpoule,A.S_Allume>;
<CtrlAmpoule,A.S_Eteint>;
edon
    
```

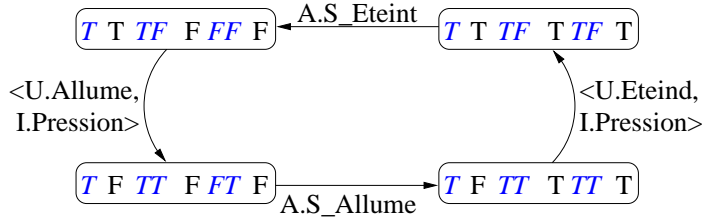


Figure 8.10: Élimination des actions de l'utilisateur par les priorités.

### 8.2.3 SYNCHRONISATIONS AVEC DIFFUSION

Lors de la conception du langage AltaRica nous avons été confrontés à certains problèmes de modélisation. Parmi ces problèmes il nous a été demandé de prendre en compte les *défaillances de causes communes* [140, 5].

Si l'on se réfère à la classification de ces phénomènes donnée dans [5], établir un modèle ou une méthodologie de modélisation systématique des défaillances de causes communes tient de la gageure. Ce problème de modélisation doit certainement être résolu au cas par cas suivant la typologie du système. Toutefois, en nous focalisant sur l'idée intuitive du vocable "défaillances de causes communes", nous avons proposé dans [141, 142] une extension du mécanisme de synchronisation par vecteurs qui permet d'exprimer qu'un événement provoque la défaillance simultanée d'un ensemble de composants. Cette généralisation des vecteurs de synchronisation permet, dans une certaine mesure, de modéliser des phénomènes ressemblant à la *diffusion* (ou *broadcast*). Cette généralisation a été formalisée dans [143].

Exemple 8.11

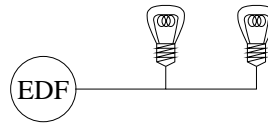


Figure 8.11: Un circuit comprenant une source et deux ampoules.

Considérons le système représenté à la figure 8.11 constitué d'une source d'électricité et de deux ampoules. Pour ce système nous supposons que deux causes peuvent entraîner la coupure du filament d'une ampoule: l'usure du filament ou une surtension à ses bornes. Considérons maintenant les modèles de composants suivants:

```

domain Tension = { nominale, surtension };
    
```

## 8.2. LA HIÉRARCHIE

```
node Source
  flow  tension : Tension;
  event Surtension;
  state mode : Tension;
  trans mode = nominale |- Surtension -> mode := surtension;
  assert tension = mode;
edon

node Ampoule
  flow  tension : Tension;
  event Surtension, Usure;
  state mode : { nominale, hs };
  trans mode = nominale and tension = surtension
    |- Surtension -> mode := hs;
    mode = nominale and tension != surtension
    |- Usure -> mode := hs;
edon

node Main
  sub
    S : Source;
    A1, A2 : Ampoule;
  assert
    S.tension = A1.tension;
    S.tension = A2.tension;
    ...
edon
```

*Une solution pour modéliser la panne simultanée des deux ampoules lors d'une surtension pourrait être d'utiliser la synchronisation  $\langle S.Surtension, A1.Surtension, A2.Surtension \rangle$ . Malheureusement, comme le montre le graphe des configurations de la figure 8.12, à partir du moment où une ampoule est tombée en panne par l'usure de son filament, aucune surtension ne peut avoir lieu (états 2, 3 et 4). Pour corriger cela, nous affaiblissons la contrainte de synchronisation de manière à prendre en compte les cas où les ampoules sont tombées en panne avant la surtension. Nous écrivons donc:*

```
node Main
  ...
  sync
    <S.Surtension, A1.Surtension, A2.Surtension>;
    <S.Surtension, A1.Surtension>;
    <S.Surtension, A2.Surtension>;
    <S.Surtension>;
edon
```

*Le problème maintenant est que, si dans un état une surtension peut avoir lieu alors zéro, une ou deux ampoules peuvent tomber en panne par cette surtension. Si l'on considère l'état 1 du graphe de la figure 8.12 alors les quatre vecteurs présentés*

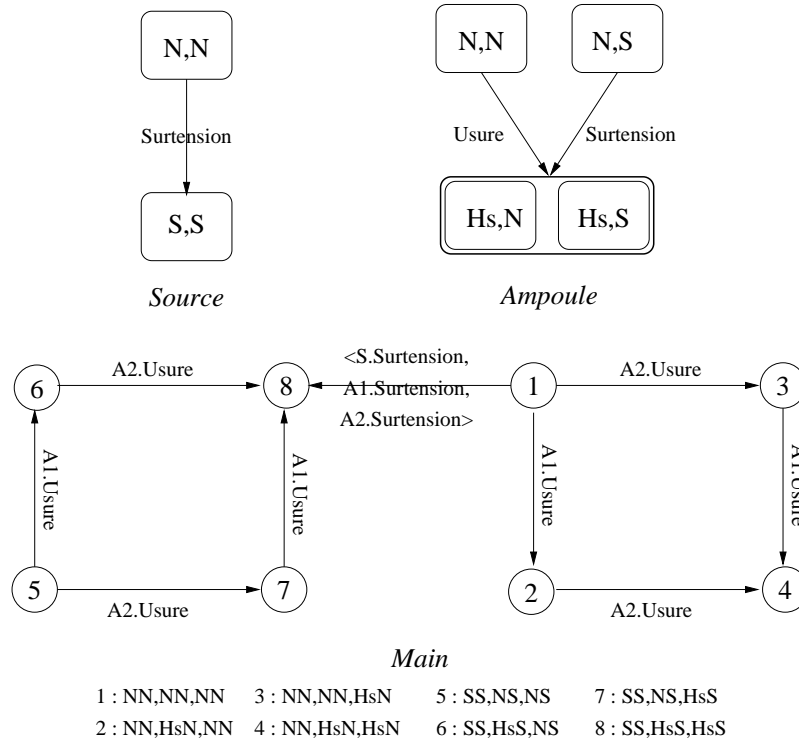


Figure 8.12: Les graphes des configurations pour la source, une ampoule et le système représenté sur la figure 8.11.

ci-dessus peuvent avoir lieu; en particulier,  $\langle S.Surtension \rangle$  qui modélise la surtension de la source sans que les ampoules ne subissent cette même défaillance. De même si l'on considère l'état 2 les vecteurs  $\langle S.Surtension, A2.Surtension \rangle$  et  $\langle S.Surtension \rangle$  sont possibles. Nous voudrions plutôt obtenir:

- pour l'état 1, seul  $\langle S.Surtension, A1.Surtension, A2.Surtension \rangle$  doit être possible parmi les quatres vecteurs pour la surtension;
- pour l'état 2, seul  $\langle S.Surtension, A2.Surtension \rangle$  doit être possible parmi les quatres;
- ...

Le résultat recherché est obtenu en reprenant le principe utilisé dans l'exemple 8.10 où le contrôleur fixe des priorités sur les vecteurs par le biais de ses événements. Il suffit d'imposer l'ordre partiel sur les vecteurs représenté sur la figure 8.13. En utilisant cet ordre nous obtenons le modèle souhaité qui est dessiné sur la figure 8.14.

L'exemple précédent montre le principe général de la synchronisation avec diffusion. Ce mécanisme consiste à considérer un ensemble d'événements de sous-composants et à poser un ordre sur toutes les combinaisons possibles de ces événements. Syntaxiquement cela consiste à spécifier un vecteur de synchronisation dans

## 8.2. LA HIÉRARCHIE

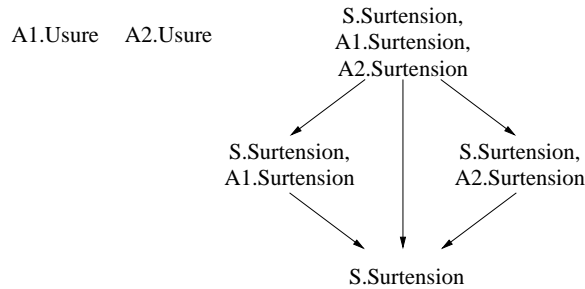


Figure 8.13:

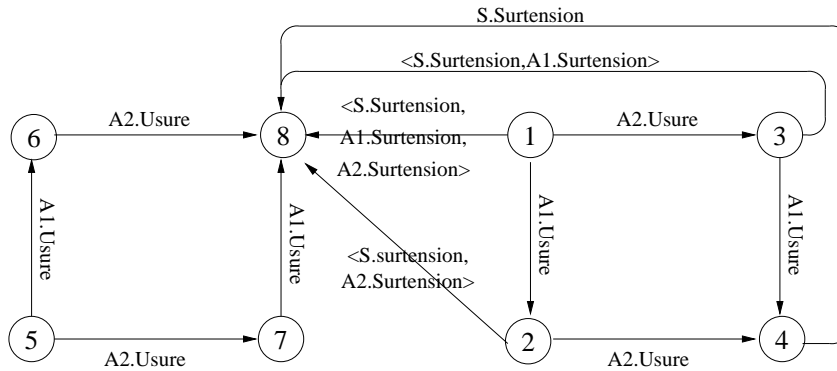


Figure 8.14:

lequel les événements qui doivent être combinés sont marqués par un « ? ». Pour l'exemple précédent, nous aurions écrit:

node *Main*

```
...
sync <S.Surtension, A1.Surtension?, A2.Surtension?>;
edon
```

Le vecteur donné ci-dessus exprime la même contrainte de synchronisation que les quatre vecteurs de l'exemple 8.11 ordonnés par la relation de la figure 8.13. L'événement de la source *S.Surtension* doit toujours être présent dans cette synchronisation; c'est pourquoi il n'est pas marqué par un « ? » (d'un certain point de vue, la source peut être considérée comme l'émettrice du message diffusé).

Le langage permet de contraindre le nombre d'événements marqués qui sont nécessaires à la synchronisation. Si, sur l'exemple 8.11, nous nous intéressons uniquement aux conséquences de la surtension sur les ampoules; le vecteur  $\langle S.Surtension \rangle$  peut alors paraître inutile. En terme de vecteur avec diffusion il suffit de spécifier que la combinaison doit contenir au moins un événement marqué ?; syntaxiquement, cela s'exprime en écrivant  $\geq 1$  à la fin du vecteur.

node *Main*

```
...
sync <S.Surtension, A1.Surtension?, A2.Surtension?> >=1;
edon
```

## 8.3 MODÈLES GRAPHIQUES

### 8.3.1 ALTARICA: UN LANGAGE GRAPHIQUE?

Bien que AltaRica ne soit pas un langage graphique, il est possible de définir une représentation graphique non ambiguë des descriptions écrites dans ce langage.

AltaRica n'est pas un langage graphique car il n'a pas été conçu comme tel. Des exemples de langages graphiques sont par exemple les *statecharts* de D. Harel [72] (mis en oeuvre dans STATEMATE [144]) ou, plus récemment, UML [145] (implémenté dans Rational Rose [146]). Il ne fait aucun doute que l'aspect graphique de tels langages a été la première étape dans l'élaboration du formalisme. En effet la raison d'être de tels modèles est d'offrir aux utilisateurs une syntaxe *graphique* facilitant la description des systèmes. Les langages graphiques de haut niveau souffrent d'une réputation de modèles semi-formels (voire informels) alors que, évidemment, ce n'est pas toujours le cas (p. ex. [72, 75, 76]).

L'objectif de cette section est une proposition de représentation graphique minimale donnant une vue synthétique d'un modèle. Nous espérons que la syntaxe graphique présentée ci-dessous pourra être utilisée pour publier, présenter ou disséminer les modèles AltaRica.

### 8.3.2 VUE GÉNÉRALE D'UN NOEUD ALTARICA

Les noeuds et les composants AltaRica sont représentés de la même manière par un cartouche comportant trois champs: *identification*, *automate* et *structure*.

<i>Identification</i>
Automate
Structure

Le premier champ (*identification*) permet de spécifier soit le nom du modèle, soit le nom du composant suivi du nom de son modèle.

Le champ *automate* permet de décrire les transitions. Nous verrons à la section 8.3.4 que cette description peut prendre deux formes syntaxiques exclusives.

Enfin, le champ *structure* permet de décrire la structure interne du noeud: ses sous-composants et leurs interactions (pour les composants cette partie est utilisée uniquement pour la description de ses assertions).

### 8.3.3 VARIABLES ET ÉVÉNEMENTS

Les variables de flux et d'état sont représentés graphiquement par, respectivement, ○ et ⊙. Les événements sont représentés quant à eux par un losange ◇.

Suivant leur type, ces objets graphiques apparaissent à différents endroits sur le cartouche. Les variables de flux et les événements sont représentés sur la périphérie du noeud: ils représentent l'interface avec l'environnement. Les variables d'état sont représentées à l'intérieur de la partie *Structure*. Chaque objet est étiqueté: par son nom et son domaine pour les variables, et par son nom et sa priorité pour les événements.

La figure 8.15 donne la représentation textuelle et graphique d'un interrupteur.

### 8.3.4 LES TRANSITIONS

La description des macro-transitions définit de manière implicite l'ensemble des transitions de l'automate sous-jacent. La représentation habituelle des automates n'est pas toujours la plus concise (par rapport à une description implicite) mais elle a l'avantage

### 8.3. MODÈLES GRAPHIQUES

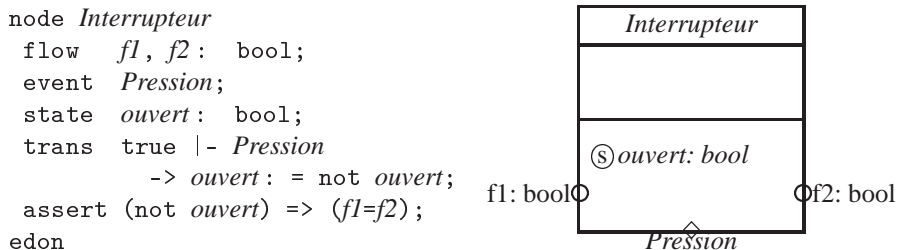


Figure 8.15: À gauche la description textuelle d'un interrupteur et à droite la représentation graphique de ses variables et de son événement.

d'être graphique et expose de manière simple les comportements modélisés. Malheureusement la syntaxe du langage AltaRica n'autorise pas une représentation systématique de l'ensemble des transitions sous la forme d'un automate. Nous autorisons pas conséquent deux écritures (graphique ou textuelle).

**La forme textuelle** est évidemment nécessaire. La partie *Automate* contiendra alors le texte des macro-transitions tel qu'il serait écrit dans un fichier AltaRica.

**Un automate** dessiné graphiquement: si l'ensemble des variables d'état du composant décrit est  $S$  alors un sous-ensemble  $S' \subseteq S$  des variables d'état est choisi afin de représenter les états de l'automate graphique. Chaque état de cet automate est une description d'une valuation des variables d'états de  $S'$ . À chaque état  $s$  (c-à-d. à chaque valuation) est associée une assertion  $a_s$  représentée à l'intérieur de l'état. L'ensemble des couples  $(s, a_s)$  se traduit par la conjonction des formules  $s \Rightarrow a_s$  où la valuation  $s$  est considérée comme une formule (c-à-d. une conjonction d'égalité  $v_s = s(v_s)$  où  $v_s \in S'$ ).

Une transition est étiquetée par: une condition  $g$  sur les variables syntaxiquement autorisées par le langage AltaRica, un événement  $e$  et une affectation  $a$  des variables de  $S \setminus S'$ . Graphiquement, les macro-transitions sont donc représentées sous la forme  $s \xrightarrow{g, e, a} s'$  où  $g$  est la condition,  $e$  l'événement et  $a$  l'affectation. Une telle représentation correspond à la macro-transition AltaRica:  $(s \wedge g, e, a \cup s')$  c'est à dire la valuation  $s$  est considérée comme une garde et  $s'$  est transformée en affectation.

La transition de l'interrupteur représenté sur la figure 8.15 est `true |- Pression -> ouvert := not ouvert`. Celle-ci peut se décrire graphiquement de deux manières (figure 8.16) suivant l'ensemble  $S'$  de variable choisi.

#### 8.3.5 LA HIÉRARCHIE

La représentation de la hiérarchie est très classique et consiste simplement à dessiner les sous-composants à l'intérieur de la partie *Structure* d'un noeud. La figure 8.17 représente graphiquement le noeud *Main* de la description ci-dessous.

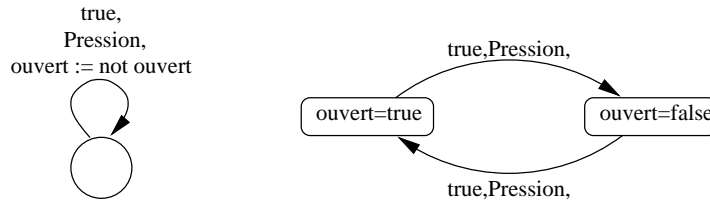


Figure 8.16: Représentation graphique de la macro-transitions de l'interrupteur de la figure 8.15. À gauche le sous-ensemble des variables représentatif de l'état est vide.

```

node Source
  flow  f: bool;
  assert f = true;
edon

//L'interrupteur de la figure 8.15
node Interrupteur
  ...
edon

node Ampoule
  flow  f: bool;
edon

node Main
  sub   S: Source;
        I: Interrupteur;
        A: Ampoule;
  assert S.f = I.f1; I.f2 = A.f;
edon
    
```

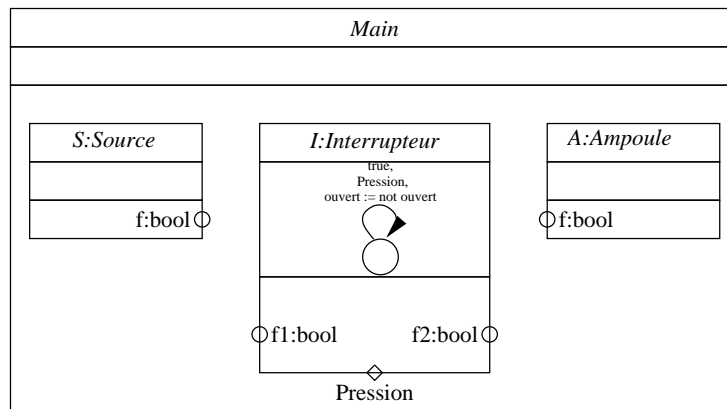


Figure 8.17: Représentation de la hiérarchie

### 8.3.6 LES INTERACTIONS : ASSERTIONS ET VECTEURS DE SYNCHRONISATION

Les interactions entre les sous-composants d'un noeud sont décrites par des assertions et des vecteurs de synchronisation. Ces deux types d'interactions se décrivent de manière similaire par un hypergraphe dont les arêtes connectent les objets impliqués dans l'interaction:

- Pour une assertion il s'agit des variables contenues dans l'expression; l'hyperarête est alors représentée par un  $\odot$ .

### 8.3. MODÈLES GRAPHIQUES

- Pour un vecteur de synchronisation, il s'agit des événements des sous-composants impliqués; l'hyper-arête est alors représentée par un  $\diamond$ .

La figure 8.18 représente le système de la figure 8.17 avec, cette fois-ci, les connexions entre les composants S,I et A.

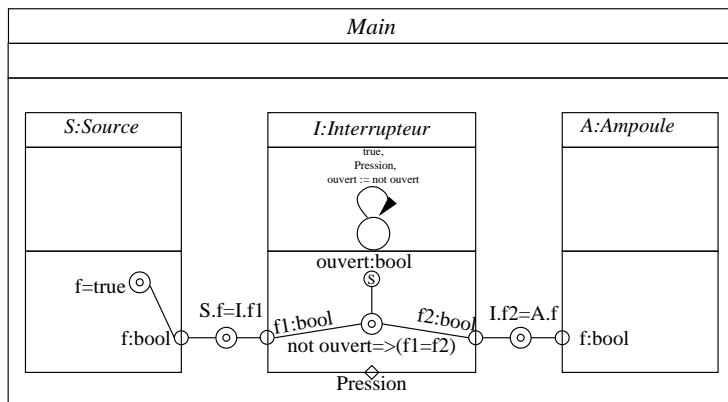


Figure 8.18: Le circuit de la figure 8.17 avec la représentation des assertions.





## SÉMANTIQUE

Dans ce chapitre nous présentons la sémantique du langage AltaRica. Celle-ci s'exprime en terme de *systèmes de transition interfacé* [143]. Nous présentons ce modèle pour lequel nous définissons une relation de bisimulation; celle-ci nous permet de mettre en évidence la compositionnalité de la sémantique.

Après la présentation des systèmes de transition interfacés nous donnons successivement la sémantique des composants et des nœuds AltaRica. Nous montrons que, moyennant une quantification existentielle des variables de flux, les nœuds peuvent être réécrit sous la forme de composants possédant la même sémantique.

Nous terminons ce chapitre en montrant que les composants AltaRica permettent sous certaines hypothèses de décrire les automates temps-réel d'Alur et Dill.

### 9.1 SYSTÈMES DE TRANSITIONS INTERFACÉS

#### Définition 9.1.1 (Système de transitions interfacé)

Un système de transitions interfacé (STI) est un quintuplet  $\mathcal{A} = \langle E, F, S, \pi, T \rangle$  où :

- $E = E_+ \cup \{\epsilon\}$  est un ensemble fini d'événements où  $\epsilon$  est un événement distingué n'appartenant pas à  $E_+$ .
- $F$  est un ensemble de valeurs de flux.
- $S$  est l'ensemble des états du STI et  $\pi : S \rightarrow 2^F$  est une application qui associe à tout état l'ensemble des valeurs de flux qu'il autorise. De plus, nous supposons que pour tout  $s \in S$ ,  $\pi(s) \neq \emptyset$ .
- $T \subseteq S \times F \times E \times S$  est la relation de transition supposée vérifier :
  - $(s, f, e, s') \in T \Rightarrow f \in \pi(s)$
  - $\forall s \in S, (s, f, \epsilon, s) \in T$ .

Une configuration d'un STI est un couple  $(s, f) \in S \times F$  tel que  $f \in \pi(s)$ .

□

#### Définition 9.1.2 (STI avec priorités)

Soit  $\mathcal{A} = \langle E, F, S, \pi, T \rangle$  un STI et  $<$  un ordre partiel strict sur  $E$ . Nous notons  $\mathcal{A} \upharpoonright \leq = \langle E, F, S, \pi, T \upharpoonright < \rangle$  le STI dont l'ensemble des transitions est défini par  $(s, f, e, s') \in T \upharpoonright <$  si et seulement si  $(s, f, e, s') \in T$  et pour tout  $s'' \in S$  et pour tout  $e' \in E$ ,  $(s, f, e', s'') \in T \Rightarrow e \not< e'$ .

□

L'ensemble  $F$  d'un système de transition interfacé peut être considéré comme un ensemble de propriétés associées aux états du système par l'application  $\pi$ . Une *bisimulation interfacée* est une relation de bisimulation qui préserve ces propriétés.

**Définition 9.1.3 (Bisimulation interfacée)**

Si  $\mathcal{A}_1 = \langle E, F, S_1, \pi_1, T_1 \rangle$  et  $\mathcal{A}_2 = \langle E, F, S_2, \pi_2, T_2 \rangle$  sont des systèmes de transition interfacés alors  $R \subset S_1 \times S_2$  est une relation de bisimulation interfacée si :

1.  $\forall s_1 \in S_1, \exists s_2 \in S_2, (s_1, s_2) \in R$  et  $\forall s_2 \in S_2, \exists s_1 \in S_1, (s_1, s_2) \in R$
2.  $\forall (s_1, s_2) \in R, \pi_1(s_1) = \pi_2(s_2)$
3.  $\forall (s_1, f, e, s'_1) \in T_1, s_2 \in S_2, (s_1, s_2) \in R \Rightarrow \exists (s_2, f, e, s'_2) \in T_2, (s'_1, s'_2) \in R$
4.  $\forall (s_2, f, e, s'_2) \in T_2, s_1 \in S_1, (s_1, s_2) \in R \Rightarrow \exists (s_1, f, e, s'_1) \in T_1, (s'_1, s'_2) \in R$

□

Dans [22] il est montré que la bisimulation de deux systèmes de transitions peut être définie à l'aide d'homomorphismes possédant les propriétés suivantes :

**Définition 9.1.4 (Homomorphisme de bisimulation interfacée)**

Soient  $\mathcal{A}_1 = \langle E, F, S_1, \pi_1, T_1 \rangle$  et  $\mathcal{A}_2 = \langle E, F, S_2, \pi_2, T_2 \rangle$  deux systèmes de transitions interfacés. Un homomorphisme de bisimulation interfacée  $h : \mathcal{A}_1 \rightarrow \mathcal{A}_2$  est une application  $h : S_1 \rightarrow S_2$  telle que :

- (h1)  $h$  est surjective
- (h2)  $\forall s \in S_1, \pi_1(s) = \pi_2(h(s))$
- (h3)  $\forall \langle s, f, e, s' \rangle \in T_1, \langle h(s), f, e, h(s') \rangle \in T_2$
- (h4)  $\forall s_1 \in S_1, s'_2 \in S_2, \langle h(s_1), f, e, s'_2 \rangle \in T_2 \Rightarrow \exists s'_1 \in S_1, h(s'_1) = s'_2 \wedge \langle s_1, f, e, s'_1 \rangle \in T_1$

□

**Théorème 9.1 ([22])**

Deux systèmes de transition interfacés,  $\mathcal{A}_1$  et  $\mathcal{A}_2$ , sont en bisimulation interfacée si et seulement si il existe un système de transition interfacé  $\mathcal{B}$  et deux homomorphismes de bisimulation interfacés  $h_1 : \mathcal{A}_1 \rightarrow \mathcal{B}$  et  $h_2 : \mathcal{A}_2 \rightarrow \mathcal{B}$ .

Le fait que deux systèmes de transition interfacés soient en bisimulation est conservé si l'on applique des priorités aux événements de ces STI.

**Proposition 9.1 (Priorités et bisimulation [143])**

Soient  $\mathcal{A}_1 = \langle E, F, S_1, \pi_1, T_1 \rangle$  et  $\mathcal{A}_2 = \langle E, F, S_2, \pi_2, T_2 \rangle$  deux systèmes de transition interfacés et  $<$  un ordre sur  $E$ . Si  $h : \mathcal{A}_1 \rightarrow \mathcal{A}_2$  est un homomorphisme de bisimulation alors  $h$  est aussi un homomorphisme de bisimulation de  $\mathcal{A}_1 \upharpoonright <$  vers  $\mathcal{A}_2 \upharpoonright <$ .

**Preuve :** En annexe, section A.1.1, page 133.

◇

## 9.2 SÉMANTIQUE DES COMPOSANTS

Dans la suite nous considérons que les expressions et formules utilisées dans le langage sont des termes ou des formules d'un langage du premier ordre ; nous notons  $\mathbb{F}$  l'ensemble des formules et  $\mathbb{T}$  l'ensemble des termes. Pour tout élément  $f$  de  $\mathbb{F} \cup \mathbb{T}$  nous notons  $vlib(f)$  l'ensemble de ses variables libres. Nous supposons que  $\mathbb{F}$  contient deux symboles de prédicats d'arité nulle  $\#$  et  $\#$  qui s'interprètent comme les constantes booléennes, respectivement, *vrai* et *faux*.

### Définition 9.2.1 (Composants – Syntaxe abstraite)

Un composant est un septuplet  $\mathcal{C} = \langle V_S, V_F, V_L, E, A, M, < \rangle$  où

- $V_S, V_F$  et  $V_L$  sont des ensembles de variables deux à deux disjoints, appelés respectivement, ensembles des variables d'états, de flux et flux non observables ; nous notons  $V_C$  l'union de ces ensembles.
- $E = E_+ \cup \{\epsilon\}$  est un ensemble d'événements où  $\epsilon$  est un événement n'appartenant pas à  $E_+$ .
- $A \in \mathbb{F}$  est l'assertion du composant ; elle vérifie  $vlib(A) \subseteq V_C$  ;
- $M \subseteq \mathbb{F} \times E \times \mathbb{T}^{V_S}$  est un ensemble de macro-transitions  $(g, e, a)$  telles que :
  - $g \in \mathbb{F}$  est un formule appelée garde ; elle doit vérifier  $vlib(g) \subseteq V_C$  ;
  - $e \in E$  est l'événement déclenchant la transition ;
  - $a : V_S \rightarrow \mathbb{T}$  est une application qui associe à toute variable d'état un terme  $a(s)$  tel que  $vlib(a(s)) \subseteq V_C$ .

L'ensemble  $M$  contient implicitement la macro-transition  $(g_\epsilon, \epsilon, a_\epsilon)$  où  $g_\epsilon = \#$  et  $a_\epsilon$  est l'identité sur  $V_S$ .

- $<$  est une relation d'ordre partielle stricte sur  $E$  telle que  $\epsilon$  soit incomparable à tout autre événement (c.-à-d. pour tout  $e \in E$ ,  $e \not< \epsilon$  et  $\epsilon \not< e$ ).

□

L'ensemble  $V_L$  des variables de flux inobservables a été introduit afin d'établir certaines propriétés de la sémantique. À terme cette notion de variables locales devrait être prise en compte dans le langage.

L'événement  $\epsilon$  est supposé incomparable aux autres événements du composant. Cette hypothèse se justifie par la sémantique intuitive de cet événement : il modélise une évolution de l'environnement alors que le composant ne change pas d'état. Pour un même système, un modèle de composants peut être utilisé dans des contextes différents ; une "bonne" modélisation d'un type de composants devrait alors être élaborée sans hypothèse particulière sur son contexte. Supposer qu'un événement soit plus (ou moins) prioritaire que  $\epsilon$  va à l'encontre de ce principe.

Pour définir la sémantique des composants et des noeuds AltaRica, nous supposons que toutes les variables prennent leurs valeurs dans un même domaine  $\mathcal{D}$ . Une valuation d'un ensemble de variables  $X$  est une application de  $X$  dans  $\mathcal{D}$  et l'ensemble des valuations de  $X$  est noté  $\mathcal{D}^X$ . Toute formule  $f \in \mathbb{F}$  s'interprète par rapport à un ensemble de variables  $X$  tel que  $vlib(f) \subseteq X$ . La sémantique d'une formule  $f$  est alors un sous-ensemble  $\llbracket f \rrbracket$  de  $\mathcal{D}^X$  ; nous posons  $\llbracket \# \rrbracket = \mathcal{D}^X$  et  $\llbracket \# \rrbracket = \emptyset$ . De manière analogue, un terme  $t$  s'interprète par une fonction  $\llbracket t \rrbracket$  de  $\mathcal{D}^X$  dans  $\mathcal{D}$ .

**Définition 9.2.2 (Composants - Sémantique)**

La sémantique d'un composant  $\mathcal{C} = \langle V_S, V_F, V_L, E, A, M, < \rangle$  est un système de transitions interfacé  $\llbracket \mathcal{C} \rrbracket = \langle E, F, S, \pi, T \rangle$  construit de la manière suivante :

- $F = \mathcal{D}^{V_F}$
- $S = \{s \in \mathcal{D}^{V_S} \mid \exists f \in \mathcal{D}^{V_F}, l \in \mathcal{D}^{V_L}, (s, f, l) \in \llbracket A \rrbracket\}$
- $\pi : S \rightarrow 2^F$  est définie par  $\pi(s) = \{f \mid \exists l \in \mathcal{D}^{V_L}, (s, f, l) \in \llbracket A \rrbracket\}$
- $T \subseteq S \times F \times E \times S$  est telle que  $T = \llbracket M \rrbracket \upharpoonright <$  où :
  - $\llbracket M \rrbracket = \bigcup_{t \in M} \llbracket t \rrbracket$
  - $\llbracket (g, e, a) \rrbracket = \{(s, f, e, s') \mid \exists l \in \mathcal{D}^{V_L}, (s, f, l) \in \llbracket A \wedge g \rrbracket \wedge s' = a[s, f, l]\}$   
où  $a[s, f, l]$  dénote la valuation des variables d'état telle que pour tout  $v \in V_S$ ,  $a[s, f, l](v) = \llbracket a(v) \rrbracket(s, f, l)$ .

□

On notera que pour tout  $s \in S$ ,  $\pi(s) \neq \emptyset$  et que la sémantique de l'unique macro-transition étiquetée  $\epsilon$  est l'ensemble  $\{(s, f, \epsilon, s) \mid f \in \pi(s)\}$ .

**Lemme 9.1**

$$\llbracket \langle V_S, V_F, V_L, E, A, M, < \rangle \rrbracket = \llbracket \langle V_S, V_F, V_L, E, A, M, \emptyset \rangle \rrbracket \upharpoonright <$$

**Preuve :** Immédiate d'après la définition de la sémantique. ◇

Le langage AltaRica ne supporte que des formules et des termes sans quantificateur. Toutefois, il suffit d'autoriser l'utilisation de la quantification existentielle dans les gardes des transitions, pour que la notion de priorité devienne un simple artifice syntaxique.

Si  $f \in \mathbb{F}$ ,  $x_i \in \mathcal{V}$  et  $t_i \in \mathbb{T}$  (pour  $i = 1 \dots n$ ) alors nous notons  $f[x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n]$  la formule obtenue en substituant simultanément toutes les occurrences des  $x_i$  par  $t_i$  (pour  $i = 1 \dots n$ ) dans  $f$ .

**Définition 9.2.3 (Priorités syntaxiques)**

Soit  $\mathcal{C} = \langle V_S, V_F, V_L, E, A, M, \emptyset \rangle$  un composant et  $<$  un ordre sur  $E$ . Nous supposons que  $V_S = \{s_1, \dots, s_n\}$ ,  $V_F \cup V_L = \{f_1, \dots, f_p\}$  et nous introduisons un nouvel ensemble de variables  $F' = \{f'_1, \dots, f'_p\}$ . Nous notons  $\mathcal{C} \upharpoonright < = \langle V_S, V_F, V_L, E, A, M \upharpoonright <, \emptyset \rangle$  le composant dont l'ensemble des macro-transitions  $M \upharpoonright <$  est construit de la manière suivante :

$$\langle G, e, a \rangle \in M \upharpoonright < \iff \exists t = \langle g, e, a \rangle \in M \wedge G = g \wedge \bigwedge_{e < e'} \neg V_E(e')$$

où pour tout événement  $e \in E$ ,  $V_E(e)$  est la formule :

$$V_E(e) = \bigvee_{\langle g, e, a \rangle \in M} V_M(\langle g, e, a \rangle)$$

et pour toute transition  $t = \langle g, e, a \rangle \in M$ ,  $V_M(t)$  est la formule :

$$V_M(t) = g \wedge \exists f'_1, \dots, f'_p, A[f_1 \leftarrow f'_1, \dots, f_p \leftarrow f'_p][s_1 \leftarrow a(s_1), \dots, s_n \leftarrow a(s_n)]$$

(On notera que  $\text{vl ib}(G) \subseteq V_C$  puisque toutes les variables  $f'_i$  sont quantifiées.)

### 9.3. SÉMANTIQUE DES NŒUDS

□

Les propriétés suivantes sont des conséquences de la définition 9.2.3 :

**Fait 9.1**  $\mathcal{C} = \mathcal{C} \upharpoonright \emptyset$

**Fait 9.2**  $\forall (s, f, l) \in \llbracket A \rrbracket, (s, f, l) \in \llbracket V_M(t) \rrbracket \iff \exists s' \in \mathcal{D}^{V_S}, (s, f, e, s') \in \llbracket t \rrbracket \wedge \pi(s') \neq \emptyset.$

**Fait 9.3**  $\forall (s, f, l) \in \llbracket A \rrbracket, (s, f, l) \in \llbracket V_E(e) \rrbracket \iff \exists s' \in \mathcal{D}^{V_S}, (s, f, e, s') \in \llbracket M \rrbracket \wedge \pi(s') \neq \emptyset$

#### Lemme 9.2

Si  $\mathcal{C} = \langle V_S, V_F, V_L, E, A, M, \emptyset \rangle$  et  $<$  est un ordre sur  $E$  alors  $\llbracket \mathcal{C} \rrbracket \upharpoonright < = \llbracket \mathcal{C} \upharpoonright < \rrbracket.$

**Preuve :** En annexe, section A.1.2, page 133.

◇

#### Corollaire 9.1

$\llbracket \langle V_S, V_F, V_L, E, A, M, < \rangle \rrbracket = \llbracket \langle V_S, V_F, V_L, E, A, M, \emptyset \rangle \upharpoonright < \rrbracket$

**Preuve :** Immédiate à partir des lemmes 9.1 et 9.2.

◇

Dans la suite, si  $\mathcal{C} = \langle V_S, V_F, V_L, E, A, M, < \rangle$  est un composant AltaRica nous noterons  $\mathcal{C} \upharpoonright$  son équivalent syntaxique  $\langle V_S, V_F, V_L, E, A, M, \emptyset \rangle \upharpoonright <.$

### 9.3 SÉMANTIQUE DES NŒUDS

#### Définition 9.3.1 (Nœuds – Syntaxe abstraite)

Un nœud est un  $(n + 5)$ -uplet  $\mathcal{N} = \langle V_F, E, <, \mathcal{N}_0, \dots, \mathcal{N}_n, V \rangle$  où :

- $V_F$  est une ensemble de variables de flux ;
- $E = E_+ \cup \{\epsilon\}$  est un ensemble d'événements où  $\epsilon$  est un événement (implicite) distingué de ceux de  $E_+ ;$
- $<$  est un ordre partiel strict sur  $E$  tel que  $\epsilon$  soit incomparable ;
- Pour  $i = 1 \dots n, \mathcal{N}_i$  est un composant ou un nœud ;  $V_{F_i}$  est son ensemble de variables de flux,  $E_i$  est son ensemble d'événements. De plus, nous supposons que leurs ensembles de variables sont disjoints ;
- $\mathcal{N}_0$  est un composant particulier appelé contrôleur. Son ensemble d'événements est  $E_0 = E$  ordonné par la relation vide (c.-à-d. tous ses événements sont incomparables). Son ensemble de variables de flux est  $V_{F_0} = V_F \cup V_{F_1} \cup \dots \cup V_{F_n}.$
- $V = V_d \cup V_{imp}$  est un ensemble de vecteurs de synchronisation avec diffusion où :

- $V_d \subseteq E_0^? \times \dots \times E_n^? \times 2^{[0, n+1]}$  est l'ensemble des vecteurs spécifiés. Pour tout  $i \geq 0$ ,  $E_i^?$  est l'ensemble défini par  $E_i^? = E_i \cup \{e^? \mid e \in E_i - \{\epsilon\}\}$ .
- $V_{imp} \subseteq E_0 \times \dots \times E_n \times \{0\}$  est un ensemble de vecteurs implicite construit de la manière suivante :
  - \*  $\langle \epsilon, \dots, \epsilon, \{0\} \rangle \in V_{imp}$ .
  - \*  $\forall i \in [0, n], \forall e_i \in E_i \setminus \{\epsilon\}, [\forall \langle e'_0, \dots, e'_i, \dots, e'_n, D \rangle \in V_d, e'_i \notin \{e_i, e_i^?\}] \Rightarrow \langle \epsilon, \dots, e_i, \dots, \epsilon \rangle \in V_{imp}$ . Le vecteur  $\langle \epsilon, \dots, e_i, \dots, \epsilon, \{0\} \rangle$  ne contient que des  $\epsilon$  hormis  $e_i$  placé à la  $i$ -ème position. Si un événement du sous-nœud  $\mathcal{N}_i$  n'est pas impliqué dans une synchronisation alors cet événement doit se produire seul (tous les autres doivent faire  $\epsilon$ ).

□

Comme cela a été présenté dans la section 8.2, les vecteurs sont implicitement complétés par des  $\epsilon$  (pour les sous-composants qui n'interviennent pas dans la synchronisation par une action différente de  $\epsilon$ ). Ici, afin de simplifier les notations, nous considérons directement des vecteurs à  $n + 1$  composantes.

Dans l'exemple 8.11, nous avons vu qu'un vecteur de synchronisation avec diffusion définit un ensemble partiellement ordonnés de vecteurs de synchronisation Arnold-Nivat. Chaque vecteur de cet ensemble est appelé une *instance* du vecteur avec diffusion.

### Définition 9.3.2 (Instances d'un vecteur de synchronisation)

Soit  $v = \langle e'_0, \dots, e'_n, D \rangle \in V$  un vecteur de synchronisation avec diffusion. Une instance de  $v$  est un vecteur  $u \in E_0 \times \dots \times E_n$  tel que  $u = \langle e_0, \dots, e_n \rangle$  et

- Pour tout  $i \in [0, n]$ ,
  - $e'_i \in E_i \Rightarrow e_i = e'_i$  (les événements non marqués apparaissent nécessairement dans l'instance).
  - $e'_i = b^? \Rightarrow e_i = b \vee e_i = \epsilon$  (les événements marqués peuvent être remplacés par  $\epsilon$ ).
- Le cardinal de l'ensemble  $\{i \mid 0 \leq i \leq n, e_i \neq \epsilon \wedge e'_i = e_i^?\}$  appartient à  $D$ .

L'ensemble  $Inst(v)$  des instances de  $v$  est ordonné par  $\sqsubset$  où :  $\langle e_0, \dots, e_n \rangle \sqsubset \langle e'_0, \dots, e'_n \rangle$  si et seulement si  $\{e_i \mid e_i \neq \epsilon\} \subset \{e'_i \mid e'_i \neq \epsilon\}$ .

□

La sémantique d'un nœud AltaRica s'obtient en construisant le produit synchronisé des sous-nœuds et du contrôleur de telle manière que les contraintes sur les flux imposées par le contrôleur soient respectées. Après cette construction deux « filtres » sont appliqués sur l'ensemble des transitions du produit : le premier consiste à conserver les transitions étiquetées par les instances de vecteur les plus prioritaires ; le second consiste à éliminer les transitions les moins prioritaires par rapport à l'ordre sur les actions du contrôleur.

Par analogie avec le produit synchronisé de Arnold et Nivat, cette opération a été appelée *produit contrôlé* (de systèmes de transitions interfacés) [143].

### Définition 9.3.3 (Nœuds – Sémantique)

Nous considérons un nœud AltaRica  $\mathcal{N} = \langle V_F, E, <, \mathcal{N}_0, \dots, \mathcal{N}_n, V \rangle$ . Nous notons  $\llbracket \mathcal{N}_i \rrbracket = \langle E_i, F_i, S_i, \pi_i, T_i \rangle$  la sémantique de  $\mathcal{N}_i$  (pour  $i=1 \dots n$ ).

## 9.4. SÉMANTIQUE SYMBOLIQUE

La sémantique du nœud  $\mathcal{N}$  est le STI  $\llbracket \mathcal{N} \rrbracket = \langle E, F, S, \pi, T \rangle$  construit de la manière suivante :

- $F = \mathcal{D}^{V_F}$
- $S = \{s \in S_0 \times \dots \times S_n \mid \pi(s) \neq \emptyset\}$
- $\pi(\langle s_0, \dots, s_n \rangle) = \{f \in \mathcal{D}^{V_F} \mid \exists \langle f, f_1, \dots, f_n \rangle \in \pi_0(s_0), \forall i, f_i \in \pi_i(s_i)\}$
- $T \subseteq S \times F \times E \times S$  est construit de la manière suivante :
  - On construit l'ensemble  $W = \bigcup_{v \in V} \text{Inst}(v) \times \{v\}$  et on munit cet ensemble de deux ordres partiels,  $<_?$  et  $<_0$ , définis par :
    - \*  $(u, v) <_? (u', v') \iff v = v' \wedge u \sqsubset u'$
    - \*  $(u, v) <_0 (u', v') \iff u = (e_0, \dots, e_n) \wedge u' = (e'_0, \dots, e'_n) \wedge e_0 < e'_0$
  - On construit ensuite l'ensemble des transitions  $T_{\mathcal{N}} \subseteq S \times F \times W \times S$  défini par  $\langle (s_0, \dots, s_n), f, (u, v), (s'_0, \dots, s'_n) \rangle \in T_{\mathcal{N}}$  si et seulement si il existe  $f_0 = \langle f, f_1, \dots, f_n \rangle \in \pi_0(s_0)$  et  $u = (e_0, \dots, e_n)$  et pour tout  $i \in [0, n]$ ,  $(s_i, f_i, e_i, s'_i) \in T_i$  et  $f_i \in \pi_i(s_i)$ .
  - Enfin la relation  $T$  est définie par :  $\langle s, f, e_0, s' \rangle \in T$  si et seulement si il existe  $(e_0, \dots, e_n, v) \in W$  tel que  $\langle s, f, (e_0, \dots, e_n, v), s' \rangle \in (T_{\mathcal{N}} \upharpoonright <_?) \upharpoonright <_0$ .

□

Le produit contrôlé est compositionnel pour la bisimulation interfacée. Ce résultat est exprimé par le théorème suivant.

### **Théorème 9.2**

Si  $\mathcal{N} = \langle V_F, E, <, \mathcal{N}_0, \dots, \mathcal{N}_n, V \rangle$  et  $\mathcal{N}' = \langle V_F, E, <, \mathcal{N}'_0, \dots, \mathcal{N}'_n, V \rangle$  sont des nœuds AltaRica tels que pour tout  $i = 0 \dots n$ , il existe un homomorphisme de bisimulation  $h_i$  de  $\llbracket \mathcal{N}_i \rrbracket$  vers  $\llbracket \mathcal{N}'_i \rrbracket$  alors il existe un homomorphisme de bisimulation  $h$  de  $\llbracket \mathcal{N} \rrbracket$  vers  $\llbracket \mathcal{N}' \rrbracket$ .

**Preuve :** En annexe, section A.1.3, page A.1.3.

◇

En d'autres termes, le théorème précédent exprime le fait que si il existe un homomorphisme de bisimulation entre  $\mathcal{N}$  et  $\mathcal{N}'$  alors  $\mathcal{N}$  peut être remplacé par  $\mathcal{N}'$  et les systèmes contenant respectivement  $\mathcal{N}$  et  $\mathcal{N}'$  seront alors équivalents pour la bisimulation. Le lecteur pourra trouver à la section 10.3 (page 98) un exemple illustrant ce théorème.

## 9.4 SÉMANTIQUE SYMBOLIQUE

À la section 9.2 nous avons montré qu'il est possible d'exprimer la sémantique des priorités d'un composant AltaRica de manière syntaxique (au prix d'une quantification existentielle). Dans cette section nous montrons qu'il en est de même pour la sémantique d'un nœud. La définition suivante présente la construction, à partir d'un nœud  $\mathcal{N}$ , d'un composant AltaRica  $\mathcal{C}_{\mathcal{N}}$  qui possède la même sémantique que  $\mathcal{N}$ .

**Définition 9.4.1 (Sémantique symbolique)**

Si  $\mathcal{N} = \langle V_F, E, <, \mathcal{N}_0, \dots, \mathcal{N}_n, V \rangle$  est un noeud AltaRica, nous notons  $\mathcal{C}_{\mathcal{N}} = \langle V_S, V_F, V_L, E, A, M, \emptyset \rangle$  le composant AltaRica construit de la manière suivante :

- Pour  $i = 0, \dots, n$  :
  - Si  $\mathcal{N}_i$  est un composant alors nous posons  $\mathcal{N}'_i = \mathcal{N}_i \upharpoonright$  ;
  - Si  $\mathcal{N}_i$  est un noeud alors nous posons  $\mathcal{N}'_i = \mathcal{C}_{\mathcal{N}_i}$  ;
  - Nous posons  $\mathcal{N}'_i = \langle V_{S'_i}, V_{F'_i}, V_{L'_i}, E'_i, A'_i, M'_i, \emptyset \rangle$ .
- $V_S = V_{S'_0} \cup V_{S'_1} \cup \dots \cup V_{S'_n}$
- $V_L = V_{L'_0} \cup V_{L'_1} \cup V_{L'_2} \cup \dots \cup V_{L'_n}$
- $A = A'_0 \wedge \dots \wedge A'_n$
- Nous reprenons l'ensemble  $W$  (ordonné par  $<_?$  et  $<_0$ ) de la définition 9.3.3.
- $M$  est défini de la manière suivante :  $(g, e_0, a) \in M$  si et seulement si il existe  $(g, \langle e_0, \dots, e_n, v \rangle, a) \in (M' \upharpoonright <_?) \upharpoonright <_0$  où  $M' \subseteq \mathbb{F} \times W \times \mathbb{T}^{V_S}$  est l'ensemble défini par :  $(g, \langle e_0, \dots, e_n, v \rangle, a) \in M'$  si et seulement si pour  $i = 0, \dots, n$ , il existe  $(g_i, e_i, a_i) \in M'_i, g = g_0 \wedge \dots \wedge g_n$  et  $\forall x \in V_S, x \in V_{S'_i} \Rightarrow a(x) = a_i(x)$ .

□

**Lemme 9.3**

Soient  $\mathcal{N} = \langle V_F, E, <, \mathcal{N}_0, \dots, \mathcal{N}_n, V \rangle$  un noeud AltaRica et  $\mathcal{C}_{\mathcal{N}} = \langle V_S, V_F, V_L, E, A, M, \emptyset \rangle$  sa réécriture en un composant. Si pour tout  $i = 1 \dots n$ ,  $\mathcal{N}_i$  est un composant alors  $\llbracket \mathcal{N} \rrbracket = \llbracket \mathcal{C}_{\mathcal{N}} \rrbracket$ .

**Preuve :** En annexe, section A.1.4, page 135.

◇

**Théorème 9.3 (Égalité des sémantiques)**

Si  $\mathcal{N}$  un noeud AltaRica et  $\mathcal{C}_{\mathcal{N}}$  sa réécriture en un composant alors  $\llbracket \mathcal{N} \rrbracket = \llbracket \mathcal{C}_{\mathcal{N}} \rrbracket$ .

**Preuve :** En annexe, section A.1.5, page 135.

◇

## 9.5 ALTARICA ET LES AUTOMATES TEMPS-RÉEL

Les variables du langage AltaRica prennent leurs valeurs dans l'ensemble des entiers relatifs, les booléens ou dans un ensemble fini de chaînes de caractères. Si l'on s'autorise l'utilisation de valeurs dans l'ensemble des réels positifs ou nul  $\mathbb{R}_+$  alors les composants AltaRica permettent de décrire les automates temporisés d'Alur et Dill [24, 147].

Un système de transitions temporisé est un système de transitions étiqueté par deux types d'événements: des actions et des délais, qui représentent des modifications discrètes ou continues de l'état du système. Si  $d \in \mathbb{R}_+$  nous notons  $\tau(d)$  une étiquette de délai et nous notons  $\mathcal{L} = A \cup \{\tau(d) \mid d \in \mathbb{R}_+\}$ .

## 9.5. ALTARICA ET LES AUTOMATES TEMPS-RÉEL

### Définition 9.5.1 (Systèmes de transitions temporisés)

Un système de transitions temporisé sur un ensemble fini d'actions  $A$  est un triplet  $S = \langle A, S, \rightarrow \rangle$  où  $S$  est un ensemble d'états et  $\rightarrow \subseteq S \times \mathcal{L} \times S$  est la relation de transition du système. Les éléments de  $\rightarrow$  sont notés  $s \xrightarrow{l} s'$  pour  $s, s' \in S$  et  $l \in \mathcal{L}$ . Enfin, la relation  $\rightarrow$  doit posséder les propriétés suivantes:

**Déterminisme du temps:**  $s \xrightarrow{\tau(d)} s' \wedge s \xrightarrow{\tau(d)} s'' \Rightarrow s' = s''$

**Continuité du temps:**  $s \xrightarrow{\tau(d)} s' \wedge s' \xrightarrow{\tau(d')} s'' \Rightarrow s \xrightarrow{\tau(d+d')} s''$

**Délai nul:**  $s \xrightarrow{\tau(0)} s' \Rightarrow s = s'$

□

Les systèmes de transitions temporisés peuvent posséder un nombre infini d'états et/ou de transitions. Un automate temporisé est une représentation (syntaxique) finie d'un système de transitions temporisé. C'est un automate classique étendu avec un ensemble fini d'horloges (des variables) à valeur dans  $\mathbb{R}_+$ . Ces horloges évoluent à la même vitesse et mesurent le temps écoulé depuis leur dernière mise à zéro. Les valeurs de ces horloges peuvent être testées ou remises à zéro par les transitions de l'automate.

### Définition 9.5.2 (Horloges et contraintes d'horloge)

Une horloge est une variable à valeur dans  $\mathbb{R}_+$ . Si  $X$  est un ensemble d'horloges, une affectation de  $X$  est une application de  $X$  dans  $\mathbb{R}_+$ . Nous notons  $\mathbb{R}_+^X$  l'ensemble des affectations de  $X$ . Si  $u \in \mathbb{R}_+^X$  est une affectation d'horloges et  $d \in \mathbb{R}_+$  alors nous notons  $u + d$  l'affectation définie par:  $\forall x \in X, (u + d)(x) = u(x) + d$ . Si  $X' \subseteq X$ , et  $u \in \mathbb{R}_+^X$  alors  $u[X' \leftarrow 0]$  est l'affectation  $u'$  définie par  $u'(x) = 0$  si  $x \in X'$  et  $u'(x) = u(x)$  si  $x \in X \setminus X'$ .

Une contrainte sur  $X$  est une partie de  $\mathbb{R}_+^X$ ; nous notons  $\mathcal{B}(X)$  l'ensemble des contraintes sur  $X$ .

□

### Définition 9.5.3 (Automates temporisés)

Un automate temporisé  $\mathcal{A}$  est un quintuplet  $\langle L, \Sigma, X, I, \Delta \rangle$  où:

- $L$  est un ensemble fini d'états;
- $\Sigma$  est un ensemble fini d'actions contenant au moins  $\epsilon$ ;
- $X$  est un ensemble fini d'horloges;
- $I : L \rightarrow \mathcal{B}(X)$  est une application qui associe à tout état une contrainte sur les horloges de  $X$ ;
- $\Delta \subseteq L \times \Sigma \times \mathcal{B}(X) \times 2^X \times L$  est un ensemble de transitions. Une transition  $\langle l, a, \phi, \lambda, l' \rangle$  représente un changement d'état de  $l$  vers  $l'$  par l'action  $a$ .  $\phi$  est une garde sur les horloges qui doit être vérifiée pour que la transition soit possible.  $\lambda \subseteq X$  spécifie un sous-ensemble d'horloges remises à zéro lors du passage de la transition.

$\Delta$  est supposé contenir une transition  $\langle l, \epsilon, \mathbb{R}_+^X, \emptyset, l \rangle$  pour tout état  $l \in L$ .

□

La sémantique d'un automate temporisé  $\mathcal{A}$  est un système de transition temporisé  $\mathcal{S}_{\mathcal{A}} = \langle \Sigma, S, \rightarrow \rangle$  défini de la manière suivante:

- $S = \{(l, u) \in L \times \mathbb{R}_+^X \mid u \in I(l)\}$
- $\forall a \in \Sigma, (l, u) \xrightarrow{a} (l', u') \iff \exists \langle l, a, \phi, \lambda, l' \rangle \in \Delta, (l, u) \in \phi \wedge u' = u[\lambda \leftarrow 0]$
- $\forall d \in \mathbb{R}_+, (l, u) \xrightarrow{\tau(d)} (l, u + d) \iff \forall 0 \leq d' \leq d, u + d' \in I(l)$ .

Nous posons  $\pi(s) \subseteq \mathbb{R}_+$  l'ensemble des durées pendant lesquelles le système est autorisé à rester dans l'état  $s$ . Du fait que  $s$  doit vérifier l'invariant  $I$ , l'ensemble  $\pi(s)$  est non vide et contient au moins 0.

Exemple 9.1

La figure 9.1 représente un automate temporel ( $n, n', m$  et  $m'$  sont des constantes).

Une transition  $\langle l, a, \phi, \lambda, l' \rangle$  est représentée sous la forme  $l \xrightarrow[\lambda]{\phi, a} l'$ .

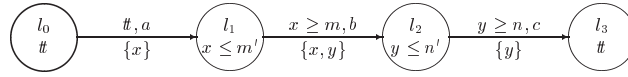


Figure 9.1: Un automate temporel (les transitions  $\epsilon$  ne sont pas dessinées). La sémantique de  $\#$  est  $\mathbb{R}_+^X$ . L'invariant  $I(l)$  est représenté dans l'état  $l$ .

Afin de comparer le modèle des composants AltaRica à celui des automates temporels nous exprimons la sémantique de ces derniers en termes de systèmes de transitions interfacés.

La sémantique d'un automate temporel  $\mathcal{A} = \langle L, \Sigma, X, I, \Delta \rangle$  est un système de transitions interfacé  $\llbracket \mathcal{A} \rrbracket = \langle E, F, S, \pi, T \rangle$  défini par:

- $E = \Sigma \cup \{\tau\}$  est l'ensemble des actions de l'automate auquel nous avons ajouté l'événement  $\tau$  pour les délais.
- $F = \mathbb{R}_+$
- $S = \{(l, u) \in L \times \mathbb{R}_+^X \mid u \in I(l)\}$
- $\pi(s) = \{d \in \mathbb{R}_+ \mid (l, u + d) \in I(l)\}$  associe à tout état l'ensemble des délais pour lesquels l'automate est autorisé à rester.
- $T \subseteq S \times \mathbb{R}_+ \times E \times S$  est définie par:
  - pour tout  $a \neq \tau$ ,  $\langle (l, u), d, a, (l', u') \rangle \in T$  si  $d \in \pi(l, u)$  il existe  $\langle l, a, \phi, \lambda, l' \rangle \in \Delta$  telle que  $u \in \phi$  et  $u' = u[\lambda \leftarrow 0]$ ;
  - $\langle (l, u), d, \tau, (l, u + d) \rangle \in T$  si pour tout  $0 \leq d' \leq d$ ,  $d' \in \pi(l, u)$ .

Étant donné un automate temporel  $\mathcal{A} = \langle L, \Sigma, X, I, \Delta \rangle$  nous pouvons construire un composant AltaRica  $\mathcal{C}_{\mathcal{A}}$  tel que  $\llbracket \mathcal{C}_{\mathcal{A}} \rrbracket = \llbracket \mathcal{A} \rrbracket$ . Ce composant est défini par  $\mathcal{C}_{\mathcal{A}} = \langle V_S, V_F, \emptyset, E, A, M, \emptyset \rangle$  où:

- $V_S = \{l\} \cup X$  où  $l$  est une variable à valeur dans  $L$ ;
- $V_F = \{d\}$  où  $d$  est une variable à valeur dans  $\mathbb{R}_+$ ;

## 9.5. ALTARICA ET LES AUTOMATES TEMPS-RÉEL

- $E = \Sigma \cup \{\tau\}$  ( $\epsilon$  est implicite);
- $A$  est la formule  $\bigwedge_{l_i \in L} (l = l_i) \Rightarrow I(l_i)$ .
- Pour tout  $e \in \Sigma$ ,  $(g, e, a) \in M$  si et seulement si il existe  $\langle l_1, e, \phi, \lambda, l_2 \rangle \in \Delta$  telle que
  - $g$  est la formule  $(l = l_1) \wedge (d = 0) \wedge \phi$ ;
  - $a$  est telle que  $a(l) = l_2$ , pour tout  $x \in \lambda$ ,  $a(x) = 0$  et pour tout  $x \in X \setminus \lambda$ ,  $a(x) = x$ .
- $(tt, \tau, a) \in M$  avec pour tout  $x \in X$ ,  $a(x) = x + d$  et  $a(l) = l$ .

### Exemple 9.2

Si l'on considère l'automate de la figure 9.1 alors celui-ci se décrit par un composant AltaRica  $\mathcal{C}_1 = \langle V_{S_1}, V_{F_1}, \emptyset, E_1, A_1, M_1, \emptyset \rangle$  de la manière suivante:

$$\begin{aligned}
 V_S &= \{l \in N = \{l_0, l_1, l_2, l_3\}, x \in \mathbb{R}_+, y \in \mathbb{R}_+\} \\
 V_F &= \{d \in \mathbb{R}_+\} \\
 E &= \{a, b, c, \tau\} \\
 A &= \begin{aligned} &(l = l_0) \Rightarrow \# \\ &\wedge (l = l_1) \Rightarrow x \leq n' \\ &\wedge (l = l_2) \Rightarrow x \leq m' \\ &\wedge (l = l_3) \Rightarrow \# \end{aligned} \\
 M &= \begin{array}{lll} l = l_0 \wedge d = 0 \wedge \# & \vdash a \rightarrow l := l_1, & x := 0, & y := y \\ l = l_1 \wedge d = 0 \wedge x \geq m & \vdash b \rightarrow l := l_2, & x := 0, & y := 0 \\ l = l_2 \wedge d = 0 \wedge y \geq n & \vdash c \rightarrow l := l_3, & x := x, & y := 0 \\ \# & \vdash \tau \rightarrow l := l, & x := x + d, & y := y + d \end{array}
 \end{aligned}$$

### Théorème 9.4

Si  $\mathcal{A}$  est un automate temporisé alors  $\llbracket \mathcal{A} \rrbracket = \llbracket \mathcal{C}_{\mathcal{A}} \rrbracket$ .

Nous nous intéressons à présent à la composition parallèle de deux automates temporisés. Suivant les auteurs, la synchronisation s'effectue soit sur les actions communes aux automates impliqués [24, 147], soit par une *fonction de synchronisation* [148]. Dans les deux cas, la synchronisation peut s'exprimer en termes de vecteurs Arnold-Nivat (en utilisant éventuellement l'action  $\epsilon$ ) qui sont renommés après le produit synchronisé.

Étant donnés deux automates temporisés définis sur des ensembles d'actions  $\Sigma$  et  $\Sigma'$ , et un ensemble de vecteur de synchronisation  $V \subseteq \Sigma \times \Sigma'$ , un *renommage de la synchronisation* est une application  $\psi$  de  $V$  dans  $\Sigma \cup \Sigma'$ .

### Définition 9.5.4 (Composition d'automates temporisés [148])

Soient  $\mathcal{A}_i = \langle L_i, \Sigma_i, X_i, I_i, \Delta_i \rangle$  pour  $i = 1, 2$  des automates temporisés,  $V$  un ensemble de vecteurs de synchronisation et  $\psi$  un renommage de  $V$ . La composition parallèle de  $\mathcal{A}_1$  et  $\mathcal{A}_2$ , notée  $\mathcal{A}_1 \overset{\psi}{\underset{V}{\parallel}} \mathcal{A}_2$ , est l'automate temporisé  $\langle L, \Sigma, X, I, \Delta \rangle$  tel que:

- $L = L_1 \times L_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $X = X_1 \uplus X_2$  où  $\uplus$  dénote l'union disjointe des deux ensembles d'horloges.
- $\langle l, a, \phi, \lambda, l' \rangle \in \Delta$  si et seulement si il existe  $\langle l_1, a_1, \phi_1, \lambda_1, l'_1 \rangle \in \Delta_1$  et  $\langle l_2, a_2, \phi_2, \lambda_2, l'_2 \rangle \in \Delta_2$  et  $(a_1, a_2) \in V$  tels que  $l = (l_1, l_2)$ ,  $\phi = \phi_1 \wedge \phi_2$ ,  $a = \psi(a_1, a_2)$ ,  $\lambda = \lambda_1 \uplus \lambda_2$  et  $l' = (l'_1, l'_2)$ .

□

La composition parallèle de deux automates temporisés s'exprimant sous la forme d'un automate temporisé, elle peut donc être traduite en terme de composant AltaRica. Toutefois, nous montrons maintenant que la composition de  $\mathcal{A}_1 \mid_V^\psi \mathcal{A}_2$  peut s'exprimer sous la forme d'un noeud synchronisant les composants AltaRica associés aux automates  $\mathcal{A}_1$  et  $\mathcal{A}_2$ .

Supposons que les composants associés à  $\mathcal{A}_1$  et  $\mathcal{A}_2$  sont, respectivement,  $\mathcal{C}_1 = \langle V_{S1}, \{d_1\}, E_1, A_1, M_1, \emptyset \rangle$  et  $\mathcal{C}_2 = \langle V_{S2}, \{d_2\}, E_2, A_2, M_2, \emptyset \rangle$  et considérons un ensemble de vecteurs de synchronisation  $V$  renommés par  $\psi$ . La composition  $\mathcal{A}_1 \mid_V^\psi \mathcal{A}_2$  s'exprime par le noeud AltaRica  $\mathcal{N} = \langle \{d\}, E_1 \cup E_2, \mathcal{N}_0, \mathcal{C}_1, \mathcal{C}_2, V \rangle$  défini par:

- $d$  est une variable prenant ses valeurs dans  $\mathbb{R}_+$ .
- $V$  est l'ensemble des vecteurs de synchronisation  $\langle e, e_1, e_2 \rangle \in (E_1 \cup E_2) \times E_1 \times E_2$  tel que  $e = \psi(e_1, e_2)$ . Nous supposons de plus que  $\langle \tau, \tau, \tau \rangle \in V$  (qui exprime la synchronisation des délais).
- $\mathcal{N}_0 = \langle V_{S0}, V_{F0}, V_{L0}, E_0, A_0, M_0, \emptyset \rangle$  est le composant AltaRica défini par:
  - $V_{S0} = \emptyset$  (le contrôleur n'a qu'un seul état, la valuation vide)
  - $V_{F0} = \{d\}$
  - $V_{L0} = \{d_1, d_2\}$
  - $E_0 = E_1 \cup E_2$
  - $A_0$  est la formule  $d = d_1 \wedge d = d_2$  (qui signifie que l'écoulement du temps est le même pour le contrôleur  $\mathcal{N}_0$  et les composants  $\mathcal{C}_1$  et  $\mathcal{C}_2$ )
  - Pour tout  $a \in E_0$ ,  $\langle \#, a, \emptyset \rangle \in M_0$

**Théorème 9.5**

$$\llbracket \mathcal{A}_1 \mid_V^\psi \mathcal{A}_2 \rrbracket = \llbracket \mathcal{N} \rrbracket.$$



## QUELQUES MODÈLES DE SYSTÈMES

### 10.1 CONCEPTION D'UN ASCENSEUR

La spécification du système considéré (un ascenseur) est inspiré de la thèse de F. Laroussinie [149]. L'objectif de cette section est de construire, à partir d'un ensemble de spécifications informelles, un modèle d'ascenseur qui vérifie les propriétés requises. Le modèle AltaRica proposé ici est tiré du Manuel Méthodologique AltaRica [150].

#### 10.1.1 SPÉCIFICATIONS INFORMELLES

L'ascenseur que nous allons étudier dessert  $n$  étages. La cabine comporte  $n$  boutons lumineux qui permettent de choisir la (ou les) destination(s); lorsqu'un bouton est allumé, il témoigne de l'existence d'une requête pour l'étage auquel il est associé. À chaque étage se trouve un bouton de même type qui permet d'appeler l'ascenseur. Lorsque la cabine s'arrête à un étage la porte s'ouvre alors automatiquement.

Le comportement de l'ascenseur est contrôlé par logiciel. Les changements d'états du contrôleur sont liés aux actions sur les boutons lumineux. Le donneur d'ordre choisira le constructeur qui pourra lui prouver que les comportements suivants sont "vrais":

1. (a) Quand un bouton est enfoncé, le voyant correspondant s'allume;  
(b) Quand la requête correspondante est satisfaite, le voyant s'éteint.
2. À chaque étage la porte n'est jamais ouverte si la cabine n'est pas là.
3. Chaque requête doit être satisfaite "un jour".
4. L'ascenseur ne dessert que les étages pour lesquels il existe une requête.
5. Lorsqu'il n'y a pas de requête, la cabine reste à l'étage où elle se trouve.
6. Lorsque la cabine se déplace, elle doit s'arrêter aux étages auxquels elle passe si une requête le concerne.
7. Lorsqu'il existe plusieurs requêtes, l'ascenseur doit traiter prioritairement celle(s) permettant de continuer dans la même direction que le dernier déplacement.

## 10.1.2 LES CHOIX INITIAUX DE MODÉLISATION

### 10.1.2.1 Le nombre d'étages

L'ascenseur modélisé dessert 4 étages pour les raisons suivantes:

1. Deux étages ne sont pas suffisants car le comportement de l'ascenseur sera obligatoirement équitable. Il passera autant de fois à chacun des étages.
2. Trois étages sont peut-être suffisants mais avec ce nombre d'étages, tout déplacement a pour départ ou pour arrivée une extrémité. Cette particularité risque de perturber les résultats pour les propriétés d'équité.
3. Cinq étages (ou plus) ne semblent pas nécessaires. Si l'on partitionne les étages de la manière suivante  $\{1\}$ ,  $\{2\}$ ,  $\{3, \dots, N-1\}$ ,  $\{N\}$  et qu'à chaque sous-ensemble on associe un numéro d'étage d'un ascenseur à quatre étages (1 pour  $\{1\}$ , 2 pour  $\{2\}$ , 3 pour  $\{3, \dots, N-1\}$  et 4 pour  $\{N\}$ ), il semble que toutes les propriétés que l'on veut vérifier sur un étage ayant 5 étages ou plus, peuvent l'être sur un système à 4 étages.

Les remarques précédentes ne sont pas des preuves mais des remarques de bon sens.

### 10.1.2.2 Les utilisateurs

L'environnement (les personnes) n'est pas modélisé pour la raison suivante. L'environnement d'un ascenseur est composé d'un nombre illimité d'utilisateurs. Dans le modèle cet environnement ne peut donc pas être décrit car cela nécessiterait de borner le nombre d'utilisateurs.

### 10.1.2.3 Quel type de modélisation?

Il y a ici un choix à effectuer concernant la description AltaRica. Doit-elle se situer au niveau fonctionnel, ou bien au niveau implémentation?

**Niveau fonctionnel** Les deux boutons qui concernent un même étage doivent être fonctionnellement regroupés, le fait que l'un soit sur le palier et l'autre dans la cabine n'a, dans ce cas, que peu d'importance.

**Niveau implémentation** Les deux boutons précédents n'utiliseront certainement pas les mêmes moyens pour communiquer avec le logiciel de contrôle du fait que l'un est statique (sur le palier) et l'autre non.

Nous plaçant dans le cadre de la conception d'un ascenseur, nous ne cherchons pas ici à reproduire un ascenseur existant afin d'étudier ses propriétés mais cherchons à en construire un ayant certaines propriétés. La description de l'ascenseur est donc fonctionnelle.

## 10.1.3 LA MODÉLISATION

Le modèle est obtenu par la description des trois objets *physiques* utilisés: le bouton, la porte et la cabine, et de leurs interactions "naturelles".

## 10.1. CONCEPTION D'UN ASCENSEUR

### 10.1.3.1 Une porte

La description de la porte est sans surprise. Il faut juste noter l'état initial qui est déjà fixé et l'exportation de la valeur de l'état par un flux.

```
node Door
  flow doorIsClosed: bool;
  event open, close;
  state closed: bool;
  trans closed |- open -> closed: = false;
      not closed |- close -> closed: = true;
  assert doorIsClosed = closed;
  extern initial_state = closed = true;
edon
```

### 10.1.3.2 Un bouton

Le flux *light* signale que depuis le dernier appui sur le bouton, il n'a pas été réinitialisé. Ce flux représente ainsi la lumière d'un bouton qui est utilisée par l'environnement (voir la description du contrôleur).

```
node Button
  flow light: bool;
  event push, off;
  state on: bool;
  trans true |- push -> on: = true;
      on |- off -> on: = false;
  assert light = on;
  extern initial_state = on = false;
edon
```

### 10.1.3.3 Un étage

Un étage est composé fonctionnellement d'une porte et de deux boutons, un situé sur le palier et l'autre dans la cabine de l'ascenseur.

Les synchronisations du modèle traduisent les phénomènes suivants:

- Pour que la porte puisse s'ouvrir, il faut qu'au moins un des deux boutons puisse s'éteindre. Par contraposée, si les deux boutons sont éteints (c-à-d. il n'y a pas de requêtes pour cet étage), la porte ne pourra pas s'ouvrir.
- Lorsque la porte se ferme, on éteint les boutons si cela est possible (c-à-d. s'ils sont allumés). Cela correspond effectivement à la notion de service effectué.

Comme le système global aura besoin de synchroniser ces événements, ils apparaissent également dans le nœud. Le logiciel de contrôle de l'ascenseur doit connaître à tout instant les requêtes existantes; le flux *request* permet d'indiquer à l'environnement si un des deux boutons pour l'étage est allumé.

```
node Floor
  flow request: bool;
  event open, close;
```

```

trans  true |- open ->;
       true |- close ->;
sub    BC, BF: Button;
       D: Door;
assert request = (BC.light or BF.light);
sync   <open, D.open, BC.off? , BF.off? > >= 1;
       <close, D.close, BC.off? , BF.off? >;
edon

```

#### 10.1.3.4 La cabine

La cabine comporte une porte et ne peut bouger que si la porte est fermée. Elle peut monter (resp. descendre) si elle n'est pas déjà au dernier étage (resp. rez-de-chaussée). Le système global ayant besoin de connaître les ouvertures et les fermetures de la porte, ces événements sont recopiés (et synchronisés) dans le nœud.

Pour le système global, le flux *cageAtFloor* informe de l'étage auquel se trouve la cabine. Nous choisissons le premier étage comme état initial de la cabine.

```

const FirstStage = 0;
const LastStage = 3;

domain Stages = [FirstStage, LastStage];

node Cage
flow  cageAtFloor: Stages;
event up, down, open, close;
state floor: Stages;
sub   D: Door;
trans floor < LastStage and D.doorIsClosed |- up -> floor: = floor+1;
      floor > FirstStage and D.doorIsClosed |- down -> floor: = floor-1;
      true |- open ->;
      true |- close ->;
sync  <open, D.open>;
      <close, D.close>;
assert cageAtFloor = floor;
extern initial_state = floor = FirstStage;
edon

```

#### 10.1.3.5 Le système complet

Le nœud *Main* représente le logiciel de contrôle de l'ascenseur. Son comportement correspond à l'idée intuitive suivante: *pour qu'un ascenseur ait un comportement équitable, il suffit qu'il fonctionne par balayage aller-retour entre le rez-de-chaussée et le dernier étage. Une optimisation simple (et classique) consiste à éviter d'atteindre les extrémités lorsque cela est inutile.*

Une ouverture à un étage donné n'est possible que si la cabine se trouve à cet étage. Le contrôleur (*Main*) utilise donc quatre événements *openX* qui autorisent l'ouverture des portes de la cabine et de l'étage X où elle se situe. Les ouvertures des portes doivent être prioritaires aux déplacements de la cabine; nous introduisons deux événements *up* et *down* qui sont de priorité inférieure aux événements *openX* et qui sont synchronisés avec les actions de déplacement de la cabine.

## 10.1. CONCEPTION D'UN ASCENSEUR

Afin de simplifier l'écriture nous introduisons quatre variables de flux booléennes, *noUp*, *noDown*, *mayUp* et *mayDown* qui s'interprètent de la manière suivante:

**noUp** est vraie lorsque pour un étage donné, tous les boutons des étages supérieurs sont éteints.

**noDown** est vraie lorsque pour un étage donné, tous les boutons des étages inférieurs sont éteints.

**mayUp** est vraie lorsque pour un étage donné, il existe un bouton d'un étage supérieur allumé.

**mayDown** est vraie lorsque pour un étage donné, il existe un bouton d'un étage inférieur allumé.

Le contrôleur a le comportement suivant: après chaque déplacement, il mémorise dans une variable d'état *lastMoveIsDown* le type de mouvement (montée ou descente) qu'il vient d'effectuer. Il s'autorise à monter (resp. descendre) si *mayUp* (resp. *mayDown*) est vraie et si, soit il continue une montée (resp. une descente), soit il n'a plus aucune raison de descendre (resp. de monter).

Pour n'avoir que des états *cohérents*, nous spécifions que l'état initial du contrôleur est *lastMoveIsDown*=true. En effet l'état initial de la cabine étant l'étage inférieur, en terme de comportement, son dernier déplacement ne peut être qu'une descente.

```
node Main
sub    F0, F1, F2, F3 : Floor;
      C : Cage;
flow  noUp, noDown, mayUp, mayDown : bool;
event open0! 1, open1! 1, open2! 1, open3! 1, up! 0, down! 0;
state lastMoveIsDown : bool;
trans C.cageAtFloor = 0 |- open0 ->;
      C.cageAtFloor = 1 |- open1 ->;
      C.cageAtFloor = 2 |- open2 ->;
      C.cageAtFloor = 3 |- open3 ->;
      lastMoveIsDown and noDown and mayUp |- up -> lastMoveIsDown : = false;
      not lastMoveIsDown and mayUp |- up ->;
      not lastMoveIsDown and noUp and mayDown |- down ->lastMoveIsDown : = true;
      lastMoveIsDown and mayDown |- down ->;
sync  <open0, C.open, F0.open>;
      <open1, C.open, F1.open>;
      <open2, C.open, F2.open>;
      <open3, C.open, F3.open>;
      <C.close, F0.close? , F1.close? , F2.close? , F3.close? > = 1;
      <up, C.up>;
      <down, C.down>;
assert mayUp =
      ((C.cageAtFloor = 0 and (F1.request or F2.request or F3.request)) or
       (C.cageAtFloor = 1 and (F2.request or F3.request)) or
       (C.cageAtFloor = 2 and (F3.request)));
      mayDown =
      ((C.cageAtFloor = 3 and (F2.request or F1.request or F0.request)) or
       (C.cageAtFloor = 2 and (F1.request or F0.request)) or
       (C.cageAtFloor = 1 and (F0.request)));
      noDown =
```

```

((C.cageAtFloor = 0) or
 (C.cageAtFloor = 1) and not F0.request) or
 ((C.cageAtFloor = 2) and not (F0.request or F1.request)) or
 ((C.cageAtFloor = 3) and not (F0.request or F1.request or F2.request)));
noUp =
 ((C.cageAtFloor = 3) or
 (C.cageAtFloor = 2) and not F3.request) or
 ((C.cageAtFloor = 1) and not (F3.request or F2.request)) or
 ((C.cageAtFloor = 0) and not (F3.request or F2.request or F1.request)));
extern initial_state = lastMoveIsDown = true;
edon

```

## 10.2 RÉGULATION DU NIVEAU D'UNE CUVE

### 10.2.1 DESCRIPTION

L'exemple présenté dans cette section est tiré de [151]. Le système étudié est représenté sur la figure 10.1. Il consiste en un réservoir contenant un liquide dont le niveau  $L$  doit être maintenu entre deux valeurs  $\alpha_1$  et  $\alpha_2$  ( $> \alpha_1$ ). Cette régulation est obtenue à l'aide d'une pompe principale  $P_1$ , d'une pompe auxiliaire  $P_2$  et d'une vanne d'évacuation  $V$ . Chacun de ces trois composants est commandé par un capteur (observant le niveau dans la cuve) supposé parfait.

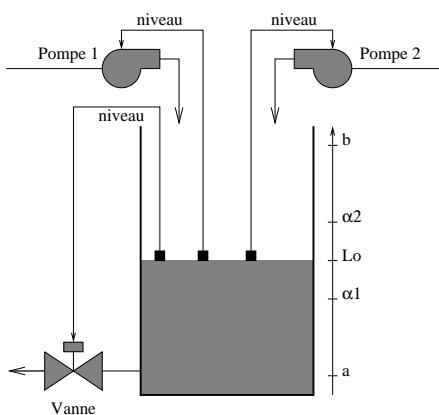


Figure 10.1: La cuve et les unités de contrôle du niveau du liquide.

Dans son état nominal, la position (ouvert ou fermé) d'un composant ( $P_1$ ,  $P_2$  ou  $V$ ) est exprimée en fonction du niveau du liquide. Le tableau 10.1 liste les positions nominales des composants par rapport à cinq zones dans la cuve.

Ces composants fonctionnent de manière indépendante et peuvent être soit ouvert (actif ou bloqué) ou fermé (actif ou bloqué); les composants sont supposés non réparables. Les pompes et la vanne ont un même débit constant au cours du temps.

Le comportement nominal du système de régulation est le suivant. Le niveau  $L$  du liquide reste égal au niveau initial  $L_0 = \frac{1}{2}(\alpha_1 + \alpha_2)$  jusqu'au moment où un composant subit une défaillance. Le niveau  $L$  peut alors évoluer jusqu'à un niveau sortant du domaine de variation autorisé  $]\alpha_1, \alpha_2]$ . Cette sortie du domaine provoque la réaction du système qui ramène  $L$  dans le domaine autorisé à moins qu'une nouvelle défaillance empêche cette régulation.

## 10.2. RÉGULATION DU NIVEAU D'UNE CUVE

Zone	$L$	$P_1$	$P_2$	$V$
1	$L < a$	assèchement de la cuve		
2	$a < L \leq \alpha_1$	ouverte	ouverte	fermée
3	$\alpha_1 < L \leq \alpha_2$	ouverte	fermée	ouverte
4	$\alpha_2 < L \leq b$	fermée	fermée	ouverte
5	$b < L$	débordement de la cuve		

Tableau 10.1: Position des composants dans leur état nominal.

L'étude de ce système consiste à déterminer l'ensemble des scénarios de défaillances de  $P_1$ ,  $P_2$  et  $V$  qui mènent à un débordement ( $b < L$ ) ou à un assèchement de la cuve ( $L < a$ ).

### 10.2.2 MODÉLISATION

La modélisation du niveau de la cuve joue un rôle important car elle influence la taille du modèle pour le système global. Le remplissage de la cuve est un phénomène continu. Les changements de position des composants sont effectués en fonction du niveau de la cuve. Plutôt que de considérer ce niveau, nous nous sommes restreint aux zones traversées par le liquide: la position d'un composant change, non pas lorsque le niveau a atteint  $x$  mètres mais lorsqu'il est dans la zone  $n$ .

Nous définissons tout d'abord un domaine *Zone* qui est l'intervalle  $[1, 5]$  (pour les cinq zones spécifiées dans le tableau 10.1). Un domaine *Debit* représente l'ensemble des taux de variation du niveau de la cuve: si l'on considère que le débit nominal d'un composant ouvert est de 1 unité alors le taux de variation du niveau est la somme des débits des pompes moins celui de la vanne. *Debit* est l'intervalle  $[-1, 2]$  ( $-1$  si la vanne est ouverte alors que les pompes sont fermées et  $2$  si la vanne est fermée alors que les deux pompes sont ouvertes).

```
domain Zone = [ 1, 5];
domain Debit = [-1, 2];
```

Le modèle de la cuve comporte deux variable de flux: *debit* qui indique le taux de variation du niveau de la cuve, et *zone* qui indique à l'environnement la zone dans laquelle se situe le niveau du liquide. La zone où se situe le liquide représente l'état de la cuve; une variable d'état *e\_zone* est donc utilisée. Une transition étiquetée par l'événement *ChangeNiveau* met à jour la variable d'état en fonction de la zone courante et du taux de variation<sup>1</sup>.

```
node Cuve
  flow  debit : Debit;
        zone : Zone;
  state e_zone : integer;
  event ChangeNiveau;
  trans 1 < e_zone and e_zone < 5 and debit! = 0
```

<sup>1</sup>Le lecteur aura certainement remarqué que la variable *e\_zone* prend ses valeurs dans l'ensemble des entiers. En fait, nous avons procédé de cette manière afin de simplifier la modification de l'état pour les cas limites: si la zone est 4 et que le débit est 2 alors la macro-transition devrait faire passer la zone à 6; or, si *e\_zone* prend ses valeurs dans *Zone*, cette transition est impossible. En spécifiant le domaine des entiers nous nous épargnons quelques tests dans les gardes de transition.

```

|- ChangeNiveau -> e_zone := e_zone+debit;
assert if e_zone <= 1 then zone = 1
      else if e_zone >= 5 then zone = 5
      else zone = e_zone;
extern initial_state = e_zone = 3;
edon

```

Les trois unités de régulation ont le même modèle *Composant*. Ce modèle utilise deux variables de flux : *active* qui vaut 0 ou 1 et qui indique à l'environnement le débit du composant (l'environnement a en charge l'interprétation de ce débit: positif pour les pompes et négatif pour la vanne), *ouvert* est un flux booléen qui vaut true si le contrôle du niveau souhaite que le composant soit ouvert.

L'état du composant est modélisé par deux variables. La position (ouvert ou fermé) est une variable entière, à valeur dans [0,1], qui vaut 0 si le composant est fermé et 1 dans le cas contraire (nous avons choisi une variable entière plutôt que booléenne afin de représenter le débit nominal ou nul du composant). Une variable booléenne *def* vaut true si le composant est tombé en panne (blocage ou action intempestive).

Les modes de défaillances du composant sont au nombre de quatre: ouverture et fermeture intempestives et blocage en position (ouvert ou fermé). Quatre événements modélisent ces pannes : *Def\_O* (resp. *Def\_F*) pour l'ouverture (resp. la fermeture) intempestive et *Def\_BO* (resp. *Def\_BF*) pour le blocage en position ouvert (resp. fermé).

Finalement l'assertion permet de spécifier le débit du composant en fonction de son état et de l'ordre d'ouverture du contrôle. Si le composant est défaillant alors le débit est celui provoqué par la panne, sinon le débit du composant est fixé par le contrôle: un ordre *ouvert* à vrai indique un débit de 1 sinon le débit est 0.

Initialement les composants sont supposés non défaillant. Leur débit initial est fixé au niveau hiérarchique supérieur.

```

node Composant
flow  active : [0,1];
      ouvert : bool;
state def : bool;
      debit : [0,1];
event Def_O, Def_F, Def_BO, Def_BF;
trans not def and not ouvert |- Def_O -> def: = true, debit: = 1;
      not def and ouvert |- Def_F -> def: = true, debit: = 0;
      not def and ouvert |- Def_BO -> def: = true, debit: = 1;
      not def and not ouvert |- Def_BF -> def: = true, debit: = 0;
assert if def then active = debit
      else if ouvert then active = 1
      else active = 0;
extern initial_state = def = false;
edon

```

La modélisation du nœud *Main* consiste en:

- La déclaration des quatre composants du système : la cuve *C* et les trois unités de régulation  $P_1$ ,  $P_2$  et *V*.
- Les ordres d'ouverture (ou de fermeture des unités) en fonction de la zone indiquée par la cuve. Ces ordres sont modélisés à l'aide d'une assertion qui décrit le tableau 10.1.

## 10.2. RÉGULATION DU NIVEAU D'UNE CUVE

- L'équation fixant le taux de variation de la cuve en fonction des débits respectifs des unités de régulation.

```
node Main
sub
  P1, P2, V: Composant;
  C: Cuve;
assert
  if C.zone <= 2 then P1.ouvert and P2.ouvert and not V.ouvert
  else if C.zone = 3 then P1.ouvert and not P2.ouvert and V.ouvert
  else not P1.ouvert and not P2.ouvert and V.ouvert;

  C.debit = P1.active+P2.active-V.active;
extern
  initial_state = P1.debit = 1, P2.debit = 0, V.debit = 1;
edon
```

### 10.2.3 SCÉNARIOS MENANT À UN ÉTAT NON SOUHAITÉ

Les scénarios menant à une situation critique (débordement ou assèchement de la cuve) sont au nombre de 24. Nous montrons ici comment se comporte le modèle du système pour deux scénarios; ceux-ci comportent trois défaillances élémentaires et mènent, pour le premier, à un assèchement et, pour le second, à un débordement de la cuve.

**Assèchement:** Initialement le niveau reste stable: la pompe principale  $P_1$  et la vanne  $V$  sont ouvertes et, ayant le même débit, le niveau de la cuve ne change pas de zone. La première défaillance se produit sur  $V$  qui se bloque en position ( $V.Def_{BO}$ ); le niveau de la cuve n'évolue pas puisque le débit de la vanne (défaillante) n'est pas modifié. La seconde défaillance provient de la pompe secondaire  $P_2$  qui se bloque dans sa position courante c'est à dire fermée ( $P2.Def_{BF}$ ); le niveau ne change toujours pas puisque  $P_2$  est toujours fermée. Finalement l'état redouté apparaîtra après la défaillance de la pompe principale  $P_1$  qui se ferme intempestivement ( $P1.Def_F$ ). À partir de ce moment, la vanne étant bloquée en position ouverte, le niveau de la cuve descend en zone 2; le contrôle du niveau tente de fermer  $V$  et d'ouvrir  $P_2$  mais échoue, jusqu'à l'assèchement de la cuve (le niveau est en zone 1).

Ce scénario est représenté sur le tableau 10.2.

**Débordement:** Le premier changement du niveau se produit après une fermeture intempestive de  $P_1$  ( $P1.Def_F$ ). Le niveau se met à baisser jusqu'à la zone 2; à ce moment là, la régulation ordonne à  $V$  de se fermer et à  $P_2$  de s'ouvrir afin de remettre le niveau dans la zone 3. Une fois le niveau en zone 3, la pompe  $P_2$  se ferme et la vanne  $V$  s'ouvre (voir le tableau 10.1) et le niveau se recommence à baisser; le niveau du liquide oscille entre les zones 2 et 3. Alors que le liquide est dans la zone 2, la vanne (fermée) se bloque en position ( $V.Def_{BF}$ ). Cette défaillance ne permet pas d'atteindre l'état critique. En effet, suivant l'oscillation précédente, le niveau remonte en zone 3, la pompe  $P_2$  s'arrête et le niveau reste stable. La situation critique apparaîtra si  $P_2$  est sujet à une ouverture intempestive ( $Def_O$ ). Le niveau entre alors en zone 4 puis en zone 5 car le contrôle ne peut fermer  $P_2$ .

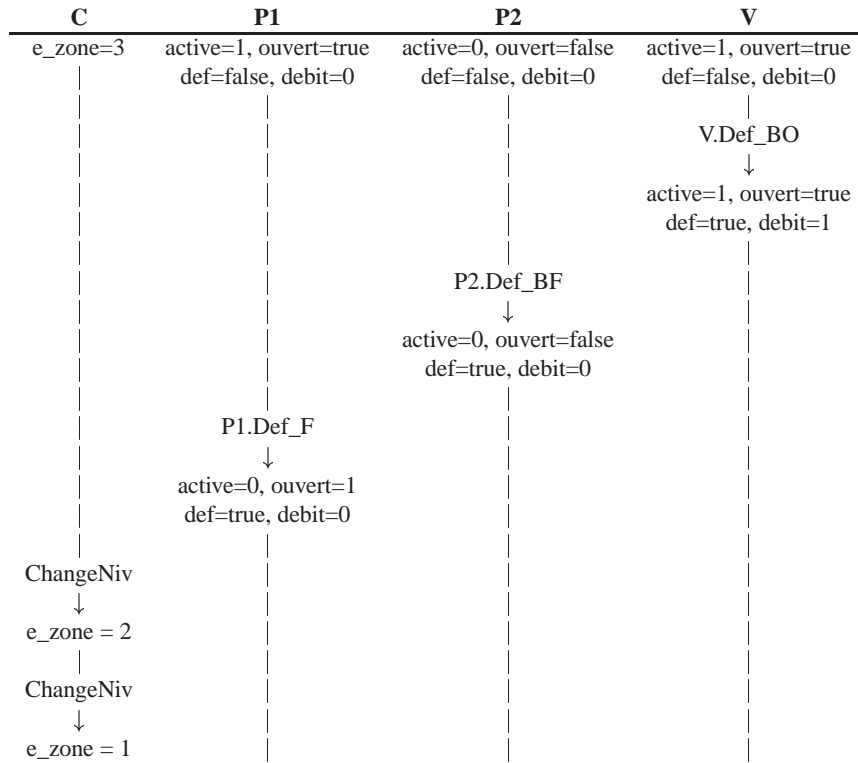


Tableau 10.2: Un scénario menant à l'assèchement de la cuve; l'ordre chronologique des événements est représenté de haut en bas.

### 10.3 RÉDUCTION PAR BISIMULATION

L'exemple que nous présentons ici est tiré de [143] et montre deux noeuds AltaRica en bisimulation interfacée (voir la section 9.3).

#### 10.3.1 DESCRIPTION

Nous considérons deux circuits (figures 10.2 et 10.3). Le premier comporte une source d'électricité  $S$ , un interrupteur  $I$  et un récepteur  $R$ . Dans le second, l'interrupteur du premier est remplacé par deux commutateurs,  $C_1$  et  $C_2$ , de manière à former un système va-et-vient.

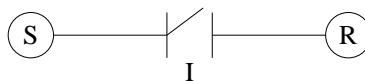


Figure 10.2: Le circuit avec un interrupteur simple  $I$ .

### 10.3. RÉDUCTION PAR BISIMULATION

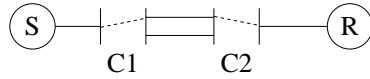


Figure 10.3: Le circuit 10.2 où  $I$  est remplacé par deux commutateurs  $C_1$  et  $C_2$ .

Nous allons montrer que si l'on ne distingue pas les actions sur  $C_1$  et  $C_2$  alors le sous-système formé des deux commutateurs est en bisimulation avec l'interrupteur  $I$  (ce sous-système peut donc être remplacé par  $I$  dans la description du circuit).

#### 10.3.2 MODÉLISATION

Nous ne nous intéressons ici qu'à la modélisation de l'interrupteur, des commutateurs et du mécanisme de va-et-vient.

La description de l'interrupteur  $I$  est sans surprise. Elle utilise deux flux  $f_1$  et  $f_2$  pour les tensions, un événement *Pression* modélisant une action sur le bouton de l'interrupteur et une variable d'état *ouvert* indiquant la position (ouvert ou fermé) du composant. Le système de transition de  $I$  est présenté sur la figure 10.4.

```
node Interrupteur
  flow  $f1, f2$ : bool;
  event Pression;
  state ouvert: bool;
  trans true |- Pression -> ouvert := not ouvert;
  assert (not ouvert) => ( $f1=f2$ );
edon
```

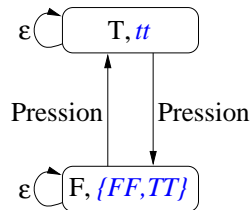


Figure 10.4: Système de transition d'un interrupteur simple.

Les commutateurs se modélisent de manière tout aussi simple. Pour ce type de composant, une variable d'état (*posHaut*) indique simplement quel fil est connecté; celui du haut ou celui du bas. Le système de transition d'un commutateur est représenté sur la figure 10.5.

```
node Commutateur
  flow  $f, haut, bas$ : bool;
  event Pression;
  state posHaut: bool;
  trans true |- Pression -> posHaut := not posHaut;
  assert posHaut => ( $f=haut$ );
         (not posHaut) => ( $f=bas$ );
edon
```

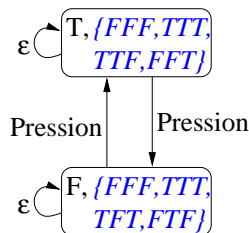


Figure 10.5: STI d'un commutateur utilisé pour un système va-et-vient. Les configurations sont représentées par des affectations du vecteur ( $posHaut, f, haut, bas$ ).

Afin de comparer le sous-système constitué par les deux commutateurs à l'interrupteur simple, nous modélisons un noeud *VaEtVient* qui intègre et connecte les deux commutateurs comme représenté sur la figure 10.6. Le modèle de ce sous-système ne comporte pas de difficulté particulière mais on notera toutefois que, puisque l'on ne distingue pas les actions sur  $C_1$  et  $C_2$ , le noeud *VaEtVient* ne possède qu'un événement *Pression* qui modélise, soit une pression sur  $C_1$  soit une pression sur  $C_2$ ; d'où les deux vecteurs de synchronisation.

Le système de transition du système va-et-vient est représenté sur la figure 10.7.

```

node VaEtVient
  flow  f1, f2 : bool;
  event  Pression;
  trans  true |- Pression -> ;
  sub    B1, B2 : Commutateur;
  sync   <Pression, B1.Pression>;
         <Pression, B2.Pression>;
  assert f1 = B1.f;
         B2.f = f2;
         B1.haut = B2.haut;
         B1.bas = B2.bas;
edon
    
```

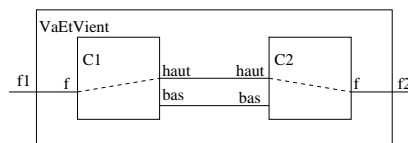


Figure 10.6: Le sous-système va-et-vient

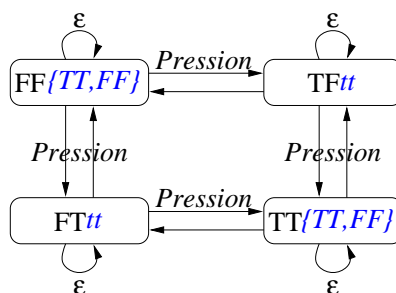


Figure 10.7: STI du système va-et-vient. Les configurations sont représentées par des affectations du vecteur ( $B1.posHaut, B2.posHaut, f1, f2$ ).

### 10.3. RÉDUCTION PAR BISIMULATION

Pour montrer que les STI du va-et-vient et de l'interrupteur (figure 10.4) sont en bisimulation il suffit de remarquer l'autobisimulation [22] pour le va-et-vient: elle nous fournit les classes d'équivalence d'états suivantes:  $\{FF, TT\}$  et  $\{FT, TF\}$ . Lorsque l'on se ramène au graphe quotient par cette relation d'équivalence, on obtient le STI de la figure 10.8 qui isomorphe à celui de l'interrupteur.

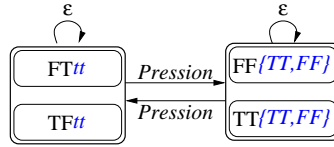


Figure 10.8: Le STI quotient du système va-et-vient par la relation d'équivalence  $\{FF, TT\}$  et  $\{FT, TF\}$ .





## SIMULATION ET ANALYSES QUALITATIVES

### 11.1 LA SIMULATION DES MODÈLES ALTAERICA

La *simulation* d'un modèle consiste à « jouer » de manière plus ou moins dirigée les règles d'évolution (données par la sémantique du modèle) du système modélisé. La simulation est employée lors de deux types d'activité:

**La validation du modèle :** avant même de se soucier du problème de la vérification de propriétés il est nécessaire de s'assurer que le modèle est valide par rapport aux comportements étudiés du système. Cette démarche consiste à se poser la question: le modèle réalisé est-il une vue abstraite du système? Pour répondre à cette question, les utilisateurs de MEC prônent l'utilisation de l'outil de vérification. En effet, puisque le *model-checker* permet de vérifier des propriétés du modèle, il autorise en particulier la vérification de sa conformité par rapport au système. Bien qu'elle augmente la confiance que l'on peut avoir en son modèle, cette méthode a le fâcheux inconvénient de nécessiter (justement) un calcul de propriété qui peut prendre du temps et de l'espace mémoire (dans MEC, le principal problème est l'obtention du modèle global par le calcul du produit synchronisé).

La méthode la plus répandue pour valider un modèle est la simulation pas-à-pas. Elle est non exhaustive mais elle permet de s'assurer que pour les scénarios joués par l'utilisateur, le modèle se comporte comme le système. Les simulateurs de modèle sont souvent dotés d'une interface graphique afin de faciliter la visualisation des comportements du système (p. ex. UPPAAL [25] ou FIABEX [1]).

**L'étude du système :** la simulation consistant à générer des comportements du système à partir du modèle, elle peut être utilisée pour la construction du graphe des états du système. De nombreux algorithmes de vérification sont basés sur cette construction.

Bien que AltaRica soit un langage essentiellement textuel, il est mis en œuvre dans un atelier de saisie graphique (l'atelier AltaRica) qui ressemble à un logiciel de CAO. Les spécifications de cet atelier comprennent l'intégration d'un simulateur graphique de modèles.

Du fait de la mise en parallèle des tâches du projet, l'atelier AltaRica ne met pas tout à fait en œuvre la sémantique du langage telle que spécifiée au chapitre 9.

La mise en œuvre de la sémantique du langage est délicate car elle nécessite le développement d'un « solveur » de contraintes. Ce solveur est nécessaire pour le calcul

des valeurs de flux pour un état donné. À l'heure actuelle le solveur utilisé n'est pas très performant; ceci n'est pas surprenant dans la mesure où nous n'avons posé aucune restriction sur les contraintes acceptées par le langage.

AltaRica est un modèle hiérarchique. La simulation d'un modèle AltaRica repose sur sa mise « à plat »; celle-ci est formellement définie par la sémantique symbolique du langage (section 9.4, page 9.4). Cette sémantique nous permet de considérer un système AltaRica comme un seul composant. Toutefois elle nécessite que le solveur soit capable de résoudre des contraintes quantifiées. Le développement d'un tel solveur est prévu à court terme afin, en particulier, d'intégrer dans l'atelier AltaRica la sémantique définitive du langage.

## 11.2 ANALYSE QUALITATIVE DE SÉQUENCES

Dans cette section nous nous intéressons à la génération et à l'analyse de séquences d'événements à partir de modèles AltaRica. Ces séquences ont la particularité de mener le système dans des états critiques non souhaités (p. ex. la perte du système). L'obtention de ces séquences permet une étude qualitative des scénarios de défaillance et la prise de décision sur l'amélioration des parties les moins fiables du système. Cette prise de décision ne peut se faire à partir d'une liste exhaustive des scénarios de panne. En effet, si l'on considère un système à  $N$  composants, chacun pouvant subir une défaillance, alors le nombre de scénarios potentiels est  $2^N$ . Cette remarque justifie le fait que l'on ne s'intéresse qu'à un sous-ensemble de scénarios dits *minimaux*.

### 11.2.1 NOTION DE SÉQUENCES MINIMALES

Dans la suite de cette section nous considérons une ensemble  $E$  d'événements. Nous notons  $E^*$  l'ensemble des mots finis sur  $E$  et  $\epsilon$  le mot vide. Étant donné un ordre  $<$  sur  $E^*$  et un sous-ensemble  $L \subseteq E^*$  représentant l'ensemble des scénarios critiques, notre objectif est le calcul des mots minimaux de  $L$  pour  $<$ . En d'autres termes, nous nous intéressons au calcul de l'ensemble  $\min(L)$  défini par :  $\min(L) = \{w \in L \mid \forall w' \in L, w' \not< w\}$ .

La pertinence du résultat du calcul des séquences est conditionnée par l'ordre  $<$ . Plusieurs critères de minimalité (l'ordre  $<$ ) pourraient être utilisés.

**La longueur :** une séquence est minimale si sa longueur est minimale. Bien qu'il soit naturel de penser à ce critère il est bien trop restrictif pour être utilisé. En effet, si l'on suppose qu'il existe deux séquences menant à l'état non souhaité,  $ab$  et  $c$  alors, l'ensemble des séquences minimales est  $\{c\}$ . Un tel résultat cache l'implication de  $a$  et de  $b$  dans la réalisation de l'événement redouté. De plus, d'un point de probabiliste, rien ne permet d'affirmer que  $ab$  se produise moins souvent que  $c$  (il suffit que  $c$  soit une défaillance d'un composant très fiable).

**L'ordre préfixe :** une séquence  $s$  est minimale si il n'existe pas dans  $L$  (l'ensemble des séquences critiques), une séquence  $s'$  qui soit un préfixe de ( $c$ 'est-à-dire qui commence)  $s$ . Ce critère n'est pas assez restrictif. En effet, l'ensemble de séquences  $\{abc, bc\}$  ne contient que des séquences minimales. Mais  $a$  ne joue qu'un rôle secondaire dans l'occurrence de l'état redouté : il est nécessaire que  $b$  et  $c$  aient lieu. Une décision efficace pour diminuer le risque d'une situation critique serait d'agir en priorité sur  $b$  et  $c$  (ou du moins sur les composants qui leurs sont associés). Cette remarque est aussi valable pour l'ordre sur les facteurs

## 11.2. ANALYSE QUALITATIVE DE SÉQUENCES

(une séquence est minimale si elle ne contient pas un facteur qui est dans  $L$ ) ou l'ordre suffixe.

...

Pour le calcul des séquences minimales nous avons opté pour l'ordre induit par la notion de *sous-mots* (une séquence  $u$  est inférieure à une séquence  $v$  si  $u$  est une sous-suite de  $v$ ). Cet ordre est nettement moins restrictif que la longueur et les ordres préfixe, facteur ou suffixe sont des cas particuliers de l'ordre par les sous-mots.

### Définition 11.2.1 (Sous-mots)

Si  $u = u_1 \dots u_n$  et  $v = v_1 \dots v_m$  sont deux mots de  $E^*$  alors  $u$  est un sous-mot de  $v$  si il existe une application  $\sigma : [1, n] \rightarrow [1, m]$  telle que :

- Pour tout  $1 \leq i < j \leq n$ ,  $\sigma(i) < \sigma(j)$  ( $\sigma$  est strictement croissante)
- Pour tout  $1 \leq i \leq n$ ,  $u_i = v_{\sigma(i)}$  (on peut projeter les lettres de  $u$  dans  $v$  en conservant l'ordre).

Nous notons  $u \sqsubseteq v$  si  $u$  est un sous-mot de  $v$  (l'ordre strict est noté  $\sqsubset$ ).

□

## 11.2.2 CALCUL DES SÉQUENCES MINIMALES

Étant donné un ensemble de séquences  $L$  nous calculons l'ensemble  $\min(L)$  à l'aide d'un opérateur *ext* (pour extraction) défini sur  $2^{E^*} \times 2^{E^*}$  qui retire d'un ensemble  $X$  les séquences qui possèdent au moins un sous-mot dans un ensemble  $Y$ . L'opérateur *ext* est défini formellement par :

$$\begin{aligned} \text{ext} : 2^{E^*} \times 2^{E^*} &\longrightarrow 2^{E^*} \\ (X, Y) &\longmapsto \{x \in X \mid \forall y \in Y, y \not\sqsubseteq x\} \end{aligned}$$

Dans la suite, si  $a \in E$  et  $L \subseteq E^*$  alors nous notons :

- $a^{-1}L = \{u \in E^* \mid a.u \in L\}$  l'ensemble *résiduel* de  $L$  par la lettre  $a$  ;
- $L_a = a(a^{-1}L)$  l'ensemble des séquences de  $L$  qui commencent par  $a$ .
- $L_{\bar{a}} = L \setminus L_a$  l'ensemble des séquences de  $L$  qui ne commencent pas par  $a$ .

### Lemme 11.1

$\forall L \subseteq E^* \setminus \{\epsilon\}, \forall a \in E, \forall w \in E^*$ ,

$$a.w \in \min(L) \iff w \in \text{ext}(\min(a^{-1}L), L_{\bar{a}})$$

**Preuve :** En annexe, section A.2.1, page 136. ◇

### Lemme 11.2

$\forall L \subseteq E^* \setminus \{\epsilon\}, \forall a, b \in E, \forall w \in E^*$ , si  $a \neq b$  alors

$$b.w \in \min(L) \iff b.w \in \text{ext}(\min(L_{\bar{a}}), L_a)$$

**Preuve :** En annexe, section A.2.2, page 136.  $\diamond$

Le théorème suivant nous permet d'obtenir l'ensemble  $\min(L)$  en appliquant l'opérateur  $ext$  sur la partition de  $L$  en  $L_a$  et  $L_{\bar{a}}$  pour une lettre  $a$  donnée.

**Corollaire 11.1**

Pour tout  $L \subseteq E^*$  et  $a \in E$ ,  $\min(L) = a.ext(\min(a^{-1}L), L_{\bar{a}}) \cup ext(\min(L_{\bar{a}}), L_a)$

**11.2.3 CODAGE DES SÉQUENCES**

**11.2.3.1 Structure de données**

Afin de coder les ensemble de séquences nous définissons une structure de donnée proche des BDD [18] (Binary Decision Diagram). Un diagramme de décision  $\mathcal{D}(E)$  sur une alphabet  $E$  est défini syntaxiquement par :

- $\{\epsilon, \emptyset\} \subseteq \mathcal{D}(E)$
- $a \in E, N_1, N_2 \in \mathcal{D}(E) \Rightarrow \Delta(a, N_1, N_2) \in \mathcal{D}(E)$

Les éléments de  $\mathcal{D}(E)$  sont appelés des nœuds. Chaque élément de  $\mathcal{D}(E)$  représente un sous-ensemble de  $E^*$ ; la fonction sémantique associée aux diagrammes  $\llbracket \cdot \rrbracket : \mathcal{D}(E) \rightarrow 2^{E^*}$  est définie inductivement par :

$$\llbracket \epsilon \rrbracket = \{\epsilon\} \quad \llbracket \emptyset \rrbracket = \{\} \quad \llbracket \Delta(a, N_1, N_2) \rrbracket = a \llbracket N_1 \rrbracket \cup \llbracket N_2 \rrbracket$$

Deux nœuds  $N, N' \in \mathcal{D}(E)$  sont dits *équivalents*, noté  $N \approx N'$ , si :

- $N = N'$
- ou si  $N = \Delta(a, N_1, N_2), N' = \Delta(a, N'_1, N'_2), N_1 \approx N'_1$  et  $N_2 \approx N'_2$

**Théorème 11.1**

Si  $N \approx N'$  alors  $\llbracket N \rrbracket = \llbracket N' \rrbracket$ .

**Preuve :** En annexe, section A.2.3, page 136.  $\diamond$

Un diagramme  $N \in \mathcal{D}(E)$  est dit *réduit* si  $N$  est une *feuille*, (c.-à-d.  $N \in \{\epsilon, \emptyset\}$ ) ou si  $N = \Delta(a, N_1, N_2)$  alors  $N_1 \neq \emptyset$  et  $N$  ne contient pas deux sous-nœuds équivalents.

Étant donné un ordre total  $<$  sur  $E$ , un diagramme  $N \in \mathcal{D}(E)$  est dit *ordonné* si  $N$  est une feuille ou si  $N = \Delta(a, N_1, N_2)$  et  $N_2 = \Delta(b, N'_1, N'_2)$  alors  $a < b$ .

**Théorème 11.2**

Si  $N$  et  $N'$  sont des diagrammes réduits et ordonnés tels que  $\llbracket N \rrbracket = \llbracket N' \rrbracket$  alors  $N \approx N'$ .

**Preuve :** En annexe, section A.2.4, page 137.  $\diamond$

## 11.2. ANALYSE QUALITATIVE DE SÉQUENCES

Si  $L \subseteq E^*$  est non vide et n'est pas le singleton  $\{\epsilon\}$  alors nous notons  $\lambda(L) = \min\{a \in E \mid a^{-1}L \neq \emptyset\}$ ;  $\lambda(L)$  est la plus petite lettre de  $E$  telle que le résiduel de  $L$  par  $\lambda(L)$  soit non vide.

La construction d'un  $\mathcal{D}(E)$  à partir d'un ensemble  $L \subseteq E^*$  de mots s'effectue de manière récursive sur  $L_a$  et  $L_{\bar{a}}$  où  $a = \lambda(L)$ . Le diagramme associé à un langage  $L$  peut être construit à l'aide de la fonction  $\text{build} : 2^{E^*} \rightarrow \mathcal{D}(E)$  définie de la manière suivante :

- $\text{build}(\emptyset) = \emptyset$
- $\text{build}(\{\epsilon\}) = \epsilon$
- $\text{build}(L) = \Delta(a, \text{build}(a^{-1}L), \text{build}(L_{\bar{a}}))$  si  $L \notin \{\emptyset, \{\epsilon\}\}$  et  $a = \lambda(L)$

Dans la suite nous considérons uniquement des diagrammes réduits et ordonnés.

Nous représenterons un diagramme sous la forme d'un graphe dirigé acyclique possédant deux feuilles étiquetés  $\epsilon$  et  $\emptyset$ . Un nœud intermédiaire du graphe représente un diagramme de la forme  $N = \Delta(a, N_1, N_2)$ .  $N$  est étiqueté par  $a$  et possède deux arcs sortants l'un vers  $N_1$  et l'autre, marqué par un point ( $\bullet$ ), relie  $N$  à  $N_2$ .

Exemple 11.1 (Construction d'un diagramme)

Considérons  $E = \{a, b, c\}$  tel que  $a < b < c$  et  $L = \{abc, bc, ac\}$ . Nous avons :

1.  $\text{build}(L) = \Delta(a, \text{build}(\{bc, c\}), \text{build}(\{bc\}))$
2.  $\text{build}(\{bc, c\}) = \Delta(b, \text{build}(\{c\}), \text{build}(\{c\}))$
3.  $\text{build}(\{bc\}) = \Delta(b, \text{build}(\{c\}), \text{build}(\emptyset))$
4.  $\text{build}(\{c\}) = \Delta(c, \text{build}(\{\epsilon\}), \text{build}(\emptyset))$

Le diagramme associé à  $L$  est représenté sur la figure 11.1.

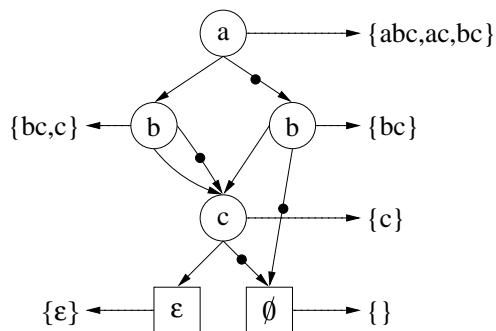


Figure 11.1: Le diagramme représentant le langage  $L = \{abc, bc, ac\}$ . À chaque nœud est associée sa sémantique.

### 11.2.3.2 Opérations sur les diagrammes

Nous définissons maintenant l'ensemble des opérations qui nous sont nécessaires pour le calcul des séquences minimales : l'union, l'intersection, la concaténation, l'extraction (de séquences) et les séquences minimales.

a) *Union*

- $\text{union} : \mathcal{D}(E) \times \mathcal{D}(E) \rightarrow \mathcal{D}(E)$
- $\text{union}(\emptyset, \emptyset) = \emptyset$
- $\text{union}(\emptyset, \epsilon) = \epsilon$
- $\text{union}(\emptyset, \Delta(a, N_1, N_2)) = \Delta(a, N_1, N_2)$
- $\text{union}(\epsilon, \emptyset) = \epsilon$
- $\text{union}(\epsilon, \epsilon) = \epsilon$
- $\text{union}(\epsilon, \Delta(a, N_1, N_2)) = \Delta(a, N_1, \text{union}(N_2, \epsilon))$
- $\text{union}(\Delta(a, N_1, N_2), \emptyset) = \Delta(a, N_1, N_2)$
- $\text{union}(\Delta(a, N_1, N_2), \epsilon) = \Delta(a, N_1, \text{union}(N_2, \epsilon))$
- $\text{union}(\Delta(a, N_1, N_2), \Delta(a', N'_1, N'_2)) = \Delta(m, N, N')$  où
  - $m = \min\{a, a'\}$
  - si  $a = a'$  alors  $N = \text{union}(N_1, N'_1)$  et  $N' = \text{union}(N_2, N'_2)$
  - si  $a < a'$  alors  $N = N_1$  et  $N' = \text{union}(N_2, \Delta(a', N'_1, N'_2))$
  - si  $a' < a$  alors  $N = N'_1$  et  $N' = \text{union}(N'_2, \Delta(a, N_1, N_2))$

b) *Concaténation*

- $\text{concat} : \mathcal{D}(E) \times \mathcal{D}(E) \rightarrow \mathcal{D}(E)$
- $\text{concat}(\emptyset, \emptyset) = \emptyset$
- $\text{concat}(\emptyset, \epsilon) = \emptyset$
- $\text{concat}(\emptyset, \Delta(a, N_1, N_2)) = \emptyset$
- $\text{concat}(\epsilon, \emptyset) = \emptyset$
- $\text{concat}(\epsilon, \epsilon) = \epsilon$
- $\text{concat}(\epsilon, \Delta(a, N_1, N_2)) = \Delta(a, N_1, N_2)$
- $\text{concat}(\Delta(a, N_1, N_2), \emptyset) = \emptyset$
- $\text{concat}(\Delta(a, N_1, N_2), \epsilon) = \Delta(a, N_1, N_2)$
- $\text{concat}(\Delta(a, N_1, N_2), \Delta(a', N'_1, N'_2)) = \text{union}(N, N')$  où
  - $N = \Delta(a, \text{concat}(N_1, N'_1), \emptyset)$
  - $N' = \text{concat}(N_2, N'_2)$
  - $N'' = \Delta(a', N'_1, N'_2)$

## 11.2. ANALYSE QUALITATIVE DE SÉQUENCES

### c) Intersection

À toute lettre  $a$  de  $E$  nous associons le diagramme  $\text{ltr}(a) = \Delta(a, \epsilon, \emptyset)$  (on notera que  $\llbracket \text{ltr}(a) \rrbracket = \{a\}$ ).

- $\text{inter} : \mathcal{D}(E) \times \mathcal{D}(E) \rightarrow \mathcal{D}(E)$
- $\text{inter}(\emptyset, \emptyset) = \emptyset$
- $\text{inter}(\emptyset, \epsilon) = \emptyset$
- $\text{inter}(\emptyset, \Delta(a, N_1, N_2)) = \emptyset$
- $\text{inter}(\epsilon, \emptyset) = \emptyset$
- $\text{inter}(\epsilon, \epsilon) = \epsilon$
- $\text{inter}(\epsilon, \Delta(a, N_1, N_2)) = \text{inter}(\epsilon, N_2)$
- $\text{inter}(\Delta(a, N_1, N_2), \emptyset) = \emptyset$
- $\text{inter}(\Delta(a, N_1, N_2), \epsilon) = \text{inter}(N_2, \epsilon)$
- $\text{inter}(\Delta(a, N_1, N_2), \Delta(a', N'_1, N'_2)) = N$  où
  - si  $a = a'$  alors  $N = \text{union}(\text{concat}(\text{ltr}(a), N'), N'')$  où :
    - \*  $N' = \text{inter}(N_1, N'_1)$
    - \*  $N'' = \text{inter}(N_2, N'_2)$
  - si  $a < a'$  alors  $N = \text{inter}(N_2, \Delta(a', N'_1, N'_2))$
  - si  $a > a'$  alors  $N = \text{inter}(N_1, \Delta(a, N_1, N_2))$

### d) Extraction

- $\text{ext} : \mathcal{D}(E) \times \mathcal{D}(E) \rightarrow \mathcal{D}(E)$
- $\text{ext}(\emptyset, \emptyset) = \emptyset$
- $\text{ext}(\emptyset, \epsilon) = \emptyset$
- $\text{ext}(\emptyset, \Delta(a, N_1, N_2)) = \emptyset$
- $\text{ext}(\epsilon, \emptyset) = \epsilon$
- $\text{ext}(\epsilon, \epsilon) = \emptyset$
- $\text{ext}(\epsilon, \Delta(a, N_1, N_2)) = \text{ext}(\epsilon, N_2)$
- $\text{ext}(\Delta(a, N_1, N_2), \emptyset) = \Delta(a, N_1, N_2)$
- $\text{ext}(\Delta(a, N_1, N_2), \epsilon) = \emptyset$
- $\text{ext}(\Delta(a, N_1, N_2), \Delta(a', N'_1, N'_2)) = \text{union}(N, N')$  où
  - si  $a = a'$  alors  $N = \text{concat}(\text{ltr}(a), \text{inter}(\text{ext}(N_1, N'_1), \text{ext}(N_1, N'_2)))$
  - si  $a < a'$  alors  $N = \text{concat}(\text{ltr}(a), \text{ext}(N_1, \Delta(a', N'_1, N'_2)))$
  - si  $a' < a$  alors  $N = \text{inter}(\alpha, \beta)$  où :
    - \*  $\alpha = \text{concat}(\text{ltr}(a), \text{ext}(N_1, \Delta(a', N'_1, \emptyset)))$
    - \*  $\beta = \text{ext}(\Delta(a, N_1, \emptyset), N'_2)$
  - $N' = \text{ext}(N_2, \Delta(a', N'_1, N'_2))$

e) Séquences minimales

- $\min : \mathcal{D}(E) \rightarrow \mathcal{D}(E)$
- $\min(\emptyset) = \emptyset$
- $\min(\epsilon) = \epsilon$
- $\min(\Delta(a, N_1, N_2)) = \text{union}(N, N')$  où
  - $N = \text{concat}(\text{ltr}(a), \text{ext}(\min(N_1), N_2))$
  - $N' = \text{ext}(\min(N_2), \text{concat}(\text{ltr}(a), N_1))$

f) Correction des opérations

**Théorème 11.3**

Pour tout  $N_1, N_2 \in \mathcal{D}(E)$  :

1.  $\llbracket \text{union}(N_1, N_2) \rrbracket = \llbracket N_1 \rrbracket \cup \llbracket N_2 \rrbracket$
2.  $\llbracket \text{concat}(N_1, N_2) \rrbracket = \llbracket N_1 \rrbracket \cdot \llbracket N_2 \rrbracket$
3.  $\llbracket \text{inter}(N_1, N_2) \rrbracket = \llbracket N_1 \rrbracket \cap \llbracket N_2 \rrbracket$
4.  $\llbracket \text{ext}(N_1, N_2) \rrbracket = \text{ext}(\llbracket N_1 \rrbracket, \llbracket N_2 \rrbracket)$
5.  $\llbracket \min(N_1) \rrbracket = \min(\llbracket N_1 \rrbracket)$

**Preuve :** En annexe, section 11.3, page 110. ◇

**11.2.3.3 Sur la taille des diagrammes pour les séquences minimales**

Il existe des langages dont la représentation est polynomiale en nombre de nœuds alors que la représentation leur ensemble de séquences minimales est exponentielle.

En effet, pour tout  $n \geq 3$ , nous fixons un alphabet de  $n$  lettres  $E = \{e_1, \dots, e_n\}$  et nous notons  $L_n$  le langage défini par :  $L_n = E^n \cup \{e.e \mid e \in E\}$ .  $L_n$  comprend l'ensemble des mots à  $n$  lettres dans  $E$  et l'ensemble des carrés de  $E$ .

**Conjecture 11.1**

Pour tout  $n \geq 3$ , le nombre de nœuds de  $\text{build}(L_n)$  est  $\frac{1}{2}(3n^2 + 3n + 2)$  tandis que celui de  $\text{build}(\min(L_n))$  est  $n2^{n-1} + n + 1$ .

Ce résultat a été établi de manière expérimentale lorsque nous avons constaté un résultat analogue sur le nombre d'état de l'automate minimal codant ces langages :

**Théorème 11.4**

Pour tout  $n \geq 3$ , le nombre d'états de l'automate minimal reconnaissant le langage  $L_n$  est  $2n + 1$  alors que le nombre d'états de l'automate minimal reconnaissant  $\min(L_n)$  est  $2^n$ .

### 11.3. VERS LES FORMULES BOOLÉENNES

**Preuve :** En annexe, section A.2.6, page 140.  $\diamond$

#### 11.2.4 GÉNÉRATION DE SÉQUENCES MINIMALES

L'algorithme mis en œuvre pour la génération de séquences critiques n'est pas spécifique au modèle AltaRica et peut s'appliquer à tout modèle dont on possède une sémantique opérationnelle en terme de système de transition étiqueté fini.

L'algorithme de calcul des séquences critiques consiste en un parcours en profondeur du système de transition considéré comme un automate dont l'ensemble  $F$  des états d'acceptation est l'ensemble des états non souhaités.

La figure 11.2 donne l'algorithme de parcours de l'automate. Dans cette figure la structure de donnée codant Langage est un  $\mathcal{D}(E)$ . Pour la mise en œuvre de cette structure nous nous sommes inspirés des techniques employées sur les BDDs [152] (p. ex. table unique de stockage des nœuds, cache sur les opérations, . . . ).

La fonction CalculSequences applique à chaque état initial la fonction ParcoursSequences qui retourne l'ensemble des traces de chemins minimales issues de l'état initial donné en argument. Le résultat est l'application de l'opérateur  $\min$  sur l'ensemble des traces calculées.

ParcoursSequences effectue un parcours en profondeur de l'automate. Lorsque la fonction rencontre un état acceptant elle retourne la séquence vide (ligne 4). Dans le cas contraire elle s'applique récursivement sur les successeurs de l'état courant (si celui-ci est un état sans successeurs alors elle retourne l'ensemble vide – lignes 6,7,11 et 12). Si un état successeur est dans la pile alors il est ignoré (ligne 8); en effet, dans ce cas le chemin courant contient un cycle et la trace correspondant n'est donc pas minimale. Les lignes 10 et 11 effectuent les calculs ensemblistes du calcul des séquences minimales.

### 11.3 VERS LES FORMULES BOOLÉENNES

Dans cette section nous nous intéressons à la génération de formules booléennes à partir de modèles AltaRica. Cette compilation permet, à l'aide d'un outil tel que Aralia[11], l'analyse qualitative et quantitative des combinaisons de défaillances qui mènent le système dans un état non souhaité.

#### 11.3.1 FORMULES BOOLÉENNES ET IMPLICANTS PREMIERS

Une *formule booléenne* est un terme construit inductivement à partir des deux constantes booléennes 0 (pour faux) et 1 (pour vrai), d'un ensemble de variables  $V$  et des opérateurs logiques  $\vee$  (disjonction),  $\wedge$  (conjonction) et  $\neg$  (négation).

Un *littéral* est une variable  $x$  ou sa négation  $\neg x$ ;  $x$  et  $\neg x$  sont dits *opposés*. Un *produit* est un ensemble de littéraux qui ne contient pas à la fois un littéral et son opposé. Un produit est assimilé à la conjonction de ses éléments et un ensemble de produit est assimilé à la disjonction de ses éléments.

Une *affectation* est une application de  $V$  dans  $\{0, 1\}$ . À toute formule  $f$  on associe à une application  $\llbracket f \rrbracket : (V \rightarrow \{0, 1\}) \rightarrow \{0, 1\}$  définie inductivement sur la structure de  $f$  par :  $\llbracket 0 \rrbracket(\sigma) = 0$ ,  $\llbracket 1 \rrbracket(\sigma) = 1$ ,  $\llbracket x \rrbracket(\sigma) = \sigma(x)$ ,  $\llbracket f_1 \vee f_2 \rrbracket(\sigma) = \llbracket f_1 \rrbracket(\sigma) \vee \llbracket f_2 \rrbracket(\sigma)$ ,  $\llbracket f_1 \wedge f_2 \rrbracket(\sigma) = \llbracket f_1 \rrbracket(\sigma) \wedge \llbracket f_2 \rrbracket(\sigma)$  et  $\llbracket \neg f \rrbracket(\sigma) = \neg \llbracket f \rrbracket(\sigma)$ .

Une affectation  $\sigma$  *satisfait* une formule  $f$  si  $\llbracket f \rrbracket(\sigma) = 1$ . Un produit peut être considéré comme une affectation partielle en affectant 1 aux variables apparaissant

```

1 Langage ParcoursSequences(Etat s)
2 début
3   Langage  $l_s$ ;
4   si s est un état acceptant alors  $l_s = \{\epsilon\}$ ;
5   sinon
6      $l_s = \emptyset$ ;
7     pour toute transition  $s \mid e \rightarrow s'$ 
8       si  $s'$  n'est pas dans la pile d'appel alors
9          $l'_s = \text{ParcoursSequences}(s')$ ;
10         $l_s = \text{union}(l_s, \text{concat}(\{e\}, l'_s))$ ;
11     $l_s = \text{min}(l_s)$ ;
12  retourner  $l_s$ ;
13 fin

14 Langage CalculSequences(Automate A)
15 début
16    $l = \emptyset$ ;
17   pour chaque état initial  $s_0$  de A
18      $l_{s_0} = \text{ParcoursSequences}(s_0)$ ;
19      $l = \text{union}(l, l_{s_0})$ ;
20  retourner  $\text{min}(l)$ ;
21 fin

```

Figure 11.2: Calcul des séquences par un parcours en profondeur de l'automate.

positivement dans le produit et 0 à celle qui apparaissent négativement. Un produit  $\sigma$  satisfait une formule  $f$  si  $f$  est satisfait par toute affectation compatible avec  $\sigma$  (c.-à-d. qui coïncide sur les littéraux du produit). Une produit qui satisfait  $f$  est un *implicant* de  $f$ . Un implicant  $\sigma$  est dit *premier* si il n'existe pas d'implicant strictement inclu dans  $\sigma$ . Un implicant premier est donc une combinaison minimale d'événements entrant la défaillance du système.

### 11.3.2 GÉNÉRATION DE FORMULES BOOLÉENNES

Le problème de la génération de formules booléennes peut se formuler ainsi :

- On considère un automate  $\mathcal{A} = \langle E, S, T, I, F \rangle$  où  $E$  est l'alphabet,  $S$  l'ensemble des états,  $T \subseteq S \times E \times S$  est la relation de transition,  $I \subseteq S$  est l'ensemble des états initiaux et  $F \subseteq S$  est un ensemble d'états d'acceptation. Dans la pratique  $F$  représente l'ensemble des états critiques du système. Nous notons  $L(\mathcal{A}) \subseteq E^*$  le langage accepté par l'automate  $\mathcal{A}$ . Si  $u$  est un mot alors nous notons  $\alpha(u)$  l'ensemble des lettres qu'il contient.
- Si  $E = \{e_1, \dots, e_n\}$  alors nous associons à tout événement  $e_i$  une variable booléenne  $x_i$ .
- Le problème de la génération de formules booléennes à partir de  $\mathcal{A}$  est celui de l'obtention d'une formule  $\phi_{\mathcal{A}}(x_1, \dots, x_n)$  telle que :  $\sigma = \{x_{i_1}, \dots, x_{i_k}\}$  est un implicant premier de  $\phi_{\mathcal{A}}$  si et seulement si il existe un mot  $u \in L(\mathcal{A})$  tel que  $\sigma = \alpha(u)$ .

### 11.3. VERS LES FORMULES BOOLÉENNES

Plutôt que de considérer  $L(\mathcal{A})$  nous pouvons nous restreindre à  $\min(L(\mathcal{A}))$ . Après le calcul de  $\min(L(\mathcal{A}))$  par l'algorithme proposé dans la section précédente nous pouvons utiliser la structure de diagramme pour générer la formule  $\phi_{\mathcal{A}}$  (cette formule n'est pas minimale). L'ensemble des séquences minimales étant codé par un nœud  $N$ , nous appliquons la transformation  $\Phi(N)$  définie inductivement par :

- $\Phi(\epsilon) = 1$
- $\Phi(\emptyset) = 0$
- $\Phi(\Delta(a, N_1, N_2)) = (x_a \wedge \Phi(N_1)) \vee \Phi(N_2)$

La formule  $\phi_{\mathcal{A}}$  considérée est alors  $\Phi(N)$ .





## CONCLUSIONS ET PERSPECTIVES

### 12.1 CONCLUSIONS

L'objectif de cette thèse était initialement la définition formelle de la sémantique de l'atelier FIABEX. Cette sémantique a été définie par celle du langage AltaRica; ce dernier sert de sémantique au successeur de FIABEX, l'atelier AltaRica.

Outre sa définition en terme de systèmes de transition interfacés, nous avons montré :

- D'une part la compositionnalité de ce modèle par rapport à une relation de bisimulation adaptée aux systèmes de transition interfacés ;
- D'autre part nous avons formalisé la « mise à plat » des descriptions hiérarchique exprimées dans ce langage et nous avons montré que cette transformation conservait la sémantique du langage. Cette transformation a pour vocation l'exploitation des modèles AltaRica par les outils d'analyse.

De manière inattendue, nous avons constaté que le modèle AltaRica permettait la description des automates temps-réel d'Alur et Dill.

Finalement nous avons étudié le codage de scénarios de défaillance par une structure de données proche des diagrammes de décision binaires. Nous avons défini sur cette structure les opérations ensemblistes usuelles. De plus nous avons défini sur cette même structure une opération de calcul des scénarios minimaux (pour les sous-mots). Cette opération est « l'équivalent » du calcul des *coupes minimales* pour les formules booléennes.

### 12.2 PERSPECTIVES

Les perspectives technologiques et scientifiques offertes par cette thèse sont nombreuses. Du fait de son appartenance à un projet industriel il est relativement naturel de les présenter par ordre de priorité. En effet, si le projet AltaRica est un projet ambitieux, sa réussite est conditionnée par le soutien et la validation des partenaires industriels impliqués. Ainsi, pour atteindre les objectifs du projet, nous nous devons de résoudre en premier lieu les problèmes tant scientifiques que technologiques susceptibles de répondre à leurs besoins.

## 12.2.1 LES PERSPECTIVES INDUSTRIELLES

### 12.2.1.1 Génération d'arbres de défaillance

Un besoin essentiel des partenaires du projet AltaRica est la compilation du langage vers les arbres de défaillance. À l'heure actuelle cette compilation est totalement inefficace car elle nécessite l'exploration du graphe des états du modèle. Deux possibilités sont à étudier :

**Réduction du graphe des états :** Bien que cette voie semble assez naturelle dans le domaine de la vérification, nous pensons qu'elle mènera à une impasse pour les systèmes réels. En effet, la prise en compte de défaillance est un facteur aggravant pour la taille du graphe des états. De plus, étant donné la taille des modèles considéré (entre 100 et 1000 composants) nous craignons que même une représentation symbolique ne soit pas très performante. Enfin, les techniques utilisées dans le domaine de la vérification sont le plus souvent dédiées à la détection de bogue du système. De ce point de vue les techniques sont performantes car on se contente en général d'une réponse par « oui » ou par « non » du programme d'analyse ; sachant cela, il est possible d'apporter un nombre conséquent d'optimisation (p.ex. la vérification « à la volée »). De telles approches auront très certainement une portée limitée pour l'obtention des scénarios de défaillance d'un modèle.

**Analyse structurelle :** Cette voix consisterait en l'analyse structurelle des descriptions. Ce type d'approche a été mise en œuvre dans FIABEX[153]. Extraire des informations de la topologie du système permettrait d'une part de construire l'arbre de défaillance de manière incrémentale (en essayant de déterminer des relations de causalité entre les défaillances) et d'autre part, d'ignorer les parties du système qui n'ont pas d'influence sur la réalisation de l'événement redouté.

Quelque soit la voix explorée nous sommes de plus en plus convaincu de la spécificité de ce problème de sûreté de fonctionnement.

### 12.2.1.2 Prise en compte des aspects stochastiques

La simulation de Monte-Carlo est un outil important en sûreté de fonctionnement (cf. chapitre 3). Cette méthode nécessite que l'on soit capable de transformer un modèle comportemental de haut niveau en un processus stochastique. Malheureusement, les aspects stochastiques n'ont pas été pris en compte dans le langage AltaRica.

Les événements des composants AltaRica sont locaux. Dans une version stochastique d'AltaRica, les lois de probabilités associées aux événements devrait donc être locales. Le problème à étudier est la sémantique d'un vecteur de synchronisation impliquant des événements probabilisés. La sémantique des vecteurs étant définie, il deviendrait envisageable d'associer un processus stochastique à une description AltaRica.

## 12.2.2 PERSPECTIVES ACADÉMIQUES

### 12.2.2.1 Le successeur de MEC

Le modèle AltaRica étant le successeur du modèle Arnold-Nivat, il est tout à fait naturel de penser au successeur du vérificateur de modèles MEC. Ce point semble être un objectif plus technologique que scientifique. En effet, pour traiter les modèles AltaRica

## 12.2. PERSPECTIVES

dans l'outil MEC il est « juste » nécessaire d'implanter dans le *model-checker* un module d'analyse syntaxique des fichiers AltaRica et le module de construction du graphe des états. Malheureusement une telle approche ne permettrait pas une pleine exploitation du langage. Du point de vue de la logique employée, le successeur de MEC devrait prendre en compte les notions de variables et de hiérarchie du modèle AltaRica.

### 12.2.2.2 Aspects temps-réels

Nous avons montré que, moyennant la prise en compte de variables réelles, il était possible de construire en AltaRica des automates temps-réels. Ce résultat restera toutefois anecdotique tant que les outils de calcul ne mettront pas en œuvre des algorithmes analogues à ceux utilisés pour traiter les automates temps-réel (cf. UPPAAL).





## BIBLIOGRAPHIE

- [1] T. Hutinet, S. Lajeunesse, and L. Martin. Atelier FIABEX, vers une intégration des études SdF en phase de conception. In *Actes du Congrès  $\lambda\mu$  94, ESREL'94*, pages 694–700, La Baule, 1994.
- [2] Alain Leroy and Jean-Pierre Signoret. *Le Risque Technologique*. Presses Universitaires de France, 1992.
- [3] Jean-François Monin. *Comprendre les méthodes formelles – Panorama et outils logiques*. Collection Technique et Scientifique des Télécommunications – CNET/ENST. Masson, 1996.
- [4] D. Kececioglu. *Reliability Engineering Handbook*, volume 1. Prentice-Hall, Inc., 1991.
- [5] A. Villemeur. *Sûreté de fonctionnement des systèmes industriels*. Collection de la Direction des Études et Recherches d'Électricité de France. Eyrolles, 1988.
- [6] Institut de Sûreté de Fonctionnement. AMDEC. Les Condensés pédagogiques de l'ISdF (numéro 4), 41, rue des Trois Fontanot, 92024 Nanterre Cedex.
- [7] H. Desille, F. Meunier, M. O'Connor, A. Rauzy, and P. Thomas. SHERLOC: Outil d'allocation d'indisponibilité pour les arbres de défaillance. In *Actes du Congrès  $\lambda\mu$ '11*, Arcachon, 1998.
- [8] Y. Dutuit and A. Rauzy. Handling boolean models with loops. In C. Guedes Soares, editor, *Proceedings of European Safety and Reliability Association Conference, ESREL'97*, volume 3, pages 2063–2070. Pergamon, 1997.
- [9] N. Limnios. *Arbres de défaillance*. Traité des Nouvelles Technologies - Série Diagnostic et Maintenance. Hermes, 1991.
- [10] A. Pagès and M. Gondran. *Fiabilité des systèmes*. Collection de la Direction des Études et Recherches d'Électricité de France. Eyrolles, 1980.
- [11] Yves Dutuit and Antoine Rauzy. Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within ARALIA. *Reliability Engineering and System Safety*, 58: 127–144, 1997.
- [12] Antoine Rauzy. New algorithms for fault trees analysis. *Reliability Engineering and System Safety*, 05(59): 203–211, 1993.

- [13] U. Berg. RISK SPECTRUM, *Theory Manual*. RELCON Teknik AB, April 1994.
- [14] IXI. *Aralia Workshop - Manuel Théorique*. Groupe Aralia - Hévéa, 1999.
- [15] Groupe Aralia. Computation of prime implicants of a fault tree within Aralia. In *Proceedings of the European Safety and Reliability Association Conference, ES-REL'95*, pages 190–202, Bournemouth – England, June 1995. European Safety and Reliability Association.
- [16] Antoine Rauzy. A brief introduction to binary decision diagrams. *European Journal of Automation, RAIRO-APII-JESA*, 30(8): 1033–1050, 1996.
- [17] R. Bryant. Graph based algorithms for boolean fonction manipulation. *IEEE Transactions on Computers*, 35(8): 677–691, August 1986.
- [18] R. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(8): 293–318, September 1992.
- [19] Macha Nikolskaïa and Antoine Rauzy. Heuristics for BDD handling of sum-of-products formulae. In *Proceedings of ESREL'99*, 1999.
- [20] Macha Nikolskaïa. *Binary Decision Diagrams and Applications to Reliability Analysis*. PhD thesis, Université Bordeaux I, Janvier 2000.
- [21] Philippe Schnoebelen. *Vérification de logiciels – Techniques et outils du model-checking*. Vuibert, 1999. Ouvrage collectif sous la direction de P. Schnoebelen.
- [22] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Masson, Paris, 1992.
- [23] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5): 1045–1079, 1955.
- [24] R. Alur and D.L Dill. Automata for modelling real-time systems. In *Automata, Languages and Programming – ICALP'90*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer Verlag, 1990.
- [25] Kim G. Larsen, Paul Petterson, and Wang Yi. UPPAAL in a nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1), 1997.
- [26] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1: 110–122, 1997.
- [27] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11<sup>th</sup> Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pages 278–292, 1996.
- [28] R. Milner. Semantics of concurrent processes. In van Leeuwen [154], pages 1201–1242.
- [29] E. A. Emerson. Temporal and modal logic. In van Leeuwen [154], pages 995–1072.
- [30] W. Thomas. Automata on infinite words. In van Leeuwen [154], pages 133–191.

- [31] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [32] V. R. Pratt. A Decidable Mu-Calculus. In *22nd IEEE Annual Symposium on Foundations of Compute Science*, pages 421–427, October 28-30 81.
- [33] A. Arnold, D. Bégay, and P. Crubillé. *Construction and analysis of transition systems with MEC*. World Scientific Publishers, 1994.
- [34] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.
- [35] P. Crubillé. *Réalisation de l'outil MEC — Spécification fonctionnelle et architecture*. Thèse, LaBRI - Université Bordeaux I, 1989.
- [36] A. Arnold and P. Crubillé. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letter*, 29(2): 57–66, 30 sept. 1988.
- [37] Denis Zampuniéris. *The Sharing Tree Data Structure – Theory and Applications in Formal Verification*. Thèse, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgique, Mai 1997.
- [38] Olivier Coudert and Jean-Christophe Madre. Symbolic computation of the valid states of a sequential machine: Algorithms and discussion. In *Proceedings of the International Workshop on Formal Methods in VLSI Design*, Miami, USA, Jan. 1991.
- [39] Claude Jard and Thierry Jéron. Bounded-memory algorithms for verification on-the-fly. In K. G. Larsen and A. Skou, editors, *Proceedings of the 3<sup>rd</sup> International Workshop, CAV'91*, volume 575 of *Lecture Notes in Computer Science*, pages 192–202, Aalborg, Denmark, July 1991. Springer-Verlag.
- [40] G.J. Holzmann. Algorithms for automated protocol verification. *AT&T Technical Journal*, 69(2), February 1990. Special Issue on Protocol Testing, Specification, and Verification.
- [41] Patrice Godefroid. *Partial order methods for the verification of concurrent systems: an approach to the state explosion problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.
- [42] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdd. In *Proceedings of TACAS'99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.
- [43] A. Arnold, D. Bégay, and J.-P. Radoux. The embedded software on an electricity meter: An experience in using formal methods in an industrial project. *Science of Computer Programming*, 28: 93–110, 1997.
- [44] François Bustany, Clément Roques, and Denis Sabatier. Les apports de la preuve en B. In *Actes du 10<sup>ième</sup> colloque  $\lambda\mu$* , Saint-Malo, Oct. 1996.
- [45] Jean-Pierre Elloy. Le temps réel. *Technique et Science Informatiques*, 7(5): 493–500, 1988. Rapport établi par le Groupe de Réflexion Temps Réel du CNRS, sous la Direction de Jean-Pierre Elloy.

- [46] Daniel Dzierzgowski. Quatre exemples de langages ou environnement pour le développement de programmes où le temps intervient. *T.S.I. - Technique et Science Informatiques*, 9(4): 289–312, 1990.
- [47] C. A. Petri. Kommunikation mit automaten. Technical report, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, Schrift Nr 2, 1962.
- [48] Gilles Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Proceedings of the IFIP Congress 74*, pages 471–475. North-Holland Publishing Company, 1974.
- [49] A. Arnold and M. Nivat. Comportements de processus. In *Colloque AFCET “Les Mathématiques de l’informatique”*, 1982.
- [50] E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton University Press, 1956.
- [51] A. M. Turing. On computable numbers with an application to the entscheidungsproblem. In *Proc. of London Mathematical Society*, volume 2, pages 230–265, 1936.
- [52] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10): 576–580, 1969.
- [53] J. M. Spivey. *La notation Z. Methodologies du Logiciel*. Masson / Prentice Hall, 1994.
- [54] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.
- [55] C. B. Jones and R. C. Shaw. *Case Studies in Systematic Software Development*. Prentice Hall, 1990.
- [56] Jean-Raymond Abrial. *Deriving Programs from Meaning*. Prentice Hall, 1994.
- [57] Jean-Raymond Abrial. *The B Book - Assigning Programs To Meanings*. Cambridge University Press, 1996.
- [58] Jonathan Bowen. *Formal Specification & Documentation Using Z – A Case Study Approach*. International Thomson Computer Press, 1996.
- [59] NASA. Formal Methods Specification and Verification Guidebook For Software and Computer Systems - Volume I : Planning and Technology Insertion. Technical Report NASA-GB-002-95, National Aeronautics and Space Administration, Washington, DC 20546, Jul. 1995.
- [60] NASA. Formal Methods Specification and Verification Guidebook For Software and Computer Systems - Volume II : A Practitioner’s Companion. Technical Report NASA-GB-001-97, National Aeronautics and Space Administration, Washington, DC 20546, May 1997.
- [61] Jean-Pierre Elloy and Olivier Roux. ELECTRE: A language for control structuring in real time. *The Computer Journal*, 28(5), 1985.

- [62] Marc Huou, Jean Perraud, and Olivier Roux. Operational semantics of a kernel of the language ELECTRE. *Theoretical Computer Science*, 97(1): 83–104, Apr. 1992.
- [63] Franck Cassez, Denis Creusot, Jean-Pierre Elloy, and Olivier Roux. Le langage réactif aynchrone ELECTRE. *Technique et Science Informatiques*, 11(5): 35–66, 1992.
- [64] Franck Cassez and Olivier Roux. Compilation of ELECTRE reactive language into finite transition systems. *Theoretical Computer Science*, 146(1–2): 109–143, 1995.
- [65] P. Argón. *Étude sur l'application de méthodes formelles à la compilation et à la validation de programmes ELECTRE*. PhD thesis, École Centrale de Nantes, 1998.
- [66] F. Cassez, A. Finkel, O. Roux, and G. Sutre. Effective recognizability and model checking of reactive FIFO automata. Technical Report LSV-98-10, École Normale Supérieure de Cachan, 1998.
- [67] V. Rusu. *Vérification Temporelle de programmes ELECTRE*. PhD thesis, École Centrale de Nantes, 1996.
- [68] P. Boisieu. *Vérification et exécution d'applications temps-réel industrielles avec ELECTRE*. PhD thesis, École Centrale de Nantes, 1999.
- [69] David Harel. Statecharts: a visual approach to complex systems. Technical Report CS84-05, Department of Applied Mathematics, The Weizmann Institute of Science, 1984.
- [70] D. Harel. StateCharts: a visual approach to complex systems. *Science of Computer Programming*, 8(3): 231–275, 1987.
- [71] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
- [72] David Harel. Some thoughts on Statecharts, 13 years later. In O. Grumberg, editor, *Proceedings of the 9<sup>th</sup> International Conference on Computer Aided Verification, CAV'97*, number 1254 in Lecture Notes in Computer Science, pages 226–227, Haifa, Israel, June 1997. Springer.
- [73] M. von der Beek. A comparison of statechart variants. In Langmaack, de Roever, and Vytupil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148, New York, 1998. Springer-Verlag.
- [74] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceeding of the First IEEE Symposium on Logic in Computer Science*, pages 54–64, New-York, 1986. IEEE Press.
- [75] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM Trans. Soft. Eng. Method.*, 5(4): 293–333, Oct. 1996.

- [76] F. Maraninchi. ARGONAUTE: graphical description, semantics and verification of reactive systems by using a process algebra. In *Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, June 1989. LNCS 407, Springer Verlag.
- [77] B. Sanscartier. Statecharts alternants récursifs - l'application des concepts de récursion et de composition dans la syntaxe algébrique et la sémantique opérationnelle des statecharts alternants. Master's thesis, Université du Québec - INRS Télécommunications, 1994.
- [78] J.R. Beauvais, R. Houdebine, P. Le Guernic, E. Rutten, and T. Gautier. A translation of statecharts into signal. In *Proceedings of the International Conference on Application of Concurrency to System Design (CSD'98)*, pages 52–62, Aizu-Wakamatsu, Japan, March 23-26 1998. IEEE Publ.
- [79] David Harel and Michan Politi. *Modeling Reactive Systems with Statecharts*. Mac Graw Hill, USA, 1998.
- [80] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993.
- [81] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Proceedings of the International Federation for Information Processing, IFIP'89*, pages 11–17, North Holland, 1989. Elsevier Science Publishers B.V.
- [82] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Another Look at Real-Time Programming, Proceedings of the IEEE*, volume 79(9), pages 1270–1282, 1991.
- [83] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirliagand, and M. Tofte, editors, *Proof, Language and Interaction: Essays in honour of Robin Milner*. MIT Press, 1998.
- [84] F. Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.
- [85] M. Jourdan and F. Maraninchi. A modular state/transition approach for programming real-time systems. In *ACM SIGPLAN Workshop on Language, compiler and tool support for real-time systems*, Orlando, Florida, June 1994.
- [86] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.
- [87] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *Proceedings of CONCUR'92*, volume 630 of LNCS, August 1992.
- [88] G. Berry and L. Cosserrat. The ESTEREL synchronous language and its mathematical semantics. In *Seminar on Concurrency*, Lecture Notes in Computer Science. Springer Verlag, 1985.

- [89] G. Berry. Preemption in concurrent systems. In *Proceedings of FSTTCS'93*, volume 761 of *Lecture Note in Computer Science*, pages 72–93. Springer-Verlag, 1993.
- [90] Rysjard Janicki. A formal semantics for concurrent systems with a priority relation. *Acta Informatica*, 24(2): 33–55, 1987.
- [91] J. L. Bergerand. LUSTRE: *un langage déclaratif pour le temps réel*. PhD thesis, Institut National Polytechnique de Grenoble, 1986.
- [92] P. Caspi, N. Halbwachs, D. Pilaud, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14<sup>th</sup> annual ACM Symposium on Principles of Programming Languages, POPL'87*, pages 178–188, Munich, 1987.
- [93] J. A. Plaice. *Sémantique et compilation de LUSTRE, un langage déclaratif synchrone*. PhD thesis, Institut National Polytechnique de Grenoble, 1988.
- [94] T. Gautier, M. Le Borgne, P. Le Guernic, and C. Le Maire. Programming real time applications with SIGNAL. In *Another Look at Real-Time Programming, Proceedings of the IEEE*, volume 79(9), pages 1321–1336, September 1991.
- [95] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48: 117–126, March 11 1986.
- [96] G. Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones; application à ESTEREL*. PhD thesis, Université d'Orsay, 1988.
- [97] T. Gautier and P. Le Guernic. Code generation in the sacres project. In *Towards System Safety, Proceedings of the Safety-critical Systems Symposium, SSS'99*, Huntingdon, UK, February 9-11 1999. Springer.
- [98] A. Ressouche and M. Robert. The Oc(Ssc)java processor overview. Technical report, INRIA Sophia-Antipolis - Projet MEIJE, 5 Oct. 1998.
- [99] F. Maraninchi and N. Halbwachs. Compiling ARGOS into boolean equations. In *Formal Techniques for Real-Time and Fault-Tolerance, FTRTFT*, LNCS, Uppsala (Sweden), September 1996. Springer Verlag.
- [100] T. Amagbagnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Programming Languages Design and Implementation*, pages 163–173. ACM, 1995.
- [101] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP'91)*, Passau, August 1991.
- [102] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In R. Kurshan, editor, *Proceedings of the International Workshop on Computer Aided Verification*, Rutgers, June 1990.
- [103] B. Buggiani, P. Caspi, and D. Pilaud. Programming distributed automatic control systems: a language and compiler solution. Technical report spectre I4, IMAG, Grenoble, Jul. 1988.

- [104] P. Aubry and P. Le Guernic. On the desynchronization of synchronous applications. In *Proceedings of the 11th International Conference on Systems Engineering, ICSE'96*. University of Nevada, July 1996.
- [105] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits, a hardware implementation of LUSTRE. In *REX Workshop on Real-Time: Theory in Practice*, volume 600 of LNCS, 1991.
- [106] G. Berry. Esterel on hardware. *Philosophical Transactions Royal Society of London A*, 339: 87–104, 1992.
- [107] G. Berry, T.R. Shiple, and H. Touati. Constructive analysis of cyclic circuits. In *Proceedings of the International Design and Testing Conference, IDTC'96*, Paris, France, March 1996.
- [108] G. Berry and H. Touati. Optimized controller synthesis using Esterel. In *Proceedings of the International Workshop on Logic Synthesis, IWLS'93*, Lake Tahoe, 1993.
- [109] E.M. Sentovitch, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. Rapport de Recherche 2943, INRIA, Sophia-Antipolis, Juillet 1996.
- [110] A. Pnueli and M. Shalev. What's in a step: On the semantics of statecharts. In *Proceedings of the Symposium on Theoretical Aspects of Computer Software*, LNCS, 1991.
- [111] A. Kountouris and P. Le Guernic. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. In *Proceedings of the IEEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, HP Labs, Bristol, UK, February 1996. IEEE.
- [112] Amar Bouali, Annie Ressouche, Valérie Roy, and Robert De Simone. The FC2TOOLS set. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8<sup>th</sup> International Conference on Computer Aided Verification, CAV'96*, volume 1102 of LNCS, New Brunswick, USA, July/August 1996. Springer-Verlag.
- [113] G. Gherardi. SAHARA: un environnement de mise au point graphique pour les programmes ESTEREL. PhD thesis, Université de Nice, 1992.
- [114] Christophe Ratel. Définition et réalisation d'un outil de vérification formelle de programmes LUSTRE: le système LESAR. PhD thesis, Université Joseph Fourier - Grenoble 1, 1992.
- [115] J. A. Plaice and J.-B. Saint. The LUSTRE-ESTEREL portable format. INRIA, Sophia Antipolis, 1987.
- [116] J.-P. Paris and J.-B. Saint. Les instructions du code intermédiaire. INRIA, Sophia Antipolis, 1990.
- [117] P. Asar. Towards a multi-formalism framework for architectural synthesis: the asar project. In *Proceedings of the Third International Workshop on Hardware/Software Codesign*, pages 25–32, Grenoble, September 22-24 1994. IEEE Computer Society Press.

- [118] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond. A multiparadigm language for reactive systems. In *IEEE International Conference on Computer Languages (ICCL)*, Toulouse, France, 1994.
- [119] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *Proceedings of European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer verlag.
- [120] J.R. Beauvais, R. Houdebine, Y.M. Tang, P. Le Guernic, E. Rutten, and T. Gautier. Une modélisation de statecharts et activitycharts en signal. In *Actes du 2ème Congrès sur la Modélisation des Systèmes Réactifs, MSR'99*, ENS, Cachan, mars 1999. Hermès.
- [121] G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating reactive processes. In *Proceedings of the 20th ACM Conference on Principles of Programming Languages, POPL'93*, Charleston, Virginia, 1993.
- [122] C. A. R. Hoare. Communicating sequential processes. *CACM*, 21(8), 1978.
- [123] Martin Richard and Olivier Roux. Conjunction of synchronous and asynchronous languages for reactive programming. In *Proc. of Euromicro Workshop on Real-Time Systems*, L'Aquila, Italy, 1996.
- [124] L. Arditi, A. Bouali, H. Boufaied, G. Clave, M. Hadj-Chaib, L. Leblanc, and R. de Simone. Using Esterel and formal methods to increase the confidence in the functional validation of a commercial DSP. In *Proceedings of ERCIM Workshop on Formal Methods for Industrial Systems*, Trento, Italy, 1999.
- [125] IEEE. *Standard VHDL Language Reference Manual*. IEEE Standard 1076–1993, 1993.
- [126] J.-P. Signoret. Moca-RP V9. Technical report, Elf-Aquitaine, 1995. rapport interne ELF Aquitaine Production – Direction Recherche et Développement Exploration Production – réf. EP/P/SE/MRT-ARF/JPS9634 – simulation de Monte-Carlo de réseaux de Petri stochastiques.
- [127] M. Bouissou. The FIGARO dependability evaluation workbench in use: Case studies for fault-tolerant compute systems. In *Proceedings of the 23th Annual Symposium on Fault-Tolerant Computing Systems, FTCS'93*, pages 680–685, 1993.
- [128] SOFRETEN. SOFIA Spécifications Générales. réf. SOFRETEN.24501.005, 1992.
- [129] S. Brleck and A. Rauzy. Synchronization of constrained transitions systems. In H. Hong, editor, *First International Symposium on Parallel Symbolic Computation (PASCOS'94)*, pages 54–62, Linz, Ostreich, 1994. World Scientific Publishing.
- [130] M.-M. Corsini and A. Rauzy. Toupie: the  $\mu$ -calculus over finite domains as a constraint language. *Journal of Automated Reasoning*, 1997.
- [131] A. Arnold. *Finite Transition Systems*. Prentice-Hall, 1994.

- [132] A. Rauzy. Utilisation du langage de contraintes Toupie pour analyser des spécifications “à la parnas”. In J.J. Lesage, editor, *Actes du congrès AFCET sur la Modélisation de systèmes réactifs*, pages 81–89, 1996.
- [133] S. Lajeunesse and A. Rauzy. Using the constraint programming system toupie for qualitative analysis of industrial system failures. In *ESREL’97*, 1997.
- [134] L. Fribourg and M. V. Peixoto. Automates concurrents à contraintes. *Technique et Science Informatique*, 13(6): 837–866, 1994.
- [135] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. Wiley series in parallel computing. John Wiley & Sons, 1995. ISBN 0-471-93059-8.
- [136] R. David and H. Alla. *Du Grafset aux réseaux de Petri*. Traité des Nouvelles Technologies, Série Automatique. Hermès, 1989.
- [137] P. Lauer, P. R. Torrigiani, and M. W. Shield. COSY: a system specification language based on path expressions. *Acta Informatica*, 12: 109–158, 1979.
- [138] R.H. Campbell. *Path Expressions: a technique for process synchronisation*. PhD thesis, University of Newcastle, 1976.
- [139] A. N. Habermann. Path expressions. Technical report, Carnegie-Mellon University, Pittsburgh, PA 15213, June 1975.
- [140] P. Eddy, E. Potter, and B. Page. *Destination désastre*. Grasset, 1977.
- [141] G. Point and A. Rauzy. AltaRica - Langage de modélisation par automates à contraintes. In *Actes du Congrès Modélisation des Systèmes Réactifs, MSR’99*, ENS, Cachan, 1999. Hermès.
- [142] G. Point and A. Rauzy. AltaRica - Constraint automata as a description language. *European Journal on Automation*, 33, 1999. Special issue on the *Modeling of Reactive Systems*.
- [143] A. Arnold, A. Griffault, G. Point, and A. Rauzy. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae*, 1999. A tribute to Peter Lauer (Soumission).
- [144] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M.B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *TSE*, 16(4): 403–414, 1990.
- [145] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Addison Wesley, 1997.
- [146] Rational Corporation. Documents on UML (the unified modeling language) version 1.0. <http://www.rational.com/uml>, 1999.
- [147] Rajeev Alur. Timed automata. In *Summer School on Verification of Digital and Hybrid Systems*. NATO-ASI, 1998.

- [148] Kim G. Larsen, Paul Petterson, and Wang Yi. Model-checking for real-time systems. In Horst Reichel, editor, *Proceedings of the 10<sup>th</sup> International Conference on Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88, Dresden, Germany, 22-25 August 1995.
- [149] François Laroussinie. *Logique temporelle avec passé pour la spécification et la vérification des systèmes réactifs*. Thèse, I.N.P. Grenoble, France, Novembre 1994.
- [150] A. Arnold, A. Griffault, G. Point, and A. Rauzy. Manuel méthodologique. Technical report, Groupe AltaRica, 1999. version 1.0.
- [151] Y. Dutuit, E. Châtelet, J.-P. Signoret, and P. Thomas. Dependability modelling and evaluation by using stochastic Petri nets: application to two test cases. *Reliability Engineering and Safety System*, pages 117–124, 1997.
- [152] K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE 0738, 1990.
- [153] S. Lajeunesse, T. Hutinet, and J.-P. Signoret. Automatical fault trees generation on dynamic systems. In *Proceedings of Probabilistic Safety Assessment and Management Conference- ESREL'96 - PSAM III*, volume 3. Springer, 1996.
- [154] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B. Elsevier, 1990.



# **Partie III**

## **Annexes**





## PREUVES DES CHAPITRES PRÉCÉDENTS

### A.1 PREUVES DU CHAPITRE 9

#### A.1.1 PROPOSITION 9.1

Soient  $\mathcal{A}_1 = \langle E, F, S_1, \pi_1, T_1 \rangle$  et  $\mathcal{A}_2 = \langle E, F, S_2, \pi_2, T_2 \rangle$  deux systèmes de transition interfacés et  $<$  un ordre sur  $E$ . Si  $h : \mathcal{A}_1 \rightarrow \mathcal{A}_2$  est un homomorphisme de bisimulation alors  $h$  est aussi un homomorphisme de bisimulation de  $\mathcal{A}_1 \upharpoonright <$  vers  $\mathcal{A}_2 \upharpoonright <$ .

**Preuve :** L'opérateur de restriction  $\upharpoonright <$  ne modifie pas les configurations du STI auquel il est appliqué. Il s'en suit que les propriétés (h1) et (h2) de l'homomorphisme  $h$  restent vraies après l'application de  $\upharpoonright <$ . Il nous faut montrer les propriétés (h3) et (h4):

- (h3) Si  $\langle s_1, f, e, s_2 \rangle \in T_1 \upharpoonright <$  alors, par construction,  $\langle s_1, f, e, s_2 \rangle \in T_1$  et donc  $\langle h(s_1), f, e, h(s_2) \rangle \in T_2$ . Supposons maintenant que  $\langle h(s_1), f, e, h(s_2) \rangle \notin T_2 \upharpoonright <$ . Il existe alors  $e' \in E$  tel que  $e < e'$  et  $s' \in S_2$  tels que  $\langle h(s_1), f, e', s' \rangle \in T_2$ . Mais, par définition de  $h$ , il existe  $s \in S_1$  tel que  $h(s) = s'$  et  $\langle s_1, f, e', s \rangle \in T_1$ . Il s'en suit que  $\langle s_1, f, e, s_2 \rangle \notin T_1 \upharpoonright <$ , d'où une contradiction.
- (h4) Si  $\langle h(s_1), f, e, s'_2 \rangle \in T_2 \upharpoonright <$  nous avons par construction  $\langle h(s_1), f, e, s'_2 \rangle \in T_2$ . Puisque  $h$  est un homomorphisme de bisimulation, il existe  $s'_1 \in S_1$  tel que  $h(s'_1) = s'_2$  et  $\langle s_1, f, e, s'_1 \rangle \in T_1$ . Si  $\langle s_1, f, e, s'_1 \rangle \notin T_1 \upharpoonright <$  il existe  $e' \in E$  et  $s''_1 \in S_1$  tels que  $e < e'$  et  $\langle s_1, f, e', s''_1 \rangle \in T_1$ . Nous obtenons alors que  $\langle h(s_1), f, e', h(s''_1) \rangle \in T_2$ , ce qui contredit l'hypothèse  $\langle h(s_1), f, e, s'_2 \rangle \in T_2 \upharpoonright <$ .

◇

#### A.1.2 LEMME 9.2

Si  $\mathcal{C} = \langle V_S, V_F, V_L, E, A, M, \emptyset \rangle$  et  $<$  est un ordre sur  $E$  alors  $\llbracket \mathcal{C} \rrbracket \upharpoonright < = \llbracket \mathcal{C} \upharpoonright < \rrbracket$ .

**Preuve :** Nous notons  $T_\emptyset$  l'ensemble des transitions de  $\llbracket \mathcal{C} \rrbracket \upharpoonright <$  et  $T_<$  celui de  $\llbracket \mathcal{C} \upharpoonright < \rrbracket$ . Puisque, les constructions de ces deux STI ne diffèrent que sur leurs ensembles de transitions il suffit de montrer l'égalité  $T_\emptyset = T_<$ .

Si  $\langle s, f, e, s' \rangle \in T_\emptyset$ , il existe  $t = (g, e, a) \in M$  telle que  $(s, f, e, s') \in \llbracket t \rrbracket$ . De plus il existe  $(G, e, a) \in M \upharpoonright <$  avec  $G = g \wedge \bigwedge_{e < e'} \neg V_E(e')$ . Puisque  $(s, f, e, s') \in \llbracket t \rrbracket$ , il existe  $l \in \mathcal{D}^{V_L}$  tel que  $(s, f, l) \in \llbracket A \wedge g \rrbracket$  et si  $e' \in E$  est tel que  $e < e'$  alors pour tout  $s''$  tel que  $\pi(s'') \neq \emptyset$ ,  $(s, f, e', s'') \notin \llbracket M \rrbracket$  (par construction de  $T_\emptyset$ ) et donc nous avons  $(s, f, l) \notin \llbracket V_E(e') \rrbracket$  (fait 9.3). Il s'en suit que  $(s, f, l) \in \llbracket G \rrbracket$  et donc  $\langle s, f, e, s' \rangle \in T_<$ .

Supposons maintenant que  $\langle s, f, e, s' \rangle \in T_{<}$ . Il existe donc  $(G, e, a) \in M \upharpoonright_{<}$  et  $t = (g, e, a) \in M$  telles que  $G = g \wedge \bigwedge_{e < e'} \neg V_E(e')$  et  $\langle s, f, e, s' \rangle \in \llbracket (G, e, a) \rrbracket$ . Il existe donc  $l \in \mathcal{D}^{V_L}$  tel que  $(s, f, l) \in \llbracket A \wedge g \rrbracket$  et pour tout  $e' \in E$  si  $e < e'$  alors  $(s, f, l) \notin \llbracket V_E(e') \rrbracket$ . Donc pour tout  $s'', \langle s, f, e', s'' \rangle \notin \llbracket M \rrbracket$  (fait 9.3) ce qui nous permet de conclure que  $\langle s, f, e, s' \rangle \in T_\emptyset$ .  $\diamond$

### A.1.3 THÉORÈME 9.2

si  $\mathcal{N} = \langle V_F, E, <, \mathcal{N}_0, \dots, \mathcal{N}_n, V \rangle$  et  $\mathcal{N}' = \langle V_F, E, <, \mathcal{N}'_0, \dots, \mathcal{N}'_n, V \rangle$  sont des nœuds AltaRica tels que pour tout  $i = 0 \dots n$ , il existe un homomorphisme de bisimulation  $h_i$  de  $\llbracket \mathcal{N}_i \rrbracket$  vers  $\llbracket \mathcal{N}'_i \rrbracket$  alors il existe un homomorphisme de bisimulation  $h$  de  $\llbracket \mathcal{N} \rrbracket$  vers  $\llbracket \mathcal{N}' \rrbracket$ .

**Preuve :** Nous posons  $\llbracket \mathcal{N} \rrbracket = \langle E, F, S, \pi, T \rangle$  et  $\llbracket \mathcal{N}' \rrbracket = \langle E, F, S', \pi', T' \rangle$ . Nous posons l'application  $h : S \rightarrow S'$  définie par :  $h(\langle s_0, \dots, s_n \rangle) = \langle h_0(s_0), \dots, h_n(s_n) \rangle$ .

Pour tout  $i = 0 \dots n$ , nous posons  $\llbracket \mathcal{N}_i \rrbracket = \langle E_i, F_i, S_i, \pi_i, T_i \rangle$  et  $\llbracket \mathcal{N}'_i \rrbracket = \langle E_i, F_i, S'_i, \pi'_i, T'_i \rangle$ . De plus, nous notons  $\phi_i$  la projection de  $F_0 = F \times F_1 \times \dots \times F_n$  sur  $F_i$ .

- (h1) Soit  $\vec{s}' = \langle s'_0, \dots, s'_n \rangle \in S'$ . Par construction nous avons  $\pi'(\vec{s}') \neq \emptyset$ . Il existe donc  $(f, f_1, \dots, f_n) \in \pi'_0(s'_0)$  tel que pour tout  $i = 1 \dots n$ ,  $f_i \in \pi'_i(s'_i)$ . Mais, par définition des  $h_i$ , pour tout  $i = 0 \dots n$ , il existe  $s_i \in S_i$  tel que  $s'_i = h_i(s_i)$  et  $\pi'_i(s'_i) = \pi_i(s_i)$ . On conclut alors que  $\langle s_0, \dots, s_n \rangle \in h^{-1}(\vec{s}')$  et  $h$  surjective.
- (h2) Soit  $\vec{s} = \langle s_0, \dots, s_n \rangle \in S$  et  $f \in \pi(\vec{s})$ . Il existe  $(f, f_1, \dots, f_n) \in \pi_0(s_0)$  tel que pour  $i = 1 \dots n$ ,  $f_i \in \pi_i(s_i)$ . Par hypothèse, pour tout  $i = 0 \dots n$ ,  $\pi_i(s_i) = \pi'_i(h_i(s_i))$  donc  $(f, f_1, \dots, f_n) \in \pi'_0(h_0(s_0))$  et pour tout  $i = 1 \dots n$ ,  $f_i \in \pi'_i(h_i(s_i))$ . On en conclut que  $f \in \pi'(h(\vec{s}))$ .
- (h3-4) Soient  $T_{\mathcal{N}}$  et  $T_{\mathcal{N}'}$  les ensembles de transitions intermédiaires utilisés pour la sémantique de  $\mathcal{N}$  et  $\mathcal{N}'$ . Il est clair que:

$$\begin{aligned} & - \forall \langle s, f, e, s' \rangle \in T_{\mathcal{N}}, \langle h(s), f, e, h(s') \rangle \in T_{\mathcal{N}'} \\ & - \forall s_1 \in S_1, s'_2 \in S_2, \langle h(s_1), f, e, s'_2 \rangle \in T_{\mathcal{N}'} \Rightarrow \exists s'_1 \in S_1, h(s'_1) = s'_2 \wedge \langle s_1, f, e, s'_1 \rangle \in T_{\mathcal{N}} \end{aligned}$$

La proposition 9.1 nous donne que ces deux propriétés de  $h$  sont conservés par les priorités et donc qu'elles restent vraies pour  $T_{\mathcal{N}} \upharpoonright_{<?}$  et  $T_{\mathcal{N}'} \upharpoonright_{<?}$  d'une part et pour  $(T_{\mathcal{N}} \upharpoonright_{<?}) \upharpoonright_{<0}$  et  $(T_{\mathcal{N}'} \upharpoonright_{<?}) \upharpoonright_{<0}$  d'autre part.

Enfin, si  $(s, f, e_0, s') \in T$  il existe  $(s, f, \langle e_0, \dots, e_n, v \rangle, s') \in (T_{\mathcal{N}} \upharpoonright_{<?}) \upharpoonright_{<0}$  et  $(h(s), f, \langle e_0, \dots, e_n, v \rangle, h(s')) \in (T_{\mathcal{N}'} \upharpoonright_{<?}) \upharpoonright_{<0}$ ; il s'en suit que  $(h(s), f, e_0, h(s')) \in T'$ .

Et pour conclure, si  $(h(s_1), f, e_0, s'_2) \in T'$  il existe  $(h(s_1), f, \langle e_0, \dots, e_n, v \rangle, s'_2) \in (T_{\mathcal{N}'} \upharpoonright_{<?}) \upharpoonright_{<0}$  et il existe  $s'_1 \in S, h(s'_1) = s'_2$  et  $(s_1, f, \langle e_0, \dots, e_n, v \rangle, s'_1) \in (T_{\mathcal{N}} \upharpoonright_{<?}) \upharpoonright_{<0}$  et donc  $(s_1, f, e_0, s'_1) \in T$ .

$\diamond$

### A.1.4 LEMME 9.3

Soient  $\mathcal{N} = \langle V_F, E, <, \mathcal{N}_0, \dots, \mathcal{N}_n, V \rangle$  un noeud AltaRica et  $\mathcal{C}_{\mathcal{N}} = \langle V_S, V_F, V_L, E, A, M, \emptyset \rangle$  sa réécriture en un composant. Si pour tout  $i = 1 \dots n$ ,  $\mathcal{N}_i$  est un composant alors  $\llbracket \mathcal{N} \rrbracket = \llbracket \mathcal{C}_{\mathcal{N}} \rrbracket$ .

**Preuve :**

Posons  $\llbracket \mathcal{N} \rrbracket = \langle E, F, S_o, \pi_o, T_o \rangle$  et  $\llbracket \mathcal{C}_{\mathcal{N}} \rrbracket = \langle E, F, S_s, \pi_s, T_s \rangle$  et pour tout  $i = 0 \dots n$ ,  $\llbracket \mathcal{N}_i \rrbracket = \llbracket \mathcal{N}'_i \rrbracket = \langle E_i, F_i, S_i, \pi_i, T_i \rangle$ . Il est clair que  $S_o = S_s$  et  $\pi_o = \pi_s$ . Montrons que  $T_o = T_s$ .

Si  $T_{\mathcal{N}}$  est l'ensemble des transitions intermédiaire utilisé dans la construction de  $T_o$  et si  $M'$  est l'ensemble intermédiaire des macro-transitions pour la définition de  $\mathcal{C}_{\mathcal{N}}$ , il nous suffit de montrer que  $T_{\mathcal{N}} = \llbracket M' \rrbracket$ . Cette remarque se justifie par le fait que la propriété recherchée est l'isomorphisme des deux STI ; cette propriété est conservée par l'application de  $|\langle \_? \rangle$  et de  $|\langle \_0 \rangle$  sur ce cas particulier d'homomorphisme (proposition 9.1).

$\langle (s_0, \dots, s_n), f, (u, v), (s'_0, \dots, s'_n) \rangle \in T_{\mathcal{N}}$  si et seulement si il existe  $f_0 = (f, f_1, \dots, f_n) \in \pi_0(s_0)$  et  $u = (e_0, \dots, e_n)$  tels que pour tout  $i = 0 \dots n$ ,  $(s_i, f_i, e_i, s'_i) \in T_i$ . Donc pour tout  $i$ , il existe  $t_i = (g_i, e_i, a_i) \in M'_i$  telle que  $(s_i, f_i, e_i, s'_i) \in \llbracket t_i \rrbracket$  c'est-à-dire il existe  $l_i \in \mathcal{D}^{V_{L_i}}$  tel que  $(s_i, f_i, l_i) \in \llbracket A'_i \wedge g_i \rrbracket$  et  $s'_i = a_i[s_i, f_i, l_i]$ . Or il existe  $t = (g, e, a) \in M'$  telle que  $g = g_0 \wedge \dots \wedge g_n$  et  $\forall x \in V_S, x \in V_{S'_i} \Rightarrow a(x) = a_i(x)$ . On obtient alors que  $(s_0, \dots, s_n, f, f_1, \dots, f_n, l_0, \dots, l_n) \in \llbracket A \wedge g \rrbracket$  et que  $(s_0, \dots, s_n) = a[s_0, \dots, s_n, f, f_1, \dots, f_n, l_0, \dots, l_n]$  et donc  $\langle (s_0, \dots, s_n), f, (u, v), (s'_0, \dots, s'_n) \rangle \in \llbracket t \rrbracket \subseteq \llbracket M' \rrbracket$ .

Réciproquement,  $\langle (s_0, \dots, s_n), f, (u, v), (s'_0, \dots, s'_n) \rangle \in \llbracket M' \rrbracket$  si et seulement si il existe  $t = (g, e, a) \in M'$  et  $(l_0, f_1, l_1, \dots, f_n, l_n) \in \mathcal{D}^{V_L}$  tels que  $(s_0, \dots, s_n, f, l_0, f_1, l_1, \dots, f_n, l_n) \in \llbracket A \wedge g \rrbracket$  et  $(s'_0, \dots, s'_n) = a[s_0, \dots, s_n, f, l_0, f_1, l_1, \dots, f_n, l_n]$ . Or, en supposant  $u = (e_0, \dots, e_n)$ , il existe pour  $i = 0 \dots n$ ,  $t_i = (g_i, e_i, a_i) \in M'_i$  telles que  $g = g_0 \wedge \dots \wedge g_n$  et  $\forall x \in V_S, x \in V_{S'_i} \Rightarrow a(x) = a_i(x)$ . En posant  $f_0 = (f, f_1, \dots, f_n)$ , on a clairement pour tout  $i = 0 \dots n$ ,  $(s_i, f_i, e_i, s'_i) \in T_i$  ; ceci qui montre la réciproque.  $\diamond$

### A.1.5 THÉORÈME 9.3

Si  $\mathcal{N}$  un noeud AltaRica et  $\mathcal{C}_{\mathcal{N}}$  sa réécriture en un composant alors  $\llbracket \mathcal{N} \rrbracket = \llbracket \mathcal{C}_{\mathcal{N}} \rrbracket$ .

**Preuve :** En annexe, section A.1.4, page A.1.4. Supposons que  $\mathcal{N} = \langle V_F, E, <, \mathcal{N}_0, \dots, \mathcal{N}_n, V \rangle$  et que la propriété soit vérifiée pour les  $\mathcal{N}_i$  c'est-à-dire  $\llbracket \mathcal{N}_i \rrbracket = \llbracket \mathcal{C}_{\mathcal{N}_i} \rrbracket$ . Si l'on pose  $\mathcal{N}' = \langle V_F, E, <, \mathcal{C}_{\mathcal{N}_0}, \dots, \mathcal{C}_{\mathcal{N}_n}, V \rangle$ , nous avons  $\mathcal{C}_{\mathcal{N}} = \mathcal{C}_{\mathcal{N}'}$ . L'égalité  $\llbracket \mathcal{N}_i \rrbracket = \llbracket \mathcal{C}_{\mathcal{N}_i} \rrbracket$  étant un cas particulier d'homomorphisme, le théorème 9.2 nous donne qu'il existe un homomorphisme  $h$  de  $\llbracket \mathcal{N} \rrbracket$  vers  $\llbracket \mathcal{N}' \rrbracket$ . Par construction,  $h$  est un isomorphisme entre  $\llbracket \mathcal{N} \rrbracket$  vers  $\llbracket \mathcal{N}' \rrbracket$ . Il suffit alors d'appliquer le lemme 9.3 sur  $\mathcal{N}'$  pour obtenir le résultat :  $\llbracket \mathcal{C}_{\mathcal{N}} \rrbracket = \llbracket \mathcal{C}_{\mathcal{N}'} \rrbracket = \llbracket \mathcal{N}' \rrbracket = \llbracket \mathcal{N} \rrbracket$ .  $\diamond$

## A.2 PREUVES DU CHAPITRE 11

### A.2.1 LEMME 11.1

Nous montrons que :  $\forall L \subseteq E^* \setminus \{\epsilon\}, \forall a \in E, \forall w \in E^*, a.w \in \min(L) \iff w \in \text{ext}(\min(a^{-1}L), L_{\bar{a}})$ .

$\Rightarrow$ : Si  $a \in \Sigma$  et  $w \in \Sigma^*$  sont tels que  $a.w \in \min(L)$  alors nous montrons que  $w \in \text{ext}(\min(a^{-1}L), L_{\bar{a}})$ . Puisque  $\min(L) \subseteq L$  nous avons  $w \in a^{-1}L$ . De plus, si  $v \in a^{-1}L$  est tel que  $v \sqsubset w$  alors nous aboutissons à une contradiction. Cette hypothèse implique que  $a.v \sqsubset a.w$  et contredit le fait que  $a.w \in \min(L)$ . On en conclut que  $w \in \min(a^{-1}L)$ . Enfin, si  $v \in L_{\bar{a}}$  est tel que  $v \sqsubset w$  alors nous aboutissons à la même contradiction car cette fois-ci l'hypothèse implique que  $v \sqsubset a.w$  pour  $v \in L$ . On en conclut que pour tout  $v \in L_{\bar{a}}, w \not\sqsubset v$ .

$\Leftarrow$ : Pour montrer la réciproque nous considérons  $w \in \text{ext}(\min(a^{-1}L), L_{\bar{a}})$  et  $v \in L$  tel que  $v \sqsubseteq a.w$ .

1. Si  $v = a.x$  ( $x \in E^*$ ) alors  $a.x \sqsubseteq a.w$  et donc  $x \sqsubseteq w$  avec  $x \in a^{-1}L$ . Nous avons une contraction avec le fait que  $w \in \min(a^{-1}L)$ .
2. Si  $v = b.x$  ( $x \in L_{\bar{a}}$ ) alors nécessairement  $b.x \sqsubseteq w$  ce qui contredit  $w \in \text{ext}(\min(a^{-1}L), L_{\bar{a}})$ .

### A.2.2 LEMME 11.2

Nous montrons que :  $\forall L \subseteq E^* \setminus \{\epsilon\}, \forall a \in E, \forall w \in E^*, b.w \in \min(L) \iff b.w \in \text{ext}(\min(L_{\bar{a}}, L_a)$ .

$\Rightarrow$ : Soit  $b.w \in \min(L)$ . Puisque  $b.w \in L$  nous avons  $b.w \in L_{\bar{a}}$ . Nécessairement  $b.w$  appartient à  $\min(L_{\bar{a}})$  car le cas contraire est en contradiction avec l'hypothèse  $b.w \in \min(L)$ . De même  $b.w \in \text{ext}(\min(L_{\bar{a}}, L_a)$  sinon il existe  $a.x \in L_a \sqsubseteq L$  tel que  $a.x \sqsubset b.w$  ce qui contredit une nouvelle fois l'hypothèse.

$\Leftarrow$ : Soient  $b.w \in \text{ext}(\min(L_{\bar{a}}, L_a)$  et  $x \in L$ ; nous supposons que  $x \sqsubseteq b.w$ :

- $x = a.u$ : dans ce cas,  $x \sqsubseteq b.w$  avec  $x \in L_a$ ; ceci contredit l'hypothèse  $b.w \in \text{ext}(\min(L_{\bar{a}}, L_a)$ ;
- $x = y.u$ : avec  $y \in E - \{a\}$ , et dans ce cas  $x \sqsubseteq b.w$  avec  $x \in L_{\bar{a}}$ ; ce qui contredit le fait que  $b.w \in \min(L_{\bar{a}})$ .

### A.2.3 THÉORÈME 11.1

#### Définition A.2.1

La hauteur  $H(N)$  d'un diagramme de séquences  $N$  est définie inductivement par :

- $H(\epsilon) = 0$  et  $H(\emptyset) = 0$ ;
- $H(\Delta(a, N_1, N_2)) = 1 + \max\{H(N_1), H(N_2)\}$ .

□

Nous montrons par induction sur  $n = H(H)$  que : si  $N \approx N'$  alors  $\llbracket N \rrbracket = \llbracket N' \rrbracket$ . Soient  $N$  et  $N'$  deux nœuds tels que  $N \approx N'$ .

- $N = \epsilon$  ou  $N = \emptyset$ . Puisque  $N \approx N'$ , nous avons nécessairement  $N = N'$  et par suite,  $\llbracket N \rrbracket = \llbracket N' \rrbracket$ .
- Si  $n > 0$  nous supposons la propriété vraie pour tout  $k < n$ . Nous avons  $N = \Delta(a, N_1, N_2)$  et  $N' = \Delta(a, N'_1, N'_2)$  avec  $N_1 \approx N'_1$  et  $N_2 \approx N'_2$ . Puisque  $H(N_1) < n$  et  $H(N_2) < n$  nous avons, pour  $i = 1, 2$ ,  $\llbracket N_i \rrbracket = \llbracket N'_i \rrbracket$  et par suite  $\llbracket N \rrbracket = a\llbracket N_1 \rrbracket \cup \llbracket N_2 \rrbracket = a\llbracket N'_1 \rrbracket \cup \llbracket N'_2 \rrbracket = \llbracket N' \rrbracket$ .

## A.2.4 THÉORÈME 11.2

Nous montrons par induction sur  $n = H(N)$  que : si  $N$  et  $N'$  sont des diagrammes réduits et ordonnés tels que  $\llbracket N \rrbracket = \llbracket N' \rrbracket$  alors  $N \approx N'$ .

Soient  $N$  et  $N'$  deux nœuds que nous supposerons différents et tels que  $\llbracket N \rrbracket = \llbracket N' \rrbracket$ .

- Si  $n = 0$  alors  $N = \epsilon$  ou  $N = \emptyset$ .  $N'$  étant réduit il ne peut être de la forme  $\Delta(a, N_1, N_2)$  car dans ce cas  $\llbracket N' \rrbracket \neq \{\epsilon\}$  et  $\llbracket N' \rrbracket \neq \{\}$  ( $N_1 \neq \emptyset$ ). Nécessairement nous avons  $N = N'$ .
- Si  $n > 0$  supposons la propriété vraie pour tout  $k < n$ . Nous avons  $N = \Delta(a, N_1, N_2)$  et  $\llbracket N \rrbracket = \llbracket N' \rrbracket$ .  $N'$  est nécessairement de la forme  $N' = \Delta(a', N'_1, N'_2)$ . D'autre part, puisque  $\llbracket N \rrbracket = \llbracket N' \rrbracket$ ,  $a = \lambda(\llbracket N \rrbracket) = \lambda(\llbracket N' \rrbracket) = a'$ . De plus, puisque  $N$  et  $N'$  sont ordonnés,  $a.\llbracket N_1 \rrbracket \cap \llbracket N_2 \rrbracket = \emptyset$  et  $a.\llbracket N'_1 \rrbracket \cap \llbracket N'_2 \rrbracket = \emptyset$ . Par conséquent nous avons  $a.\llbracket N_1 \rrbracket = a.\llbracket N'_1 \rrbracket$  et  $\llbracket N_2 \rrbracket = \llbracket N'_2 \rrbracket$ . Par hypothèse de récurrence nous obtenons  $N_1 \approx N'_1$  et  $N_2 \approx N'_2$ . On conclut que  $N \approx N'$ .

## A.2.5 THÉORÈME 11.3

### A.2.5.1 Union

Nous montrons par induction sur  $n = H(N) + H(N')$  que  $\llbracket \text{union}(N, N') \rrbracket = \llbracket N \rrbracket \cup \llbracket N' \rrbracket$ .

Le cas  $n = 0$  est évident. Soient  $N$  et  $N'$  deux nœuds et  $n > 0$  tels que  $n = H(N_1) + H(N_2)$ . Nous supposons que la propriété est vraie pour  $k < n$ . Nous montrons la propriété que pour le cas le plus général où  $N = \Delta(a, N_1, N_2)$  et  $N' = \Delta(a', N'_1, N'_2)$ ; les autres cas sont triviaux.

- a = a' :** Par définition  $\llbracket \text{union}(N, N') \rrbracket = a.\llbracket \text{union}(N_1, N'_1) \rrbracket \cup \llbracket \text{union}(N_2, N'_2) \rrbracket$ . L'hypothèse de récurrence nous donne  $\llbracket \text{union}(N_1, N'_1) \rrbracket = \llbracket N_1 \rrbracket \cup \llbracket N'_1 \rrbracket$  et  $\llbracket \text{union}(N_2, N'_2) \rrbracket = \llbracket N_2 \rrbracket \cup \llbracket N'_2 \rrbracket$ . On obtient alors  $\llbracket \text{union}(N, N') \rrbracket = a.\llbracket N_1 \rrbracket \cup a.\llbracket N'_1 \rrbracket \cup \llbracket N_2 \rrbracket \cup \llbracket N'_2 \rrbracket = \llbracket N \rrbracket \cup \llbracket N' \rrbracket$ .
- a < a' :** Par définition  $\llbracket \text{union}(N, N') \rrbracket = a.\llbracket N_1 \rrbracket \cup \llbracket \text{union}(N_2, N') \rrbracket$ . L'hypothèse de récurrence étant valable pour le couple  $(N_2, N')$  nous avons  $\llbracket \text{union}(N_2, N') \rrbracket = \llbracket N_2 \rrbracket \cup \llbracket N' \rrbracket$ . Nous obtenons encore le résultat  $\llbracket \text{union}(N, N') \rrbracket = a.\llbracket N_1 \rrbracket \cup \llbracket N_2 \rrbracket \cup \llbracket N' \rrbracket = \llbracket N \rrbracket \cup \llbracket N' \rrbracket$ .
- a > a' :** Analogue au cas précédent.

### A.2.5.2 Concaténation

Nous procédons de manière analogue à l'union pour montrer que  $\llbracket \text{concat}(N, N') \rrbracket = \llbracket N \rrbracket \cdot \llbracket N' \rrbracket$ .

Le cas  $n = 0$  est évident. Soient  $N$  et  $N'$  deux noeuds et  $n > 0$  tels que  $n = H(N_1) + H(N_2)$ . Nous supposons que la propriété est vraie pour  $k < n$ . Nous montrons la propriété que pour le cas le plus général où  $N = \Delta(a, N_1, N_2)$  et  $N' = \Delta(a', N'_1, N'_2)$ ; les autres cas sont triviaux.

Par définition  $\llbracket \text{concat}(N, N') \rrbracket = a \cdot \llbracket \text{concat}(N_1, N') \rrbracket \cup \llbracket \text{concat}(N_2, N') \rrbracket$ . L'hypothèse de récurrence nous permet d'écrire que  $\text{concat}(N_1, N') = \llbracket N_1 \rrbracket \cdot \llbracket N' \rrbracket$  et  $\text{concat}(N_2, N') = \llbracket N_2 \rrbracket \cdot \llbracket N' \rrbracket$  donc  $\llbracket \text{concat}(N, N') \rrbracket = a \cdot \llbracket N_1 \rrbracket \cdot \llbracket N' \rrbracket \cup \llbracket N_2 \rrbracket \cdot \llbracket N' \rrbracket = (a \cdot \llbracket N_1 \rrbracket \cup \llbracket N_2 \rrbracket) \cdot \llbracket N' \rrbracket = \llbracket N \rrbracket \cdot \llbracket N' \rrbracket$ .

### A.2.5.3 Intersection

Nous montrons par induction sur  $n = H(N) + H(N')$  que  $\llbracket \text{inter}(N, N') \rrbracket = \llbracket N \rrbracket \cap \llbracket N' \rrbracket$ .

Nous ne montrons que le cas général. Soient  $N = \Delta(a, N_1, N_2)$  et  $N' = \Delta(a', N'_1, N'_2)$  deux noeuds et  $n > 0$  tels que  $n = H(N_1) + H(N_2)$ . Nous supposons que la propriété est vraie pour  $k < n$ .

**a = a' :** Par définition  $L = \llbracket \text{inter}(N, N') \rrbracket = a \cdot \llbracket \text{inter}(N_1, N'_1) \rrbracket \cup \llbracket \text{inter}(N_2, N'_2) \rrbracket$ . Par hypothèse de récurrence,  $\llbracket \text{inter}(N_1, N'_1) \rrbracket = \llbracket N_1 \rrbracket \cap \llbracket N'_1 \rrbracket$  et  $\llbracket \text{inter}(N_2, N'_2) \rrbracket = \llbracket N_2 \rrbracket \cap \llbracket N'_2 \rrbracket$ . On obtient  $L = a \cdot \llbracket N_1 \rrbracket \cap a \cdot \llbracket N'_1 \rrbracket \cup \llbracket N_2 \rrbracket \cap \llbracket N'_2 \rrbracket$ . Puisque  $a \cdot \llbracket N_1 \rrbracket \cap \llbracket N_2 \rrbracket = \emptyset$  et  $a \cdot \llbracket N'_1 \rrbracket \cap \llbracket N'_2 \rrbracket = \emptyset$  nous obtenons  $L = (a \cdot \llbracket N_1 \rrbracket \cup \llbracket N_2 \rrbracket) \cap (a \cdot \llbracket N'_1 \rrbracket \cup \llbracket N'_2 \rrbracket) = \llbracket N \rrbracket \cap \llbracket N' \rrbracket$ .

**a < a' :** Par définition  $L = \llbracket \text{inter}(N, N') \rrbracket = \llbracket \text{inter}(N_2, N') \rrbracket$  et par hypothèse de récurrence,  $L = \llbracket N_2 \rrbracket \cap \llbracket N' \rrbracket$ . D'autre part,  $\llbracket N \rrbracket \cap \llbracket N' \rrbracket = a \cdot \llbracket N_1 \rrbracket \cap \llbracket N' \rrbracket \cup \llbracket N_2 \rrbracket \cap \llbracket N' \rrbracket$  mais  $a < a'$  implique  $a^{-1} \llbracket N' \rrbracket = \emptyset$  d'où  $a \cdot \llbracket N_1 \rrbracket \cap \llbracket N' \rrbracket = \emptyset$  et  $L = \llbracket N \rrbracket \cap \llbracket N' \rrbracket$ .

**a > a' :** Analogue au cas précédent.

### A.2.5.4 Extraction

Avant le montrer la correction des équations de l'extraction nous énonçons un ensemble de propriétés (facilement vérifiables) de cette opération.

Si  $A, B \subseteq E^*$  et  $a, b \in E$  alors :

**Fait A.1**  $\text{ext}(A, B \cup C) = \text{ext}(A, B) \cap \text{ext}(A, C)$

**Fait A.2**  $\text{ext}(A \cup B, C) = \text{ext}(A, C) \cup \text{ext}(B, C)$

**Fait A.3**  $\text{ext}(a.A, a.B) = a \cdot \text{ext}(A, B)$

**Fait A.4**  $a \neq b \Rightarrow \text{ext}(a.A, b.B) = a \cdot \text{ext}(A, b.B)$

**Fait A.5**  $\text{ext}(A, B) = \text{ext}(A_a, B_a) \cap \text{ext}(A_a, B_{\bar{a}}) \cup \text{ext}(A_{\bar{a}}, B)$

**Fait A.6**  $\text{ext}(A, B) = a \cdot [\text{ext}(a^{-1}A, a^{-1}B) \cap \text{ext}(a^{-1}A, B_{\bar{a}})] \cup \text{ext}(A_{\bar{a}}, B)$

Nous pouvons maintenant montrer par induction sur  $n = H(N) + H(N')$  que  $\llbracket \text{ext}(N, N') \rrbracket = \text{ext}(\llbracket N \rrbracket, \llbracket N' \rrbracket)$ .

Nous ne considérons que le cas général où  $N = \Delta(a, N_1, N_2)$  et  $N' = \Delta(a', N'_1, N'_2)$ , et nous supposons que la propriété est vraie pour  $k < n$ . Nous posons  $L = \llbracket \text{ext}(N, N') \rrbracket$ . Trois cas sont à considérer :

**a = a'** : Nous avons  $L = a.(\llbracket \text{ext}(N_1, N'_1) \rrbracket \cap \llbracket \text{ext}(N_1, N'_2) \rrbracket) \cup \llbracket \text{ext}(N_2, N') \rrbracket$ .  
D'après l'hypothèse de récurrence, nous obtenons que :

$$\begin{aligned} L &= a.(\text{ext}(\llbracket N_1 \rrbracket, \llbracket N'_1 \rrbracket) \cap \text{ext}(\llbracket N_1 \rrbracket, \llbracket N'_2 \rrbracket)) \cup \text{ext}(\llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= (\text{ext}(a.\llbracket N_1 \rrbracket, a.\llbracket N'_1 \rrbracket) \cap \text{ext}(a.\llbracket N_1 \rrbracket, \llbracket N'_2 \rrbracket)) \cup \text{ext}(\llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= \text{ext}(a.\llbracket N_1 \rrbracket, a.\llbracket N'_1 \rrbracket \cup \llbracket N'_2 \rrbracket) \cup \text{ext}(\llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= \text{ext}(a.\llbracket N_1 \rrbracket, \llbracket N' \rrbracket) \cup \text{ext}(\llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= \text{ext}(a.\llbracket N_1 \rrbracket \cup \llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= \text{ext}(\llbracket N \rrbracket, \llbracket N' \rrbracket) \end{aligned}$$

**a < a'** : Nous avons  $L = a.\llbracket \text{ext}(N_1, N') \rrbracket \cup \llbracket \text{ext}(N_2, N') \rrbracket$ . D'après l'hypothèse de récurrence, nous obtenons que :

$$\begin{aligned} L &= a.\text{ext}(\llbracket N_1 \rrbracket, \llbracket N' \rrbracket) \cup \text{ext}(\llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= \text{ext}(a.\llbracket N_1 \rrbracket, \llbracket N' \rrbracket) \cup \text{ext}(\llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= \text{ext}(a.\llbracket N_1 \rrbracket \cup \llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= \text{ext}(\llbracket N \rrbracket, \llbracket N' \rrbracket) \end{aligned}$$

**a > a'** : Nous avons  $L = a.\llbracket \text{ext}(N_1, \Delta(a', N'_1, \emptyset)) \rrbracket \cap \llbracket \text{ext}(\Delta(a, N_1, \emptyset), N'_2) \rrbracket \cup \llbracket \text{ext}(N_2, N') \rrbracket$ . D'après l'hypothèse de récurrence, nous obtenons que :

$$\begin{aligned} L &= a.\text{ext}(\llbracket N_1 \rrbracket, a'.\llbracket N'_1 \rrbracket) \cap \text{ext}(a.\llbracket N_1 \rrbracket, \llbracket N'_2 \rrbracket) \cup \text{ext}(\llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= \text{ext}(a.\llbracket N_1 \rrbracket, a'.\llbracket N'_1 \rrbracket) \cap \text{ext}(a.\llbracket N_1 \rrbracket, \llbracket N'_2 \rrbracket) \cup \text{ext}(\llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= \text{ext}(a.\llbracket N_1 \rrbracket, a'.\llbracket N'_1 \rrbracket \cup \llbracket N'_2 \rrbracket) \cup \text{ext}(\llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= \text{ext}(a.\llbracket N_1 \rrbracket \cup \llbracket N_2 \rrbracket, \llbracket N' \rrbracket) \\ &= \text{ext}(\llbracket N \rrbracket, \llbracket N' \rrbracket) \end{aligned}$$

### A.2.5.5 Séquences minimales

Nous montrons par induction structurale que pour tout nœud  $N \in \mathcal{D}(E)$ ,  $\llbracket \min(N) \rrbracket = \min(\llbracket N \rrbracket)$ .

**(B)** Les cas où  $N \in \{\epsilon, \emptyset\}$  sont triviaux.

**(I)** Supposons la propriété vraie pour deux nœuds  $N_1$  et  $N_2$  et considérons une lettre  $a \in E$  et le nœud  $N = \Delta(a, N_1, N_2)$ . Nous avons :  $\llbracket \min(N_1) \rrbracket = \min(\llbracket N_1 \rrbracket)$  et  $\llbracket \min(N_2) \rrbracket = \min(\llbracket N_2 \rrbracket)$  et, par construction,  $\llbracket \min(N) \rrbracket = a.\text{ext}(\llbracket \min(N_1) \rrbracket, \llbracket N_2 \rrbracket) \cup \text{ext}(\llbracket \min(N_2) \rrbracket, a.\llbracket N_1 \rrbracket) = a.\text{ext}(\min(\llbracket N_1 \rrbracket), \llbracket N_2 \rrbracket) \cup \text{ext}(\min(\llbracket N_2 \rrbracket), a.\llbracket N_1 \rrbracket)$ . Mais  $\llbracket N_1 \rrbracket = a^{-1}\llbracket N \rrbracket$  et  $\llbracket N_2 \rrbracket = \llbracket N \rrbracket_{\bar{a}}$ , nous obtenons, par le théorème 11.1,  $\llbracket \min(N) \rrbracket = \min(\llbracket N \rrbracket)$ .

**A.2.6 THÉORÈME 11.4**

Pour tout  $n \geq 3$ , le nombre d'états de l'automate minimal reconnaissant le langage  $L_n$  est  $2n$  alors que le nombre d'états de l'automate minimal reconnaissant  $\min(L_n)$  est  $2^n$ .

Les langages  $L_n$  et  $\min(L_n)$  étant fini, le nombre d'états des automates minimaux reconnaissant ces langages est le nombre de résiduels de ces langages. Si  $L$  est un langage alors l'ensemble de ses résiduels est  $\text{Res}(L) = \{X \subseteq E^* \mid u \in E^*, u.X \subseteq L\}$ .

**Lemme A.1**

Le nombre de résiduels du langage  $L_n$  est  $2n + 1$ .

**Preuve :** Si  $u$  un mot de  $E^*$  alors nous considérons 5 cas :

1. si  $|u| = 0$  alors  $u^{-1}L_n = \epsilon^{-1}L_n = L_n$  ;
2. si  $|u| = 1$  alors  $u = a \in E$  et clairement  $u^{-1}L_n = \{a\} \cup E^{n-1}$  ;
3. si  $|u| = 2$  alors :
  - (a) si  $u = aa$ , avec  $a \in E$ , alors  $u^{-1}L_n = \{\epsilon\} \cup E^{n-2}$  ;
  - (b) si  $u = ab$ , avec  $a \neq b$  alors  $u^{-1}L_n = \emptyset \cup E^{n-2}$  ;
4. si  $|u| \in [3, n]$  alors  $u^{-1}L_n = u^{-1}E^n = \emptyset \cup E^{n-|u|}$  ;
5. si  $|u| > n$  alors  $u^{-1}L_n = \emptyset$ .

De ces différents cas nous déduisons que l'ensemble des résiduels non vides est :  $\{L_n, E^{n-2}, \{\epsilon\} \cup E^{n-2}\} \cup \{E^{n-i} \mid i = 3 \dots n\} \cup \{\{a\} \cup E^{n-1} \mid a \in E\}$ . Le nombre d'états de l'automate est par conséquent  $3 + n - 2 + n = 2n + 1$ .  $\diamond$

Si  $u \in E^*$  nous notons  $|u|_a$  le nombre d'occurrences de la lettre  $a$  dans  $u$ .

**Lemme A.2**

$\min(L_n) = \{aa \mid a \in E\} \cup \{u \in E^n \mid \forall a \in E, |u|_a = 1\}$

**Preuve :**

$\supseteq$  : Puisqu'ils sont minimaux pour la longueur, tous les carrés sur  $E$  sont nécessairement minimaux pour les sous-mots.

Considérons un mot  $u \in \{u \in E^n \mid \forall a \in E, |u|_a = 1\}$  et supposons qu'il existe un mot  $v \in \min(L_n)$  tel que  $v \neq u$  et  $v \sqsubseteq u$ . Nous avons évidemment  $|v| < |u|$ . Mais  $\min(L_n) \subseteq L_n$  donc  $v$  est soit de longueur 2, soit de longueur  $n$ . Nous en déduisons que  $|v| = 2$  et par suite que  $v$  est un carré (les seuls mots de longueur 2 dans  $L_n$ ). Le fait que  $v$  soit un carré est en contradiction avec  $v \sqsubseteq u$  car  $u$  ne contient pas deux lettres identiques ( $\forall a \in E, |u|_a = 1$ ).

$\subseteq$  : Si  $u \in \min(L_n)$  alors deux cas sont possibles :

1. si  $|u| = 2$  alors  $u$  est un carré sur  $E$  ;
2. si  $|u| = n$  alors  $u$  étant minimal pour les sous-mots, il ne doit pas contenir un carré sur  $E$  c'est à dire ne pas posséder deux lettres identiques.

◇

Comme précédemment nous étudions le nombre de résiduels non vides engendrés par des mots de longueur donné. Nous montrons tout d'abord le lemme suivant :

**Lemme A.3**

Si  $u$  et  $v$  sont des mots de même longueur  $l \geq 2$  qui ne sont pas des carrés sur  $E$  alors nous avons :

$$\forall a \in E, |u|_a = |v|_a \iff u^{-1} \min(L_n) = v^{-1} \min(L_n)$$

**Preuve :** Soient  $u$  et  $v$  deux mots de  $E^*$  tels que  $|u| = |v| = l, l \geq 2$  et qui ne sont pas des carrés. Si il existe une lettre  $a$  telle que  $|u|_a > 1$  alors  $u^{-1} \min(L_n) = \emptyset$  (car aucun mot de  $\min(L_n)$  ne peut avoir plus d'une lettre) ; nous supposons donc  $|u|_a \leq 1$  et  $|v|_a \leq 1$  pour tout  $a \in E$ .

$\Rightarrow$  : supposons que pour tout  $a \in E, |u|_a = |v|_a$  (ce qui revient à dire qu'ils possèdent les mêmes ensembles de lettres). Nous avons pour tout  $w \in E^{n-l}, u.w \in E^n$  et  $v.w \in E^n$  ; de plus  $|u| \geq 2$  et  $u$  n'est pas un carré donc nous avons  $u^{-1} \min(L_n) = u^{-1} E^n$ . Enfin, nous avons  $u^{-1} E^n = \{w \in E^{n-l} | \forall a \in E, |w|_a \leq 1 - |u|_a\}$  mais puisque pour tout  $a, |u|_a = |v|_a$  nous obtenons  $u^{-1} E^n = \{w \in E^{n-l} | \forall a \in E, |w|_a \leq 1 - |v|_a\} = v^{-1} E^n = v^{-1} \min(L_n)$ .

$\Leftarrow$  : supposons maintenant que  $u^{-1} \min(L_n) = v^{-1} \min(L_n)$  et considérons  $x \in u^{-1} \min(L_n) = u^{-1} E^n$ . Nous avons  $ux \in \min(L_n)$  donc pour tout  $a \in E, |ux|_a = 1$  donc  $|u|_a = 1 - |x|_a$ . En suivant le même raisonnement pour  $v$  nous arrivons  $|v|_a = 1 - |x|_a = |u|_a$ .

◇

Nous notons  $R_l = \{u^{-1} \min(L_n) / |u| = l\}$  et  $\text{card}(R_l)$  sont nombre d'éléments.

**Corollaire A.1**

Pour tout  $l \geq 3, \text{card}(R_l) = C_n^l$  éléments.

**Preuve :** En effet, puisque les mots de longueur  $l$  qui possèdent le même ensemble de lettres engendrent le même résiduel de  $\min(L_n)$  le nombre

d'éléments de  $R_l$  est le nombre de combinaison de  $l$  lettres choisies parmi  $n$  (sans répétition); d'où le résultat.  $\diamond$

Nous pouvons maintenant conclure la preuve du théorème avec le lemme suivant :

**Lemme A.4**

Le nombre de résiduels du langage  $\min(L_n)$  est  $2^n$ .

**Preuve :** Pour montrer ce résultat nous procédons comme précédemment en dénombrant le nombre résiduels engendrés par des mots de longueur fixée. Considérons un mot  $u$  de  $E^*$ ; nous avons :

- $\text{card}(R_0) = 1 = C_n^0$  car si  $|u| = 0$  alors  $u^{-1}\min(L_n) = \epsilon^{-1}\min(L_n) = \min(L_n)$ ;
- $\text{card}(R_1) = n = C_n^1$ . En effet, si  $|u| = 1$  alors  $u$  est une lettre de  $E$  et nous obtenons  $u^{-1}\min(L_n) = \{u\} \cup \{v \in E^{n-1} \mid \forall a \in E, |v|_a = 1 - |u|_a\}$ ;
- $\text{card}(R_2) = 1 + C_n^{n-2} = 1 + C_n^2$ . En effet, si  $|u| = 2$  alors
  - si  $u = aa$  pour  $a \in E$  alors  $u^{-1}\min(L_n) = \{\epsilon\}$ ;
  - si  $u = ab$  pour  $a \neq b \in E$  alors  $u^{-1}\min(L_n) = \{v \in E^{n-2} \mid \forall a \in E, |v|_a = 1 - |u|_a\}$ ;
- Pour tout  $l \in [3, n]$ ,  $\text{card}(R_l) = C_n^l$  d'après le corollaire A.1.

Le nombre de résiduels de  $\min(L_n)$  est donc  $\sum_{p=0}^{p=n} C_n^p = 2^n$  (le résiduel  $\{\epsilon\}$  de  $R_2$  est compté dans  $\text{card}(R_n)$ ).  $\diamond$



## GRAMMAIRE DU LANGAGE ALTARICA

Cette annexe présente la syntaxe concrète du langage Altarica. La grammaire est exprimée en BNF. L'abréviation *RegExp(E)* désigne les chaînes de caractères appartenant au langage associé à l'expression régulière *E*. La notation *[ construction ]* désigne une construction optionnelle. Nous utilisons l'abréviation *construction+* pour désigner une liste non vide de *construction*. Les symboles non terminaux sont écrits en *italique* et les mots-clés sont notés en gras (p.ex. `true`) et les éléments lexicaux entre cotes (p.ex. '=').

### MODÈLES ALTARICA

*modèle-altarica* ::= *liste-déclarations*

*liste-déclarations* ::= *déclaration+*

*déclaration* ::= *déclaration-de-constante*

| *déclaration-de-domaine*

| *déclaration-de-modèle*

### DÉCLARATION DES CONSTANTES

*déclaration-de-constante* ::= `const` *identifiant* '=' *expression-constante*

### DÉCLARATION DES DOMAINES

*déclaration-de-domaine* ::= `domain` *identifiant* '=' *domaine*

*domaine* ::= *domaine-intervalle*

| *domaine-énuméré*  
| *domaine-pré-défini*

*domaine-intervalle* ::= '[' *expression-constante-numérique* ',' *expression-constante-numérique* ']'

*domaine-énuméré* ::= '{' *liste-expressions-constantes* '}'

*domaine-pré-défini* ::= *bool* | *integer* | *symbol*

## DÉCLARATION DES MODÈLES DE COMPOSANTS

*déclaration-de-modèle* ::= *node* *identifiant* *composante-de-modèle* +  
*edon*

*composante-de-modèle* ::= *déclarations-variables*  
| *déclarations-événements*  
| *déclarations-sous-composants*  
| *définitions-transitions*  
| *définitions-assertions*  
| *définitions-vecteurs*  
| *déclarations-pour-outils*

## DÉCLARATIONS DES VARIABLES

*déclarations-variables* ::= *type-de-variable* ( *liste-identifiants* ':' ' *domaine* ['; ' ] ) +

*type-de-variable* ::= *flow* | *state* | *local*

*liste-identifiants* ::= *identifiants* ',' *liste-identifiants*  
| *identifiants*

## DÉCLARATIONS DES ÉVÉNEMENTS

*déclarations-événements* ::= *event* *liste-événements* ['; ' ]

*liste-événements* ::= *définition-événement* ',' *liste-événements*



*liste-instances-événements* ::= *instance-événement* ',' *liste-instances-événements*  
 | *instance-événement*

*instance-événement* ::= *identifiant-instance* ['?' '']

*identifiant-instance* ::= *identifiant* ',' *identifiant*  
 | *identifiant*

*contrainte-vecteur* ::= *op-contrainte-vecteur* *expression-entière-constante*

*op-contrainte-vecteur* ::= '=' | '<' | '<=' | '>' | '>='

### INFORMATIONS SUPPLÉMENTAIRES POUR LES OUTILS

*déclarations-pour-outils* ::= *extern* *déclarations-données*+

*déclaration-données* ::= *identifiant* '=' *RegExp(.\*)*;

### INFORMATIONS EXISTANTES

Au cours du projet AltaRica, certaines directives nécessaires aux outils externes ont été définies; en particulier l'état initial du composant.

*déclaration-état-initial* ::= 'initial\_state' '=' *liste-initialisations* ';' ;

*liste-initialisations* ::= *initialisation* ',' *liste-initialisations*  
 | *initialisation*

*initilisation* ::= *instance-de-variable* '=' *expression-constante*

*instance-de-variable* ::= *identifiant* ',' *instance-de-variable*  
 | *identifiant*

### EXPRESSIONS

*expression-constante* ::= *expression*

*expression-constante-numérique* ::= *expression*

*expression-constante-entière* ::= *expression*

*liste-expressions-constantes* ::= *expression* ',' *liste-expressions-constantes*  
| *expression*

*liste-expressions-booléennes* ::= *expression-booléenne* ',' *liste-expressions-booléennes*  
| *expression-booléenne*

*expression-booléenne* ::= *expression*

*expression* ::= *expression-disjonction* *op-ou* *expression*  
| *expression-disjonction*

*op-ou* ::= '|' | **or**

*expression-disjonction* ::= *relation* *op-et* *expression-disjonction*  
| *relation*

*op-et* ::= '&' | **and**

*relation* ::= *atome-booléen* *op-rel-bool* *atome-booléen*  
| *atome-symbolique* *op-rel-sym* *atome-symbolique*  
| *terme-numérique* *op-rel-num* *terme-numérique*

*op-rel-bool* ::= '=' | '!' | '=' | *op-imply*

*op-imply* ::= '=>' | **imply**

*op-rel-num* ::= '=' | '!' | '=' | '<' | '<=' | '>=' | '>'

*op-rel-sym* ::= '=' | '!' | '='

*atome-booléen* ::= **true**  
| **false**  
| *instance-de-variable*  
| *op-not* *atome-booléen*  
| *si-alors-sinon*  
| *cardinalité*  
| *expression-parenthésée*

*op-not* ::= '~' | **not**

ANNEXE B. GRAMMAIRE DU LANGAGE ALTARICA

*op-card* ::= '@' | card

*si-alors-sinon* ::= ite '(' *expression-booléenne* ',' *expression* ',' *expression* ')'  
 | if *expression-booléenne* then *expression* else *expression*

*cardinalité* ::= *op-card* '[' *bornes-cardinalité* ']' '(' *liste-expressions-booléennes* ')'

*bornes-cardinalité* ::= *expression-constante-entière*  
 | *expression-constante-entière*, *expression-constante-entière*

*expression-parenthésée* ::= '(' *expression* ')'

*atome-symbolique* ::= *identifiant*  
 | *instance-de-variable*  
 | *si-alors-sinon*  
 | *expression-parenthésée*

*terme-numérique* ::= *terme-multiplicatif* '+' *terme-numérique*  
 | *terme-multiplicatif* '-' *terme-numérique*  
 | *terme-multiplicatif*

*terme-multiplicatif* ::= *atome-numérique* '\*' *terme-multiplicatif*  
 | *atome-numérique* '/' *terme-multiplicatif*  
 | *atome-numérique*

*atome-numérique* ::= + *atome-numérique*  
 | - *atome-numérique*  
 | *entier*  
 | *instance-de-variable*  
 | *si-alors-sinon*  
 | *expression-parenthésée*

*identifiant* ::= *RegExp*([a-zA-Z][\_0-9a-zA-Z]+)  
 | """ *RegExp*(.\*) """

*entier* ::= *RegExp*([0-9]+)

## COMMENTAIRES

```
commentaire ::= '/* RegExp(.*) */'  
                | '// RegExp(.*)'
```